Dissertation

submitted to the Combined Faculty of

Natural Sciences and Mathematics

of Heidelberg University, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

Oliver Julien Breitwieser

born in: Essen, Germany

Oral examination: 2021 July 15th

Learning
by
Tooling:

Novel Neuromorphic Learning Strategies

in

Reproducible Software Environments

Referees:
Dr. habil. Johannes Schemmel (Heidelberg University)
Prof. Dr. Holger Fröning (Heidelberg University)

The ultimate fate of all intelligent beings has
always been to become as grand as their thoughts.

— Liu Cixin, Death's End

Abstract

Neuromorphe Hardware ermöglicht neuartige Rechenparadigmen. Hierzu stellen wir zwei innovative Lernstrategien vor: Zum einen führen wir Spike-basiertes Deep Learning mit LIF-Neuronen in einem Time-To-First-Spike Kodierungsschema durch, das sich darauf konzentriert, Klassifizierungsergebnisse mit so wenigen Spikes so schnell wie möglich zu erreichen. Dies ist entscheidend für biologische Akteure, die unter Umgebungsdruck schnelle Reflexe benötigen bei gleichzeitiger Energieeffizienz. Wir leiten exakte Lernregeln ab und führen Backpropagation mit Spike-Zeiten von LIF-Neuronen sowohl in Software als auch auf der BrainScaleS-Hardwareplattform durch. Zum anderen präsentieren wir schnelle energieeffiziente analoge Inferenz auf BrainScaleS-2. In diesem nicht-spikenden Modus verwenden wir CNNs zur Überprüfung medizinischer EKG-Daten auf Vorhofflimmern. Das neu in Betrieb genommene BrainScaleS-2 Mobilsystem hat dabei erfolgreich am Wettbewerb “Energieeffizientes KI-System” des Bundesministeriums für Bildung und Forschung teilgenommen und bewiesen, dass es zuverlässig arbeitet. Diese neuen Rechenparadigmen von Grund auf zu entwickeln ist eine Herkulesaufgabe in Bezug auf den erforderlichen Arbeitsaufwand und die Menge an beteiligten Personen. Daher stellen wir Methoden vor, die kollaborative Entwicklung sowie Einsatz von wissenschaftlicher Software ermöglichen. Insbesondere konzentrieren wir uns auf die explizite Verfolgung getrennter Gruppen von Software-Abhängigkeiten mittels Spack, einem existierenden Paketmanager für Hochleistungsrechnen. Sie werden als monolithische Singularity-Container in einem kontinuierlichen Veröffentlichungsschema nach gründlicher Überprüfung bereitgestellt. Diese Praktiken ermöglichen es uns die Entwicklung unserer neuromorphen Plattform voranzutreiben und gleichzeitig die Reproduzierbarkeit von Experimenten zu fördern, ein noch nicht gelöstes Problem in den softwaregestützten Wissenschaften. Durch die Einführung von quiggeldy, einem Micro-Scheduling-Service, der die verschachtelte Ausführung von Experimentschritte verschiedener Nutzer ermöglicht, erreichen wir bessere Hardware-Interaktivität, Stabilität und Experimentdurchsatz.

Neuromorphic hardware enables novel modes of computation. We present two innovative learning strategies: First, we perform spike-based deep learning with LIF neurons in a Time-To-First-Spike coding scheme that focuses on achieving classification results with as few spikes as early as possible. This is critical for biological agents operating under environmental pressure, requiring quick reflexes while conserving energy. Deriving exact learning rules, we perform backpropagation on spike-times of LIF neurons in both software and on the BrainScaleS hardware platform. Second, we present fast energy-efficient analog inference on BrainScaleS-2. In this non-spiking mode, we use convolutional neural networks to check medical ECG traces for atrial fibrillation. The newly commissioned BrainScaleS-2 Mobile system has successfully participated and proven to operate reliably in the “Energy-efficient AI system” competition held by the German Federal Ministry of Education and Research. Developing these new computing paradigms from the ground up is a Herculean effort in terms of work required and people involved. Therefore, we introduce tooling methods to facilitate collaborative scientific software development and deployment. In particular, we focus on explicitly tracking disjoint sets of software dependencies via Spack, an existing package manager aimed at high performance computing. They are deployed as monolithic Singularity containers in a rolling-release schedule after thorough verification. These practices enable us to confidently advance our neuromorphic platform while fostering reproducibility of experiments, a still unsolved problem in software-aided sciences. By introducing quiggeldy, a micro-scheduling service operating on interleaved experiment-steps by different users, we achieve better hardware interactivity, stability and experiment throughput.

Contents

1 Motivation & Outline **1**

I Background **7**

2 Machine Learning **9**
- 2.1 Supervised Learning 10
- 2.2 Deep Learning 12
- 2.3 Spiking Neural Networks 15

3 Neuromorphic Hardware: The BrainScaleS platform **19**
- 3.1 BrainScaleS-1 21
- 3.2 BrainScaleS-2 27

4 Software Development in Science **35**
- 4.1 Importance of High-Quality Software 35
- 4.2 Scientific Software Development 38
- 4.3 Reproducibility of Software-Aided Science 39
- 4.4 Software Concepts 42

II Facilitating Collaborative Software Development in Science **57**

5 Motivation & Outline **59**

6 The BrainScaleS-2 Software-Stack: An Overview **63**

7 Workflow: Continuous Integration **71**
- 7.1 Software Build Automation: Jenkins 71
- 7.2 Code Review: Gerrit 74
- 7.3 Building the full Stack: `waf` 78

8 Managing and Deploying an Evolving Set of Software Dependencies **85**
- 8.1 Managing Dependencies in HPC Environments: Spack 86
- 8.2 Software Environments via `visionary` Containers 97
- 8.3 Container Build Process Automation: `yashchiki` 112

9 Packing up the Cluster: Slurm in Containers **125**
- 9.1 Slurm Deployment 126
- 9.2 Deploying Binaries Cluster-wide: `clusterize` 132

10 Avoid con{ges,ten}tion via Micro-Scheduling: `quiggeldy` **137**
- 10.1 The Case for interactive analog Hardware 137

10.2 Core Principles . . . 140
10.3 Integration into BrainScaleS-2 Software Stack . . . 141
10.4 Reinitialization to enforce Structure in larger Experiments . . . 143
10.5 Implementation . . . 145
10.6 Integration into other Layers . . . 159
10.7 Deployments . . . 160
10.8 Overhead-Evaluation . . . 164
10.9 Transparent Cluster-Integration: `hagen-daas` . . . 167

11 Remaining Challenges for Deployment 171

III Robust Learning Strategies in Neuromorphic Hardware 177

12 Neuromorphic Learning with Time-to-first-Spike Coding 179
12.1 Background: Learning with IF-neurons . . . 180
12.2 Extending to LIF-neurons . . . 182
12.3 Results . . . 195

13 Fast Analog Inference on BrainScaleS-2 211
13.1 Poor Man's Introduction to Electrocardiography . . . 213
13.2 Classifying ECG with CNNs . . . 215
13.3 Results . . . 227

14 Discussion & Conclusion 237

IV Appendix 245

A Contributions 247
A.1 In Thesis . . . 247
A.2 Supervision . . . 248
A.3 Publications . . . 249
A.4 Software . . . 253

B Supplementary information 255
B.1 Full List of `quiggeldy` CLI Arguments . . . 255
B.2 Parameter & Software States . . . 258

C Acronyms and Technical Terms 267

D List of Figures 275

E List of Tables 277

V Bibliography **279**

Motivation & Outline 1

A core principle of evolution is that it favors the more productive. Those more efficient in their ability to accrue and employ resources have a higher likelihood to prevail [Akkerhuis et al., 1999]. Over time, this has led to increasingly complex biological structures, culminating (so far) in what we perceive to be the most complex structure known: the human brain [Fischbach, 1992]. One of its advantages is the ability to create tools in order to perform previously impossible tasks or drastically reduce the time required; a trait shared with only a few other species [Shumaker et al., 2011].

At large, human society seems to follow the same principle, always striving to accomplish more with less. Machines are key to this, first replacing manual [Deane et al., 1979] then mental labor [O'Regan, 2012]. Throughout history, humankind tinkered with devices to perform computation with increasing intricacy. Notable mechanical examples include the Antikythera mechanism [Lin et al., 2016], used to calculate star positions, mechanical calculators first conceptualized in the 18th century [Müller et al., 1786], programmable Jacquard looms[1] [Posselt, 1887] and Charles Babbage's famed Difference[2] and Analytical engines [Menabrea, 1842], for which Ada Lovelace wrote the first program computing Bernoulli numbers [Lovelace, 1843; Wolfram, 2015].

The modern computing landscape as we know it today emerged in the 20th century, first via vacuum tubes[3] then MOSFET[4] technology, largely based on the von Neumann paradigm [Neumann, 1945]. While the fabled *Moore's law* [Moore et al., 1965], "predicting" a doubling of integration density every 18–24 months, is still holding, it is soon bound to fail [Theis et al., 2017]: Semiconductor manufacturing is able to produce structures down to 2 nm resolution [Ye et al., 2019], merely one order of magnitude larger than involved atoms themselves ($\approx 0.1\,\mathrm{nm}$). Further miniaturization, the main source for improved integration density, will not be possible using the same technology.

Judging from recent trends, there is an ever increasing demand for more compute power. For example, the field of machine learning has made enormous progress over the last 30

[1]Arguably, Jacquard looms did *not* perform computation, however, they were the first commercial machine allowing for changes in behavior via *programming*, using punched cards without any other physical modifications.

[2]Babbage originally planned for his difference engines to compute up to seventh order polynomials with thirty digits precision (almost 100 bit). While never realized, they have been shown to work as designed [O'Regan, 2012, Chapter 12.3]. Only a scaled-down working prototype was constructed in his lifetime, able to compute up to second order differences with 6-digit numbers (almost 20 bit) [Snyder, 2011, chap. 8].

[3]There are already many notable examples for the "first generation" of digital computers using vacuum tubes, including the Atanasoff-Berry computer (ABC) (developed at the University of Iowa in 1942), the Colossus (developed at Bletchley Park in 1943), the ENIAC (developed in 1946), the UNIVAC I and the Whirlwind computer (both developed in 1951) [O'Regan, 2012, chap. 2.4.1].

[4]Metal-Oxide-Semiconductor Field-Effect Transistor

years. Here, applying concepts of brain-like information processing has led to tremendous progress in image recognition [Krizhevsky et al., 2012; LeCun et al., 2015], strategic decision making [Silver et al., 2017] and language processing [Brown et al., 2020], even leading to various commercial-grade applications,[5] e.g., [Wu et al., 2016]. Unfortunately, these impressive results come at a *massive* computational cost [Schwartz et al., 2020], often in the form of relatively simple operations scaled to great concurrency. It is expected that, by 2030, 8–50 % of the global energy consumption will be spent on computation [Andrae et al., 2015].

Going in reverse – building neuroscientific models to understand inner workings of the brain – is still limited to small bottom-up approaches by lack of compute power. [Yamazaki et al., 2021] achieve human-scale simulations of the cerebellum, using a model consisting of approximately 68 billion neurons and 5.4 trillion synapses. Even using K computer[6] with 82 944 CPUs, one minute of cerebellar (non-functional) activity took 10 h to simulate. This is a slowdown by a factor of about 600 compared to realtime – without any meaningful forms of I/O[7] or exchange with an environment. Hence, even if we had a complete theory of the inner workings of the brain, we lack the computational capabilities to efficiently verify it.

New computing paradigms are needed. They are typically specialized to particular use-cases [Jouppi et al., 2017]. One potential avenue is neuromorphic computing [Mead, 1989; Mead, 1990]. It covers various technological concepts whose unifying property is mimicking brain-like information processing directly in hardware – just like machine learning does in an algorithmic sense (cf. Chapter 3). While, relatively speaking, still a young field, it has steadily increased in relevance over the past 30 years [Jaeger, 2021]. The aim is to construct particularly fast and energy-efficient[8] forms of computation.

In this thesis, we focus on a particular neuromorphic platform: BrainScaleS.[9] It achieves accelerated computation at a factor of $10^3 - 10^5$ compared to realtime by *emulating* analog neuron dynamics. There is no layer of abstractions solving differential equations: One set of physical systems (electric potentials in CMOS[10] transistors) emulates another (membrane potentials of biological neurons in the brain) in a behavioral sense at an accelerated time-scale. This is of particular interest when compared to the "speed-up" factor of $1.7 \cdot 10^{-3}$ in conventional simulations, mentioned above. Emulated neurons communicate by exchanging digital messages, so-called *spikes*, hence the term "mixed-signal".

Building such a neuromorphic platform from the ground up is a Herculean effort in terms of work required. In today's world, most of this is aided by the latest generation of tools:

[5]The hype for "general artificial intelligence" seems to supersede even "clean cold fusion energy". It is the author's opinion that both will be available at roughly the same time.

[6]K computer was ranked 1 (2011) to 20 (2019) among supercomputers in the world, cf. `https://www.top500.org/system/177232/` (visited on 2021-05-03)

[7]Input/Output

[8]Especially terms of energy-efficiency, we have a long way to go: The brain's total energy consumption is estimated to be 20 % of total calorie intake, about 20 W [Raichle et al., 2002].

[9]BrainScaleS Mixed-Signal Accelerated Neuromorphic Systems, [Schemmel et al., 2008; Schemmel et al., 2010; Schemmel et al., 2017; Schemmel et al., 2020]

[10]Complementary Metal-Oxide-Semiconductor

software. It serves as a way to record design blueprints, describe models and precisely control the behavior of all kinds of machines. Especially in the case of an accelerated neuromorphic substrate, we require high-quality software to precisely control and keep up with experiment execution. But, as with most tools, wielding it must be learnt [Slaughter et al., 1998].

First and foremost, scientists want to probe ideas and generate knowledge. Hence, they focus more on producing immediate results rather than preserving their results for others to reproduce [Anzt et al., 2020]. This is software-aided sciences' very own *replication crisis* [Baker, 2016; Krafczyk et al., 2021]. In large-scale projects such as BrainScaleS with scope and duration vastly exceeding the average stay of any one person involved, this is a *problem that needs to be addressed.* While every new sub-project aims to build atop of methods developed previously, verifying correctness while ensuring continued functionality requires conscious effort [National Academies of Sciences et al., 2019]. Without it, it is often easier to re-invent the wheel rather than verifying an existing one to roll downhill properly.

In this thesis we introduce methods for *collaborative software development and deployment in a scientific environment* (cf. Part II). By introducing methods from professional software engineering, we streamline development. Thoroughly tracking all moving parts and verifying changes to the system, we increase confidence: Confidence in the reliability and usability of previous work as well as confidence that new improvements will not result in unforeseen side-effects in far remote corners of the ecosystem.

Following these principles allows for the development of *novel and robust neuromorphic learning strategies* on BrainScaleS, two of which we detail in this manuscript (cf. Part III). They show-case both fundamental aspects of neuromorphic hardware, being *fast* and *energy-conserving*, while employing two completely different computing paradigms. One performs learning on spike-times to solve image classification problems with as few spikes as early as possible (cf. Chapter 12), while the other uses the same circuitry in an analog approach to conventional machine learning to classify real-world medical data as efficient as possible (cf. Chapter 13). This is relevant for both TinyML applications [Lin et al., 2020; Banbury et al., 2021] – an emerging field of machine learning that focusses on resource-efficient and low-powered learning on embedded devices – as well as edge computing [Shi et al., 2016; Park et al., 2018; Dongarra et al., 2019; Chen et al., 2019] – where data processing (including inference) happen in direct proximity to data storage, highly relevant for I/O-bound tasks in data centers.

Thesis Outline

The manuscript is structured as follows:

Part I is intended to bring all readers up to speed if concepts in later parts are unfamiliar. In Chapter 2 we give a brief overview of machine learning and emerging trends, especially with regards to SNNs.[11] Chapter 3 introduces neuromorphic hardware and the BrainScaleS platform in particular with its different generations. Finally, in Chapter 4 we motivate why

[11]Spiking Neural Networks

proper software engineering is important, especially in a scientific context. It illustrates the core problems related to reproducibility (cf. Section 4.3) and gives an overview of and alternative solutions to concepts used in Part II (cf. Section 4.4).

Part II then focuses on how to structure and facilitate collaborative software development and deployment in a scientific environment. We start by giving a detailed overview of the software stack for the most recent BrainScaleS software generation (cf. Chapter 6), intended to showcase the scale at which we are working to foster collaboration. Chapter 7 details how basic software engineering concepts are implemented at Electronic Vision(s)[12] such as code review and CI.[13]

A core result, detailed in of this thesis is the explicit tracking of software dependencies (cf. Section 8.1) that are then built into easily-accessible containers (cf. Section 8.2) via a fully automated procedure (cf. Section 8.3) that preserves all existing functionality. This concept is extended to the whole cluster deployment (cf. Chapter 9) allowing for streamlined development of changes to the cluster software that can be deployed with confidence.

Another important result is the development of `quiggeldy` (cf. Chapter 10), a micro-scheduler for neuromorphic hardware, that ensures better hardware interactivity for users while ensuring users to properly track used hardware parameters.

Part III puts these methods to the test by presenting two novel learning strategies: In Chapter 12, Time-To-First-Spike, a spike-based deep learning approach that focuses on achieving classification results with as few spikes as early possible. We derive exact spike-time input-output relations for spike-times of neurons that allow for exact learning rules which we evaluate in software and on hardware.

Finally, we present fast analog inference on BrainScaleS-2, the latest generation of prototype chips (cf. Chapter 13). In this non-spiking mode we classify medical ECG traces via CNNs in a fast and energy-efficient manner. The newly commissioned BrainScaleS-2 Mobile system has successfully participated and proven to operate reliably in the "Energy-efficient AI system" competition held by the German Federal Ministry of Education and Research.

For full details on the author's contributions to each topic discussed, please refer to Appendix A.

[12]Electronic Vision(s) Group at the Kirchhoff-Institute for Physics in Heidelberg
[13]Continuous Integration

If You Want to Go Fast, Go Alone.
If You Want to Go Far, Go Together.

African Proverb

I
Background

Machine Learning 2

The discipline of machine learning, in its simplest[1] definition, concerns itself with the idea of having a machine automatically solve a problem without being explicitly programmed how to achieve it [Samuel, 1959]. This can be understand as finding a universal function approximation: a mapping between two arbitrary spaces. These two spaces are typically denoted *input* and *output* data. We are given conjoined examples from both input and output (in the case of supervised learning) or from input only (in the case of unsupervised learning). Mixtures are also possible (denoted semi-supervised learning). The *model* is then able to map each input value to an output value. This is called *inference*. Finding the model, typically through an intricate step of algorithmic optimization, is called *training*. Typically,[2] training is performed on *a lot* of data samples: The more samples, the better a sufficiently sized model becomes [Sejnowski, 2020].

For example, the input for the fundamental MNIST[3] dataset[4] is comprised of 28×28 pixel values while the output denotes which handwritten digit is shown in the image, classified by human experts. More advanced versions of this task are concerned with mapping natural images of varying sizes to one of 1000 label categories [Krizhevsky et al., 2012] or natural language sentences describing the images' content [Karpathy et al., 2015]. Other machine learning tasks involve mapping the current state of an ongoing game of Go to the best next move to perform [Silver et al., 2016]. Here, it becomes obvious that generating the necessary training data is a whole challenge in itself because there are far less human experts able to play Go than there are to classify images [Silver et al., 2017]. Nevertheless, machine learning advanced to the point were models are able to beat the best humans in Go [Silver et al., 2017] and most intricate video games [OpenAI et al., 2019]. Other advances include speech recognition [Hinton et al., 2012a; Graves et al., 2013], mapping from audio data to written words, and natural language processing [Brown et al., 2020], mapping a text prompt to even more text that appears to be written by a human.

Of course, the list of examples given is by no means complete. For further details and a practical introduction into machine learning, we refer to [Mehta et al., 2019] as well as [Bishop, 2006].[5]

[1]Because we do not need it for the concepts presented in this manuscript, we mostly leave out treating machine learning in the reference frame of probability theory. The reader is advised to consult [Bishop, 2006] for a thorough introduction to machine learning in terms of a Bayesian interpretation and maximum a-posteriori learning.

[2]There are exceptions to every rule, here it is one-shot learning, which is only performed on very few data samples [Fei-Fei et al., 2006], something the brain is arguably rather proficient at [Lee et al., 2015].

[3]MNIST Database, `http://yann.lecun.com/exdb/mnist/` (visited 2021-04-10), [LeCun et al., 1998]

[4]Exemplary images are found in Figure 12.2.

[5]In particular, [Bishop, 2006] offers a concise, clear and well-written style that, unfortunately, is rather rare in scientific literature, sparking the author's initial interest in the field.

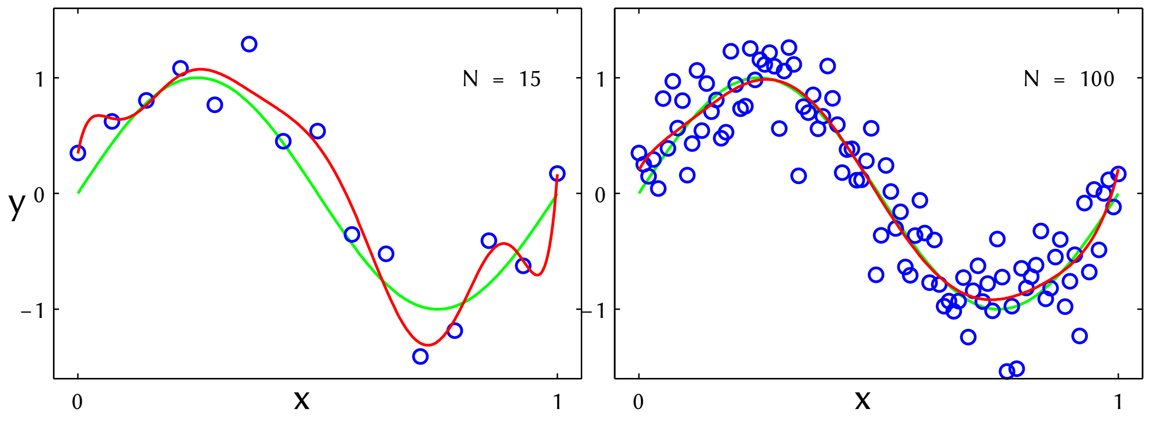


Figure 2.1: Fitting a 9th-order polynomial (red) to data (blue) sampled from a sin-function (green) with noise. **Left:** When the number of model parameters and training samples are roughly equal, the model typically overfits to the data. **Right:** By increasing the amount of training data, our model becomes less susceptible to the errors of each data point. Another possibility is adding regularizing terms to the loss function that punish large polynomial coefficients or reducing the amount of free parameters. Adapted from: [Bishop et al., 2006, Figure 1.6]

2.1 Supervised Learning

For the concepts discussed in Part III, it is sufficient to limit our discussion to supervised learning. However, there are also unsupervised [Bengio et al., 2012] or semi-supervised [Zhu et al., 2009] variants in which all none or only part of the training data is labelled.

We denote input and output data as $(\mathcal{X}, \mathcal{Y})$.

$$\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots\} \qquad \mathcal{Y} = \{\mathbf{y}_1^*, \mathbf{y}_2^*, \dots\} \qquad |\mathcal{X}| = |\mathcal{Y}| =: N \tag{2.1}$$

Here, $\mathbf{v}$ denotes a vector, v_i corresponds to a single vector entry and N is the cardinality of both sets. The star is used to differentiate between actual data and model predictions $\mathbf{y_i}$. Correspondingly, $\mathbf{M}$ denotes a matrix with elements m_{ij} throughout this manuscript. A model is then defined by a function assigning a prediction to every input data point.

$$\mathbf{x} \mapsto \mathbf{y} = \varphi(\mathbf{x}; \boldsymbol{\theta}) \tag{2.2}$$

where $\boldsymbol{\theta}$ denotes the set of parameters of φ. Furthermore, we define a loss function

$$\mathcal{L}[\varphi(\mathcal{X}; \boldsymbol{\theta}), \mathcal{Y}] = \frac{1}{2} \sum_i \|\varphi(\mathbf{x}_i; \boldsymbol{\theta}) - \mathbf{y}_i^*\|^2 \tag{2.3}$$

that measures how well (or not) φ describes the data across the full or a subset of the dataset. Equation (2.3) uses mean squared error, a "typical" choice of loss function using $\|\cdot\|^2$, i.e., Euclidean L^2 norm. This is by far not the only choice.[6] The loss function can be augmented further to include other regularization terms which aim to prevent overfitting

[6]For example, in Chapter 12 we use a soft-max loss function over spike-times.

(cf. Figure 2.1). Choosing which loss function and regularization to apply has great influence on the trained model [Zou, 2006]. The final learning task is to minimize the loss function, finding parameters $\hat{\boldsymbol{\theta}}$

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \mathcal{L}[\varphi(\mathcal{X};\boldsymbol{\theta}),\mathcal{Y}] \tag{2.4}$$

where $\hat{\boldsymbol{\theta}}$ ideally describes a global minimum, however, in most realistic cases it will be a local one. We then train the model by calculating the gradient of the loss function with regard to model parameters and update the model parameters in the opposite direction (since we are minimizing the loss):

$$\begin{aligned} \Delta\boldsymbol{\theta} &= -\eta\nabla_{\boldsymbol{\theta}}\mathcal{L}[\varphi(\mathcal{X};\boldsymbol{\theta}),\mathcal{Y}] \\ \boldsymbol{\theta} &\mapsto \boldsymbol{\theta} + \Delta\boldsymbol{\theta} \end{aligned} \tag{2.5}$$

where η is a learning rate controlling the step size. There are several ways of performing efficient gradient descent [Ruder, 2017]. Often it is wasteful to compute the gradient over the whole dataset. Instead we compute the loss for smaller *mini-batches* and average prior to applying the update. This also helps to prevent overfitting and stabilizes learning.

Adam-Optimizer SGD[7] is known to get stuck in saddle-points, drastically reducing learning progress [Dauphin et al., 2014]. We therefore choose an optimizer with momentum: Adam.[8]

Adam keeps a running average of both first gradient ($\mathbf{m}_t$) and squared gradient ($\mathbf{v}_t$), where the subscript denotes the training epoch.

$$\begin{aligned} \mathbf{m}_t &= \beta_1\mathbf{m}_{t-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}}\mathcal{L} \\ \mathbf{v}_t &= \beta_2\mathbf{v}_{t-1} + (1-\beta_2)\left(\nabla_{\boldsymbol{\theta}}\mathcal{L}\odot\nabla_{\boldsymbol{\theta}}\mathcal{L}\right) \end{aligned} \tag{2.6}$$

where $\odot$ corresponds to element-wise multiplication. These correspond to estimates of the mean (first order) and uncentered variance (second order) of the gradients. The first corresponds to a momentum term helping to avoid saddle-points while the latter ensures an effective scaling of the learning rate based on observed variance of the gradient. Furthermore, these estimates are bias-corrected to offset zero-initialization.

$$\begin{aligned} \hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1-\beta_1} \\ \hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1-\beta_2} \end{aligned} \tag{2.7}$$

So that the actual update rule becomes:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t}+\epsilon}\,\hat{\mathbf{m}}_t \tag{2.8}$$

[7]Stochastic Gradient Descent, [Robbins et al., 1951]
[8]ADAptive Moment estimation, [Kingma et al., 2014]

The square-root in the denominator is meant as an element-wise operation and ϵ ensures numerical stability in case of a vanishing denominator. Each parameter in $\boldsymbol{\theta}$ is treated separately. Throughout this thesis, we use the Adam' proposed default values of

$$\begin{aligned} \beta_1 &= 0.9 \\ \beta_2 &= 0.999 \\ \epsilon &= 10^{-8} \end{aligned} \tag{2.9}$$

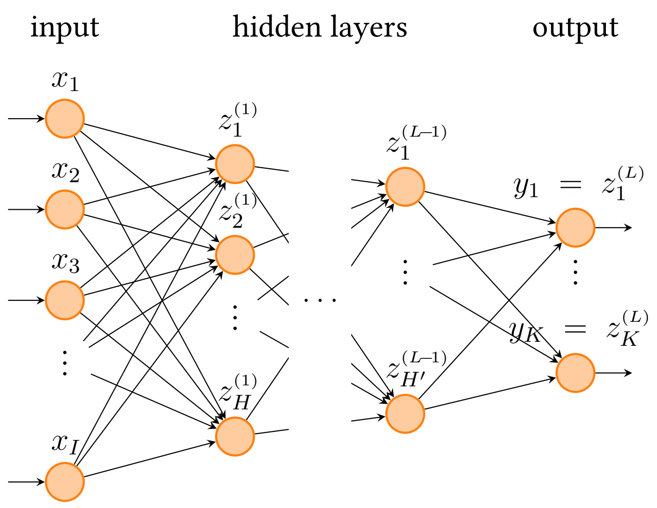


Figure 2.2: Schematic of a densely-connected multi-layer feed-forward artificial neural network with L layers, I inputs and K label units. The neurons of successive layers are connected by the weights $w_{ij}^{(l)}$. Adapted from: [Emmel, 2020, Figure 2.1].

2.2 Deep Learning

Deep learning is one of the main driving forces behind machine learning's rise in popularity in the last two decades [LeCun et al., 2015; Goodfellow et al., 2016]. In discussions of the general public, the term is often used interchangeably with AI,[9] indicating how impressive its recent achievements have been [Brooks et al., 2012; Ng, 2016; Hassabis et al., 2017; Sejnowski, 2018; Richards et al., 2019]. Most of the examples presented above employ deep learning strategies.

Deep learning takes great inspiration from biology, especially the visual pathway [McCulloch et al., 1943; Rosenblatt, 1962]. ANNs[10] can be seen as a rather high-level abstraction

[9] Artificial Intelligence
[10] Artificial Neural Networks

of their biological counterparts. Each "unit" in an ANN performs a simple operation:

$$y = \varphi\left(\underbrace{\sum_j w_j x_j}_{=:u}\right) \tag{2.10}$$

where w_j are simple weights and φ is a non-linear operation. The summed argument to the non-linearity is denoted u due to its resemblance of a neuron's membrane potential. These are then stacked into L layers (denoted by the superscript), each computing a vector of activations from the previous layer in parallel (cf. Figure 2.2)

$$\begin{aligned} \mathbf{u}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} & \mathbf{z}^{(l)} &= \varphi\left(\mathbf{u}^{(l-1)}\right) \text{ for } l \in \{1, \ldots, L-1\} \\ \mathbf{u}^{(l)} &= \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} & \mathbf{y} &= \varphi\left(\mathbf{u}^{(L)}\right) \end{aligned} \tag{2.11}$$

where $\mathbf{W}^{(l)}$ is the weight matrix between $(l-1)$th and lth layer, $\mathbf{x}$ still corresponds to the input data and $\mathbf{z}^{(l)}$ are intermediate activations. The Lth layer is typically called *label layer* which defines the final outputs $\mathbf{y}$ of the model. All layers but input and label layer are called *hidden*.

We highlight the importance of the non-linearity φ, often called *activation-function*: Without it, Equation (2.11) is a linear operation. Applying several linear operation in succession when stacking layers is still a (reversible) linear operation with the same discriminative power: A hyperplane through the dimensionality of the input dataset. Hence, we need non-linearities to perform more powerful transformations. What *exact* kind of non-linear operation φ performs is of lesser concern [Ramachandran et al., 2017], as long as its differentiable (see below). While early models used logistic non-linearities [Hinton et al., 2006],

$$\varphi_{\text{log}}(x) = \frac{1}{1+e^{-x}} \tag{2.12}$$

nowadays ReLUs[11] are preferred [Krizhevsky et al., 2012] due to their simplicity.

$$\varphi_{\text{ReLU}}(x) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases} \tag{2.13}$$

The Backpropagation-Algorithm After defining all nomenclature, the backpropagation algorithm [Linnainmaa, 1970; Ivakhnenko, 1971; Werbos, 1982; Rumelhart et al., 1986] follows from performing Equation (2.4) via Equation (2.5). Here, the parameter vector is a collection of all weights $w_{ij}^{(l)}$. It is important that φ is differentiable for backpropagation to

[11]Rectified Linear Units

apply. By applying the chain rule of differentiation, we find the following update rules:

$$\begin{aligned} \boldsymbol{\delta}^{(L)} &= \varphi'\left(\mathbf{u}^{(L)}\right) \odot \nabla_{\mathbf{y}}\mathcal{L}\left[\mathbf{y}, \mathbf{y}^*\right] \\ \boldsymbol{\delta}^{(l)} &= \varphi'\left(\mathbf{u}^{(l)}\right) \odot \mathbf{W}^{(l+1),T}\boldsymbol{\delta}^{(l+1)} \text{ for } l \in \{1, \ldots, L-1\} \\ -\Delta\mathbf{W}^{(l)} \propto \frac{\partial\mathcal{L}}{\partial\mathbf{W}^{(l)}} &= \boldsymbol{\delta}^{(l)}\, \varphi^T\left(\mathbf{u}^{(l-1)}\right) \text{ for } l \in \{1, \ldots, L-1\} \end{aligned} \tag{2.14}$$

where φ' denotes the derivative of the non-linearity, $\mathbf{y}^*$ are the training examples, T is the matrix transpose and $\boldsymbol{\delta}^{(l)}$ is the error-signal of a given layer. We see that error-signals effectively *propagate back* from the label layer through the whole network. It is interesting to note that despite being discovered in the 1980s, training deep multilayer networks only became feasible once enough computing power was available through GPUs.[12] Since most update steps correspond to MACs,[13] GPUs are particularly suited for the task. Using non-polynomial activation functions, ANNs are able to approximate any function using a sufficient number of hidden neurons [Leshno et al., 1993].

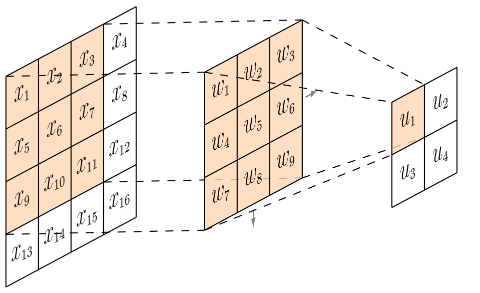


Figure 2.3: Example of a discrete convolution operation. A 3×3 filter kernel w_j is strode in both directions over a 4×4 input x_i to form the output u_k. The active regions to compute the output value are shaded orange. For multi-channeled input a corresponding number number of filters is needed. Each filter can have more than one output dimension. This way the output may also include several channels. Adapted from: [Emmel, 2020, Figure 2.4]

Convolutional Neural Networks Convolutional neural networks are a subclass of ANNs that mimic the visual cortex even stronger. In the years prior, a large part in the field of image recognition was concerned with how to extract the correct features from images. Approaches included SIFT[14]-vectors (based on difference of Gaussian), SURF[15] based on Haar-wavelets or Gabor-filters that were also identified in the visual cortex [Fogel et al., 1989; Olshausen et al., 1996].

The key insight for CNNs[16] is to simply let the network learn applicable feature extractors on its own [Krizhevsky et al., 2012]. To that end, so-called convolutional layers perform multidimensional convolution (cf. Figure 2.3). In other words, whereas fully-connected layers described above correspond to one large matrix operation, here *several* smaller

[12] Graphics Processing Units
[13] Multiply-ACcumulate operations
[14] Scale-Invariant Feature Transform, [Lowe, 1999]
[15] Speeded-Up Robust Features, [Bay et al., 2006]
[16] Convolutional Neural Networks

matrices are convolved along the input data. The reasoning is that if a feature is useful to be extracted at one position of the image, there is a large likelihood it will help discriminate input data when extracted everywhere. During training, convolution matrices are initialized randomly to encourage specialization towards different features. Furthermore, their gradients are averaged across all operations they participate in. Therefore, they can be seen as a form of regularization for fully-connected ANNs that are prone to overfitting due to their sheer number of free parameters.

Furthermore, the models are typically made translation-invariant by applying max-pooling layers. Here, the input over a variably sized window is replaced with the maximum value found, greatly reducing data dimensionality. By virtually increasing the training data due to fuzzing (e.g., extracting random slices from larger images), the model become resilient to particular input locations.

Additionally, drop-out layers can be used to reduce overfitting further. During training they randomly zero their output in order for the model to not rely too much on any particular feature. For inference, they are disabled.

There are several more regularization methods than could be covered in this broad overview. For further information, readers are referred to [LeCun et al., 2015; Goodfellow et al., 2016]. In Chapter 13, we incorporate CNNs into the model.

2.3 Spiking Neural Networks

The average human brain is comprised of about $50 \cdot 10^9$ to $100 \cdot 10^9$ neurons [Bartheld et al., 2016], interconnected by about $15 \cdot 10^{14}$ synapses [Pakkenberg et al., 2003]. Extrapolating from the fact that this text is currently not (yet) being read by a machine, it is safe to assume that they perform some form of information processing. Since the focus of this thesis is machine learning and its applications to neuromorphic hardware, we skip a detailed treatment of their biological aspects.[17]

One of the main differences between ANNs and SNNs[18] is their treatment of time. ANNs model abstract input/output relations in the mathematical sense: For every input applied we generate an output. SNNs, on the other hand, represent a dynamical system that have an explicit concept of time. Their dynamics are described by a set of differential equations evolving over time that elicit binary signals, spikes, at discrete time points [Gerstner, 2001; Izhikevich, 2004].

When compared directly, SNNs still lag behind ANNs in terms of discriminative power as well as scalability [Pfeiffer et al., 2018]. There is no consensus as to why this is the case. Despite many approaches for finding suitable forms of spike-based coding and corresponding computing paradigms [Gerstner, 1998; Maass, 2016; Davies, 2019], we

[17]For detailed introductions to biological aspects of neurons, especially relevant to neuromorphic hardware, readers are referred to [Petrovici, 2016; Dold, 2020; Kungl, 2020; Baumbach, 2021].

[18]Spiking Neural Networks

might not have found the “right one” yet. When trying to adapt from ANNs to SNNs, a common first step is to replace all artificial neurons in the ANN with their spiking counterpart and use rate-coding to approximate their real-valued output signals [Cao et al., 2015; Diehl et al., 2016; Schmitt et al., 2017; Petrovici et al., 2017a]. Afterwards, parameters are tuned in such a way that output spike rates in the SNN correspond to real-valued outputs in the SNN. This is not only inefficient, it also completely ignores information potentially embedded in the precise timing of single spike events.

Time information embedded into spikes does promise to hold important advantages. For example, the Tempotron, a supervised synaptic learning algorithm, employs a coding scheme that utilizes both spatial and temporal dimensions [Gütig et al., 2006].

Another aspect incorporates spike-times in the context of inference [Petrovici et al., 2013; Neftci et al., 2014; Petrovici et al., 2016; Neftci et al., 2016; Leng et al., 2018; Kungl et al., 2019; Dold et al., 2019; Jordan et al., 2019; Korcsak-Gorzo et al., 2021]. In the Neural Sampling paradigm [Buesing et al., 2011], a neuron k with membrane potential u_k is said to encode the state of a binary random variable z_k. Here, $z_k = 1$ whenever the neuron is refractory (i.e., it has just spiked and cannot yet spike again) and 0 otherwise. If the Neural Computability Condition [Buesing et al., 2011] holds,

$$u_k(t) = \ln \frac{p(z_k = 1|\mathbf{z}_{\setminus k}(t))}{p(z_k = 0|\mathbf{z}_{\setminus k}(t))} \tag{2.15}$$

the network samples from an underlying probability distribution p. In other words, if the membrane potential encodes the log-odds of an underlying (conditional) probability distribution,[19] any time point in network dynamics can be seen a sample from this distribution. If this underlying distribution is Boltzmann [Hinton et al., 1984],

$$p(\mathbf{z}) = \frac{1}{Z} \exp\left[\sum_{i,j} \frac{1}{2} z_i\, W_{ij}\, z_j + \sum_k b_k\, z_k\right] \tag{2.16}$$

where $\mathbf{W}$ is a symmetric weight matrix with empty diagonal, $\mathbf{b}$ a bias-vector and Z a normalizing constant, we retrieve “regular” neuron dynamics when inserting Equation (2.16) into Equation (2.15),

$$u_k(\mathbf{z}_{\setminus k}(t)) = \sum_{j \neq k} W_{kj} z_j(t) + b_k \tag{2.17}$$

albeit with rectangular PSPs.[20] These can also be trained, for example using contrastive divergence [Hinton et al., 2006].

We can extend this approach to biologically more realistic neuron models, i.e., non-rectangular PSPs and deterministic dynamics driven by external stochasticity [Petrovici et al., 2016]. In order to facilitate faster simulations for Neural Sampling, `sbs`[21] was devised

[19]It means the probability of a particular neuron being active or not, *conditioned* on the state of all other neurons in the network.

[20]Post-Synaptic-Potentials

[21]Spike-Based Sampling – a library for fast Neural Sampling, [Breitwieser et al., 2020; Breitwieser, 2015]

during [Breitwieser, 2015] and extended over the course of this thesis. Serving as an easy-to-use abstraction for sampling networks, it allows users to separate calibrating activation functions from specifying abstract weight matrices and bias vectors, as well as reconstructing probability distributions from samples. It has been used as the computational basis for several publications (cf. Appendix A.3).

Another interesting aspect is using these sampling network to perform a spike-based form of expectation maximization [Bill et al., 2015; Breitwieser, 2015; Spilger, 2018], one of the core algorithms used in unsupervised machine learning. Here, spikes from a so-called cause-layer form samples from an underlying latent probability distribution corresponding to the expectation step, while the maximization step is achieved via plasticity in the synapses. These networks can be stacked hierarchically, enabling them, for example, to classify MNIST [Guo et al., 2017].

Applying back-propagation was not applicable to spike-times until recently [Bohte et al., 2000; Zenke et al., 2018; Huh et al., 2018; Tavanaei et al., 2019; Neftci et al., 2019; Wunderlich et al., 2020]. The key component here is to find some expression for the activation function to allow application to regular backpropagation. Approaches include surrogate gradients that approximate the true gradient sufficiently to perform SGD and exact methods via back-propagation of adjoint variables that track synaptic currents at the time of spiking. We present TTFS,[22] a novel approach to performing exact inference in a backpropagation setting on spike-times only, in Chapter 12.

Alternatives to backpropagation have also been proposed [Bellec et al., 2019; Bellec et al., 2020]. Here, a recurrent SNN is trained with an online version of BPTT,[23] a common algorithm for training recurrent ANNs that works by treating recurrent connections as unrolled copies of the same network. Biologically plausible eligibility traces are propagated forward and combined with online learning signals to learn several tasks that involve local memory.

Overall, machine learning with SNNs is a promising field, expected to become more competitive with “regular” ANNs in the coming decades.

[22]Time-To-First-Spike

[23]BackPropagation Through Time

Neuromorphic Hardware: The BrainScaleS platform

3

The field of neuromorphic computing is, by all accounts, still in its infancy. Originally envisioned more than 30 years ago and referring to mixed-signal neuron emulations in VLSI[1] [Mead, 1989; Mead, 1990], it has since grown into a plethora of different approaches. The underlying cause for their emergence is the fact that conventional von Neumann-based [Neumann, 1945] computing is approaching the physical limitations of further miniaturization, thereby plateauing out in terms of processing capabilities and speed [Theis et al., 2017]. While clock speeds of modern processors have stayed within 3–4 GHz for the past 20 years, progressively limiting single threaded performance growth,[2] the infamous Moore's law still provided increased numbers of transistors per area. Nowadays, these are predominantly used to facilitate concurrent programming via more and more CPU[3] cores.

This is in line with the rise of machine learning – in particular "deep learning" (cf. Chapter 2) – in recent years [Cireşan et al., 2012; Krizhevsky et al., 2012], marking a paradigm shift away from SISD[4] to large-scale SIMD[5]: Typical "real world" classification tasks such as vision require processing of large quantities of parallel data streams, best performed by specialized hardware such as GPGPUs[6] [Owens et al., 2007; Navarro et al., 2014] that emerged from GPUs. Executing a single instruction stream on a universal but speed-limited compute engine is simply not performant enough.

Another important aspect is energy efficiency. The nowadays ubiquitous smart phone and other mobile embedded devices are rated not by their processing speed but by the total amount of energy they require to perform a certain task. Here, miniaturization has been shown to break down Dennard scaling, impeding further energy-efficiency of conventional computing [Esmaeilzadeh et al., 2011]. New technologies could explore other avenues of energy-efficient computation off the beaten path. This is also important for *edge computing*, a recent trend where distributed cheap and low-power but highly specialized compute units perform preliminary data processing "on the *edge*" of data storage [Davis et al., 2004], typically involving inference on pre-trained machine learning models.

One branch of new approaches to computation takes inspiration from our current understanding of how the brain processes information: Hence, they are labelled *neuro-*

[1]Very Large Scale Integration
[2]Single-threaded performance is still increasing via other optimization techniques such as speculative branch-execution or pipelining.
[3]Central Processing Unit
[4]Single Instruction stream – Single Data stream
[5]Single Instruction stream – Multiple Data streams
[6]General Purpose Graphical Processing Units

morphic [Jaeger, 2021]. However, this is their only unifying property. They range from immediate applicable to exploratory research, from repurposing existing technologies[7] to designing new chips,[8] from energy-efficient but conventional digital computing[9] to analog approaches,[10] oftentimes trying to to avoid the so-called von Neumann-bottleneck[11] by experimenting with new memory layouts,[12] sparse coding[13] or new memristive[14] components altogether.

Neuromorphic approaches include, but are not limited to:

FPGA[15]-based emulators such as DeepSouth cortex [Wang et al., 2018] are the most straightforward neuromorphic computing architecture, because they can use complete toolchains already existing for FPGAs. Due to the fact that they are digitally programmable, they can be adapted on-the-fly to specific tasks. Since they can implement essentially arbitrary digital circuitry, their data paths can be scaled to saturate the von Neumann-bottleneck for a specific task, thereby allowing for more efficient computation. Furthermore, since their energy costs scale with the implementation size, they can be more energy efficient than conventional computing, but not as efficient as ASICs,[16] which fit the same logic in a smaller area. They are also adaptable to prototype learning algorithms [Mostafa et al., 2017].

Fully digital custom ASICs aim to model other modes of computation, typically spike-based. Here, neuromorphic cores simulate various forms of neuron models that exchange spike-like messages between each others. Examples include SpiNNaker [Furber et al., 2014; Mayr et al., 2019], a collection of ARM-based processing cores with a custom interconnected network to perform neural simulations in realtime, TrueNorth [Akopyan et al., 2015], executing simple time-multiplexed neuron models interconnected via binary synapses in an energy-efficient manner, Darwin [Shen et al., 2016], implementing SNNs with configurable synaptic delays, Loihi [Davies et al., 2018; Davies et al., 2021], focussing on fast on-chip learning by supporting highly configurable update rules with access to many observable, and Tianjic [Pei et al., 2019], offering a hybrid approach by supporting both SNN as well as ANN execution.

Mixed-Signal approaches use the analog properties of conventional CMOS[17] transistors to *emulate* neuronal dynamics directly. I.e. there is no logical abstraction layer simulating

[7] [Furber et al., 2014; Mayr et al., 2019]

[8] [Akopyan et al., 2015; Shen et al., 2016; Davies et al., 2018; Davies et al., 2021; Moradi et al., 2018; Pei et al., 2019]

[9] [Furber et al., 2014; Mayr et al., 2019; Akopyan et al., 2015; Davies et al., 2018; Davies et al., 2021]

[10] [Schemmel et al., 2010; Pfeil et al., 2013; Benjamin et al., 2014; Qiao et al., 2015; Schemmel et al., 2017; Neckar et al., 2018; Schemmel et al., 2020]

[11] The von Neumann bottleneck describes a decrease in efficiency due to limited bandwidth between CPU and attached memory.

[12] [Moradi et al., 2014]

[13] [Mostafa et al., 2017]

[14] [Jo et al., 2010; Bill et al., 2014; Li et al., 2018]

[15] Field-Programmable Gate Array

[16] Application-Specific Integrated Circuits

[17] Complementary Metal-Oxide-Semiconductor

neuron models: One physical system emulates another. These approaches can be further divided into sub-threshold and supra-threshold designs, referring the operating point of the involved transistors. While the former aims ultra-low power consumption while emulating with time constants close to realtime, the latter offers accelerated neuron dynamics with greater precision at a speed-up factor of $10^3 - 10^5$ at the cost of higher power consumption. Examples for sub-threshold designs include the Neurogrid project [Benjamin et al., 2014] as well as the ROLLS [Qiao et al., 2015], BrainDrop [Neckar et al., 2018] and DYNAPs chips [Moradi et al., 2018], whereas the BrainScaleS[18] platform discussed below executes in the accelerated supra-threshold regime. Because of their intricacies they often require models to be adapted when implemented [Petrovici et al., 2014; Petrovici, 2016].

New Materials are new electronics components that can perform certain aspects of neuromorphic computing in and of themselves [Lee et al., 2019]. The most prominent are *memristors*, that first have been proposed as the missing component of a theoretical quartet of fundamental two-terminal components along resistor, capacitor and inductor [Chua et al., 1976]. Their fundamental property is that their conductivity (or resistance) is variable and depends on the precise history of how charge was transferred, similar to a synapse in biology [Jo et al., 2010; Li et al., 2018]. While offering low power consumption and fast read-write speeds, they still suffer from large fixed-pattern noise and trial-to-trial variability. However, recent chips already feature up to 10^3—10^4 memristive elements [Cai et al., 2019].

Please note that GPGPUs (such as Nvidia's TPU[19] or Intel's Habana Gaudi [Medina et al., 2020]), while used to accelerate machine learning models, are generally not considered to be neuromorphic devices because they "merely" extend conventional computation towards maximum MAC-bandwidth.

Recent reviews of different aspects to neuromorphic engineering include [Indiveri et al., 2011b; Indiveri et al., 2011a; Vanarse et al., 2016; Furber, 2016; Nawrocki et al., 2016; Schuman et al., 2017; Thakur et al., 2018; Li et al., 2018; Pfeiffer et al., 2018; Lee et al., 2019; Roy et al., 2019; Rajendran et al., 2019; Zhu et al., 2020]. For a more extensive summary of these reviews, we refer to [Kungl, 2020, Chapter 2.3].

We now focus on the BrainScaleS platform, a mixed-signal neuromorphic compute platform and its various generations, mainly developed at Electronic Vision(s).[20]

3.1 | BrainScaleS-1

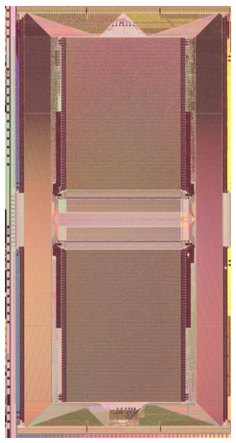
BrainScaleS-1[21] marks the first generation of large-scale mixed-signal neuromorphic accelerator platforms. Realized in 180 nm CMOS technology, it is the successor to Spikey,

[18]BrainScaleS Mixed-Signal Accelerated Neuromorphic Systems, [Schemmel et al., 2008; Schemmel et al., 2010; Schemmel et al., 2017; Schemmel et al., 2020]

[19]Tensor Processing Unit, [Jouppi et al., 2017; Coral, 2020]

[20]Electronic Vision(s) Group at the Kirchhoff-Institute for Physics in Heidelberg

[21]BrainScaleS-1 Wafer-Scale Mixed-Signal Accelerated Neuromorphic System, [Schemmel et al., 2008; Schemmel et al., 2010]


(a) Detailed view of a single HICANN chip. The two large synaptic arrays at the top and bottom are most prominent, the neuron circuits are located in between.
Photo taken from [Klähn, 2017].

(b) A fully assembled wafer module. Photo taken from [Schmitt et al., 2017, Figure 2b].

Figure 3.1: Photographs of BrainScaleS-1.

a standalone smaller scale chip [Pfeil et al., 2013]. While initially developed during the name-giving BrainScaleS project [BrainScaleS, 2011], its development continued in the HBP[22] [Markram, 2012]. It provides means to perform accelerated large-scale emulations of SNNs. The system operates at an acceleration factor of 10^3–10^5 compared to realtime. By default the speed-up factor is set to 10^4, meaning that 1 ms in biological time corresponds to 0.1 µs realtime. Analog neuron circuits emulate dynamics while communicating via digital spike signals, hence the term *mixed signal*.

BrainScaleS-1 is comprised of several wafer-modules (cf. Figure 3.1b). Presently, there are five racks deployed with four wafers each. Each wafer is made up of 48 reticles, each of which consists of 8 HICANN[23] chips – the smallest conceptual building block of the system, depicted in Figure 3.1a. In total, each wafer contains 384 HICANNs can emulate up to $\sim$180 000 neurons and $\sim$40 000 000 synapses, depending on the implemented network structure.

HICANN Building Block The smallest conceptual building block of the BrainScaleS-1 is the HICANN. It was developed within BrainScaleS and has a symmetric structure: Top and bottom half are mirrored versions of each other (see Figure 3.1a). Analog circuitry is contained within the ANC.[24] It consists of two parts: The 2×256 neuron circuits, DenMems,[25] and the much bigger synapse arrays. The DenMems are subdivided further into 8 blocks of 64 circuits each. In each block, membrane potentials of several DenMems can be short-circuited to create larger neurons or simply combined to form

[22]Human Brain Project
[23]High Input Count Analog Neural Network
[24]Analog Network Core
[25]Dendritic Membranes

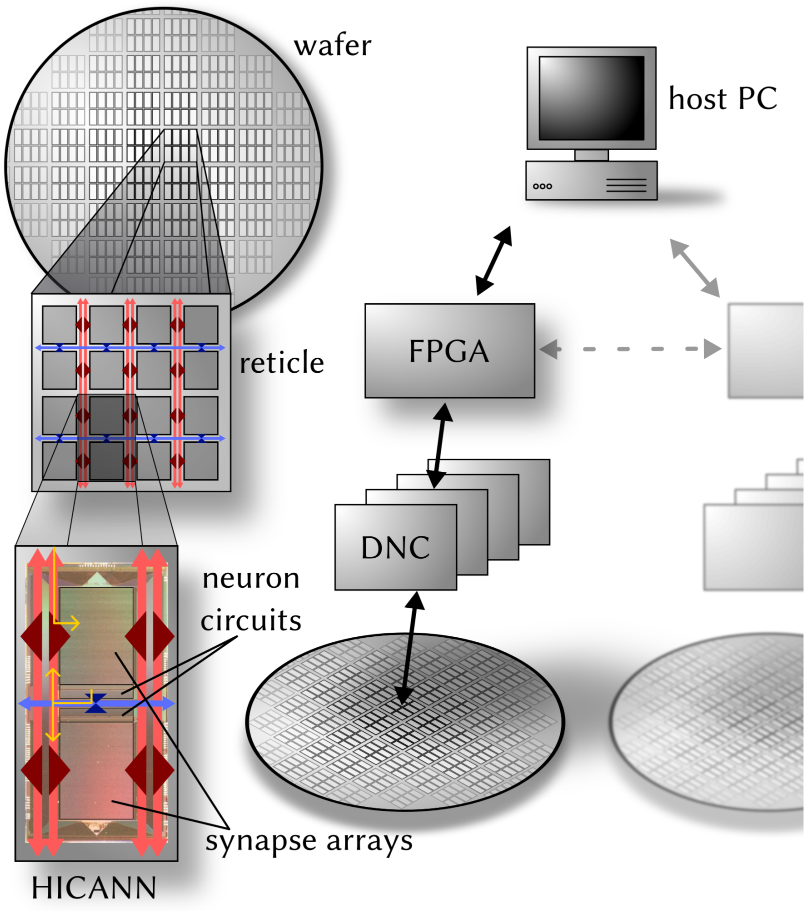


Figure 3.2: **Left:** The HICANN building block has two symmetric halves with synapse arrays and neuron circuits. Red and blue lines indicate L1 communication of synaptic activity. Exemplary routes a spike could travel on the chip are indicated in yellow: An external spike arrives off-HICANN and is routed to the synapse driver. This causes a neuron to spike so that its spike packet is emitted back into the routing network. **Right:** Off-wafer communication is achieved by a packet-based hierarchical L2 network via DNCs and FPGAs, interfacing the on-wafer routing buses of the HICANNs. Adapted from: [Petrovici et al., 2014, Figure 2].

multi-compartment models [Millner, 2012].

Each DenMem implements the AdEx[26] neuron model [Brette et al., 2005] with CoBa[27] synapses (Section 12.2.5) with two synaptic input channels and the possibility for constant external current injection. The AdEx model is a two dimensional extension to the simpler LIF[28] neuron model applied in Chapter 12, extending it with additional adaption and exponential terms. Its implementation on BrainScaleS-1 is able to reproduce more firing patterns observed in nature, such as tonic spiking, spike frequency adaption and chaotic spiking [Tran, 2013]. Due to its modular design, all extension terms can be deactivated via configuration parameters, decaying the model back to LIF (used in Section 12.3.3). Analog

[26]Adaptive Exponential
[27]Conductance-Based
[28]Leaky-Integrate-and-Fire

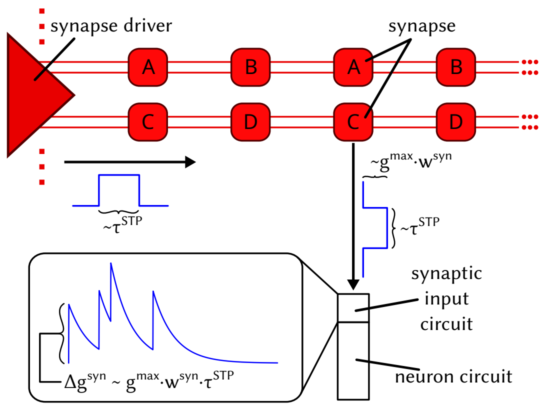


Figure 3.3: Conceptual overview of the synapse driver in BrainScaleS-1. Each synapse row driver listens for incoming spike pulses and routes them onto one of four strobe lines based on top two bits of the afferent neuron's address and emits a pulse packet with length τ^{STP} (the length is determined by a possibly active TM mechanism). Each synapse is connected to one of the four strobe lines (indicated by A-D). If the lower four bits match, the synapses then reroute the pulse into the corresponding column. The strength of the synaptic conductance is modulated by the window length τ^{STP}, the maximum conductance g^{max} (set for the whole row) and the actual 4-bit synaptic weight w^{syn} so that the total charge applied corresponds to the TM-modulated synaptic weight. Adapted from: [Breitwieser, 2015, Figure 3.2].

neuron parameters are configured via 10 bit FGs[29] [Srowig et al., 2007; Schemmel et al., 2010; Koke, 2017]. They work by using the tunnel effect at high voltages to store charge in an isolated (i.e., *floating*) transistor gate which afterwards induces a voltage serving as parameter reference for analog circuitry. 4 bit synaptic weights are stored in SRAM.[30]

Spike Routing On the wafer itself, spikes are routed via a network of asynchronous buses: the L1[31] communication network (cf. Figure 3.2 bottom left). The system was designed with scalability in mind, allowing for the inter-connection of several wafers via a synchronous packet-based communication network called L2.[32] In each HICANN, synaptic connections are realized via 224×256 synapse array. It is fed by 2×56 synapse drivers, each one supplying two synapse rows of the array. Each DenMem receives input from one synaptic column, leading to 224 possible inputs per DenMem and up to $224 \times 64 = 14336$ possible inputs per neuron (assuming a completely connected DenMem block).

Spike Processing Once a neuron spikes it generates a digital spike pulse packet, consisting only of its 6-bit source address. In the merger tree, this packet is then either time-stamped and sent off-wafer via the L2 network or injected onto one of the 64 horizontal buses. The buses route it throughout the wafer via sparse crossbar switches at the intersection of vertical and horizontal buses until it reaches the target synapse driver. Based on the top two weights of the afferent neuron's address, the signal is routed onto one of four *strobe lines*. Both lower 4 address bits and a reference signal of length τ^{STP}

[29]Floating Gatess
[30]Static Random-Access Memory
[31]Layer-1
[32]Layer-2

(modelling STP[33] via the TM[34] model, either depressing or facilitating) are transmitted into the synapse array. Each synapse is statically connected to one of the four strobe lines. If the four bits match the address stored in a synapse, a 4-bit DAC[35] generates the final current signal representing the conductance with height proportional to the maximum conductance g^{max} multiplied by the 4-bit weight w^{syn} for duration τ^{STP} so that the total charge transmitted corresponds to the STP-modulated synaptic weight of the connection. See Figure 3.3 for an illustration. The g^{max} of two adjacent synapse drivers can be configured to be a fixed multiple of each other, thereby increasing the weight resolution to 8 bit while halving the number of synapses for these synapse drivers.

Furthermore, besides the neuron circuits, there are 8 LFSRs[36] present on each HICANN. They can serve as source of pseudo-randomness by injecting spikes onto the L1 buses.

Dealing with Fixed-Pattern Noise Every physical system experiences fixed-pattern noise to varying degree. Current transistor-based manufacturing technologies have reached a level where small deviations between single transistors are inevitable due to the miniaturized scale. The task is then to tweak the signal-to-noise ratio such that the system is usable. In digital systems this is achieved by translating analog signals to mere binary values while adjust supply voltages and clock-frequencies to ensure the translation holds at all times. For analog systems, such as BrainScaleS-1, this is not as straightforward. Via calibration, we can greatly reduce fixed-pattern noise albeit not quench it completely [Schwartz, 2013; Koke, 2017; Kleider, 2017].

The greatest source of noise is the trial-to-trial variation of FGs [Kononov, 2011; Kungl, 2016]. Despite writing the same 10 bit value, the resulting neuron parameters still exhibit large variation. The trial-to-trial variability is so large, unfortunately, that the established best-practice is setting analog parameters only once per experiment and then perform learning only on the digital parameters, i.e., synaptic weights [Schmitt et al., 2017].

Software Stack Operating neuromorphic computing platforms holds many challenges in terms of precise system control, data pre-/post-processing as well as data exchange. Similar to other digital hardware platforms, software is *the* key component to make complex hardware systems accessible to users in structured and reliable way [Kacher et al., 2020; Rueckauer et al., 2021]. While in this thesis we focus on the software stack for the successor generation to BrainScaleS-1, described in Chapter 6, its core fundamentals have been established during BrainScaleS-1's development. Figure 3.4 gives an overview over the software workflow. Experiments are specified in a high-level DSL[37] (often implemented as API[38] or software library, in this case PyNN[39]). This description is then converted to a valid configuration of neuromorphic hardware, taking into account various

[33]Short Term Plasticity

[34]Tsodyks-Markram, [Tsodyks et al., 1997; Markram et al., 1998; Tsodyks et al., 1998]

[35]Digital-to-Analog Converter

[36]Linear-Feedback Shift Registers

[37]Domain Specific Language

[38]Application Programming Interface

[39]A Python package for simulator-independent specification of Neuronal Network models, [Davison et al., 2009]

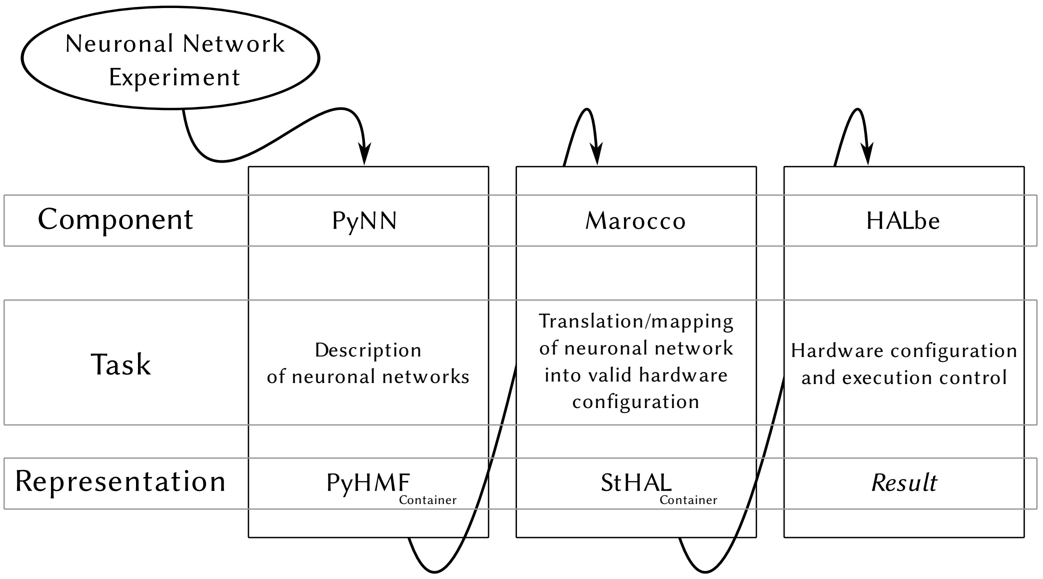

Neuronal Network Experiment
Component
PyNN
Marocco
HALbe
Task
Description of neuronal networks
Translation/mapping of neuronal network into valid hardware configuration
Hardware configuration and execution control
Representation
PyHMF Container
StHAL Container
Result

Figure 3.4: Overview of the BrainScaleS-1 software stack. A given experiment is specified in a high level abstract DSL such as PyNN. Lower level software layers then take care of translating biological parameters to their corresponding hardware counterparts. Furthermore, the network is mapped onto the hardware, thereby automatically configuring buses, crossbar switches, etc. (cf. Figure 3.2). Details of the BrainScaleS-1 software stack are found in [Jeltsch, 2014; Müller, 2014; Müller et al., 2020b]. A similar approach was taken for the software stack in BrainScaleS-2, detailed in Chapter 6. Adapted from: [Müller, 2014, Figure 2.23].

other information such as calibration data. For example, this allows for the selection of neurons by their calibrated parameter ranges.

BrainScaleS-1's wafer-scale connectivity, while flexible to allow for many configurations, does impose constraints on connectivity, for example in terms of which neurons can emit spikes to certain buses and how these connecting buses are mapped across the wafer. It can be shown to be an NP-complete problem to perfectly map any user-defined structure onto BrainScaleS-1 given its connectivity constraints [Cook, 1971; Jeltsch, 2014]. Hence, the mapping layer *marocco* employs a set of heuristics to achieve good mapping results [Jeltsch, 2014; Passenberg, 2019; Kaiser, 2020]. It is able to correctly place networks consisting of neurons on the order of hundreds without synapse-loss. Finally, neurons can be hand-placed, but this is a rather time-consuming endeavor.

It should be noted that this mapping problem afflicts all systems performing physical emulation. Because emulation actually takes place in real-time and is not performed virtually by solving differential equations in an abstract computing paradigm, communication needs to be accomplished in real-time and at sufficient bandwidth as well. Here, digital systems and simulations have an advantage in that they can sacrifice simulation speed in order to process more inputs.

For details of the BrainScaleS-1 software stack please refer to [Jeltsch, 2014; Müller, 2014; Müller et al., 2020b].

3.2 BrainScaleS-2

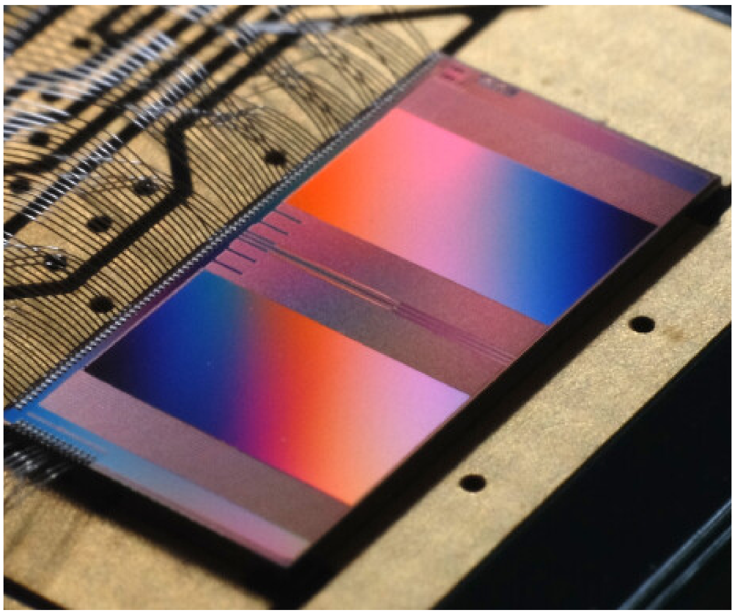
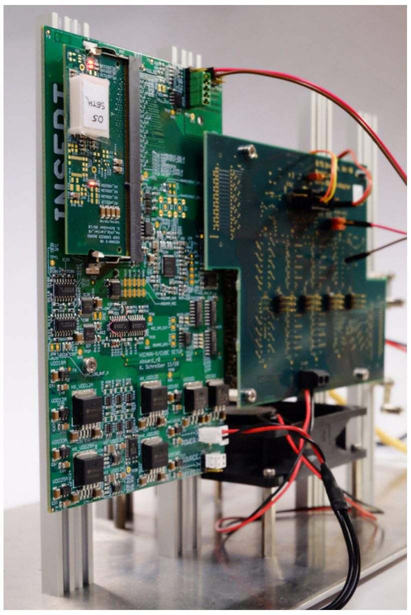



Figure 3.5: BrainScaleS-2 (HICANN-X) single-chip “cube” setup. **Left:** Photograph of the bonded HICANN-X chip. **Right:** BrainScaleS-2 single-chip “cube” setup The white cap (top left) covers one HICANN-X chip which is bonded onto the underlying chip-carrier PCB; other PCBs connect each chip to one FPGA (invisible on the back). The host computer and FPGAs are linked via 1 Gbit Ethernet. Each HICANN-X chip is comprised of 512 AdEx neurons and $512 \times 256 = 131072$ CuBa synapses. Photos taken from: `https://www.kip.uni-heidelberg.de/vision/outreach/chip-gallery/` (visited on 2021-05-03) and [Müller et al., 2020a].

BrainScaleS-2[40] is the second generation of BrainScaleS hardware developed. Its development took place during the HBP [Markram, 2012]. It is manufactured in a smaller form factor, 65 nm CMOS, which allows for more complex structures to occupy the same chip area. The most significant features include a different, more reliable neuron parameter storage implementation [Hock et al., 2013] as well as an embedded SIMD microprocessor for on-chip plasticity [Friedmann et al., 2013].

The software stack for BrainScaleS-2 is detailed in Chapter 6 and [Müller et al., 2020a].

3.2.1 Prototype Generations

There have been several prototype chips so far, namely the HICANN-DLS[41] and the HICANN-DLS-SR-HX,[42] often shortened to HICANN-X, each of which went through

[40]BrainScaleS-2 Analog Neuromorphic Hardware System, [Schemmel et al., 2017; Schemmel et al., 2020]
[41]HICANN Dreieck Ludwighafen Süd: successor to HICANN chip and based on the technology test chip route65 which inspired the reference to BAB65, [Aamir et al., 2018; Friedmann et al., 2017]
[42]HICANN Dreieck LudwighafenSüd: Spikey Replacement with HAGEN eXtensions, [Schemmel et al., 2020]

several iterations.[43] So far, each prototype has been deployed in a standalone fashion with a controlling FPGA. Additionally, a fully standalone setup with a SoC[44]-based host is also available in the form of the BrainScaleS-2 Mobile[45] [Stradmann et al., 2021].

Plasticity Processing Unit The first key difference to BrainScaleS-1 is the inclusion of the so-called PPU,[46] an embedded SIMD microprocessor [Friedmann et al., 2013; Friedmann, 2013]. It implements a subset of the PowerISA 2.06 specification for 32-bit architectures [PowerISA, 2010] with 16 kB SRAM and can be programmed using standard tools, namely a C[47] or C++[48]-based runtime. Additionally, it is equipped with a custom vector unit extension developed during [Friedmann, 2013], providing digital integer and fixed-point arithmetics which hold up with the parallelism in the analog core. Especially, the synapse array can be accessed and manipulated in a parallel fashion, operating on either 128 1-byte entries or 64 2-byte entries at once. The PPU is programmed by loading a given program into SRAM. Execution is gated by a reset pin. For the BrainScaleS-2 Mobile system (described below) it also supports accessing off-chip memory regions, e.g., the FPGA's DRAM.[49]

The PPU has access to several observables, including spike counts and pair-based correlation measurements in each synapse. The latter is implemented via two correlation capacitors that are charged at every pre- or every post-synaptic spike, respectively, and leak current with an adjustable time constant. Upon observing the complimentary spike ("post after pre" or "pre after post", respectively), the current charge of the corresponding correlation capacitor is applied to a larger accumulating capacitance which can then be read out via PPU. The accumulated charges serve as eligibility traces needed for STDP.[50]

Capacitive Memory Parameter Storage The second major improvement is the implementation of analog parameters using CapMem[51] cells [Hock et al., 2013; Hock, 2014]. They provide the option to set analog parameters with 10 bit accuracy. The digital parameters are stored locally in SRAM and are then converted to either 0.2–2 V or 0–2 µA depending on the storage cell type. Compared to FGs, these storage cells have limited storage time and have to be refreshed even during experiments. For this, a novel concept for the refresh process was developed. A reference generator applies a slow linearly increasing voltage to all cells simultaneously. Alongside, a running counter counts up since last ramp reset. Each storage cell then effectively applies the reference voltage when its internal and the global running counter coincide, setting the analog parameter. The ramp generator can be configured in its update frequency that is on the order of $\sim 1\,\mathrm{ms}$. It

[43]In this summary we only cover the major prototype chip types and leave out differences between iterations unless relevant to make a point.
[44]System on a Chip
[45]BrainScaleS-2 Mobile Analog Neuromorphic Hardware System, [Stradmann et al., 2021]
[46]Plasticity Processing Unit
[47]C Programming Language, [ISO, 2018]
[48]C++ Programming Language, [ISO, 2017]
[49]Dynamic Random-Access Memory
[50]Spike-Timing Dependent Plasticity
[51]Capacitive Memory

is part of calibration to find parameter settings balance between potential cross-talk effect during programing, delay after updating parameters and long-term stability. Calibration is also necessary to translate digital counter values, stored in SRAM and typically given in LSB, to their biological counterparts influencing neuron dynamics [Weis et al., 2020; Weis, 2020].

On-Chip Voltage Recording BrainScaleS-2 features on-chip voltage recording capabilities via ADCs.[52] There are two types: CADC[53] and MADC[54] [Schreiber, 2021]. The CADC is primarily used to read out correlation measurements from synapses. It is able to read out one full row in the synapses (256 neurons with (a-)causal accumulation =512 voltages) in parallel. Using a design that is effectively a reverse CapMem implementation, it drives the same kind of linear voltage ramp but records the points in time when measured value and ramp coincide. This is massively parallel, but relatively slow. For faster sampling – e.g., of the evolving membrane potential – we can use the MADC that samples at roughly 30 MHz with 10 bit accuracy. Besides the eponymous membrane, the MADC can be used to record most analog observables in hardware – a tremendously helpful tool, not only for debugging. The downside, however, is that it can only be connected to a single observable at a time.

HICANN-DLS The HICANN-DLS was the first prototype chip to be designed [Schemmel et al., 2017]. It features 32 neurons with the simpler LIF model with CuBa[55] synapses [Aamir et al., 2016; Aamir et al., 2017; Aamir et al., 2018]. Conceptually, synaptic input is implemented in a similar manner as BrainScaleS-1. Each neuron is associated with a column of 32 synapse circuits that receive input from the chip's digital backend. Per synapse, both 6 bit label and 6 bit weight are stored in local SRAM. If an input event matches the stored weight label, the event is propagated to the associated neuron with a current pulse proportional to the synaptic weight. Spike routing is implemented in the controlling FPGA. Despite its small size, HICANN-DLS's capabilities have already been demonstrated in several studies, including: [Friedmann et al., 2017; Billaudelle et al., 2019; Wunderlich et al., 2019; Billaudelle et al., 2020; Schreiber et al., 2020; Schreiber, 2021].

HICANN-X The HICANN-DLS-SR-HX, often shortened to HICANN-X, is the first full-size prototype of BrainScaleS-2 [Schemmel et al., 2020]. A photograph can be seen in Figure 3.5. It features a total of 512 AdEx neurons with CuBa synapses. Each neuron is connected to a column of 256 synapses, allowing for up to $512 \times 256 = 131072$ synapses. Each quadrant of the synaptic array is driven by 128 synapse drivers. The chip features two independent embedded PPUs, each able to execute a separate program. Spike routing is accomplished via an event router embedded in the digital core. HICANN-X has been successfully used in several studies (some of which are presented in Chapters 12 and 13), including [Czischek et al., 2020; Baumbach, 2021; Göltz et al., 2021; Stradmann et al., 2021].

[52] Analog-to-Digital Converters
[53] Correlation ADC
[54] Membrane ADC
[55] Current-Based

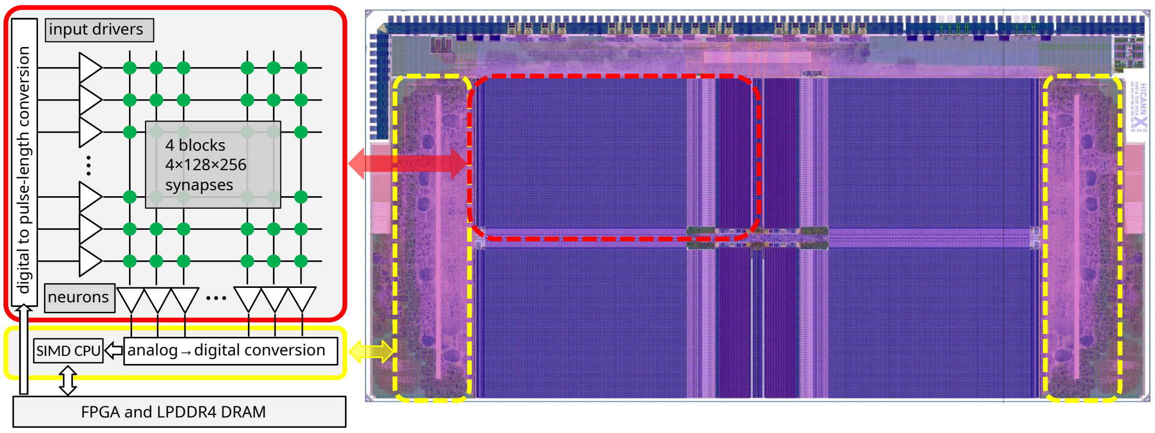


Figure 3.6: **Left:** Internal structure of the HICANN-X ASIC. The analog network core consists of four quadrants, each containing 128 neurons and 128×256 synapses (red). A total of 512 parallel ADC channels allow for readout of various analog parameters by two embedded SIMD processors (yellow). **Right:** position of the described functional units on a layout drawing of the BrainScaleS-2 ASIC. Taken from: [Stradmann et al., 2021, Figure 4].

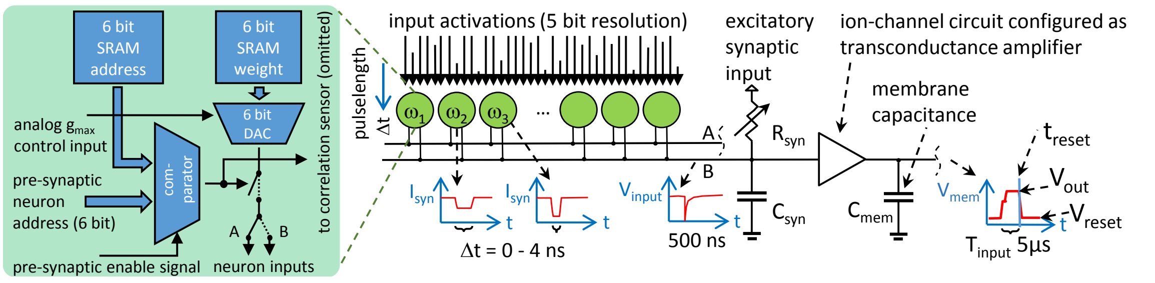

6 bit SRAM address
6 bit SRAM weight
6 bit DAC
analog g_max control input
pre-synaptic neuron address (6 bit)
com-parator
A
B
pre-synaptic enable signal
neuron inputs
to correlation sensor (omitted)
pulselength Δt
input activations (5 bit resolution)
ω1
ω2
ω3
excitatory synaptic input
ion-channel circuit configured as transconductance amplifier
membrane capacitance
R_syn
C_syn
C_mem
I_syn
V_input
V_mem
t
Δt = 0 - 4 ns
500 ns
t_reset
V_out
V_reset
T_input
5µs

Figure 3.7: Operation principle of analog computation in BrainScaleS-2. **Left:** Synaptic operations similar to BrainScaleS-1, but with increased synaptic resolution ($4 \rightarrow 6$ bit). In spiking-mode, each synapse compares its stored weight label with the arriving pre-synaptic event label. In case they match, a current pulse is forwarded to one of two neuron inputs (typically excitatory and inhibitory). Pulse height corresponds to the weight, pulse length is determined by potential STP effects. In non-spiking HAGEN mode, the STP circuitry modifies the pulse length in accordance to the lower 5 bit of the weight label, thereby implementing vector entries that then get multiplied with the stored weight values. The overall transmitted charge (pulse length $\Delta t \times I_{\text{syn}}$ pulse height) then corresponds to the multiplication result. **Right:** Circuitry implementing a LIF neuron. In non-spiking HAGEN mode, the leakage-term is deactivated. During the calculation period T_{input} the membrane simply accumulates all charge arriving from synapses. The final voltage V_{out} of a single neuron represents the result of the analog MAC. Adapted from: [Stradmann et al., 2021, Figure 5].

3.2.2 HAGEN-Mode: Accelerated Multiply-Accumulate

HICANN-X introduces the eponymous HAGEN[56] eXtensions, named after a rather early chip generation that implemented perceptrons [Schemmel et al., 2004]. In this non-spiking mode, the chip is able to perform fast analog MACs with 5 bit input vector resolution and 6 bit weight matrix resolution. The result can be computed with up to 8 bit resolution. Same as the Tianjic architecture [Pei et al., 2019], HICANN-X supports hybrid execution,

[56]Heidelberg AnaloG Evolvable neural Network

where part of the chip is executing in spiking mode and another part in non-spiking MAC mode.

The HAGEN-mode is effectively an extension to the STP-circuitry already present in BrainScaleS-1 (cf. Figure 3.3). A conceptual overview is given in Figure 3.7. Core components of a MAC are the input vector $\mathbf{x}$, the matrix $\mathbf{W}$ it is multiplied with, and a way to accumulate intermediate results in each row. The output vector $\mathbf{y}$ is then given by

$$\underbrace{y_i}_{\substack{\text{CADC}\\\text{readout}}} = \underbrace{\sum_j}_{\substack{\text{charge}\\\text{accumulation}\\\text{on membrane}}} \underbrace{W_{ij}}_{\substack{\text{synaptic}\\\text{weight}}} \underbrace{x_i}_{\substack{\text{input}\\\text{event}}} \tag{3.1}$$

where we have identified the hardware equivalent of every component. Each entry of the input vector is sent to one synapse driver, its value encoded in the weight label. Here, the STP circuitry modifies the pulse length between 0–4 ns realtime depending on the lower 5 bit of the weight label. The 6 bit weight stored in the synapse's SRAM then determines the height of the pulse (same as in spiking mode). Hence, the overall transmitted charge corresponds to the multiplication of both values. A transconductance amplifier in each neuron then translates synaptic charge to a proportional current, charging the membrane. Neurons have their leakage term deactivated in HAGEN mode so that all synaptic events effectively get accumulated on the neuron membrane. After the accumulation phase of roughly 5 µs the CADC is used to read out the membrane potential with 8 bit accuracy via a readout amplifier from each neuron to the correlation readout lines. In order to boost the signal, one vector can be send in several times during one accumulation phase to increase accumulated charge on the membrane, thereby boosting the signal-to-noise ratio.

Attentive readers will have noticed that only 5 bit out of the 6 weight label bits are used to determine the vector entries. The final bit is called the label bit. It can be used to differentiate between vector entry types: Each event sent into the synapse array has the label bit either set or unset. Active synapses then listen to only one of those two event types. This is useful when unrolling matrices, as seen in Section 13.2.2.

Each synaptic row can be set to be either excitatory or inhibitory. In order to support arbitrary MACs, half of all synapses needs to be excitatory and the other half inhibitory. Therefore, the maximum parallel input vector length is 128. For smaller matrices it can be doubled virtually by duplicating the weight matrix such that it interacts with the second vector half that is sent in with opposite label bit.

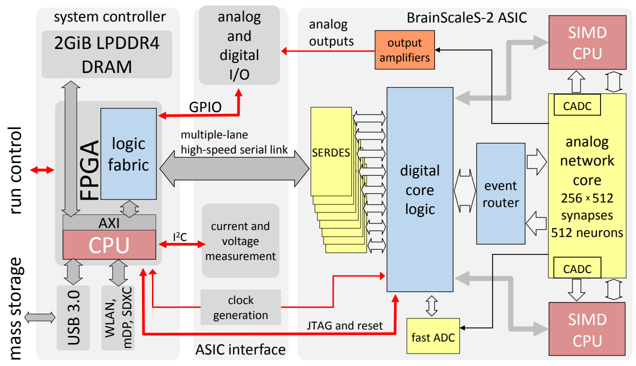
While the synaptic circuits can process back-to-back events within 8 ns, corresponding to an event rate of 125 MHz, or

$$125\,\mathrm{MHz} \cdot 256 \cdot 512 \cdot 2\,\mathrm{Op} = 32.8\,{}^{\mathrm{TOp}}\!/\mathrm{s} \tag{3.2}$$

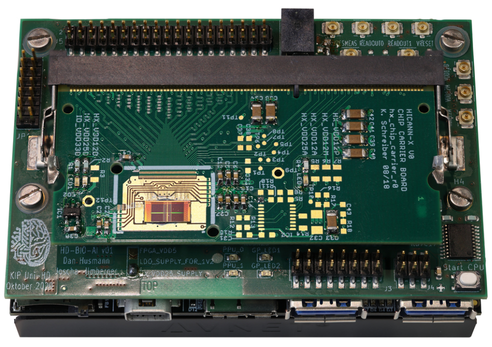
as maximum achievable operation rate, when we count addition and multiplication as individual operations. The aforementioned integration time of 5 µs reduces this MAC frequency to

$$200\,\mathrm{kHz} \cdot 256 \cdot 512 \cdot 2\,\mathrm{Op} \approx 52\,{}^{\mathrm{GOp}}\!/\mathrm{s} \tag{3.3}$$

Nevertheless, the system is not yet fully optimized for non-spiking MAC acceleration. Rather, HICANN-X prototypes serve as an avenue to explore this mode of operation that was integrated without disturbing existing spiking mode functionality. Since it has now been shown to be feasible and – most important – functional (see Chapter 13), future chip iterations will improve its raw performance. For example, incorporating specialized circuits to to perform integration of synaptic output currents during HAGEN-mode would push the throughput of a single chip well beyond 10 TOp/s. Finally, by splitting the synapse array into several smaller chunks with correspondingly shorter wires, we could achieve even higher synaptic operating frequencies.



(a) Overview of BrainScaleS-2 Mobile Analog Neuromorphic Hardware System (from left to right): FPGA-based controller, ASIC interface consisting of two PCB – the ASIC adapter PCB and the ASIC carrier board –, and BrainScaleS-2 ASIC. The system controller provides the USB interface for the USB mass storage device with test and result data as well as the run control handshake signals for the energy measurement protocol of the inference competition detailed in Chapter 13. Taken from: [Stradmann et al., 2021, Figure 1]



(b) Photo of the BrainScaleS-2 Mobile system (from bottom to top): FPGA-based system controller, ASIC adapter PCB, ASIC carrier board with the latest BrainScaleS-2 ASIC directly wire-bonded to the PCB. The system has the mechanical footprint of a credit card (84 mm × 55 mm) at a height of approximately 40 mm. It weighs roughly 155 g with and 70 g without the FPGA's heatsink respectively. Taken from: [Stradmann et al., 2021, Figure 2].

Figure 3.8: Overview of BrainScaleS-2 Mobile

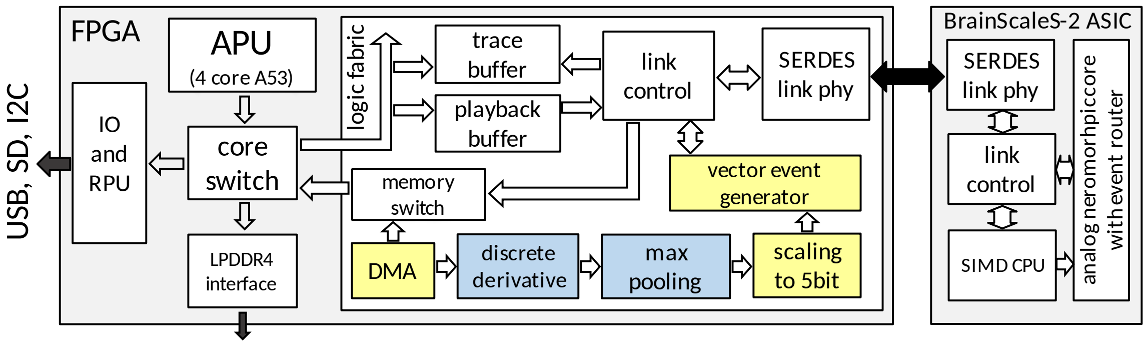


Figure 3.9: Block diagram of the major functional units of the FPGA, the part inside the logic fabric has been realized as custom RTL in SystemVeriolog.[57] The DMA controller, preprocessing chain elements and vector event generator create the input activation events representing the vector in the vector-matrix multiplication. Some of the preprocessing (blue) is problem-specific for the medical ECG dataset discussed in Chapter 13. To the right side the major blocks of the BrainScaleS-2 ASIC are shown as well to illustrate the complete communication path from the embedded PPU to the DRAM memory. The arrows denote the control flow direction from initiator to follower of the internal (hollow) and external (filled) data buses shown in the figure. Taken from: [Stradmann et al., 2021, Figure 6].

3.2.3 BrainScaleS-2 Mobile System

The latest addition to the BrainScaleS-2 lineup is BrainScaleS-2 Mobile.[58] In contrast to the “cube” setup (cf. Figure 3.5) that consists of a controlling FPGA with BrainScaleS-2 ASIC and is reachable via Ethernet, BrainScaleS-2 Mobile is a full SoC with roughly the same area as a credit card, containing a base board consisting of a low-power FPGA with an embedded quad-core microprocessor [Xilinx, 2019; AVNET, 2020] and 2 GiB of LPDDR4 DRAM, USB 3.0 (device & host), SDXC, 802.11b/g/n Wi-Fi as well as Bluetooth 4.2 (BLE) communication circuits (cf., “system controller” in Figure 3.8a), a custom adapter PCB[59] (cf., “ASIC interface” in Figure 3.8a), interfacing the different connectors of the FPGA board to a single SO-DIMM[60] connector for the ASIC carrier board, and the BrainScaleS-2 ASIC directly bonded to a carrier PCB using a SO-DIMM edge connector.

The embedded quad-core microprocessor eliminates the need for a host computer and allows for fully standalone execution and mobile deployments. A schematic overview is given in Figures 3.8a and 3.9, while a photograph is shown in Figure 3.8b. The PPUs on the BrainScaleS-2 ASIC are able to instruct the FPGA via a DMA[61] engine, mapping additional functionality into the PPUs memory range. The FPGA can then be used to perform task specific preprocessing (blue shading in Figure 3.9) and generate vector events. Using a lookup table inside the FPGA allows for arbitrary mapping of input vector elements onto the synapse array. Multiple handshake signals are used to synchronize result readout in the PPUs to input submission via vector event generator. The quad core microprocessors, while present, are not necessary for experiment execution besides initialization (cf. later Section 13.2.4). However, when not aiming for power efficient inference, the quad cores can serve as host computer, for example allowing users to submit experiment steps from remote sites via an intermediate scheduler developed during this thesis (cf. Chapter 10).

Finally, the ASIC adapter PCB provides several shunt-based power monitoring ICs[62] [Texas Instruments, 2020]. They allow for monitoring of individual supply currents of the BrainScaleS-2 ASIC. The whole readout chain was optimized for maximum sampling frequencies in order to allow for accurate calculations of energy consumption via integrated power samples. Overall, the temporal resolution was 294 Hz for sensors on the base PCB, while 4.4 kHz were reached for sensors on the ASIC adapter board. This was used extensively to perform measurements during the BMBF[63] Pilotinnovationswettbewerb “Energieeffizientes KI-System” (*Energy-efficient AI system*), detailed in Chapter 13.

[58]BrainScaleS-2 Mobile Analog Neuromorphic Hardware System, [Stradmann et al., 2021]
[59]Printed Circuit Board
[60]Small Outline Dual In-line Memory Module
[61]Direct Memory Access
[62]Integrated Circuits
[63]Federal Ministry of Education and Research (*Bundesministerium für Bildung und Forschung*)

4 Software Development in Science

This chapter serves as supplementary information to Part II.

Planning, testing and reviewing has become a staple in software development ever since its first formalization in the 1970s [Fagan, 1976]. There are many principles and workflows ensuring software quality [Gilb et al., 1993; Chrissis et al., 2011], including an ISO[1] norm [ISO, 2005]. Here, we bring attention to some often overlooked concepts in the context of science and give an overview of possible alternatives to the solutions presented in Part II.

4.1 Importance of High-Quality Software

In this section we give an overview and reasons as to why everyone should care about proper software engineering.

Software is ubiquitous. It is used to plan food production [Yang et al., 2011], control infrastructure [Jones et al., 2004], guide medical appliances [Leveson et al., 1993] or entertainment [Mnih et al., 2013; Justesen et al., 2017]. Ensuring proper rigor and quality assurance is therefore of utmost importance. It should not be treated as an afterthought. Even as early as 2002, the annual costs attributed to inadequate software were estimated to be 22.2–59.5 billion US dollars in the United States alone [RTI, 2002]. Recently, Jeff Hawkins said that Numenta[2] spends about half its budget on developing its software framework and treats usability for external users among the highest priorities.[3] Whenever software is not a priority, things can go horribly wrong.

One very famous example is the Therac-25 incident, summarized succinctly in the report [Leveson et al., 1993]. The Therac-25 was a machine for radiation therapy, administered automatically by a computer-controller procedure. Between 1985 and 1987, it was involved in at least six accidents where patients suffered from massive radiation overdoses.

An investigation revealed various grievances in the way software work was handled: Essentially, the controlling software had been adapted from the software stack controlling a previous hardware iteration, the Therac-6, by *a single person*. It was written in PDP 11 assembly language over the course of several years with no discernible documentation.

[1]International Organization for Standardization

[2]Numenta is a machine intelligence company developing a cohesive theory, core software, technology and applications based on the principles of the neocortex. `https://numenta.com/` (visited on 2021-05-03)

[3]Personal communication at round of questions after a talk given at NICE 2021.

Although the system was tested both in a simulator as well as fully integrated for about 2700 hours of use, this was not enough to uncover various bugs. Programming was conducted in a concurrent paradigm despite there being no support for proper atomic operations. A hardware interlock, still present in previous generations, mechanically prevented some combinations of settings. It was removed because the software was thought to also check for these conditions. The accidents were, essentially, race conditions where the machine assumed it was operating at different beam energies than had been set, leading to massive overdoses. Furthermore, the probability for these accidents to occur were related to how fast the human operator entered their instructions to the machine, again explaining why these glitches were not uncovered during testing but only occurred under real world conditions after personnel became acquainted with the device.

Despite the Therac-25 incident now lying about 35 years in the past, there are more recent examples. A last minute change to optimize runtime of division operations in Pentium processors introduced the now infamous "Pentium bug" [Pratt, 1995]. It caused results to be slightly wrong in one of every 40 billion random single precision divisions. Proper verification of the design in software would have caught the error.

From 1999, the UK introduced the Horizon system into their Post Office network [Peachey, 2021]. Developed by the Japanese company Fujitsu, it was used for tasks such as transactions, accounting and stocktaking. Unfortunately, it suffered from substantial bugs in regards to transactions, sometimes amounting to several thousands of pounds. Subpostmasters were accused of theft and convicted because the proprietary Horizon was seen as infallible. Some even used their own money to cover-up discrepancies caused by software errors. There was no way for them to properly inspect the software to prove their innocence. It was only this year, 22 years later, that their names were cleared.

The grounding of all Boeing 737 MAX was a combination of both cutting development time to save costs and overconfidence in the capability of software to make decisions overruling humans [Johnston et al., 2019]. It resulted in two crashes of aircraft in Indonesia and Ethiopia between October 2018 and March 2019, resulting in 346 casualties. Because of design limitations of attaching larger more efficient engines to a smaller airplane frame, automated software was used to prevent over-steering. Again, implementation was done hastily: There were no fall-back solutions, a lack of proper documentation and inadequate training for pilots to be aware of how to overwrite the system on misbehavior.

But also apart from the main headlines, there are little quirks. A recent airworthiness directive[4] enforces that all Boeing 787 must effectively be turned off and on again every 51 days. If not, this could lead to "display of misleading data". This is not as bad as a previous error of the Boeing 787,[5] causing a shutdown of the plane's electricity generator every 248 days due to a memory overflow bug. Of course, Boeing is not the only manufacturer affected by these bugs, as Airbus recently patched a bug in the A350 that required a reboot of the plane every 149 hours.[6]

[4]https://ad.easa.europa.eu/ad/US-2020-06-14 (visited on 2021-04-10)

[5]https://s3.amazonaws.com/public-inspection.federalregister.gov/2015-10066.pdf (visited on 2021-04-10)

[6]https://ad.easa.europa.eu/ad/2017-0129R1 (visited on 2021-04-10)

Scientific software is impacted as well. In particular, [Eklund et al., 2016] performed an analysis on fMRI,[7] a common method for identifying active areas in the brain while subjects perform certain tasks. Using real resting-state data, i.e., without any particular activity, they validate common statistical fMRI-methods. Despite expecting a false-positive rate for significant data of 5 % from theory, they found false-positive rates of up to 70 % with the most common software packages for fMRI. Besides concluding a "need of validating the statistical methods being used in the field of neuroimaging", they also found software bugs that lead to overestimation of significance for found results. One bug in particular had been present for *at least 15 years* despite active usage. This questions the validity of a *large* number of fMRI studies, that unfortunately, "Due to lamentable archiving and data-sharing practices", are unlikely to ever be redone.

[Soergel, 2015] investigate the influence of software errors further. They estimate that for a typical medium-scale bioinformatics analysis with 100 000 lines of code "the probability of a wrong output is effectively 100 %", whereas a smaller, more focused analysis with 1000 lines of code only has a 5 % chance of a wrong output. Furthermore, they stress that, in contrast to work performed in a lab, "software errors produce outcomes that are inaccurate, not merely imprecise" because even small errors can have significant ramifications downstream. Overall, they call "for all scientists to recognize the urgent need to verify computational results". This also plays a key part in the reproducibility-crisis in (software-aided) sciences that is discussed further in Section 4.3.

Finally, improper implementation, review and security auditing enables criminal exploitation. So called ransomware uses security vulnerabilities to enter systems and encrypt all stored data [Kharraz et al., 2015]. Users are then given the option to pay the name-giving *ransom* in exchange for the decryption cipher. These have affected companies but also hospitals and other critical infrastructure. In recent years, attacks are steadily increasing in frequency.

The conclusions from the Therac-25 report [Leveson et al., 1993] were as valid back then as they are today:

- Documentation should not be an afterthought.
- Software quality assurance practices and standards should be established.
- Designs should be kept simple.
- Ways to get information about errors – for example: software audit trails – should be designed into the software from the beginning.
- The software should he subjected to extensive testing and formal analysis at the module and software level: system testing alone is not adequate.

They are in line with other reviews of the subject. These conclude that good quality software and introduction of quality assuring policies, while more expensive initially, reduce costs [Jones, 1994; Slaughter et al., 1998]: "Focus on quality, and productivity will follow". One of the earliest thorough investigations into the subject of why planning and

[7] functional Magnetic Resonance Imaging

managing software projects is hard can be found in [Brooks, 1978]. Already back then, it concludes:

Costly and late projects invest most of the extra work and time in finding and repairing errors in specification, in design, in implementation.

Of course, operating a neuromorphic hardware platform does not entail the same risk for loss of life, still, we should employ the same rigor during development.

4.2 Scientific Software Development

As stated in Section 4.1, software has become a core component of everyday life. It is only natural that it permeated sciences in much the same way [Wilson et al., 2014; Anzt et al., 2020]. In general, scientists are curiosity- and result-driven people. They aim to gather new insights into their respective fields. Software is yet another tool in their belt alongside other measuring equipment or even mathematics; a means to an end.

This poses a problem with regards to reuse. While the original author implicitly knows the inner workings and limitations of their software, anyone else merely adopting it with the intent to safe time might not. Without proper verification, people can merely assume that other people's software is working correctly. Additionally, since scientists often prototype ideas, this nature is reflected in code. It typically grows organically, branching to try out new ideas whenever necessary. This is true for any software under use because requirements shift and therefore involve changes [Brooks, 1987].

Slowly, but steadily, this situation is being noticed. Some key challenges identified by [Anzt et al., 2020] are:

Lack of benefit for the individual
Researchers committing time to software are at a disadvantage compared to their peers, who have relatively more time to produce publications. However, the benefit for the community as a whole is clear because funds are freed from reinventions of the wheel.

Lack of suitable incentive systems
There is no proper reward-system for scientific products other than traditional text-based publications. For example, software citations are not always mandatory in publications. In most cases, research software repositories that are published alongside a paper are not maintained afterwards.

Lack of awareness
Research software sustainability is still somewhat of a niche topic, lacking both visibility and acceptance. This thesis aims to do its part for changing this.

Lack of expertise
Not every scientist is interested or knows how to design, implement and maintain software properly. However, on average, the level of knowledge is rising across the community.

This is also the case at Electronic Vision(s) despite interest for high-level modeling still outweighing low-level system engineering.

Lack of impact measures

The question of how to measure research software's impact is still unclear. This includes quality, re-usability or general benefit to the scientific community.

All methods presented in this thesis face the same problem. Feedback from users has been positive, but it is hard to quantify beyond anecdotal worst-case scenarios that are clearly solved (cf. intro to Chapter 8).

Infrastructure issues

There is not enough evidence to support either centralized or decentralized infrastructure when it comes to the application of scientific research software.

At Electronic Vision(s), we are clearly in favor of a decentralized approach and make no use of external services but rather self-host. However, this takes dedicated personnel to set-up and maintain continuously. Not every research group has the luxury to afford this, either among their PhD candidates or permanent employees. Some of the infrastructure in place at Electronic Vision(s) is presented in Part II.

Slow adoption of research software engineering as a profession

Career options of research software work are still in flux with some progress being made in UK. In Germany, US, and the Netherlands this is still considered work in progress.

They summarize that proper software work is at odds with the prevailing “publish or perish”-mantra in science. This needs to change. As Turing Award winner Fred Brooks argues in his infamous “No Silver Bullet” essay [Brooks, 1987]:

> *There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.*

Sustainable software development is hard work, doubly so in a scientific context. Once the foundation has been laid out, it needs to be sustained and extended organically in small, incremental steps, each verified individually, thereby promoting reuse. The wheel should only be invented once.

4.3 Reproducibility of Software-Aided Science

Reproducibility is one of the key aspects of the scientific method. If a phenomenon is not reproducible, it cannot be studied scientifically. No matter the area of research, significant insights are only accepted if they are reproduced independently by several parties. This principle has helped weeding out many false claims in the past, including physics [Goodstein, 2010, Chapter 6]. Recent trends, however, show that it becomes harder and harder to replicate results [Baker, 2016; Mesnard et al., 2016; Krafczyk et al., 2021].

Since reproducibility in lab-based experiments is not within scope of this thesis, we focus on reproducibility in software-related experiments. But, as the analysis-step of most lab-based experiments is software-aided, they are included by proxy. [National Academies of Sciences et al., 2019] gives the following definition:

> *We define reproducibility to mean computational reproducibility – obtaining consistent computational results using the same input data, computational steps, methods, code, and conditions of analysis; and replicability to mean obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data. In short, reproducibility involves the original data and code; replicability involves new data collection and similar methods used by previous studies.*

This is not always a given in computational sciences. The authors of [Stodden et al., 2018] examined 308 articles from the Journal of Computational Physics, studying how supplementary code was treated. They were able to obtain code for 55 articles. Allowing for up to four hours per article, they were *unable to reproduce* any article's *complete* set of results.

[Krafczyk et al., 2021] build upon this by limiting the scope to 7 articles, but allowing for up to 40 hours of human wall clock time (i.e., excluding computation time) to read up on references and allowed reaching out to the original authors. Their efforts are detailed in 18 small vignettes worth reading, with many rather relatable scenarios. In conclusion, they are able to "regenerate numerically identical results to some of those in the articles, and visually similar figures". This is a more positive outcome than [Stodden et al., 2018], but in an ideal world we would expect all results able to be regenerated within a few invested human-minutes, not counting any computational expenses, of course.

Reproducibility is no small feat, also challenging in non-scientific areas [Goswami et al., 2020]. In the best-case scenario one researcher simply is unable to get the code of other researcher to run (a detectable fail-state); in the worst-case scenario, however, code can be executed as expected, but differs slightly, but significantly in its output due to mismatched environments[8] (a hard-to-detect fail-state), see [Krafczyk et al., 2021, Vignette 14]. It is especially the latter scenario that is worrying because the experiment fails to be reproduced not because the presented idea is wrong but because of mere technical reasons that are avoidable. Overall, it reduces confidence in reported scientific results.

Software dependencies are often treated as an afterthought [Cox, 2019]. They are expected to simply work. However, if not properly tracked, they can pose a significant obstacle to reproducing work of another researcher. It is in part because of this that building software is a considerable time-sink in general [Dubois et al., 2003].

There are numerous attempts at solving these dependency management issues. These include, but are definitely not limited to: [Belguidoum et al., 2007; Zhang et al., 2017; Tovar et al., 2018; Sampedro et al., 2018; Pouchard et al., 2019; Bhatt, Asti et al., 2020]. Typically, each solution is tailored to its specific use case. As the scientific community as a whole

[8]The quote "it works on my machine" has become a far too well-known saying among software-developers and computer-aided scientists alike.

moves towards an era in which data is FAIR[9] [Stall et al., 2018] and open,[10] reproducibility and availability become even more important concepts as time advances [Krafczyk et al., 2019; Anzt et al., 2020].

Another challenge is a high fluctuation-rate of people. Typical research stays range from months to a few years. Here, again, people are primarily expected to produce results, i.e., theses or publications. There is no additional benefit in helping subsequent researchers re-using one's code. In the context of this thesis this is a problem for experiments that are intended to show-case the capability of neuromorphic hardware as they should be kept in a functional state to serve as example and verification tool. If they are not constantly verified, they could silently fail due to minor changes to the codebase with unintended side effects.

A recent investigation into reproducibility of published computer-generated results is [Krafczyk et al., 2021]. The authors give three guidelines of reproducible computational research:

P1. Provide transparency regarding how computational results are produced

P2. When writing and releasing research software, aim for ease of (re-)executability

P3. Make any code upon which the results rely as deterministic as possible.

They define a *Reproduction Package* with guidelines derived from their own attempts at reproduction (see above), such as CI[11] or clear information on expected results, computational effort and explicitly named entry point scripts, i.e., `run.sh`. Their guidelines include[12]:

G1. Make all artifacts that support published results available, up to legal and ethical barriers.

G2. Connect published scientific claims to the underlying computational steps and data.

G3. Specify versions and unique persistent identifiers for all artifacts.
Each visionary container presented in Section 8.2 is uniquely named, based on date.

G4. Declare software dependencies and their versions.
Each visionary package presented in Section 8.1.4 represents a dependency specification. Each visionary container presented in Section 8.2 contains a full list of software versions in `/opt/spack_specs`.

G5. Refrain from using hard coded parameters in code.

G6. Avoid using absolute or hard-coded filepaths in code.

G7. Provide clear mechanisms to set and report random seed values.

G8. Report expected errors and tolerances with any published result that include any uncertainty from software or computational environments.

[9]Findable, Accessible, Interoperable, and Reusable
[10]`https://plos.org/open-science/` (visited on 2021-04-11)
[11]Continuous Integration
[12]Where applicable, we *highlight* how the methods presented here already fulfill these

All results in this thesis are reported including uncertainties.

G9. Give implementations for any competing approaches or methods relied upon in the article.

For methods presented in Chapters 12 and 13, implementations are available at Appendix B.2.

G10. Use build systems for complex software.

We use `waf`[13] *(cf. Section 7.3) to build the complete software stack for both BrainScaleS generations presented in Section 3.1 and Chapter 6. Combined with visionary containers, the user does not need to install any dependencies.*

G11. Provide scripts to reproduce visualizations of results.

Tools like `gridspeccer`,[14] *developed (primarily by the author) to facilitate easier plotting, can help to reproduce visualizations. Prior to becoming a standalone tool,* `gridspeccer` *was already embedded into several paper repositories (including [Petrovici et al., 2017a; Petrovici et al., 2017b; Kungl et al., 2019; Göltz et al., 2021]) and allows for easy reproduction of plots from pre-generated raw experiment results.*

G12. Disclose resource requirements for computational experiments.

Typically, all papers regarding neuromorphic computing make it a feature point which platform was used to execute. The same is true for this manuscript.

They also give recommendations for how to achieve each of the guidelines. In the context of neuromorphic hardware that is not yet available to the general public, not all guidelines are applicable (yet). For experiments presented in this thesis, their software environment is described in Appendix B.2.

4.4 Software Concepts

This section gives an overview over software concepts used in Part II. In particular, we explain core concepts of used version control as well as explain the problem of package managing and how containerization is accomplished in Linux. Furthermore, we give an overview over both package managers and container implementations.

4.4.1 Version Control: `git`

Ever since its inception, `git`[15] has become one of the most popular version control systems. It was initially created by Linus Torvalds after losing access to the existing version control for the Linux kernel source. Since it is a powerful tool that focuses on productivity, its use

[13] Waf: the meta build system, [Nagy, 2005]

[14] `https://github.com/obreitwi/gridspeccer` (visited on 2021-04-12)

[15] Git – a distributed version-control system for tracking changes in source code during software development, see Section 4.4.1, [Torvalds et al., 2005]

must be properly learnt. For a full conceptual tutorial, we recommend [Duan, 2010]. Over the years, it has helped several students grok the inner workings of `git`.

In this section we explain certain `git`-related terms that are used throughout this thesis. They are ordered such that the list can be understood in one read-through.

repository
: A repository is a collection of code snapshots, forming its history. It contains commits.

commit
: A commit corresponds to a snapshot of all tracked files in the workspace. Most commits have exactly one parent commit, forming a history of snapshots. It is also valid to have zero or several parents (see below). Its most important metadata are author, a (hopefully informative) commit message describing its content and a hash, generated from its contents (including parent information). This hash can be used to uniquely identify any commit.

commit-trees
: Most commits have at least one parent. They therefore form a treelike[16] data structure: the commit-tree. It represents the history of a repository.

  The only non-"treelike" property is merger of two commit histories into one. In this case merge-commit is created that with several parents. Other than that, they are just like regular commits, i.e., a snapshot of the workspace.

merge-commit
: Signifies the merging of two commit-trees and therefore has several parents.

branch
: Branches designate entry-points into a repository. Effectively they are just names for certain commits, typically leafs in the commit-tree.

  Naming conventions typically use the term `master` or `main` for the stable branch. When developing features or trying out different ideas, developers typically create differently named feature-branches.

tag
: A tag is like a branch in that it is a different name for a specific commit, but is not expected to change. Usually different releases, milestones or otherwise significant commits are tagged.

checkout
: To checkout a commit means to unpack its snapshotted contents into the current workspace.

`HEAD`
: `HEAD` is an implicitly defined branch pointing to the currently checked-out commit.

cherry-picking
: To cherry-pick a commit means to apply the changes of a commit (i.e., the exact

[16]Since it contains loops through feature-branches that are merged back, the commit history is not a real tree-structure.

differences to its parent) to `HEAD` and creating a new but *different* (i.e., a new hash) commit with the otherwise same metadata.

fetch
To fetch from a remote repository means download its commit-tree but not change the current workspace in any way. Typically, all designated branches and their history are downloaded.

pull To pull from a remote repository means to first fetch it and then try to align the currently checked-out branch with its remote counterpart. Different pull strategies can be chosen from, including merging and rebasing.

fast-forward
When pulling from a remote repository and all new commits are direct descendants of `HEAD`, we can simply move the branch (fast-) forward down the tree to the new leaf to match the remote branch. No new local (merge-)commits have to be created.

rebasing
A rebase corresponds to an "uprooting" of a commit-tree to a different parent commit. Each commit is then applied like a cherry-pick, i.e., the changes to its original parent are replayed onto the current `HEAD`. A (trivial) example rebase is exemplified later in Figure 7.2.

clone
Downloading a repository from a remote site is called "cloning".

fork A fork is another word for a clone or copy of a repository, typically hosted at a different location.

upstream
A colloquial name for the "official" repository hosting a given project.

living at `HEAD`
The term "living at `HEAD`" means that the current `HEAD` in the upstream repository, i.e., the stable branch, should *always* be in a usable state and never point to a commit that cannot be built or fails to pass all tests.

4.4.2 Package managers

Package managers, as the name suggests, manage software packages, i.e., they aim to track dependency relations between disjoint software projects and provide means to resolve them. Since the problem of solving dependencies is universal, package managers exist at several different levels of generality. Here, we give a brief overview over existing solutions.

For a single language, tools like `python-setuptools`[17] for Python[18] (in combination with

[17]Python Setuptools, [PyPA2006]
[18]Python Programming Language, [Rossum, 2000]

pip[19]), `cargo`[20] for Rust[21] or `stack`[22]/`cabal`[23] for Haskell[24] allow packages to express their dependence on other packages *within* the same language in a corresponding file.[25] Other examples of intra-language dependency tracking include `go`[26]-modules[27] or `npm`[28] for Javascript. Building[29] (or installing) a package then causes the tool to fetch missing dependencies, often querying a remote database for additional information. Meta-build tools – such as Bazel,[30] Contractor,[31] MixDown[32] or `symwaf2ic`[33] (used at Electronic Vision(s) and discussed in Section 7.3) – can be regarded as crude package managers with much smaller scope, in the sense that they ensure dependencies for a single package are met.

Then, there are *binary package managers* managing installed software in binary format on a OS[34]-level, such as APT,[35] RPM,[36] YUM[37] or `pacman`[38] for Linux distributions. They do not concern themselves with how packages are built and only model their interdependencies. Software-packages are retrieved and installed with their dependencies from remote sites, often called mirrors or repositories (not to be confused with `git`-repositories). Furthermore, they *track conflicts*/incompatibilities; if an update to one package would render one of its dependents unusable (e.g., because it has yet to receive an update), the package manager blocks the new version from being installed until the conflict is resolved. Ideally, installing new or updating existing packages *never*[39] bricks the system. Overall, though, they deal with software installed at the *system* level: Every package is installed at most at one version and package contents are installed at fixed locations[40] – two versions of the same package or two packages providing the same functionality would be installed to the same location.

[19]Package Installer for Python, [PyPA2008]
[20]`cargo`: A Package Manager for Rust, [Katz et al., 2014]
[21]Rust Programming Language, [Matsakis et al., 2014]
[22]Haskell Tool Stack, [FP Complete, 2015]
[23]Haskell Cabal, [Jones et al., 2005]
[24]Haskell Programming Language, [Marlow et al., 2010]
[25]`requirements.txt` for Python, `Cargo.toml` for Rust and `stack.yaml`/`cabal.project` for Haskell
[26]go Programming Language, [Pike, 2009]
[27]`https://golang.org/doc/modules/managing-dependencies` (visited on 2021-05-03)
[28]Node Package Manager, `https://www.npmjs.com/` (visited 2020-05-03)
[29]Python packages are typically not built but rather transposed to bytecode and packed along with any static resources, unless there are (C-)extensions.
[30]`https://bazel.build/` (visited on 2021-05-03)
[31]`https://home.fnal.gov/~amundson/contractor-www/` (visited on 2020-12-06)
[32]MixDown: Meta-build tool for managing collections of third-party libraries, [Epperly et al., 2010]
[33]Electronic Vision(s)-specific fork of `waf`
[34]Operating System
[35]Advanced Packaging Tool, [Silva, 2001]
[36]RPM Package Manager, [Troan et al., 1995]
[37]Yellowdog Updater, Modified, [Vidal, 2011]
[38]Package Manager, [Vinet et al., 2002]
[39]Depending on maturity and level of sophistication in both package manager and managed packages, updates breaking an installation are more, or less, likely; just ask any Linux enthusiast what distribution they are using, btw…
[40]Typical install locations are at the root-level, i.e., `/usr/{bin,include,lib,man,share}`.

As a side remark, even modern “app stores”[41] can be regarded as package managers as well, however, all packages (i.e., “apps”) are self-contained and there are no “direct” interdependencies in the sense that users install one app per desired functionality.

At the non-system, i.e., user-level, there are also binary package managers, most prominently `conda`[42] and its distribution Anaconda,[43] often employed in machine learning and data science communities. They install a specific set of packages in a user’s home-directory, completely separate from any system-installation. Packages are only evaluated to be compatible with each other, which can make it hard to get external software to interact with a `conda`-environment. Each instance only manages software packages for a single user, i.e., every user will have their own personal copy of all software packages in their home folder. Recently, mamba[44] was introduced as an open-source reimplementation of `conda` in C++ [Vollprecht et al., 2020]. It is part of a larger ecosystem that aims to lessen the dependence on Anaconda which is only partially open-source.[45]

There are also means to distribute single applications as images. Here, all dependencies such as libraries are bundled along with the application itself in order to keep external dependencies to a minimum. The result is a portable image file that is compatible with a wide range of Linux distributions, at the cost of larger file size. Popular examples include Snap,[46] Flatpak,[47] AppImage[48] and ZeroInstall.[49]

When it comes to HPC,[50] there are some more aspects to consider: There are many different users in the same environment, each one potentially requiring a specific version of each package to run their software. Hence, package managers need to provide a way of installing and maintaining several versions of the same package installed at the same time. A specific version is then selected by modifying the user’s environment in such a way that binaries are automatically found by the runtime (via `${PATH}`) and dynamically linked libraries by the loader (via `${LD_LIBRARY_PATH}`). These environment modifications are typically performed via `dotkit`,[51] GNU modules[52] or Lmod.[53] Environment modifications are reversible by tracking them via so-called module files. Another way of ensuring binaries find dynamically linked libraries is by using `RPATH`s: Instead of providing dynamic

[41]The most prominent app stores include `https://www.apple.com/app-store/` for Apple products, `https://play.google.com/store/apps` for Android devices, but even `https://www.microsoft.com/en-us/windows/windows-10-apps` for Windows 10 (all visited on 2020-12-01).
[42]conda: A Cross-Platform, Python-Agnostic Binary Package Manager, [CA2017]
[43]Anaconda Software Distribution, [CA2016]
[44]The Fast Cross-Platform Package Manager, `https://github.com/mamba-org/mamba` (visited on 2021-02-22)
[45]Besides its free “Anaconda Individual Editition”, Anaconda, Inc. provides both “Team” and “Enterprise Edition” as commercial applications. `https://www.anaconda.com/pricing` (visited on 2021-04-14)
[46]`https://snapcraft.io/` (visited on 2021-05-03)
[47]`https://github.com/flatpak/flatpak` (visited on 2021-05-03)
[48]`https://appimage.org/` (visited on 2021-05-03)
[49]`http://0install.net/` (visited on 2021-05-03)
[50]High-Performance Computing
[51]`dotkit`, Simple Module Files via Shell Scripts, `https://dotkit.sourceforge.io/` (visited on 2012-12-06)
[52]GNU Environment Modules, [Furlani, 1991]
[53]Lua-based Module System, [McLay et al., 2011; Geimer et al., 2014]

linking information at runtime, the absolute path to specific libraries is embedded directly into the binary, so that they can be loaded without any further information from the environment. The module files are then only used to provide non-linking-related environment information, such as `${MANPATH}`.

Package managers suitable for HPC include `smithy`,[54] a command line installation tool that can generate module files and features passive dependency resolution, i.e., like regular build tools, it can check if all prerequisites are present prior to installing, but does not provide automatic handling of dependencies. It is not under active development anymore. EasyBuild,[55] a Python-based package manager, enables generation of Lmod module files and provides ways to automatically resolve dependencies and link binaries. It does not set `RPATH`s.

Recently, newer approaches such as Nix[56] offer a different take on escaping the "dependency hell": Every package is deployed in isolation, whereby its deploy-location is determined by a cryptographic hash of its configuration and dependency information. These deploy-locations are not human-readable, preventing users from discovering which software packages are currently installed by "typical" Unix-interactions, such as inspecting `{/,/usr/}bin` – the information has to be provided by other means. No dependency on software deployed system-wide (as above) is permitted, enforcing packages to know hashes of all dependencies in order to link/interact with them in any form. This minimizes the risk of untracked dependencies and allows for features such as atomic upgrades, side-by-side installation of different versions of the same package at the same time and even downgrades. "Atomic" here means that when updating software packages to new versions, there is no point in time such that a package is in a mixed state of partially upgraded and non-upgraded files that could potentially lead to errors in itself or dependents. Furthermore, dependents need to be explicitly "reinstalled" with updated dependency-information or they will simply keep making use of their old dependencies. Dependency relations and build instructions are specified in a functional language that is evaluated lazily. Via automated garbage collection, particular versions of packages can be safely removed once there are no more dependents.

HashDist[57] also employs dependency resolution by hashing and – same as Nix – ensures correct linking of binaries to dynamic libraries via `RPATH`s. However, it is not under active development anymore.

Spack,[58] the choice for package management employed at Electronic Vision(s), is discussed in Section 8.1. It incorporates hashing as a way to track dependency information like Nix, but offers an easy-to-use Python DSL to express dependencies and adjust build flows to fit into the Spack ecosystem.

[54]Smithy, [Jones et al., 2008]
[55]EasyBuild: Building Software with Ease., [Hoste et al., 2012]
[56]Nix Package Manager, [Dolstra et al., 2004]
[57]HashDist, [Ahmadia et al., 2012]
[58]Supercomputing PACKage manager, [Gamblin et al., 2015]

4.4.3 Lightweight Containers

Docker containers are kind of neat. They are also kind of a craven surrender to the rotting mess of excessive software complexity.

— John Carmack[59]

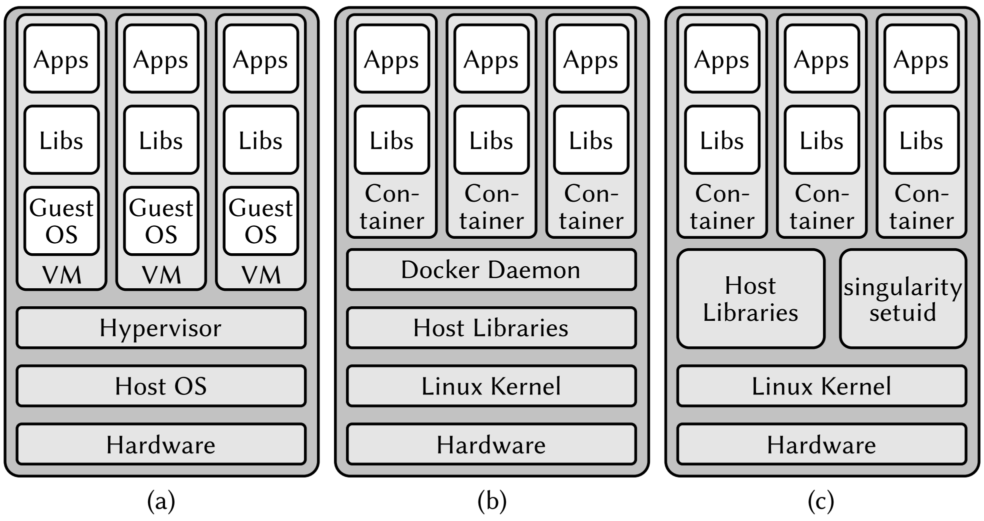


Figure 4.1: Overview over different types of virtualization. **(a)** "Traditional" virtualization works by emulating the complete VM, including OS. In fact, the host OS is not needed to run a hypervisor. The host machine may not even need to run a full host OS. **(b)** The Docker[60] daemon, running with elevated rights, takes on the task of the hypervisor, i.e., managing lightweight containers. It needs to be running for as long as Docker[61] is to be used. **(c)** Singularity, by contrast, uses a `setuid`-binary to perform all namespace-related operations on container startup. Elevated rights are dropped at the earliest possibility. Afterwards, the process running within the container is a regular process. The kernel handles all clean-up upon exit.

In "traditional" VMs[62] [Smith et al., 2005] one system *emulates* another at the *instruction level* in terms of compute devices, OS and kernel. This often results in a performance penalty unless mitigated by special hardware-features [Chen et al., 2008]. Additionally, creating and maintaining them requires a certain amount of effort and is therefore used to isolate distinct services from each other. In general, it is infeasible to spin up a virtual machine in order to execute a single short process. Popular VM-implementations are Hyper-V,[63] QEMU,[64] KVM,[65] Xen [Barham et al., 2003] and VMware [Nieh et al., 2000]. On top of these virtualization technologies there are management middlewares such as libvirt [Bolte et al., 2010] and Vagrant [Hashimoto, 2013].

Oftentimes, full stack emulation is not even necessary and far too cumbersome. Therefore, we can make use of specific OS kernel features, such as namespaces and cgroups [Rosen,

[59]`https://twitter.com/ID_AA_Carmack/status/1385103110977179649` (visited on 2021-04-22)
[62]Virtual Machines
[63]`https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-on-windows-server` (visited on 2021-05-03)
[64]Quick EMUlator, [Bellard, 2005]
[65]Kernel-based Virtual Machine, [Kivity et al., 2007]

2013], to isolate processes from each other. Whereas namespaces are used to virtualize resources, cgroups are used to limit access to system resources on a per-process or per-process-group level. This allows for different processes on the same physical machine to exist in vastly different environments, including available hardware devices, filesystems and software environments. These lightweight environments, set up on-demand by the OS-kernel, are called *containers* [Bernstein, 2014]. The only common denominator across all such processes is the OS kernel, for obvious reasons. By making use of DBT,[66] even different ISAs[67] can be emulated within containers, albeit at a performance penalty [Bellard, 2005; Cota et al., 2017]. Core differences between virtualization principles are summarized in Figure 4.1. Especially for deployment, containers represent a huge step forward in software management as they help to decouple disjoint software packages. Ideally, applications become portable standard units of software that only interact via a few well-defined interfaces. Containers have been shown to outperform classical VM approaches [Felter et al., 2015].

4.4.3.1 | Linux Namespaces

Namespaces are a feature of the Linux kernel, allowing for partitioning of kernel resources so that sets of processes can be assigned different resources. This offers a form of process isolation. It is a feature effectively back-ported from Plan 9 [Pike et al., 1995], the ill-fated successor OS to Unix [Ritchie et al., 1978]. While they originated in 2002 (kernel `2.4.19`), it was the addition of user namespaces in kernel `3.8` that allowed for the implementation of containers [Kerrisk, 2013].

As of kernel `5.6`, there are eight different kinds of namespaces. Each can be nested. Every process belongs to exactly one namespace of each kind. It is then only able to access resources associated with its namespaces (and possibly descendant namespaces).

For new processes, the namespace configuration can be adjusted via `clone()`[68]-syscalls,[69] a more versatile variant of `fork()`,[70] whereas a current process can disassociate parts of its execution context via `unshare()`[71] or join other namespaces via `setns()`,[72] presuming sufficient permissions. It provides an extensive API (that exceeds the limit of this overview) to create new processes with fine grained control over, in particular, namespaces.

In detail, the namespaces[73] are:

[66]Dynamic Binary Translation
[67]Instruction Set Architectures
[68]`https://www.man7.org/linux/man-pages/man2/clone.2.html` (visited on 2021-05-03)
[69]System Calls
[70]`https://www.man7.org/linux/man-pages/man2/fork.2.html` (visited on 2021-05-03)
[71]`https://man7.org/linux/man-pages/man2/unshare.2.html` (visited on 2021-05-03)
[72]`https://man7.org/linux/man-pages/man2/setns.2.html` (visited on 2021-05-03)
[73]Full information can be found at `https://man7.org/linux/man-pages/man7/namespaces.7.html` and in particular at `https://man7.org/linux/man-pages/man7/user_namespaces.7.html` (both visited on 2021-01-28).

`mnt`: Mount
: Allows each process to see different sets of mounted filesystems. Typically, already existing bind-mounts are copied to a new namespace, but unless shared subtrees[74] are used, any binds added afterwards do not propagate back to the "parent" namespace.

`pid`: Process IDs
: As the name suggests, this namespace facilitates process isolation. `PID`[75]-namespaces form a tree. Any process is only able to "see" processes (i.e., target with them via signals) within its own and any child namespaces. In particular this means that processes in disjoint `PID` namespaces can appear to have the same `PID`.[76] The first process in a new `PID` process acts as `init`-process, i.e., it adopts "orphaned"[77] children. Furthermore, if the `init` process of a given `PID` namespace terminates, the kernel recursively terminates all processes contained in it and all descendant namespaces.

`net`: Network
: Each physical network device is attached to exactly one network namespace. Via virtual network devices,[78] tunnels can be created between different network namespaces. These can also be used for bridging to a physical network device in another namespace. All virtual network devices are destroyed upon namespace termination.

`ipc`: Interprocess Communication
: This namespace governs isolation of System V IPC objects[79] and POSIX[80] message queues.[81] In particular, processes in different `ipc` namespaces are *unable* to communicate by mapping the exact same memory region (e.g., via `shm`[82]). As a preluding side-note: `sctrltp`,[83] presented in Chapter 6, makes use of the `shm`-interface to send and receive data from the user process with zero-copy, only by moving pointers around.

`uts`: UNIX Time-Sharing
: This namespace allows for the same physical host to have different host and domain names.

`user`: User ID
: As written above, `user` namespaces are the core component to make containers work. They isolate many security-related identifiers and attributes, e.g., capabilities[84] or

[74]`https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt` (visited on 2021-05-03)
[75]Process ID
[76]In reality, every process has a unique `PID` for every parent `PID` namespace it is part of. Within each namespace, `PID`s are unique.
[77]Child processes whose parent process has terminated.
[78]`https://man7.org/linux/man-pages/man4/veth.4.html` (visited on 2021-05-03)
[79]`https://man7.org/linux/man-pages/man7/sysvipc.7.html` (visited on 2021-05-03)
[80]Portable Operating System Interface
[81]`https://man7.org/linux/man-pages/man7/mq_overview.7.html` (visited on 2021-05-03)
[82]`https://man7.org/linux/man-pages/man7/shm_overview.7.html` (visited on 2021-05-03)
[83]Slow ConTRoL Transport Protocol
[84]`https://www.man7.org/linux/man-pages/man7/capabilities.7.html` (visited on 2021-05-03)

the root directory. They allow for completely new mappings of UIDs[85]/GIDs[86] between their parent namespace and themselves. Whenever a process tries to access a resource, both UID and GID get mapped back into whatever namespace the resource was defined in. In particular, this allows for a process to effectively have administrative rights *within* its namespace (either by holding the `CAP_SYS_ADMIN` capability or having the `root`-UID `0`), for example allowing them to perform bind-mounts or change (virtual) network devices defined in the same namespace. When accessing a file, for example, UID/GID will be mapped to their counterparts in the namespace where the filesystem was mounted, i.e., values *different* from root. This provides far more efficient and flexible process isolation than previous implementations such as `chroot`[87] which are known to be breakable [Simes, 2002].

`cgroup`: Control group
Added in kernel `4.6`,[88] this namespace kind allows obscuring `cgroup` related information. Processes are then only able obtain `cgroup` information relative to their own, hiding true control group position and identity.

`time`
The latest namespace kind, integrated in kernel `5.6`,[89] allows processes to effectively operate in different system times.

Namespaces have applications beyond containers. For example, various browsers use it to isolate code running in each browser tab.[90] Programs like Firejail[91] enable easy isolation of untrusted processes using namespaces.

4.4.3.2 Overview: Container Implementations

The features as outlined above are only accessible via the kernel C-API and, hence, not very user-friendly. Therefore, there are *a lot* of tools and implementations aiming to make usage of and interaction with containers more approachable. Here, we give an overview with no claim of completeness. Please note that this overview does not contain container orchestration software such as Kubernetes[92] or OpenShift[93] as these build on top of container implementations presented here to allow for easy management of deployed containers “in the cloud”, i.e., cloud services like Google’s Compute Engine,[94] Microsoft’s

[85]User IDentifier numbers
[86]Group IDentifier numbers
[87]change root directory syscall, `https://man7.org/linux/man-pages/man2/chroot.2.html` (visited on 2021-04-11)
[88]`https://lkml.org/lkml/2016/3/18/564` (visited on 2021-05-03)
[89]`https://www.phoronix.com/scan.php?page=news_item&px=Time-Namespace-In-Linux-5.6` (visited on 2021-05-03)
[90]`https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux/sandboxing.md` (visited on 2021-05-03)
[91]`https://firejail.wordpress.com/` (visited on 2021-05-03)
[92]`https://kubernetes.io/` (visited on 2021-05-03)
[93]`https://www.openshift.com/` (visited on 2021-05-03)
[94]`https://cloud.google.com/compute` (visited on 2021-05-03)

Azure[95] or Amazon Web Services.[96] It was of no real concern for the topic of this thesis.

Docker [Merkel, 2014] is, at the time of writing, the most prominent and wide-spread container runtime and the de-facto standard for running isolated micro-services. Originally based on LXC[97] (see below), it uses a daemon process running with elevated user rights (i.e., as `root`) to handle container creation and management. While this is fine when the administrator has full control over the container images being deployed, as is typically the case in industry, it poses security risks in HPC environments where users potentially use externally provided containers that are not trustworthy [Combe et al., 2016]. There exist several solutions to mitigate these security risks (discussed below).

`containerd` In order to make different container runtimes inter-operable, there are efforts to standardize both runtimes as well as formats. The Linux Foundation[98] initiated OCI[99]: a lightweight open governance structure that aims to establish implementation-agnostic APIs and container format specifications. Docker[100] donated its core container runtime `runc`[101] as a reference implementation for the OCI runtime specification to the CNCF,[102] a Linux Foundation project which aims to accelerate adoption of microservices, containers and cloud-native apps.

In much the same way, `containerd` was contributed by Docker to the CNCF to serve as a full runtime, built on top of `runc` as executor. It is an industry-standard container runtime with a self-proclaimed emphasis on simplicity, robustness and portability, managing the complete lifecycle of its host system from image transfer to container execution.

`udocker`[103] is a minimal runtime that support running Docker containers without `root`-privileges. It "executes" containers by providing a `chroot`-like environment over extracted containers, incorporating many other tools and libraries to provide this functionality. For obvious reasons, any `root`-requiring functionality such as mounting filesystems or listening to privileged TCP[104] ports (below 1024) is not supported.

NsJail[105] is not a full container implementation but focusses on process isolation, utilizing the same kernel features as container runtimes. Additionally to utilizing namespaces and cgroups, it also provides programmable `seccomp-bpf`[106] syscall filters, allowing for even more fine grained control of what syscalls might be performed by the isolated process, and cloned as well as isolated Ethernet interfaces. Hence, it could serve as the basis of a container runtime but is more used for isolating networking services from the rest of the OS. Therefore, it also does not define any form of container image format.

[95] `https://azure.microsoft.com` (visited on 2021-05-03)
[96] `https://aws.amazon.com/` (visited on 2021-05-03)
[97] LinuX Containers, `https://linuxcontainers.org/` (visited on 2021-01-12)
[98] `https://www.linuxfoundation.org/` (visited on 2021-01-26)
[99] Open Container Initiative, `https://opencontainers.org/` (visited on 2021-01-26)
[100] Docker, [Merkel, 2014]
[101] runC, `https://github.com/opencontainers/runc` (visited on 2021-01-26)
[102] Cloud Native Computing Foundation, `https://www.cncf.io/` (visited 2021-03-02)
[103] `https://github.com/indigo-dc/udocker` (visited on 2021-05-03)
[104] Transmission Control Protocol, [RFC793]
[105] `https://nsjail.dev` (visited on 2021-01-29)
[106] `https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html` (visited on 2021-01-29)

`systemd-nspawn`[107] is a wrapper to run isolated processes and OCI-compliant containers directly from within `systemd`.[108] It also adheres to its own container interface definition,[109] allowing for `systemd-nspawn` to be used by more highlevel container managers.

Charliecloud [Priedhorsky et al., 2017] is a very lightweight solution, allowing to run containers in an unprivileged, i.e., user-space, setting. It is able to import and convert Docker images, but requires images to be unpacked to a directory prior to execution. Overall, it is implemented in around 500 lines of C and 300 lines of shell code, making extensive use of aforementioned *user namespaces* (cf. Section 4.4.3.1). These might not be available or enabled in older production systems. However, because of its focussed set of features, it requires a lot of user effort to set up containers in non-trivial use cases.

Shifter [Canon et al., 2016; Belkin et al., 2018] is a more sophisticated, but also not yet OCI-compliant Docker-image converter aimed at HPC-environments. To that end, it requires setup of a gateway service, typically on a specialized node, that pulls container images from a registry and repacks them to a suitable file format for HPC such as SquashFS.[110] Originally, it was based on `chroot` which has known security issues by design[Simes, 2002] but has since moved to a more secure runtime.

Buildah[111] is a container build service. It allows for the construction of OCI-compliant containers without a full container runtime or daemon installed. Developed by RedHat and part of its OpenShift orchestration software, it was developed to relax its dependency on Docker for container building.

`rkt`[112] (pronounced "rocket") was a container runtime released by CoreOS[113] as an alternative to Docker for their similarly named Linux derivate aimed at providing infrastructure for clustered deployments. Following the acquisition of CoreOS by RedHat, `rkt` was discontinued[114] and – same as `containerd` – donated to the CNCF.

Sarus [Benedicic et al., 2019] is an OCI-compliant container engine, again with a focus on HPC-environments. It consists of several software components, such as an image manager that, like Shifter, imports container images from other registries and converts them to Sarus' own file format. Each container consists of a bundle of a container image and a configuration file in JSON[115] format. The runtime-component of Sarus prepares these bundles and then calls into an OCI-compliant executor (e.g., `runc`) to launch the container. Via OCI hooks, Sarus enables integration for various HPC use cases, such as MPI,[116] GPU

[107] `https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html` (visited on 2021-01-29)
[108] systemd System and Service Manager, `https://cgit.freedesktop.org/systemd/systemd/tree/README` (visited on 2021-01-29)
[109] `https://systemd.io/CONTAINER_INTERFACE/` (visited on 2021-01-29)
[110] `https://github.com/plougher/squashfs-tools` (visited on 2021-01-26)
[111] `https://buildah.io/` (visited on 2021-05-03)
[112] `https://github.com/rkt/rkt` (visited on 2021-05-03)
[113] `https://www.openshift.com/learn/topics/coreos` (visited on 2021-05-03)
[114] `https://github.com/rkt/rkt/issues/4024` (visited on 2021-01-29)
[115] JavaScript Object Notation, [RFC8259]
[116] Message Passing Interface, [Graham et al., 2006]

acceleration or integration with HPC workload managers such as Slurm.[117] A hook is a customizable action performed at specific points during container lifetime.

LXC[118] is one of the first container implementations. It was the original container execution driver for Docker, but was made optional in `v0.9`[119] and then dropped in `v1.10`.[120] At the time of writing, apart from the original LXC toolset, it is comprised of several other projects: LXD is a revamp of the original LXC user interface while also allowing to manage container via REST[121] API. LXCFS is a FUSE[122] filesystem providing overlays for `cpuinfo`, `meminfo`, `stat` and `uptime` in container contexts. Finally, `distrobuilder` is an image build tool for LXC and LXD, reading in YAML[123]-based definition files.

`cntr` [Thalheim et al., 2018] is a toolbox to support interactive debugging of container applications. The main idea is that the core application is deployed in a "slim" container. In case a developer needs to interact with it, they can dynamically attach parts of a "fat" container image (e.g., to make tools available), effectively extending the slim image at runtime. Written in Rust, using FUSE, it supports the full Linux filesystem API, making it compatible with all container implementations listed here. The authors report "reasonable" performance while reducing the Top-50 images available of Docker Hub by 66.6 % in size, on average.

SCStore [Zhang et al., 2017] is the closest solution in terms of functionality visible to end-users to what we describe in Chapter 8. Based on Docker, they provide reproducible container builds for the Taihu-light Supercomputer and the campus computing facility of Tsinghua University. Howevervia, instead of tracking dependency relations and then building software packages, they extract dependency information directly from binaries. It then keeps a set of Docker containers that are mapped in a DAG[124] similar to Spack's dependency graphs. New application images can then be built from generated `Dockerfiles`. This also allows for in-place "offline" backup. Unfortunately, no source code for SCStore is available.

Furthermore, super-computing sites, like the JSC,[125] have started to provide their own dedicated container build systems.[126] **Singularity**[127], the container runtime chosen for the approach employed at Electronic Vision(s) and presented in this thesis, is discussed

[117]Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management, [Yoo et al., 2003]

[118]LinuX Containers, `https://linuxcontainers.org/` (visited on 2021-01-12)

[119]`https://www.docker.com/blog/docker-0-9-introducing-execution-drivers-and-libcontainer/` (visited on 2021-05-03)

[120]`https://docs.docker.com/engine/release-notes/prior-releases/#1100-2016-02-04` (visited on 2021-05-03)

[121]REpresentational State Transfer

[122]Filesystem in Userspace, `https://github.com/libfuse/libfuse` (visited on 2021-05-03)

[123]YAML Ain't Markup Language, `https://yaml.org/spec/1.2/spec.html` (visited on 2021-04-08)

[124]Directed Acyclic Graph

[125]Jülich Supercomputing Centre

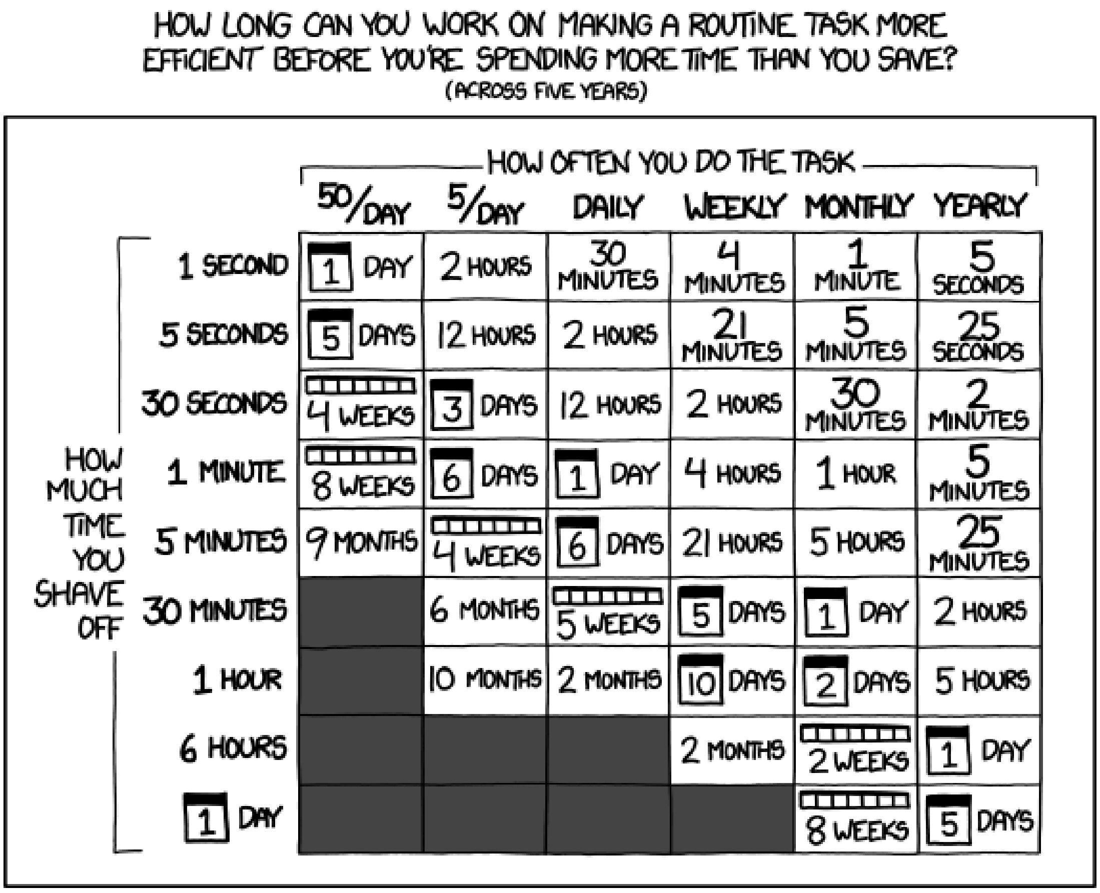
[126]`https://apps.fz-juelich.de/jsc/hps/jusuf/cluster/container-runtime.html#container-build-system` (visited on 2021-01-28)

[127]Singularity Container, [Kurtzer et al., 2017]

	Singularity	Shifter	Charlie Cloud	Docker
Privilege model	SUID/UserNS	SUID	UserNS	Root Daemon
Supports current production Linux distros	Yes	Yes	No	No
Internal image build/bootstrap	Yes	No*	No*	No***
No privileged or trusted daemons	Yes	Yes	Yes	No
No additional network configurations	Yes	Yes	Yes	No
No additional hardware	Yes	Maybe	Yes	Maybe
Access to host filesystem	Yes	Yes	Yes	Yes**
Native support for GPU	Yes	No	No	No
Native support for InfiniBand	Yes	Yes	Yes	Yes
Native support for MPI	Yes	Yes	Yes	Yes
Works with all schedulers	Yes	No	Yes	No
Designed for general scientific use cases	Yes	Yes	No	No
Contained environment has correct perms	Yes	Yes	No	Yes
Containers are portable, unmodified by use	Yes	No	No	No
Trivial HPC install (one package, zero conf)	Yes	No	Yes	Yes
Admins can control and limit capabilities	Yes	Yes	No	No

Table 4.1: Comparison between different container implementations: Singularity [Kurtzer et al., 2017], Shifter [Canon et al., 2016], Charliecloud [Priedhorsky et al., 2017] and Docker [Merkel, 2014]. Taken from: [Kurtzer et al., 2017, Table 1].
*relies on Docker
**with security implications
***depends on upstream

in Section 8.2. A comparison by its authors with other implementations can be found in Table 4.1.



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES	■	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR	■	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS	■	■	■	2 MONTHS	2 WEEKS	1 DAY
	1 DAY	■	■	■	■	8 WEEKS	5 DAYS

— https://xkcd.com/1205/

II

Facilitating Collaborative Software Development in Science

Motivation & Outline 5

In Section 4.3, we discussed the current state of reproducibility in science. We restate the three core guidelines of [Krafczyk et al., 2021]:

P1. Provide transparency regarding how computational results are produced

P2. When writing and releasing research software, aim for ease of (re-)executability

P3. Make any code upon which the results rely as deterministic as possible.

This part presents approaches Electronic Vision(s)[1] employs to achieve these three goals. Over the course of this thesis, these approaches were substantially extended towards robustness and streamlining. Many mundane tasks that were necessary in day-to-day development operations have been automated. These include reducing the number of commands needed to set up a work environment[2] or reducing friction when working with several disjoint sets of features.[3]

The approaches presented here are applicable to any large-scale software-project that

- involves multiple developers working on distinct aspects of the codebase.
- uses several programming languages at different levels of abstraction.
- is expected to be usable in a productive way already during development.
- is worked on by people of varying skill levels from aspiring developers with a passion for clean and maintainable code to students that mainly want to get their project done but need a certain feature not implemented yet. Especially for contributions of the latter kind there should be a safe pathway into the codebase.

However, our focus is on neuromorphic hardware development, especially for BrainScaleS-2[4] (cf. Section 3.2). Neuromorphic mixed-signal hardware development is a joint effort of many disciplines. The core system components, analog neuron-emulating circuitry and their synaptic connectivity, are created by analog hardware designers. Digital hardware designers then have to interconnect these components and make sure that they can be configured and read out in a reliable and timely fashion. Low-level software layers like `sctrltp`[5] and `fisch`[6] are then tasked to provide logical abstractions of raw bits that flow in and out of the system, reducing mental load and increasing robustness by detecting erroneous access patterns. Intermediate-tier software like `stadls`[7]/`haldls`[8] is then needed

[1]Electronic Vision(s) Group at the Kirchhoff-Institute for Physics in Heidelberg

[2]Users enter distinct environments within containers, explained in Chapter 3 and Section 8.3.

[3]Our build tool is able to automatically merge non-conflicting sets of changes, see Section 7.3.2.

[4]BrainScaleS-2 Analog Neuromorphic Hardware System, [Schemmel et al., 2017; Schemmel et al., 2020]

[5]Slow ConTRoL Transport Protocol

[6]FPGA Instruction Set Compiler for HICANN

[7]STAteful encapsulation for HICANN-DLS

[8]Hardware Abstraction Layer for HICANN-DLS

to define and model the most primal access patterns to interact with hardware systems. This includes utilities "close-to-metal", for example `calix`[9] to ensure proper hardware calibration. These mid-tier abstractions already allow hardware-usage by experts that might not necessarily be hardware designers. Finally, high-level software frameworks focus on expressivity, these include `pynn.brainscales2`[10] and `hxtorch`.[11] They are more streamlined and easy to use, intended for both hardware-experts and non-hardware-experts alike to allow for a clean separation between model-intent and technical implementation. Each software layer requires different forms of expertise in software developers. All of these software abstractions are then used by modelers and theorists to investigate computational models for machine learning and neuroscience.

It should be noted that these lines are not strict, but rather blurry. After a core design has been established, hardware developers are encouraged to guide the implementation of low level abstractions[12] or even provide them themselves. Modelers are often interested in contributing to the API[13]-side of higher level software layers since they know which usage patterns are most convenient when specifying models. All of these contributions and interactions need to be taken into consideration. Once these layers are finished, intermediate and less maintainable collection of scripts[14] can be abandoned, mostly written to perform pioneering work and achieve results as soon as possible.

On all levels, verification as is *crucial* to ensure progress.

> *Every component is to be considered broken until verified*

is a mantra employed by most forms of TDD.[15] This includes both hardware *and* software. Especially software is often expected to "just work", with no regard for the amount of work involved to achieve this goal. This is especially important given the relatively early stage neuromorphic computing is in. A recent survey of neuromorphic computing and its application to hardware remarks that neuromorphic computers represent a new challenge for developing software frameworks to enable non-experts to effectively use them [Schuman et al., 2017]. At the time of writing, we have not yet reached said goal. This is evidenced by most recently published experiments on neuromorphic hardware: They are either performed by the designers themselves on conducted in close collaboration with them, for example [Esser et al., 2016; Kreiser et al., 2017; Albada et al., 2018; Kungl et al., 2019; Billaudelle et al., 2019; Göltz et al., 2021; Czischek et al., 2020; Stradmann et al., 2021]. Therefore, this work represents a crucial step towards a fully integrated collaborative platform that facilitates usage by non-experts.

[9]CALIbration Framework for HICANN-X, [Weis, 2020]

[10]PyNN-backend for BrainScaleS-2

[11]`PyTorch` for BrainScaleS-2, [Spilger et al., 2020]

[12]At Electronic Vision(s) these are called *containers*, not be to confused with software containers discussed in Sections 8.2 and 8.3.

[13]Application Programming Interface

[14]The infamous "blackbox", officially called `model-hx-strobe`, is an early Python library, written with the precise aim of interacting with BrainScaleS-2 prototype chips as soon as possible. It contains a lot of magic constants set by experts. Time has shown repeatedly that, once the original creators move on from such projects, they are essentially dead and not adaptable to new tasks.

[15]Test-Driven Development, [Beck, 2003]

Our overarching goal is to achieve similar software support for BrainScaleS[16] platforms as commercial GPGPUs,[17] such as TPUs.[18] Upon inception, they – similar to BrainScaleS – were only accessible to experts with hand-tailored code, but the availability of useful mid-level abstractions such as the CUDA[19] API allowed for integration into widely used machine learning frameworks such as `PyTorch`,[20] TensorFlow[21] or Theano.[22] Nowadays, end users can execute their models via Python[23] scripts and – for the most part – do not have to concern themselves with the underlying workings. Given the analog nature of the BrainScaleS platforms, our challenge is a more interesting one since we aim for ease of use while still enabling users to harness the advantages of analog and spike-based computing at the same time. This is a delicate balance to strike.

Outline

Chapter 6 starts by giving a broad overview over the different layers in the BrainScaleS-2 software stack in order to motivate why it is important to use sophisticated tooling during development. As long as experimenters reference which state of the software stack they used, we are halfway to fulfilling P1. Chapter 7 then focuses on how development within the software stack proceeds in a tractable manner, ensuring functionality is maintained despite adding and/or modifying features. This addresses P3, i.e., tractability. Chapter 8 explains how dependencies, i.e., the parts of software not under active development, are maintained to provide a robust and well-defined environment across machines for hardware experiments to run in and development to take place in. By updating in a snapshotted rolling release schedule, downtimes are kept to a minimum. In terms of [Krafczyk et al., 2021] conclusion, this addresses both P1 and P2, i.e., (re-)executability concerns and transparency. Chapter 9 explains how the ideas presented in Chapter 8 are extended to the existing Electronic Vision(s) cluster. Finally, Chapter 10 introduces `quiggeldy`, a micro-scheduler designed to increase experiment throughput on hardware while enforcing robustness by imposing encapsulating design-constraints on hardware experiments, and how it is integrated into the cluster. Chapter 11 ends Part II by summarizing outstanding challenges encountered during adoption of this deployment scheme and gives an outlook to how they can be tackled.

[16]BrainScaleS Mixed-Signal Accelerated Neuromorphic Systems, [Schemmel et al., 2008; Schemmel et al., 2010; Schemmel et al., 2017; Schemmel et al., 2020]
[17]General Purpose Graphical Processing Units
[18]Tensor Processing Units, [Jouppi et al., 2017; Coral, 2020]
[19]Compute Unified Device Architecture, [Nickolls et al., 2008]
[20]Python-Implementation of Lua-library `torch`, [Paszke et al., 2019]
[21]TensorFlow, [Abadi et al., 2015]
[22]Theano: A Python framework for fast computation of mathematical expressions, [Theano Development Team, 2016]
[23]Python Programming Language, [Rossum, 2000]

6 The BrainScaleS-2 Software-Stack: An Overview

Operating neuromorphic computing platforms holds many challenges in terms of precise system control, data pre-/post-processing as well as data exchange. Similar to other digital hardware platforms, software is *the* key component to make complex hardware systems accessible to users in a structured and reliable way [Kacher et al., 2020; Rueckauer et al., 2021]. Based on the experience of operating BrainScaleS-1[1] [Müller et al., 2020b], software layers for BrainScaleS-2 follow similar design principles [Müller et al., 2020a; Spilger et al., 2020]. The architectural overview is sketched in Figure 6.1.

Most of the repositories discussed in this chapter are publicly available at

`https://github.com/electronicvisions`

or upon request. Development happens via an internal code review service at

`https://gerrit.bioai.eu`

that is only accessible for group members and selected collaborators. It is discussed in Section 7.2.

All software contributions performed by the author over the course of this thesis are summarized in Appendix A.4 and detailed where relevant for concepts presented in this thesis (e.g., Section 10.5.2).

`pynn.brainscales2`: PyNN-backend for BrainScaleS-2 is the topmost layer of the BrainScaleS-2-software stack – and the primary API used by external users. It adapts BrainScaleS-2 to being accessed via PyNN in spiking mode. PyNN aims to unify descriptions of neuroscientific network models by providing a backend-agnostic API written in Python. Ideally, experimenters write their network description once and are then able to run their model on a plethora of different simulators as well as hardware backends by simply exchanging a single line of code. Of course, reality often gets in the way, and especially in the case of neuromorphic hardware there will be mismatch between the abstract biological model and its realization on hardware [Petrovici et al., 2014]. `pynn.brainscales2` therefore exposes additional neuron models that more closely resemble physical neurons as implemented on the chip. This way, we may lose the ability to move backends by

[1]BrainScaleS-1 Wafer-Scale Mixed-Signal Accelerated Neuromorphic System, [Schemmel et al., 2008; Schemmel et al., 2010]

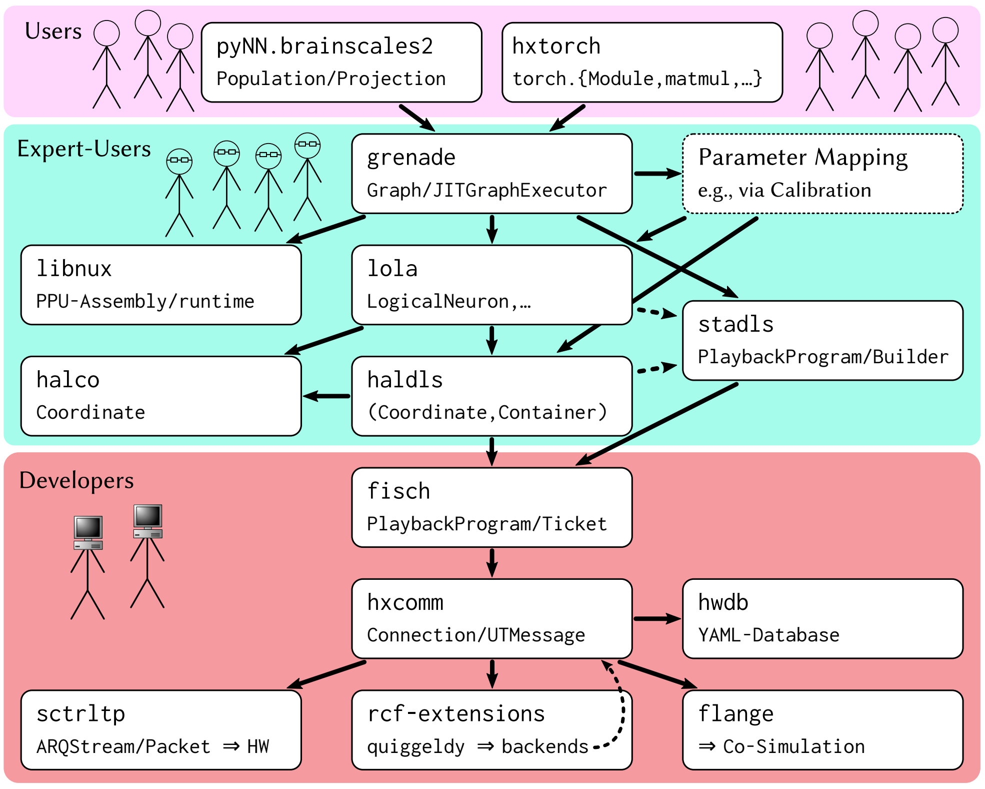


Figure 6.1: Overview of different levels of abstraction in the BrainScaleS-2 software stack and their targeted user group. Each layer is shown with its predominant form of data representation (non-exhaustive). End-users specify their experiment via a high-level API such as `pynn.brainscales2` or `hxtorch`. Using a form of parameter mapping (e.g., pre-defined calibrations obtained via `calix`), the abstract experiment specification is then translated to hardware parameters and scheduled via `grenade`. They are translated into `stadls`-programs that provide control flow for stateless `haldls`-instructions (possibly generated from logically more concise abstractions in `lola`). Overall, these Coordinate/Container-Pairs (separating *what* is written *where*) handle chip setup, experiment control and data readout. They are en-/decoded into `UTMessages` (i.e., FPGA words) via `fisch` and then sent to one of many possible backends via `hxcomm`, designated in the lowest level in the figure via ⇒. At the same time `libnux` provides runtime abstractions for code executed on PPUs and as well as simple experiment I/O via a so-called mailbox. Because of high-bandwidth requirements, communication to and from the controlling FPGA is handled by `sctrltp`, a highly optimized library that handles data transfer via direct network buffer access in a concurrent daemon process. Other targets include a SystemVeriolog-based co-simulation of hardware, accessed via `flange`, and `quiggeldy`, a micro-scheduling service developed during this thesis that can proxy any other backend. `quiggeldy` is discussed in detail in Chapter 10.

changing a single line of code, but still we achieve familiarity with core concepts of the API when users adapt experiments to different substrates. Hardware parameters can be set explicitly and in hardware units, whereas the standard PyNN models are specified in biological units. This allows expert-users to define and execute elaborate experiments in PyNN without resorting to the lower levels of the software stack that are more complicated to use and not as stable – if they are exposed in Python at all. Initial implementation was done in [Czierlinski, 2020]. In the future, automatic conversion from biological models to hardware parameters is planned with the help of calibration frameworks such as `calix`.

Implementation-wise, `pynn.brainscales2` employs `grenade`[2] to translate the provided network configuration into a set of `haldls`-containers encapsulated in a `stadls` `pbmem`[3] (see below).

`hxtorch`: `PyTorch` for BrainScaleS-2 [Spilger et al., 2020] is an extension for the machine learning framework `PyTorch`. It allows for BrainScaleS-2 to serve as computing backend for machine learning in a similar fashion to how `pynn.brainscales2` integrates with PyNN for neuroscientific simulations. BrainScaleS-2 provides a non-spiking operation mode that enables fast analog vector-matrix multiplications (cf. Section 3.2.2, [Stradmann et al., 2021]). These can be easily executed using `hxtorch`. Furthermore, `hxtorch` provides support for FPGA[4]-aided convolutions, corresponding software-based automatic differentiation techniques, to allow for hardware-in-the-loop training and automatic partitioning of neural networks onto one or more chips to execute arbitrary network sizes on the fixed-sized substrate in as few operations as possible. The latter is achieved via `grenade` (see below). It therefore allows for the integration of BrainScaleS-2 into arbitrary deep learning architectures [Weis et al., 2020]. Utilizing `grenade`, `hxtorch` also has support to perform spiking, non-spiking and hybrid-mode experiments on BrainScaleS-2. [Müller et al., 2021] will detail how hybrid-mode execution is facilitated.

`grenade`: GRaph-based Experiment Notation And Data-flow Execution allows for efficient use of limited hardware resources in both spiking and non-spiking operating mode. Necessary computations are tracked via a hardware-centric data flow dependency graph, similar graph-based computation techniques employed in other machine learning frameworks such as TensorFlow[5] or `PyTorch`.[6] Within the graph, vertices resemble statically configurable computation or hardware circuits, whereas edges represent analog as well as digital signal/data flow. This enables efficient partitioning of computations on limited hardware resources in a just-in-time fashion. Since there is at most implicit time-evolution in non-spiking operation mode, operations are free to be scheduled around to fit on the chip. The dependency graph allows us to execute operations as soon as its needed inputs are available. Pre- and post-processing of instruction/response streams of disjoint operations can happen in parallel hardware execution. The result is seamless batch support of arbitrary network sizes without need for manual placement of operations onto the chip. At the time of writing, due to the fact that BrainScaleS-2 currently being at the single-prototype-chip-level, no dedicated mapping layer[7] has been developed yet. In the future, we plan for `grenade` to handle mapping of networks to multiple prototype chips. It is implemented in [Spilger, 2021].

`calix`: CALIbration Framework for HICANN-X is a calibration framework written in Python during [Weis, 2020]. While not considered part of the core BrainScaleS-2 software-stack, it can be used to calibrate both neuron and synaptic hardware parameters

[2]GRaph-based Experiment Notation And Data-flow Execution
[3]PlayBack MEMory program
[4]Field-Programmable Gate Array
[5]TensorFlow, [Abadi et al., 2015]
[6]Python-Implementation of Lua-library `torch`, [Paszke et al., 2019]
[7]A BrainScaleS-2-counterpart to *marocco* [Jeltsch, 2014; Passenberg, 2019; Kaiser, 2020] for BrainScaleS-1.

of BrainScaleS-2. Both `hxtorch` and `pynn.brainscales2` support loading arbitrary low-level calibration routines, composed of low-level operations as explained below. For non-spiking vector-matrix multiplications, due to a lack of temporal dynamics, only a subset of parameters needs to be calibrated. Each neuron can receive its unique set of target parameters to calibrate. Calibration is done in-the-loop, i.e., settings are adjusted and measured on the host until convergence. Afterwards, a finished calibration can be saved to disc to load prior to experiment execution. `calix` makes use of `stadls` (see below) via Python-bindings and provides serialization via `cereal`.[8]

`stadls`: STAteful encapsulation for HICANN-DLS facilitates experiment control and experiment encapsulation. Experiment runs on hardware are encapsulated as sequence of execution-steps, called `pbmem`.[9] Each step corresponds to a CoCo,[10] a pair of `halco`[11]-coordinate and `lola`[12]/`haldls`-container[13] (see below), that is either written to or read from hardware. Instructions can be executed best-effort or after specific time delays, for example when reading out results after an experiment run. These time delays are specified in terms of FPGA-cycles. Post-run, users can extract result-data from the playback program via a ticketing mechanism that uniquely identifies read-requests. In particular, this means that users must decide on what to read out from hardware prior to executing their experiment. For performance and tractability reasons – strong typing allows catching misconfiguration early – it is written in C++,[14] but provides `genpybind`[15]-generated Python-bindings as well as serialization via `cereal`.

`lola`: LOgical LAyer implements high-level configuration containers that encapsulate practical logical entities on-chip. They allow for mentally more convenient configuration of hardware resources for expert-level users that need full control over hardware. A prime example is `AtomicNeuron` which holds all relevant CapMem[16]-cell settings to fully configure a single neuron circuit. Upon execution, `lola`-containers are translated to a set `haldls`-containers (see below). All parameters are specified in hardware-units (e.g., reset potentials are given in DAC[17]-values and not mV). Same as `stadls`, `lola` is written in C++ with `genpybind`-generated Python-bindings.

`haldls`: Hardware Abstraction Layer for HICANN-DLS gathers all low-level data structures for configuring BrainScaleS-2-hardware. Hardware configuration is modelled as a hierarchical set of containers. Only leaf-nodes in this tree-like structure contain actual configurable parameters, encapsulating smallest chunks of information that need to be written onto hardware as single instructions. Examples include single CapMem-cells,

[8]a C++11 Header-only Library for Serialization, [Grant et al., 2017]
[9]PlayBack MEMory program
[10]Coordinate/Container-Pair
[11]Hardware Abstraction Layer providing COordinates for BrainScaleS-1-based and BrainScaleS-2-based neuromorphic systems
[12]LOgical LAyer
[13]Please note that these containers are not to be confused with lightweight containers discussed in Section 8.2 to provide consistent software environments.
[14]C++ Programming Language, [ISO, 2017]
[15]Autogeneration of Python Bindings from Manually Annotated C++ Headers, [Klähn et al., 2020]
[16]Capacitive Memory
[17]Digital-to-Analog Converter

readout configuration or SynapseQuads (four adjacent synapses always configured, i.e., written, at the same time). Each leaf-container is a mapping from logical entity with a set of configuration states to fisch-Registers, i.e., another form of container that abstracts away atomically reading/writing to a register-like hardware location (see fisch below). This is much safer than letting users write arbitrary data to arbitrary addresses. Illegal configuration states can hence be detected without experiment execution. Additionally, haldls-containers are backend-aware, as not all parameters can be read back on actual hardware but only in co-simulation (a logic-level simulation of hardware, see flange[18] below). Adding a read-instruction for a parameter that cannot be read back on hardware will flag the pbmem and stadls will refuse to execute it on actual hardware. Same as stadls and lola, haldls is written in C++ with genpybind-generated Python-bindings and provides serialization via cereal.

libnux: Library to Interface with PPU Codenamed Nux is a support-library for writing programs executed on the PPU[19] during experiments. libnux provides limited C-runtime support, e.g., stack-based memory management, support for std::vector-processing by PPU's vector unit, limited fixed-point math-capabilities and convenience macros to execute these operations on the vector unit. Hardware-specific functionality such as sending spikes is wrapped in utility functions. Simple experiment I/O[20] can be performed via a so-called mailbox: a designated area of shared memory used for data exchange with the host-computer. Together with a patched version of gcc[21] and binutils,[22] vector registers of the PPU's can be addressed and used in assembly. It was initially developed during [Friedmann, 2013], but has been extended in [Heimbrecht, 2017; Spilger, 2018; Wunderlich et al., 2019].

halco: Hardware Abstraction Layer providing COordinates for BrainScaleS-1-based and BrainScaleS-2-based neuromorphic systems is a library implementing on-chip coordinates on both hardware generations (BrainScaleS-1, BrainScaleS-2). It is intimately aware of how many hardware resources exist and their relations to one another, both physically and logically. This allows for uniquely addressing all entities on hardware, such as PPUs, neurons, synapses, buses, repeaters, etc. Accessing a resource that does not exist – e.g., a neuron beyond the limits of the chip – leads to an error. There are convenience functions to enumerate all entities of a certain type as well as functions mapping to (physical) neighbors or (logical) parents. However, halco itself does not store information how to access these resources: haldls uses halco-coordintates to compute the actual address ranges for accessing the corresponding resource. Same as stadls, lola and haldls, halco is written in C++ with genpybind-generated Python-bindings.

fisch: FPGA Instruction Set Compiler for HICANN is a library concerned with translating fisch-pbmems emitted by stadls, i.e., streams of Registers, to a stream of

[18]Linking C++ Software Stacks with SystemVeriolog using DPI

[19]Plasticity Processing Unit

[20]Input/Output

[21]GNU Compiler Collection, https://gcc.gnu.org/onlinedocs/gcc-11.1.0/gcc/ (visited on 2021-05-02)

[22]GNU Binutils, [Free Software Foundation, 2020]

hxcomm[23]-UT[24]-messages [Karasenko, 2020]. A UT-message corresponds to a single FPGA-specific-instruction (e.g., read, write, wait_until). Typically, several UT-messages are necessary to implement functionality of a single fisch-Register. For example, JTAG[25] operations are performed by sending a separate UTMessage for the instruction and payload. fisch aims to abstract away these technical incongruences, thereby separating high-level haldls-hardware-description from its technical implementation. It also provides a ticket-based approach for reading observables from hardware. In particular, this allows for observables to be read at different points in time during execution in a deterministic manner. Hence, when instructing the pbmem to perform a readout, users are returned a ticket. After execution, each ticket can be used to uniquely identify the read-back observable's value at the corresponding experiment time. Same as stadls, lola, haldls and halco, fisch is written in C++ with genpybind-generated Python-bindings.

hxcomm: Low-Level Communication With HICANN-X via Hostarq is a header-only library concerned with implementing UT-messages and executing them on variety of possible Connections, i.e., backends. The implemented Connections include ARQConnection, an Ethernet-based connection to a controlling FPGA-board with attached BrainScaleS-2-hardware resources, SimConnection, a RCF[26]-based connection to a flange-based co-simulation that is simulating BrainScaleS-2-hardware in software (extensively used in verifying functionality of new BrainScaleS-2-chips during design [Grübl et al., 2020]), and QuiggeldyConnection, a proxy-connection to allow rapid experiment execution that is explained in detail in Chapter 10. Each connection might apply further, more efficient data-packing techniques to improve performance [Karasenko, 2020, Chapter 5]. hxcomm is able to decide on which backend to use at runtime based on process-environment. Furthermore, timing statistics of message-execution as well as debug and failure state information for each Connection is provided. Same as stadls, lola, haldls and halco, fisch is written in C++ with genpybind-generated Python-bindings.

sctrltp: Slow ConTRoL Transport Protocol is a highly optimized library that handles data transfer for hxcomm's ARQConnection via Ethernet to the controlling FPGA-board. In order to achieve a high experiment rate, high-throughput bandwidth is needed. This is especially true for neuromorphic hardware systems operating at a relative speed-up factor of 1000, such as BrainScaleS-2. Otherwise, the majority of operation time could be spent on transferring configuration data to and from the substrate [Albada et al., 2018, Fig. 9]. This is achieved by handling queueing and retrieving data from buffers in parallel and passing around pointers instead of copying data (zero-copy policy). Actual transmission happens via sliding-windows in a concurrent daemon process with shared memory that is separated from other executed user-code. For performance reasons, sctrltp was initially written in C[27] [Schilling, 2010] and has since been adapted for C++ with pybind11[28]-based

[23]Low-Level Communication With HICANN-X via Hostarq
[24]Universal Translator
[25]Joint Test Action Group industry standard for verifying designs and testing printed circuit boards after manufacture, [IEEE, 2001]
[26]Remote Call Framework – a cross-platform interprocess communication framework for C++, [Delta V Software, 2020]
[27]C Programming Language, [ISO, 2018]
[28]Seamless operability between C++11 and Python, [Jakob et al., 2019]

Python-bindings.

`rcf-extensions/lib-rcf`: Library Wrapping and Extending RCF is used to facilitate interprocess communication across the network. rcf allows for the definition of communication interfaces directly in C++ without an intermediary IDL.[29] It supports both synchronous as well as asynchronous system calls and is primarily used in `flange` and `hxcomm` (see below and Chapter 10). Especially to facilitate the latter, `lib-rcf` contains a series of generic extensions built on top of RCF, uninspiredly named `rcf-extensions`. Examples include fast micro-schedulers and utilities to upload data on demand. Through the use of template metaprogramming, they can be adapted to any specific use-cases. Python bindings can be provided by whatever library is making use of these extensions.

`flange`: Linking C++ Software Stacks with SystemVeriolog using DPI enables us to use a simulation of future or current hardware chips as backend for the software stack. Instead of sending the stream of UT-messages via `sctrltp` to a physical FPGA-controlled backend, it is redirected in `flange` via RCF to a SystemVeriolog[30]-based simulation of the chip and injected via SystemVeriolog DPI.[31] This allows verification of both components in lock-step. The software-stack can be adapted to hardware differences while core functionality of the hardware can be verified without adapting tests for co-simulation. `flange` was primarily developed by Philipp Spilger. Resulting both in a more advanced software-stack at tape-out and more robust hardware platforms in general, this design philosophy was extensively used in [Grübl et al., 2020].

`hwdb`: Electronic Vision(s) Hardware Database is the single source of truth regarding hardware configuration data. It stores information about IP addresses, ports, ADC[32] configuration and hardware chip versions, for both BrainScaleS-1 and BrainScaleS-2 hardware components. Slurm[33] makes extensive use of its information to schedule user jobs (cf. Section 9.1.2). At its heart, it contains a YAML[34]-database that can be accessed via APIs for C and C++. The latter is wrapped to Python.

[29]Interface Definition Language

[30]SystemVerilog Programming Language, [IEEE, 2018]

[31]Direct Programming Interface

[32]Analog-to-Digital Converter

[33]Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management, [Yoo et al., 2003]

[34]YAML Ain't Markup Language, `https://yaml.org/spec/1.2/spec.html` (visited on 2021-04-08)

7 Workflow: Continuous Integration

As discussed in Section 4.4, one aspect of successful software projects is to ensure that their requirements are met and upheld in a maintainable way. A method to achieve this is CI: Continuous Integration [Kaiser et al., 1989; Booch, 1990]. As the name suggests, its main idea is to continually integrate and verify all changes done to a single repository against the remainder of the code base. Also in science it is of increased interest [Krafczyk et al., 2019].

The basic workflow is sketched in Figure 7.1: Modifications to the software stack (cf. Figure 6.1) are tracked on a per-repository level in Gerrit (see Section 7.2). Repositories include software layers from compute stacks, hardware RTL[1] code and publications such as papers and personal theses. Developers can propose changes that are then reviewed and commented on by others. In order to ensure that these changes leave the repository (and thereby the whole software-stack) in a working state, each change is automatically verified via Jenkins (see Section 7.1). In case adjustments are requested by either human or automatic verification, the change can be updated and re-evaluated. Changes can only be submitted to the repository if they are both approved by at least one other developer as well as verified in a well-defined execution environment that is distinctly *not* the developer's environment. This leads to an overall increase of code quality as functionality is maintained by testing and comprehensibility by manual review. The chance for *quick fixes*, i.e., code changes done under time pressure with insufficient validation,[2] making it into the code base is severally diminished.

7.1 Software Build Automation: Jenkins

Jenkins [Armenise, 2015; Kawaguchi et al., 2011] is an open source automation server written in Java.[3] It allows for the execution of arbitrary user-defined CI-jobs. The contents of each job are defined in a so-called `Jenkinsfile` in Apache Groovy [Strachan et al., 2020], a dynamically typed Java-dialect supporting closures akin to a scripting language. While `Jenkinsfiles` can be managed in Jenkins directly, they are usually integrated into the corresponding repositories. A job is organized into a series of build steps. Steps can be

[1]Register-Transfer Level

[2]The author has been guilty of this as well, pushing a change concerning how `waf` determined Gerrit credentials without waiting for *all* verification jobs to complete. Unfortunately, hardware-related verifications differed ever so slightly in their settings, leading to all of them failing over night. Related change: `https://gerrit.bioai.eu/c/waf/+/3353` (visited on 2020-04-18).

[3]Java programming language

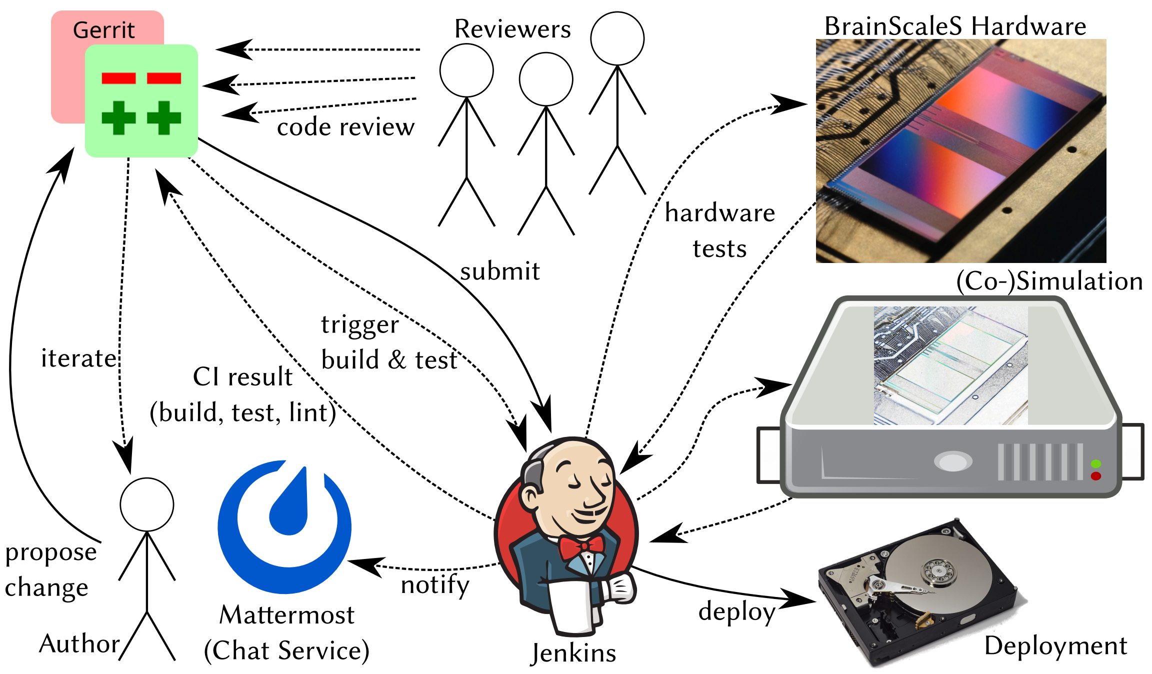


Figure 7.1: Overview of the general software development workflow at Electronic Vision(s): A developer proposes changes to one of the software repositories via Gerrit. Others then review and comment the code. Automatic verification is performed via Jenkins, which schedules and executes all software tests that depend on code in the repository in question. In case of failures, notifications can be sent to specific channels in our group-internal Mattermost chat service. This is mostly used for nightly tests verifying long-term stability that are not supposed to fail. Software tests include both hand-written unit-tests verifying functionality as well as static code analysis validating coding standards (e.g., mandatory documentation in some places or code formatting). Additionally, changes in hardware design repositories are verified via co-simulation while changes in "regular" software repositories are verified on existing hardware. This leads to an iterative process in which comments or test failures call for adjustments to the originally proposed change which need to be re-verified by both human and machine. Only if the change is finally approved by other developers as well as verified in testing can it be merged into the codebase.
Figure extended from Eric Müller. Logos of Gerrit, Jenkins and Mattermost taken from: Wikimedia Commons, `https://handbook.mattermost.com`.

run in sequence or in parallel if possible, e.g., when performing disjoint tests. Each might be executed in parallel on a variety of executors (also called "agents"). There are a number of plugins adding support for various source control systems and execution backends (examples include execution of agents via `ssh`,[4] integration with Gerrit or `jenlib` below). Furthermore, Jenkins provides a REST[5] API for automated interactions.

Jobs can be triggered in several ways: The most common trigger is a new or updated change in Gerrit, which causes building of the affected repository as well as all repositories that depend on it. But also time-based triggers are possible, rebuilding repositories every night and executing extended hardware tests to detect regressions. These typically represent more involved tests that are too time-consuming to run for each individual change, but – at least in theory – cover the same aspects. Typical examples are completed experiments

[4]Secure SHell, [RFC4250]
[5]REpresentational State Transfer

that are verified with regards to *functional* criteria. It is possible to have one build job trigger other others. Of course, these tests can always be triggered manually for more involved software changes, they are simply *not required* for every benign change. Each build job reports a status that is one of the following[6]:

`SUCCESS`	Relevant software was built and all verification steps executed successfully.
`FAILURE`	Building relevant software failed to build or crucial functionality could not be verified. Additional work on the change is required.
`UNSTABLE`	While relevant software was indeed built successfully, some test cases, static code analysis checks or dependent build jobs failed to verify. It is up to the job configuration to draw the line between `UNSTABLE` and `FAILURE`. Generally, build jobs are required to pass with `SUCCESS` for a Gerrit change to be eligible for submission.

At the time of writing, there are 225 active Jenkins jobs defined, about 50 of which are "nightlies". Per day, there are about 26–30 nightlies run every night. Their runtime is between 10 s and almost 5 h, whereas the median is 277 s. 68.3 % of all jobs finish under 10 min.[7]

7.1.1 Jenkins in HPC-environments: `jenlib`

Jenlib – Shared Library for Visionary Jenkins Pipelines [Stradmann, 2019] is a plugin for Jenkins. Among other things, it provides interoperability with HPC[8] systems, in particular Electronic Vision(s)'s cluster system via Slurm (see Section 9.1). Parts of build jobs can be encapsulated by `onSlurmResource`. This causes a dynamic agent to be created and scheduled to run on the cluster with requested resources (e.g., a certain partition or number of CPU[9] cores). The build job is paused until the cluster job is ready in order to not consume compute resources while waiting. Once the dynamic agent is scheduled by Slurm, it registers itself with Jenkins and the build job can progress. After the build job finishes execution, the Slurm job is terminated[10] and the agent automatically deleted. Similarly, `inSingularity` is provided to either execute parts of build jobs in Singularity[11] containers (see Section 8.2), while `wafDefaultPipeline` can be used to build and test a software repository according to our default build tool `waf`[12] (see Section 7.3).

By default, Jenkins assumes disjoint build nodes with their network storage. Users are expected to explicitly mark files that should be preserved between build steps. Exclusive access to a directory is only guaranteed while the current agent is allocated. However, the

[6]There are other states, such as `ABORTED`/`NOT_BUILT`, but they are for book-keeping purposes only.
[7]Estimated from Slurm statistics between March and May 2021.
[8]High-Performance Computing
[9]Central Processing Unit
[10]via `scancel`
[11]Singularity Container, [Kurtzer et al., 2017]
[12]Waf: the meta build system, [Nagy, 2005]

Electronic Vision(s)-cluster features an NFS[13]-based file storage. This means that multiple agents could execute in the same working directory, thereby potentially overwriting each other's files in a non-deterministic manner. `jenlib` solves this problem by assigning a unique working directory for each build job instead of each agent. This allows multiple agents to work on the same directory, allowing for complex work-flows: Compiling on powerful multi-core nodes with a lot of storage and then using less-powerful compute nodes to interface with neuromorphic hardware that feature better Ethernet-capabilities. Unused workspaces are purged in regular intervals.

Finally, `jenlib` provides support to integrate with Electronic Vision(s) primary form of in-group communication: Mattermost[14] (cf. Figure 7.1). Failing nightly jobs, indicating a regression in supposed-to-be-working code, cause a notification to be printed to the corresponding appropriate channels so that an investigation is triggered.

7.2 Code Review: Gerrit

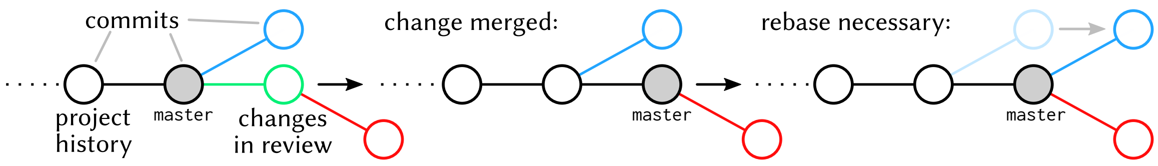


Figure 7.2: Gerrit workflow.
Left: In a given repository (black circles), users add their proposed changes as commits (colored) on top the current stable history (`master`-branch). Changes can be stacked on top of each other (red on green). The changes then get reviewed and automatically built via Jenkins (see Section 7.1). While in review, commits can be modified in any way, as long as the `Change-Id` in their commit message stays constant.
Middle: After a successful merge, the `master`-branch now points to the new latest change.
Bottom: All changes that do not yet include the newest additions in their history need to be rebased (blue). This ensures a single consistent repository history.

Gerrit Code Review [Harris, 2020] is the collaborative code development tool used by Electronic Vision(s). It started out as a fork of Rietveld,[15] exchanging `svn`[16] for `git`,[17] but was completely rewritten in version `2.0`.[18] The workflow is visualized in Figure 7.2. Changesets to repositories are created as new commits in the corresponding repository and submitted by pushing to a special branch `refs/for/<branch>` where `<branch>` is the target development branch. For each repository, Gerrit is able to track several branches, but at Electronic Vision(s) we usually have only one, typically named `master`. Once uploaded, Gerrit notifies Jenkins which triggers build jobs for the current repository and

[13]Network File System, [RFC8881]
[14]Mattermost, [Mattermost2015]
[15]Rietveld Code Review Tool, [Rossum, 2008]
[16]Apache Subversion, [CollabNet, Inc. et al., 2000]
[17]Git – a distributed version-control system for tracking changes in source code during software development, see Section 4.4.1, [Torvalds et al., 2005]
[18]Both Rietveld and Gerrit are named after Dutch designer Gerrit Rietveld.

Repository	# of Changes	$\frac{\sum \text{top}<N>}{\text{total}}$	# of Patchsets per Change		Insertions + Deletions per Change	
			Mean	Median	Mean	Median
haldls	844	6.7 %	9.8	5	636.1	45
spack	732	12.5 %	4.8	3	81.1	7
hicann-dls-private	575	17.0 %	5.1	3	6464.0	54
marocco	555	21.4 %	6.5	3	1254.9	31
sthal	521	25.6 %	5.3	3	608.7	21
halbe	448	29.1 %	4.7	3	512.2	18
yashchiki	428	32.5 %	7.8	4	32.9	6
halco	415	35.8 %	5.9	4	808.7	32
cake	389	38.9 %	4.1	2	143.1	13
hicann-system	305	41.3 %	4.0	2	403.8	22
grenade	292	43.6 %	17.7	12	181.5	66
hxcomm	273	45.8 %	10.2	4	133.3	28
hxfpga	272	47.9 %	6.3	3	280.0	20
hmf-fpga	265	50.0 %	4.6	2	9225.7	36
waf	261	52.1 %	3.9	3	52.9	8
jenlib	244	54.0 %	5.3	2	62.8	14
model-hw-hdbioai	235	55.9 %	8.2	6	150.4	41
fisch	233	57.7 %	8.2	4	129.0	32
model-hw-hxsampling	231	59.6 %	5.6	3	201.3	20
libnux	228	61.4 %	6.1	4	175.1	38
vision-bibtex	224	63.2 %	2.0	2	19.7	10
doc-visionshome	224	64.9 %	2.2	2	50.3	12
sctrltp	223	66.7 %	4.6	2	89.2	22
sw-macu	181	68.1 %	2.9	2	351.1	27
frickel-dls	177	69.5 %	3.7	2	312.3	27

Table 7.1: Gerrit statistics: number of changes in the most-active repositories, number of patchsets (iterations) per change and number of insertions/deletions. $\sum \text{top} < N > / \text{total}$ denotes the fraction of changes in Gerrit accounted for by the top N repositories. Large variance in changed lines per change – shown by the difference between mean and median – are due to changes involving large binary files or adjustments to code-linting. The 25 most active repositories account for almost 70 % of all changes. Data snapshot is from May 2021.

all repositories depending on it. Furthermore, same as Jenkins, Gerrit provides a REST API for automated interactions.

Other developers then review and comment the code while Jenkins performs automated tests. Developers critique the sensibility of the presented approach and its implementation whereas Jenkins verifies correctness. In our development process at Electronic Vision(s), changesets are scored by reviewers voting from -1 to +2. A score of +2 indicates approval for submission, +1 general approval with some minor changes requested (or the reviewer does not feel confident enough to approve the changeset), 0 is used for general comments or remarks without a full review and `-1` expresses serious concerns with the changeset (that need to be re-evaluated by this particular reviewer once addressed). Additionally, there is a hard-blocking "nucular"[sic][19] `-1`-voting option to indicate that the changeset is ill-fated and should never be merged. Jenkins also scores changesets with `-1` (`FAILURE`), `0` (`UNSTABLE`) or `+1` (`SUCCESS`). As noted above, a change can only be submitted if at least one other developer voted `+2`, it has no `-1` votes (they are "blocking") and Jenkins voted with `+1` as well.

In case of emergencies, administrators are able to override Jenkins votes, but these instances are few and far between.

If modifications are requested from reviewers or failing tests, the changeset can be adjusted by amending the corresponding commit. Gerrit associates commits with changesets via a special line at the bottom of the commit message starting with "`Change-Id: I`" followed by a forty digit hexadecimal number. `Change-Id`s can be inserted manually or, more conveniently, via `git`-hooks prior to submission for review. This allows for all aspects of a commit to be modified, including author, commit-order, afflicted files and commit message, as long as the `Change-Id` is not altered. Different iterations of a changeset are called patchsets. After a successful review, a changeset is "submitted" and the commit associated with the latest patchset becomes part of the immutable repository-history and thus cannot be modified anymore. All changes, in terms of `git`-terminology, need to be rebased on top of the current `HEAD`-commit of the branch in order to be submittable. This "single-history" workflow is distinctly different from branch-based workflows,[20] where a sequence of commits is developed in isolation and then merged back into the main branch via merge-commits. While merge-commits are supported in Gerrit, they are mainly used to integrate external commits into the codebase that need to no review. One example for this is upstream code that is reintegrated into a local fork, done, for example, in Spack[21] (see Section 8.1).

Statistics At the time of writing, we have about 150 code-related projects with 1–850 changes. Table 7.1 shows some statistics about changes as of May 2021. We see that the distribution is heavily skewed towards the core repositories, indicated by the fact that 14 out of 148 repositories account for more than 50 % of all changes tracked in Gerrit. Over the last six years at Electronic Vision(s), we have merged or are working on more than 12 500 Gerrit-tracked changes. This demonstrates that Gerrit is actively used and an essential part of day-to-day work.

7.2.1 Inter-repository dependencies: `Depends-On` and topics

Sometimes, changesets have inter-repository dependencies on one another. These can be expressed via another form of special commit message line: `Depends-On:`-statements. The

[19] A reference to Homer Simpson's mispronunciation of "nuclear" in "The Simpsons": season 9, episode 19.
[20] `https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows` (visited on 2020-04-18)
[21] Supercomputing PACKage manager, [Gamblin et al., 2015]

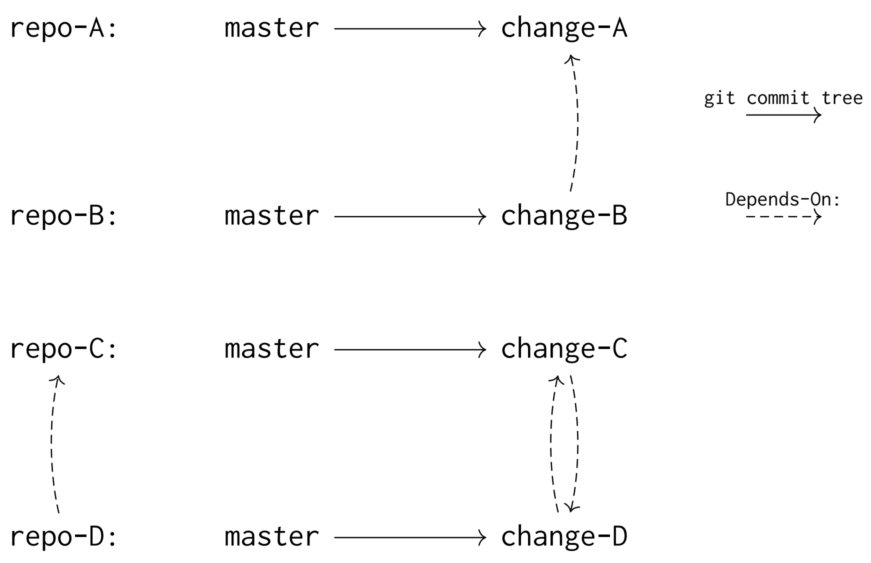


Figure 7.3: Examples for inter-repository dependencies mapped via `Depends-On`.
Top: Changeset `change-A` in repository `repo-A` introduces a new feature that changeset `change-B` in repository `repo-B` wants to make use of it. Hence, `change-B` can only be submitted once `change-A` is submitted. Furthermore, `repo-B` can only be successfully tested if `repo-A` is on changeset `change-A` during testing of `change-B`.
Bottom: The API repository of `repo-C` is reworked in changeset `change-C`. Repository `repo-D` depends on repository `repo-C`. Submitting `change-C` on its own would leave `repo-D` in a non-working state because it would fail to build. This is against Electronic Vision(s)-policy which mandates that `HEAD`-commits of all repositories should always be in a usable (i.e., buildable) state. Repository `repo-D` therefore needs to be updated to the changes of changeset `change-C` in another changeset `change-D`. Both `change-C` and `change-D` need to be submitted in lock-step. See the text for further details.

examples presented in Figure 7.3 could be expressed as indicated by the arrows: `change-B` would have `Depends-On: change-A` because it would fail to build otherwise. In the second example, `change-C` and `change-D` would have `Depends-On:` lines for each other in their respective commit messages. As they need to be submitted together, they should also be put into the same topic.

Gerrit is then able to extract dependency information and prevents submission until all dependency-changesets are submitted or submittable. For technical reasons[22] Gerrit cannot reliably detect if all dependencies are submittable (unless all changes are in the same topic; see below). Hence, the developer has to check manually and indicate this via a `Dependency-Beer-Promise`-vote.[23] `waf` (explained shortly below) was extended to read out `Depends-On:`-information and therefore allow for testing of all changesets with their respective dependencies, resulting in meaningful Jenkins-votes (see Section 7.3).

Concurrent submission of several changesets across multiple repositories (required for the second scenario) is achieved via topics. Each changeset can be assigned to at most one

[22]The exact submission requirements vary slightly on a per-repository basis.
[23]The name originates from a tongue-in-cheek fee of one beverage crate for breaking `HEAD` by ignoring dependency constraints.

topic. All changesets belonging to the same topic can only be submitted together, i.e., they *all* need to have `+2`/`+1`/no `-1` votes.

7.3 Building the full Stack: `waf`

`waf`[24] is the main build tool utilized at Electronic Vision(s). Written in Python, it is a meta build-system, i.e., in contrast to single language build tools such as `python-setuptools`[25] for Python or `cargo`[26] for Rust[27] it has support for a variety of languages such as such as C, C++, Fortran,[28] or Java and can be easily extended to add support for more. At its core, `waf` only tracks a graph of interdependent `Task`s. Each `Task` has a set of sources, i.e., input files, typically one target, i.e., output file, and a rule governing how the target is produced from all sources. The target of one `Task` may then act as source of several others. On subsequent calls, `waf` is able to determine which targets actually need to be rebuilt by computing and comparing the hash of each source.

As per usual, a `waf` build is split up into distinct phases, such as `configure` (ensuring all dependencies are present), `build` (translating high-level code to libraries/binaries) and `install` (copying files into their expected locations and executing tests). The build-flow is configured via a set of `wscript`-files that contain simple Python code. In each phase, so-called `waf-Tools` can be loaded that provide additional functionality, e.g., checking for dependencies, building binaries/shared libraries in a C-context or running tests with a certain framework.

7.3.1 Multi-Repository Builds via `setup`-Command

At Electronic Vision(s), a local fork of `waf`, deemed `symwaf2ic`, is maintained that periodically contributes back upstream. One of `symwaf2ic`'s main features is the `setup`-command.[29] It allows for the expression of inter-repository dependencies that are needed when building, for example, the BrainScaleS-2 software stack (see Chapter 6). Each repository's `wscripts` can be extended to include their dependencies on other repositories:

```python
def depends(dep):
    dep("repo-A")
    dep("repo-B", branch="special-branch")
    dep("repo-C", ref="refs/tags/special-tag")
```

Repositories can depend on special branches or a specific `git` reference (such as a tag) of other repositories. From this information, `waf` is able to build another dependency graph,

[24]Waf: the meta build system, [Nagy, 2005]
[25]Python Setuptools, [PyPA2006]
[26]`cargo`: A Package Manager for Rust, [Katz et al., 2014]
[27]Rust Programming Language, [Matsakis et al., 2014]
[28]Fortran Programming Language, [ISO/IEC, 2018]
[29]The `setup`-command was implemented by the author far prior to this thesis.

this time between repositories. All build-phases are then executed in dependency-first order, i.e., each repository can access information and reference defined Tasks from its dependencies. The whole stack of repositories is treated as one big build project.

Furthermore, the workflow when using waf changes slightly: Instead of checking out the repository of interest and then invoking waf within it, the build process is shifted to a sandbox in which all involved repositories are checked out and built side-by-side. The user navigates to a suitable folder and executes

```
$ waf setup --project <project>
```

where <project> corresponds to the top-level repository, i.e., the repository of interest. Several top-level repositories can be specified. Using a project database,[30] waf is able to map repository names to URLs.[31] Starting with the top-level repositories, they are checked out and their wscripts scanned for dependencies. Missing dependencies are then processed in an iterative scheme and checked out as well. The user does not need to manually perform any checkout. waf was extended to automatically parse Depends-On:-lines from commits after checkout and retrieve the dependency-commits from changesets in Gerrit. Additionally, the user is able to specify --gerrit-changesets at the command line to incorporate Gerrit changesets into the sandbox. Overall, this is especially useful for Jenkins-builds (see Section 7.1).

7.3.2 Contributions

During this thesis, waf was extended in various ways, including several minor bugfixes, refactoring, quality-of-life changes and feature additions (see Appendix A.4).

Dependency-Resolution of disjoint Changeset-Stacks was vastly improved, after first refactoring the existing codebase. Originally, the resolution algorithm worked as follows: Starting with any specified --gerrit-changes, waf would recursively scan commit-messages for additional dependency statements and organize all dependencies in a list per repository. For each repository, each entry in the dependency list would be applied by either checking out the given commit – in case the repository was at master in the current sandbox – or cherry-picking it on top the current HEAD. As outlined in Figure 7.4, this lead to problems as soon as dependency chains got slightly complicated. Working with disjoint stacks of changes was even more time consuming. Because another stack of changesets could only be included if each entry was manually tracked via Depends-On:, features were developed in isolation and not tested together early. If features did depend on each other while still being under active development, developers would oftentimes work with snapshots that were manually updated periodically because Gerrit's support for changesets belonging to multiple developers is lacking. There was hardly any testing

[30] At the time of writing, the public project database of Electronic Vision(s) is available at: https://github.com/electronicvisions/projects

[31] Uniform Resource Locators

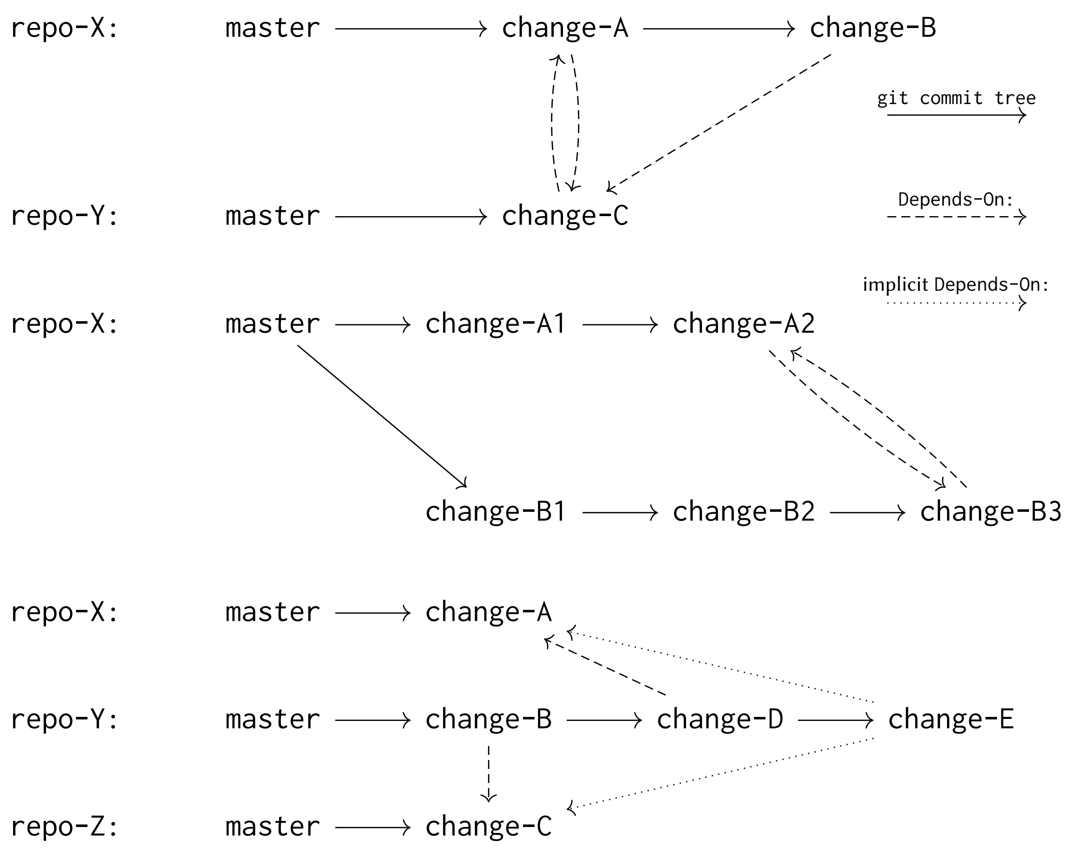


Figure 7.4: Examples of dependency relations that lead to errors, but are supported now. Solid lines indicate parent-relations in the `git`-commit-tree, whereas dashed lines indicate a `Depends-On:`-relation.
Top: Two repositories contain changesets that are developed in lock-step and depend on each other (`change-A` and `change-C`). Then, there is a new changeset, `change-B`, that builds on top the previous two. Because `change-A` is a direct parent of `change-A`, its functionality is available when checking out `change-B`. `change-C` is included via a `Depends-On:`-line, but then includes `change-A` as well. As the old implementation relied on cherry-picking whenever `HEAD` was not `master`, this led to an error because `change-A` was applied twice. The solution is to cherry-pick conditionally, only if the `Change-Id:` of the to-be-applied change is not yet present in the commit tree.
Middle: Two developers work on distinct stacks of changes in the same repository that should be tested together (e.g., one is a new feature and the other one an adjustment for a new FPGA-bitfile only needed for some testing setups). Previously, `change-A2` would need to depend on all three changes `change-B{1,2,3}`, whereas `change-B3` would need to depend on both `change-A{1,2}`. This was especially tedious if one stack was forced to introduce new commits deeper in the tree as they had to be tracked by the other stack manually. The same applied if a changeset in another repository wanted to incorporate both stacks: All changes had to be manually tracked – in the correct order. Now, as long as stacks are disjoint, it is enough to depend on the latest change of the other stack. Dependencies are then resolved automatically by walking up the tree of changesets.
Bottom: Transitive dependencies in a stack of changes. Previously, if parent changes depended on specific changes in other repositories, they had to be manually tracked in newer changes. In this example, `change-E` would fail to build if it did not have explicit `Depends-On:` lines on changesets `change-{A,C}`, even though the dependency is implicit (indicated by the dotted lines). See text for more details.

of disjoint[32] changeset stacks in Jenkins.

waf retrieves information about Gerrit changesets via an ssh-based interface,[33] that is then parsed from JSON[34] into plain Python dictionaries. Previously, only information such as commit hashes was directly extracted. By now wrapping these dictionaries in Python-objects with a streamlined API, more advanced options become available: Instead of just including dependencies that were explicitly added (and thereby forcing developers to include all changes from another stack, see middle in Figure 7.4), symwaf2ic's default was changed[35] to now treat parent changesets as explicit dependencies. By "walking up" the commit tree until master is reached, all dependencies of parent changes are found and taken into account (cf. bottom in Figure 7.4). The same is true for dependencies, allowing to incorporate other stacks with a single Depends-On: (cf. middle in Figure 7.4). Furthermore, the list of changes for each repository in the sandbox is reordered using ancestral information: Parents are applied before their children. Thereby, the need to specify Depends-On:-lines in a specific order is eliminated, minimizing the chance for errors during cherry-picking.

Due to technical limitations the final repository-checkout procedure is conducted via a single command line call[36] and hence waf was adapted to generate a simple bash[37]-script that checks for each changeset if the given Change-Id: is already present in the current commit tree and only performs the cherry-pick if it is not.

Improving repos-update command is a direct result of changes mentioned above. waf repos-update can be issued to update all repositories in a given sandbox to include all submitted changes, i.e., synchronize master. Previously, these updates would only work for repositories in the sandbox without checked out changesets. All others had to be manually updated. Switching to rebasing repositories alleviated the problem. Additionally, the same logic performing the initial checkout mentioned above also ensures that changesets that get merged but are still checked out out locally will be dropped on repos-update once their Change-Ids are found in the commit tree.

Application: Straight-forward Deployment of tagged Changes Overall, many potential time sinks have been eliminated from waf to streamline the development process. In particular, during the preparation of a hands-on tutorial for NICE 2021,[38] it allowed for straight-forward module[39] deployment during development: Developers could tag any change to be included with hashtag:nice2021 and an automatic Jenkins job would deploy a software module containing these changes. Via the contributions discussed above,

[32]Stacks are disjoint if their commits do not modify the same line in a single file so they can be freely cherry-picked on top of each other in any order.

[33]https://gerrit-review.googlesource.com/Documentation/cmd-index.html

[34]JavaScript Object Notation, [RFC8259]

[35]The original behavior can be enabled explicitly by adding a No-Parent-Depends-On-line to the commti message.

[36]Repositories are managed via myrepos: https://myrepos.branchable.com/ (visited 2021-04-18)

[37]The Bourne-Again SHell, https://www.gnu.org/software/bash/ (visited on 2020-12-15), [Fox, 1988]

[38]Neuro-Inspired Computational Elements Conference 2021

[39]Environment modules are discussed in Section 4.4.2.

```groovy
@Library("jenlib") _

try {
  withCcache() {
    wafDefaultPipeline(
      projects: ["hxtorch", "pynn-brainscales", "calix"],
      container: [app: "dls"],
      moduleOptions: [modules: ["ppu-toolchain"]],
      setupOptions: "--gerrit-changes='hashtag:nice2021'",
      notificationChannel: "#jenkins-trashbin")  // success checked
       ↪  globally
  }

  conditionalStage(name: "Module Deployment",
  skip: !params.DEPLOY_MODULE) {
    runOnSlave(label: "frontend") {
      inSingularity(app: "dls") {
        deployModule([name  : "bss2-stack-for-nice",
        source: "bin/ lib/ repos_log.txt"])
      }
    }
  }
  } catch (Throwable t) {
  notifyFailure(mattermostChannel: "#dls-software")
  throw t
}

if (currentBuild.currentResult != "SUCCESS") {
  notifyFailure(mattermostChannel: "#1nicedemo")
}
```

Listing 1: Complete `Jenkinsfile` needed to build an environment module from Gerrit hashtag. We just have to specify which projects we want to build in which container application (by default the latest container is chosen) with what modules of cluster-wide deployed software. Additionally, we merely supply the given Gerrit `hashtag:nice2021` to `symwaf2ic`'s setup call. Everything else works "automagically". Afterwards, we make the built software available via module. In case of errors, we immediately send a notification to the corresponding Mattermost channel so that it is investigated immediately.
This is the culmination of work that went into `waf` and `jenlib`, introducing useful abstractions to make deployment more straightforward. The `Jenkinsfile` was set up by Yannik Stradmann, who is also the main author of `jenlib`, described at Section 7.1.1.

`waf` is able to integrate all tagged changes, plus the changes they explicitly depend on *and* their parents (which they *implicitly* depend on). This allowed different developers to work on different parts of the code, in particular `quiggeldy` deployment, calibration and demo-creation, while verifying all changes *in unison* without every developer constantly integrating other developers' changes *all the time*. During stress-test and the hands-on tutorial itself, user code would automatically load the generated module. All changes needed to run the hands-on tutorial were immediately available. To the end user, *it simply worked*. The `Jenkinsfile` to achieve all this is so small that we can print it in this thesis in its entirety, see Listing 1. This is the culmination of the work that went into `waf` and `jenlib`, introducing useful abstractions to make deployment more straightforward. For more details, see Section 10.7.2.

8 Managing and Deploying an Evolving Set of Software Dependencies

As we discussed in Section 4.2, the requirements for performing software-aided science are numerous. Accordingly, as described in Chapter 6, the overall level of software engineering at Electronic Vision(s) is rather sophisticated. This naturally results in a large quantity of software dependencies that needed to be maintained. In "the old days", it took new members joining the group an increasing amount of time to be brought up to speed in terms of setting up their development environment. Most dependencies installed at the system level – such as compiler the widely used `boost` libraries[1] – were kept at a fixed version in order not to break existing installations by updating. If there was a good enough reason to perform a particular update, e.g., because of a newly introduced feature, it had to be announced well in advance and, once performed, typically resulted in some unexpected component breaking. Fixing these resulted in unexpected downtime for experimenters whose reaction was often inversely correlated to the distance from their next deadline. More experienced users typically managed their own set of tools and additional software dependencies. This typically worked acceptably well – until they started collaborating with others…

The final breaking point was a cluster-upgrade from Debian Wheezy to Jessie,[2] that could not be postponed anymore. Despite best efforts to prepare, announce and perform the upgrade, software packages started breaking in a lot of unexpected places: Be it support libraries that changed APIs or small fire-and-forget tools that were written once and then simply expected to work. The result was unwanted downtime of the whole cluster for about two weeks, during which scientists were confined to their own systems for any compute-related tasks. Overall, it took far longer to put out all software-related fires that prevented scientists from interacting with the cluster system in productive ways. It then became clear that more effort needed to be put into maintainability and verification of potential updates in order to minimize unwelcome surprises. We came to the conclusion that the core points to consider were:

- *Know your tools:* Trace the runtime and build environment so that dependency relations between developed internal and external software components are known.
- *Never change a running system (but if you have to, keep a record):* Have the environment be snapshotted so that a known-to-be-working software state can easily be restored, which is especially useful for bisecting or preserving old unmaintained

[1]`https://boost.org` (visited on 2021-04-12)
[2]`https://wiki.debian.org/DebianReleases` (visited on 2020-12-04)

(experiment) code. Furthermore, if an update introduces unforeseen bugs, we *significantly* reduce the recovery time in these cases.

- *Assume everything not tested is broken:* Verify that changes to the environment leave software libraries and experiments in a working-state by testing them in the new environment.

All of these principles are well known in software engineering, but oftentimes not followed through in scientific environments.

In this chapter we present a way to track and manage distinct sets of software environments. The core component is a package manager that is embedded into lightweight container images. Changes can be proposed easily and verified to not break existing deployments, all the while leaving day-to-day operations undisturbed. Images are then released as needed in a rolling-release system.

In particular, this addresses [Krafczyk et al., 2021]'s second core point P2 to achieve reproducibility:

> *When writing and releasing research software, aim for ease of (re-)executability.*

8.1 Managing Dependencies in HPC Environments: Spack

After the brief overview of package managers given in Section 4.4.2, here we focus on Spack,[3] the package manager employed at Electronic Vision(s). It is used to manage all major software dependencies, including those needed to operate both BrainScaleS-1 and BrainScaleS-2 software stacks (the latter is detailed in Chapter 6).

As a fairly recent package manager, Spack takes inspiration from recent developments such as Nix,[4] but focusses on HPC. Here, it is important that many different versions of software packages can be built and stored in the filesystem side-by-side. Oftentimes – as is the case with tools like EasyBuild[5] and `smithy`[6] – packages are stored in locations which are derived from their version and a fixed set of "hyperparameters", like version, architecture, version of certain dependencies, build settings, etc. However, these naming schemes are not exhaustive and run into problems: In time, there is always another build feature not covered by the scheme or certain dependencies are not explicitly expressed. This is is known as the "matrix problem" [Geimer et al., 2014]. To that end, Spack adopts cryptographic hashes from HashDist[7] and Nix. That means that for each package, its full description in terms of features/build-options *and* dependencies is hashed and determines its deploy location.

[3]Supercomputing PACKage manager, [Gamblin et al., 2015]
[4]Nix Package Manager, [Dolstra et al., 2004]
[5]EasyBuild: Building Software with Ease., [Hoste et al., 2012]
[6]Smithy, [Jones et al., 2008]
[7]HashDist, [Ahmadia et al., 2012]

Recently, Spack has been deployed to more and more sites. These include CERN [Stewart et al., 2020], LRZ[8],[9] Microsoft Azure[10]or Amazon AWS.[11] Even Intel has integrated Spack into their High Performance Computing Reference Stack v2.0.[12] Mainstream adoption is increasing as well, with recent blog posts deferring component installation to simply using Spack.[13]

8.1.1 A DSL to model package configuration: spec-synatx

Using hashes in deploy-locations poses a problem: They are unintelligible for users. Hence, users need a different way to discover, install and load installed packages into their environment. To accommodate for this, Spack implements its own DSL[14]: A complete package configuration is described by a spec.[15]

A spec is a special syntax to specify a software package with possible constraints. As an example, we use `visionary-nest`, a collection of custom NEST[16] models, developed predominantly during [Breitwieser, 2015] but extended and packaged during this thesis. In its simplest form, it consists of just the package name:

```
$ spack install visionary-nest
```

Would just install install the package at its preferred version. Specific versions can be set via `@`:

```
$ spack install visionary-nest@1.0
```

Dependencies can also be enforced to be a certain version spec syntax via `^`:

```
$ spack install visionary-nest@1.0 ^nest@2.14.0+gsl~mpi
```

Whereas variants – Spack's name for software features – are en-/disabled via `+<variant>` or ∼`<variant>`. There is also support for string-based variants.

Last but not least, compiler (`%`) and computer architecture (i.e., the instruction set with which compiled machine-code is expected to be executed; `arch=`) can be given.

An example can be found in Listing 2. Each definition corresponds to a `class` in Python, deriving from a special, potentially build-tool specific, baseclass. Besides meta-information

[8]Leibniz Supercomputing Center
[9]`https://spack.io/lrz-using-spack/` (visited on 2021-04-11)
[10]`https://archive.is/bEszb` (visited on 2021-04-11)
[11]`https://jiaweizhuang.github.io/blog/aws-hpc-guide/` (visited on 2021-04-11)
[12]`https://software.intel.com/content/www/us/en/develop/articles/high-performance-computing-reference-stack-v2-0-now-available.html` (visited on 2021-04-11)
[13]`http://pramodkumbhar.com/2020/03/architectural-optimisations-using-likwid-profiler/` (visited on 2021-04-11)
[14]Domain Specific Language
[15]Specification of Package Configurations as used by Spack
[16]NEural Simulation Tool, [Diesmann et al., 2002]

```python
1  class VisionaryNest(CMakePackage):
2      """This repository contains many NEST models developed within the
3      Electronic Vision(s) group, compiled into a single nest module."""
4  
5      url = "https://brainscales-r.kip.uni-heidelberg.de/" \
6            "projects/model-visionary-nest"
7      homepage = "https://brainscales-r.kip.uni-heidelberg.de/" \
8                 "projects/model-visionary-nest"
9      git = "git@gitviz.kip.uni-heidelberg.de:model-visionary-nest.git"
10 
11     version('1.2', commit="693455678a0ed645c8f1c006200a1f16a2a3de9c",
12             preferred=True)
13     version('1.0', commit="1e4c5a4611875a97379b49a08b8769d3e2b76108")
14     version('master', branch="master")
15 
16     depends_on('nest@2.14.0:+modules')
17 
18     def cmake_args(self):
19         args = ["-DCMAKE_CXX_FLAGS=-I{0}".format(
20                 join_path(self.spec["nest"].prefix, "include", "nest"))]
21         return args
22 
23     def setup_environment(self, spack_env, run_env):
24         run_env.append_path("NEST_MODULES", "visionarymodule")
25 
26     @property
27     def root_cmakelists_dir(self):
28         return "nest-module"
```

Listing 2: Example Spack spec for `visionary-nest`, a collection of custom NEST models, developed predominantly during [Breitwieser, 2015]. In Spack, package configurations are specified in a DSL embedded in Python. Software versions can be specified via `version` (line 11–14) in a variety of ways, including checksummed archives or (in this case) specific commit-hashes. Dependencies on other packages are specified via depends_on (line 16) and can include version constraints (semantic version ranges are supported; here `v2.14.0` and onward) as well as specific variants (enabled support for NEST-modules).

such as the homepage and a quick description, it defines available `versions` and how these are obtained (download of checksummed archives, checkout of a particular commit from a `git`-repository, etc.). Having checksummed artifacts does offer protection against supply chain attacks in which open source dependencies are infested with malicious code.[17] Of course, this requires maintainers to thoroughly verify every archive prior to including it in package definitions. Furthermore, variants can be defined via the corresponding `variant`-directive. Dependencies are defined via depends_on-directives, linking different

[17]`https://archive.is/GLVrD` (visited on 2021-02-26)

package definitions in a DAG.[18] Circular dependency relations are explicitly forbidden. Spack distinguishes between build-time, runtime and linking dependencies:

`build` Dependency is only needed when building the package, but not when running. I.e., it will be included in environmental variables like `PATH` and `PYTHONPATH` at build-time.

`link` Dependency is only needed while linking. Dependency will be added to Spack's compiler wrappers to inject the appropriate linker flags (see below).

`run` Dependency is made available in the same manner as `build`-dependencies during build-time. This includes `spack load` as well as any generated module files.

`test` Dependency is only needed to run tests of the given package. It will be added like `build`-dependencies but only if the user specifies `--test` when installing.

Dependencies can be of several dependency types. Spack will allow uninstalling of `build`- and `test`-dependencies, but not `link`- or `run`-dependencies. There is also support for *virtual* dependencies, i.e., several implementations of the same software API/functionality that can be used interchangeably (e.g., Java or MPI[19]).

The main advantage of introducing the new spec-syntax is that it can be used *within* package definitions. Most of the directives accept a `when=`-argument that limits their scope to only apply if the given spec-definition is satisfied. The simplest example would be expressing conditional dependencies, such as a package depending on Python if its Python-related features are enabled[20]:

```
depends_on('python@2.7:', when='+python')
```

Version ranges in Spack are inclusive and can be open ended on one side. But also complex version-constraints can be expressed. This is especially useful if different versions of a package require different minimal versions of dependencies (cf. Listing 3).

```
class PyScipy(PythonPackage):
  # ...
  depends_on('py-numpy@1.5.1:+blas+lapack', # ... # )
  depends_on('py-numpy@1.6.2:+blas+lapack', when='@0.16:', # ... # )
  depends_on('py-numpy@1.7.1:+blas+lapack', when='@0.18:', # ... # )
  depends_on('py-numpy@1.8.2:+blas+lapack', when='@0.19:', # ... # )
  depends_on('py-numpy@1.13.3:+blas+lapack', when='@1.3:', # ... # )
```

Listing 3: Example for expressing more complex dependency relations. Here, different versions of Python package `scipy` require different minimal versions of `py-numpy`. In all cases, `py-numpy` is required to have `blas` and `lapack` variants enabled. For brevity, `type=('build', 'run')` was omitted in each `depends_on`.

[18] Directed Acyclic Graph
[19] Message Passing Interface, [Graham et al., 2006]
[20] Of course this requires a `python`-variant to be defined.

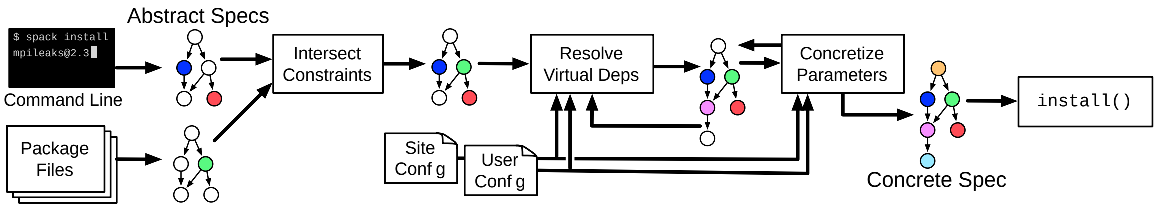


Figure 8.1: General concretization workflow in Spack. The user specifies which packages they want to have built along with possible constraints (specific versions/variants). The corresponding package files then might impose another set of constraints (e.g., because of known incompatibilities or constraints). Together with possible user/site configurations, these abstract specs are then "concretized", whereby all virtual dependencies are resolved to an actual package providing the required interface. After the concrete versions of all involved packages are known, the installation procedure can commence. Adapted from: [Gamblin et al., 2015, Figure 6]

Another point worth highlighting is the fact that package definitions in Spack are written in Python which enjoys increasing popularity in the scientific community [Oliphant, 2007; Virtanen et al., 2020]. This increases the chances of users – which primarily consist of scientists in a HPC environment – being able to read and understand package definitions than if they were written in a bespoke functional dialect as is the case with Nix. It lowers the barrier of entry for users – often *not* trained software engineers – to add needed software packages or fix issues on their own (see Section 8.3) because the syntax is already familiar.

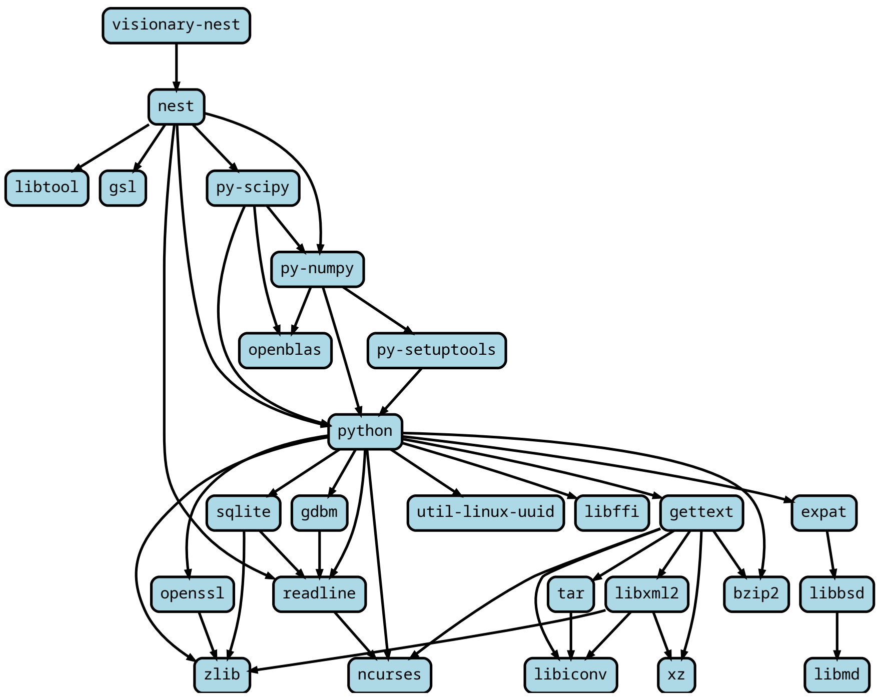


Figure 8.2: Fully concretized dependency graph resulting from Listing 2. For simplicity, all concrete version information was omitted, but all additional dependencies from enabled variants are shown.

8.1.2 | Concretization

The overall Spack workflow is sketched in Figure 8.1. Initially, a DAG is created from user-supplied packages to install – possibly with constraints – and their dependencies. Any further constraints present in package definitions are incorporated. In a second step, any virtual dependencies are resolved and replaced by a concrete package providing it. This "concretization" is then continued for all specs in the graph until every package is assigned a concrete version and set of variants (cf. Figure 8.2). After that, building packages may commence.

8.1.3 | Build process

When installing a package, Spack first extracts the contents of the chosen version's archive or `git` repository to a temporary build location.[21] In order to accommodate all workflows, Spack provides a `patch`-directive that allows applying patches to perform arbitrary code modifications. Patches can either be distributed with the Spack-repository or downloaded (just like other resources) in a checksummed manner from arbitrary locations. Again, `patch`-directives support the `when=`-argument, allowing for fine-tuned control about when patches are applied.

The whole package install process is separated into distinct phases, as one has come to expect of (meta-)build-tools. Per default, there is only a single `install`-phase, however, Spack allows build-tools specific baseclasses to add more. This is useful to accommodate all kinds of software packages. Whereas a typical Python package merely needs to be installed, i.e., copied, to the right place, intricate C++ frameworks might need several steps of configuration, code-generation, building and installing. Each of these phases can be customized in the package definition – or extended via the `when=`-supporting `@run_after`/`@run_before`-decorators that allows for inclusion of arbitrary methods into each phase. Often, it is enough to provide arguments to the build tool (cf. Listing 2), but sometimes more complex operations are needed. Here, the concretized spec comes in handy, as it can be queried via `self.spec.satisfies('<spec>')`. Like the `when=`-argument to directives, this allows for conditional adjustment of package definitions to very specific edge cases, without having users resort to leaving the Spack-framework.

While installing a package, Spack is very adamant about *environment isolation.* Each `install`-call is run in its own process so that package authors are free to modify the environment as they need without changes propagating into other install processes, which might cause undetermined behavior. Environment variables relevant[22] to the build tool are pre-populated with paths to the dependencies. Same as Nix and HashDist, Spack uses `RPATH`s to link binaries to dynamic libraries to minimize the risk of runtime errors by

[21]The authors of [Gamblin et al., 2015] report a speedup when building on locally attached storage because indirections, such as NFS, often decrease performance when reading/writing many small files. We can confirm this and even experienced building on NFS to be outright impossible altogether, cf. `https://github.com/bazelbuild/bazel/issues/2042#issuecomment-258429160` (visited on 2020-12-10).

[22]Examples include `PATH`, `PKG_CONFIG_PATH`, `CMAKE_PREFIX_PATH`, and `LD_LIBRARY_PATH`.

loading wrong, potentially ABI[23]-incompatible, libraries. Spack achieves this by having environment variables most build system use to determine compiler binaries point to wrapper scripts.[24] These wrapper scripts then modify arguments to insert include- (`-I`) and library-related (`-L`) flags as well as linker options for `RPATHs`[25] (`-Wl,-rpath`). This ensures each compiler-call is only able to access header files and link to libraries present in its dependencies, making the build *reproducible*.[26]

Finally, Spack is also able to generate module files for `dotkit`,[27] GNU modules[28] and Lmod.[29] These are important for users when loading packages including anything besides binaries, as they ensure auxiliary environment variables such as `MANPATH` are set up to provide `man`[30]-pages.

8.1.4 | Contributions

At Electronic Vision(s), we operate a local fork[31] of Spack that contains fixes and package updates not yet merged upstream as well as to-be-merged features our container build flow (described in Section 8.3) relies upon. In regular intervals, the state of the upstream repository[32] is merged back into our local fork and verified in our regular container workflow (that will be introduced in Section 8.3). During the adaption of Spack to the Electronic Vision(s)-workflow (see Sections 8.2 and 8.3), several contributions were made. Apart from simple bug fixes, they are as follows:

`visionary-` Meta-Packages are used to track dependencies to for all disjoint software stacks within Electronic Vision(s). They are implement as essentially empty packages that depend on a set of actual dependency packages. Due to the rolling release nature of our Singularity-based container build (see Sections 8.2 and 8.3), they are kept at a single version and constitute more or less separate environments. Please note that nowadays Spack supports an explicit handling of environments, however, our implementation of `visionary-`packages predates this feature. At the time of writing, they include (`visionary-`prefix omitted):

`dls-core`	Includes all dependencies of the core BrainScaleS-2-software stack as described in Chapter 6. All Jenkins-tests executed must succeed using only these dependencies. It contains 35 direct dependencies, amounting to 128 packages in total.

[23]Application Binary Interface
[24]Variables include `CC`/`CXX` for C/C++ or `F77`/`FC` for Fortran.
[25]Run-Time Search Paths
[26]One caveat is that build-systems, in an effort to be as painless as possible, are often very good at finding system-dependencies that might have been forgotten to be specified in the Spack package itself. Hence, Spack packages need to be written with care!
[27]`dotkit`, Simple Module Files via Shell Scripts, `https://dotkit.sourceforge.io/` (visited on 2012-12-06)
[28]GNU Environment Modules, [Furlani, 1991]
[29]Lua-based Module System, [McLay et al., 2011; Geimer et al., 2014]
[30]Terminal-accessible Manual Page
[31]`https://github.com/electronicvisions/spack` (visited on 2020-12-15)
[32]`https://github.com/spack/spack` (visited on 2020-12-15)

`dls`	Includes all dependencies of *all* experiments actively developed on BrainScaleS-2 that are verified via Jenkins. For obvious reasons, `dls-core` is included, as well as all analytics-related software packages needed for experiment execution and plotting of results. It contains 21 direct dependencies, amounting to 219 packages in total.
`nux`	Includes all dependencies of `libnux`. It contains 8 direct dependencies, amounting to 18 packages in total.
`simulation`	Includes simulators and helper libraries typically used for neuroscientific simulations and their analysis within Electronic Vision(s). Jenkins jobs for `sbs`[33] are executed using only these dependencies. It contains 13 direct dependencies, amounting to 78 packages in total.
`spikey`	Includes all dependencies of the legacy Spikey[34] software stack. It contains 14 direct dependencies, amounting to 78 packages in total.
`wafer`	Includes all dependencies of the BrainScaleS-1 software stack and *all* experiments actively developed on BrainScaleS-1 that are verified via Jenkins. The split into `wafer` and `wafer-core` in the same manner as the `dls` stack is planned but has not happened yet. It contains 40 direct dependencies, amounting to 230 packages in total.
`wafer-visu`	Includes all dependencies of the visualizer of hardware mappings for BrainScaleS-1, namely a C++ to Javascript[35] compiler and helper libraries. It contains 2 direct dependencies, amounting to 230 packages in total.
`xilinx`	Includes runtime dependencies of hardware development tools. It contains 10 direct dependencies, amounting to 230 packages in total.
`deep-loop`	Legacy package that includes software dependencies used for the publication [Schmitt et al., 2017]. It is currently not actively integrated into stable container builds (cf. Section 8.3). It contains 11 direct dependencies, amounting to 144 packages in total.
`slurmviz`	Includes dependencies of the containerized Slurm-deployment at Electronic Vision(s) (see Section 9.1). All Slurm-related services must run and are tested with only these dependencies. It contains 17 direct dependencies, amounting to 73 packages in total.
`unicore`	Includes all dependencies for support of UNICORE.[36]

[33]Spike-Based Sampling – a library for fast Neural Sampling, [Breitwieser et al., 2020; Breitwieser, 2015]
[34]Spikey chip, [Pfeil et al., 2013]
[35]ECMAScript 2020, [ECMA2020]
[36]Uniform Interface to Computing Resources, [Benedyczak et al., 2016]

	It contains a single direct dependency: a Java-provider.
`clusterservices`	Includes the sum of all dependencies that are needed for any cluster-related service/deployment. It contains 2 direct dependencies, amounting to 76 packages in total.
`dev-tools`	Collects all development tools (e.g., editors, utilities and helper packages). These include all software packages people would also use on a regular cluster-frontend node. All user-facing `visionary-` packages – i.e., those that have their own "app" in the container (as will be described in Section 8.3) – have an optional dependencies on this package. This avoids potential incompatibilities when providing a development environment as all development tools will be subject to any concretization constraints the corresponding package imposes: In particular, the `visionary-wafer` environment will be built with Python 2, whereas `visionary-dls` will be built with Python 3. It contains 70 direct dependencies, amounting to 238 packages in total.

Updated and new Packages During initial setup, over the course of this thesis and continuing beyond that, dependencies to the various meta-packages described above are added that are not yet or not fully integrated into the Spack ecosystem. These packages are contributed back upstream in regular intervals. At the time of writing, more than 300 package definitions have been pushed back upstream, mainly by Andreas Baumbach, who oversaw pushing package contributions back upstream, after they had been verified to work in our local fork. Sometimes quick fixes – or adjustments done due to unique characteristics the Electronic Vision(s)-setup which might break for other users – are held back until we can be sure they are stable. While *very* time-consuming, it pays back in the long run: Once a package (update) is pushed upstream, all following changes in the upstream repository will have take it into account. Hence, when we merge these upstream changes back into our own fork, we do not have to spent any time making them compatible to our local package version.

Rework of `view`-Command and extensions Spack features a generic `extension`-mechanism that allows us to express a plugin/module/extension/etc.-relationship between packages. This indicates that some packages do not provide functionality in form of binary executables or libraries when installed, but rather extend functionality of a parent package. Typically, this is the case for all interpreted languages. To illustrate: All Python packages are extensions to the Python-package because they provide code that is only executable by a Python interpreter.

Due to the way some software projects are designed, installing an extension might require modifications in the physical install location of the parent package. A prime example here are older Python-packages installed via `python-setuptools` requiring modifications in `easy-install.pth` at the target location.

Furthermore, Spack provides entry points for parent packages to customize how their extensions are activated or deactivated: Java packages need to be installed differently from Python libraries. Previously, Spack supported *one set* of extensions per installed package. Extensions could be "activated" and "deactivated" within their parent package's installation. In particular, this meant that a single Python installation could not have different versions of the same library activated at the same time. This conflicted with the way we set up disjoint environments in the form of different `visionary-` packages (see above) that potentially required different versions of the same Python library.

A way to work around this limitation is Spack's `view`-command. It allows for the creation of filesystem *views* for a particular set of packages. In its simplest description, a view is an arbitrary directory in the filesystem, denoting the view's *root*. File contents and structure of all involved packages are linked[37] *into* this root folder in the same structure they are installed. In a sense, the view appears to be a combined installation of all packages contained within. The only difference is all files are linked to their Spack-installed counterpart. Users can then simply adjust their environment to include the view's root path to make full use of all packages added to it; no further interaction with any module system required.

The original implementation of the `view`-command, however, was a plain, thinly-wrapped recursive link-operation. This meant that the set of activated extensions for each package was mirrored as well, leading to errors: One could not use the `view`-command and have activated extensions at the same time.

Therefore, the `view`-command was reworked[38] to allow a separate set of extensions in each view location. Especially, this means that the same Python installation can be present in several views with different sets of potentially incompatible libraries. Views are the basis for different environments in our container solution (see Section 8.3[39]). With almost all `visionary-`packages containing Python-related packages, adding the ability to have different sets of Python libraries, i.e., activated extensions, was *crucial*.

`fetch`-ing from specfiles Via the `fetch`-command it is possible to merely download all required data (archives, repositories patches) needed to install a particular set of packages. In its original implementation, it is concretizing the given specs on its own to determine what needs to be downloaded. This is a problem for two reasons:

i) As seen above, `visionary-` meta-packages have quite a lot of dependencies, sometimes in very particular configurations. Concretization for a single package is constantly improving, but still takes on the order of minutes. Hence, as will be explained in Section 8.3, it is important to concretize *once* and then re-use the concretization result.

ii) Since many packages appear in several meta-packages, the sets of required data to download will overlap between meta-packages. This means that meta-packages should not be fetched in parallel as this both wastes bandwidth and possibly triggers anti-DDOS[40] measurements at some sites. Furthermore, a naïve implementation

[37]Spack supports both hard- and symbolic links.

[38]`view`-rework-pull request:`https://github.com/spack/spack/pull/3227` (visited on 2021-01-03)

[39]Each visionary package has a view created under `/opt/spack/spec_views/visionary-<name>` containing all its dependencies.

[40]Distributed Denial-of-Service Attack

with several Spack instances downloading in parallel was also able to corrupt the download cache via concurrent writes. This has been fixed in the meantime.

Overall, we would be forced to fetch (and concretize) each packages in turn, wasting time. Instead, the `fetch`-command was modified[41] to also accept pre-concretized specs in the form of `specfiles`. This functionality already existed for other commands such as `install`. Now, a set of packages, pre-concretized in parallel, can be fetched at once and without conflicts.

`stack`-Command When evaluating or debugging Spack-packages locally prior to inclusion into `visionary`-meta-packages, it is often desirable to rely on pre-installed packages from another remote and read-only Spack-installation. By avoiding to build all dependencies anew this saves a lot of time, especially when dealing with a relatively benign package with a lot of dependencies. The `stack`-command[42] works by creating symbolic links in the local repository pointing to the remote one. The local Spack database can then index the packages and treat them similar to locally installed ones. Since binaries are linked via `RPATH`s still pointing to the original locations, they continue to work. It has since been replaced with `chain`-ing,[43] a more tightly-integrated iteration of the same concept. Users can point their local Spack installation to an `upstream`[44] instance, for example the one in the current container.[45] The local Spack database is then able to directly access the remote database, eliminating the need for emulation at the filesystem level. This allows for, among other things, removal of remote packages to be more easily detectable locally. However, as our workflow (see Section 8.3) does not rely on removing packages, simply `stack`-ing a selected number packages from remote Spack installations was sufficient.

Helper functions to express dependency-relations Some packages are rather tightly connected. In fact, sometimes there might even be a one-to-one correspondence between them, such as `protobuf`,[46] where each version of the Python implementation can optionally use the corresponding C++-implementation for speed-up purposes. Expressing this kind of dependency requires a lot of boilerplate code, i.e., one entry for every version. We therefore introduce a new `same_version_as`-directive[47] that maps these dependencies.

```python
# ... in package definition of py-protobuf ...
same_version_as("protobuf", when="+cpp",
                pkg_to_dep_version=lambda v: v.up_to(3))
```

Suitable versions of the dependency are identified using an optional helper function. In the example above, C++ implementations must match up to patch-level, but by default, the identity function is used.

[41] `fetch`-pull request: `https://github.com/spack/spack/pull/13106` (visited on 2021-01-03)
[42] `stack`-pull request: `https://github.com/spack/spack/pull/7081` (visited on 2020-12-15)
[43] `chain`-pull request: `https://github.com/spack/spack/pull/8772` (visited on 2020-12-15)
[44] `https://spack.readthedocs.io/en/latest/chain.html` (visited on 2021-05-02)
[45] Within visionary containers, the Spack installation resides at `/opt/spack`, cf. Figure 8.6.
[46] `https://developers.google.com/protocol-buffers` (visited on 2021-01-04)
[47] `same_version_as`-pull request: `https://github.com/spack/spack/pull/14002` (visited on at 2021-01-04)

Conclusion Overall, tracking dependencies via Spack has proven to be very beneficial for the Electronic Vision(s) group. It eases introducing and tracking new dependencies into vastly different software stacks. An anonymous quote from the 2020 Spack user survey[48] summarizes quite well:

> *I'm still in dependency hell, but Spack took me from the 7th circle (violence – for the violence I'd like to commit against my keyboard while building things) to the 3rd circle (gluttony – for the voracious appetite I now have for Spack-installed packages and the indulgent number of dependencies they require).*

8.2 Software Environments via Customized `visionary` Singularity-Containers

Even after tracking and managing software dependencies with Spack (described in Section 8.1), we still face one crucial problem: Evolution. After having a Spack installation in place, how do we allow updates and modifications while at the same time keeping it stable and running for all users? While Spack is already in a usable state, it is still under active development. This means that core functionality might be extended or changed, sometimes in backwards-incompatible ways. Therefore, one cluster-wide Spack-deployment with a stable TUI[49] that everyone can use is out of scope in the near future.

A first solution was to have Jenkins (see Section 7.1) automatically deploy snapshots of our local `visionary`-branch Spack cluster-wide via NFS. Users could load Spack from these snapshots and in turn use it to load package files. For generating the list of modules to load, Spack needed a noticeable amount of time, i.e., on the order of a few hundred milliseconds up to seconds. In an interactive setting – with users actively waiting for the next prompt to appear – these delays are far less tolerable than in batch execution, causing frustration. Loading Spack-generated GNU modules-files directly, while possible, proved to be cumbersome since dependencies were only tracked in Spack and not in the module files themselves. Users can use Spack to generate module-loading scripts, but since each Jenkins-deployed Spack-snapshot resided under a unique NFS-path with unique hashes for each package, all these scripts had to be regenerated for each newly deployed snapshot. Yet again, this forces users to perform unnecessary manual labor which we want to avoid. Furthermore, building snapshots in different locations takes a lot of time, even when using Spack's `buildcache` feature (see Section 8.3.3). Finally, these Spack deployments are only usable from machines with access to our NFS-based storage. They cannot be used on external computers such as laptops, standalone servers or other cluster sites.

Hence, Jenkins-based Spack-snapshots were discontinued and replaced by Singularity-based containers. Using containers to manage dependencies is well established in HPC environments [Zhang et al., 2017]. This section focuses on Singularity and internal structure of "visionary" containers. Their build process and structure will be discussed immediately after this in Section 8.3.

[48] `https://spack.io/spack-user-survey-2020/` (visited on 2020-12-15)
[49] Text-based User Interface

8.2.1 | Overview

After the brief overview in Section 4.4.3, Singularity [Kurtzer et al., 2017] appears – at first glance – as yet another lightweight container solution aimed at HPC-environments. However, at the time of deciding which container solution to use for the approach presented in this thesis, it was one of the most complete implementations in terms of focus, feature set, user adoption (in HPC settings) and developer support. Furthermore, it is employed and used in production at a variety of HPC-sites [Kurtzer et al., 2017, Table 4].[50] Singularity was written in C, `bash` and Python up until version `3.0` and is now written in go.[51] Its unique characteristics, besides a strong focus on security and support to seamlessly access computing devices such as GPUs[52] from within the container, are that a single image file fully defines a container. However, it also supports running containers from sandboxes, i.e., plain directories.

In contrast to Shifter or Sarus, Singularity does not offer a local gateway service that automates container image creation. Instead, a set of CLI[53]-tools is provided to create and operate containers. Using these tools, a pipeline for image creation was developed during this thesis that is used throughout Electronic Vision(s) (see Section 8.3). There are *no running services*[54] needed to operate Singularity containers, whatsoever. Some functionalities, such as image creation, require `root`-permissions. Singularity has support to build containers from base images pulled from remote sites or repositories. The latter includes Docker[55]-related hubs as well as its own hubs.[56]

Starting with version `v3.1`, Singularity provides an OCI[57] compliant runtime via a special `oci`-command subgroup.[58] As all commands require `root` permissions to execute, it is not primarily intended for end-users, but rather interoperability with other OCI-compliant services, namely Kubernetes[59] via Singularity-CRI.[60] Furthermore, Singularity offers support to verify container contents via hashing, using a direct implementation of SHA-256 [NIST2015]. User can therefore verify the identity of a given container's contents which allows for reproducible software execution. Within Electronic Vision(s), we currently do not make use of this feature.

Workflow Singularity's workflow is sketched in Figure 8.3. In short, building containers requires `root`-permissions and typically happens on user-owned devices (such as lap-

[50] `https://doi.org/10.1371/journal.pone.0177459.t004` (visited on 2021-02-01)
[51] go Programming Language, [Pike, 2009]
[52] Graphics Processing Units
[53] Command-Line Interface
[54] In order to save resources, containers can be spun up as instances. Instances are isolated persistent versions of containers that can be used concurrently to spawn new processes, thereby decreasing setup time required to spawn a new container-image. It also allows to operate services in a Docker-like fashion.
[55] Docker, [Merkel, 2014]
[56] `https://singularity-hub.org/` (visited on 2020-01-27) and `https://cloud.sylabs.io/library` (visited on 2020-02-08)
[57] Open Container Initiative, `https://opencontainers.org/` (visited on 2021-01-26)
[58] `https://sylabs.io/guides/3.1/user-guide/oci_runtime.html` (visited on 2021-01-28)
[59] `https://kubernetes.io/` (visited on 2021-01-28)
[60] `https://sylabs.io/guides/cri/1.0/user-guide/index.html` (visited on 2021-01-28)

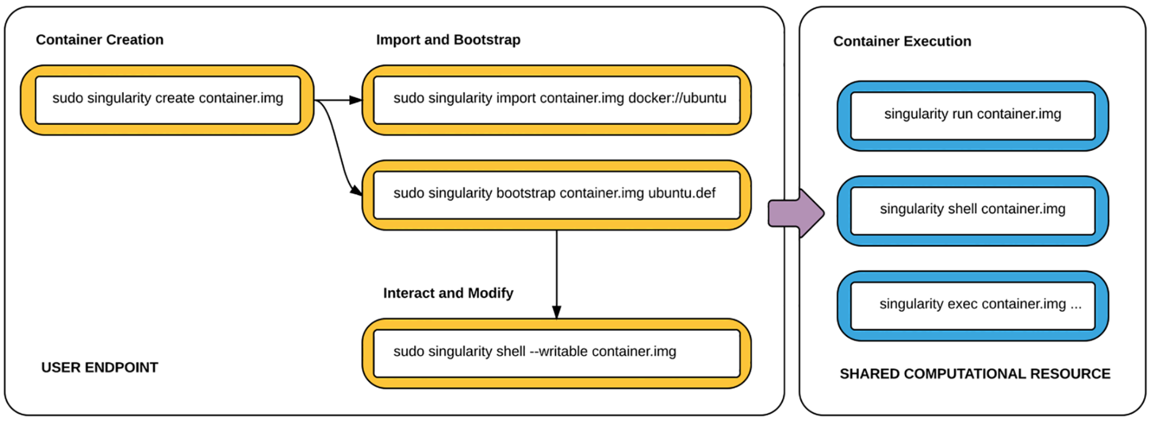

Container Creation
sudo singularity create container.img
Import and Bootstrap
sudo singularity import container.img docker://ubuntu
sudo singularity bootstrap container.img ubuntu.def
Interact and Modify
sudo singularity shell --writable container.img
USER ENDPOINT
Container Execution
singularity run container.img
singularity shell container.img
singularity exec container.img ...
SHARED COMPUTATIONAL RESOURCE

Figure 8.3: Singularity usage workflow. **Left:** Container creation requires `root` rights and typically occurs locally on user endpoints (such as desk-/laptops) or dedicated build nodes (see Section 8.3). Container can be bootstrapped from existing (possibly Docker-based) images that are pulled in from a remote hub and modified in place. *(Not shown:)* Instead of image-files, plain directories can also be used as containers, called *sandboxes*. These sandboxes can then be packed into compressed images. **Right:** Afterwards, the container can be deployed to and used on shared computational resources where the user does not have `root`-permissions. Typical workflows involve `run`ning a default application of the container (if defined), working via `shell` within the container environment or `exec`uting arbitrary binaries within the container.
Adapted from: [Kurtzer et al., 2017, Figure 1].

or desktops) or dedicated build nodes (as is the case in Section 8.3). Deployed container images are then mainly used in two ways: Either, a user executes a single process within the container environment or "enters" the container environment by spawning a new shell (comparable to `ssh`-ing to another node within the cluster). Of course, the latter is just syntactic user convenience to execute an interactive shell process "automagically". It is also possible to transparently wrap in-container execution of a binary via `clusterize`, a small tool developed during this thesis (see Section 9.2).

8.2.2 Technical Background

Under the hood, Singularity works in the same manner as all container implementations (see Section 4.4.3.1): The container image consists of a base image that is mounted as immutable root filesystem. By making use of OverlayFS,[61] a temporary filesystem is mapped *on top of* the root filesystem, allowing the containerized process to modify the filesystem where its user permissions permit it without changes propagating outwards. This is especially important if temporary files – such as log files – need to be written. These changes can also be made persistent via the usage of overlays. Overlays are plain writable `ext3`[62]-images. They are especially useful to test small adjustments to containers without rebuilding the whole container. By default,there is no default mapping of UIDs[63] or GIDs.[64]

[61]OverlayFS, `https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html` (visited on 2021-01-26)
[62]third EXTended filesystem, [Tweedie, 2000]
[63]User IDentifier numbers
[64]Group IDentifier numbers

That means that within a container image, users have as much or as little permission as the filesystem permits. Users are able to specify bind-mounts (read-only or read-write) from the host-filesystem to various in-container locations. By default, Singularity spawns a new filesystem namespace for in-container processes, though isolation in other namespaces (PID, network, IPC etc) can be enabled as well.

Since the container image is mounted as a loopback[65] device, an operation that only `root` may perform, Singularity needs to elevate its rights momentarily upon invocation. This is achieved via `setuid`[66] which is a filesystem flag that allows binaries to run as the owning UID or with admin-selected capabilities.[67] In case of Singularity, it is a `root`-owned `starter-suid` binary that is launched from and operates in tandem with the regular `singularity` binary. After performing all (bind-)mount related tasks, it immediately exits. Singularity has support for user-namespaces, discussed in Section 4.4.3.1, which do not need to perform any operations with `root`-permissions. However, this approach is only able to run sandboxes, i.e., plain folders, as containers because then no loop-devices need to be set up. The base image can be in plain `ext3`, SquashFS[68] or (starting with Singularity `3.0`) SIF[69] format.

Definition Files Same as `Dockerfiles`[70] for Docker, Singularity containers can be created from recipe-like Definition Files[71] (`.def`-suffix). It allows for flexible specification of how to build a given container. The definition file is split into header and sections. The header contains options such as the bootstrap agent: It determines what kind of image the container will be based on and how to obtain it. Possible bootstrap agents include those pulling a remote image from Docker or Singularity related container hubs, but also those providing default installations from a variety of Linux distributions – including Debian,[72] Arch[73] and CentOS,[74] minimalist setups like BusyBox[75] and local or even completely empty base images.

Further sections in the definition file then describe what other steps have to be performed in order to create the image – such as manual installation of software or copying certain files from the host into the container – or how the different environments (called `apps`) are defined. The container image used to create visionary containers is detailed in Section 8.3.

[65]`https://man7.org/linux/man-pages/man8/losetup.8.html` (visited on 2021-02-01)
[66]`https://man7.org/linux/man-pages/man2/setuid.2.html` (visited on 2021-02-01)
[67]`https://man7.org/linux/man-pages/man7/capabilities.7.html` (visited on 2021-02-01)
[68]SquashFS, `https://github.com/plougher/squashfs-tools` (visited on 2021-02-01)
[69]Singularity Image Format, [Godlove, 2019]
[70]`https://docs.docker.com/engine/reference/builder/` (visited on 2021-02-08)
[71]`https://sylabs.io/guides/3.7/user-guide/definition_files.html` (visited on 2021-02-08)
[72]`https://www.debian.org` (visited on 2021-02-08)
[73]`https://archlinux.org` (visited on 2021-02-08)
[74]`https://www.centos.org/` (visited on 2021-02-08)
[75]`https://busybox.net/` (visited on 2021-02-08)

8.2.3 Visionary Containers

During this thesis, a container-based workflow was established. The core idea to provide a single, all-encompassing *visionary container* image that is sufficient for *all* work to be carried out. It contains all "slow-changing"[76] software dependencies needed for building the software stack (e.g., Chapter 6) as well as running experiments on different hardware platforms or as plain, potentially GPU-based, simulations. Since different environments – such as different hardware platforms – have potentially incompatible requirements (e.g., different Python-versions), they are provided disjoint from each other. Upon container invocation, the user can chose which environment to operate in via a command line switch (see below). Over time, almost all workflows at Electronic Vision(s) were migrated to use this container.[77] At the time of writing, it is about 7 GB in size.

Update Workflow As mentioned above, updates to container are performed in a rolling-release system when needed (see Section 8.3.2). Updates are released as new images, indexed by date and iteration counter. Within the Electronic Vision(s)-cluster, new images are available under `/containers`, accessible from all user-facing nodes via NFS. Its structure is outlined in Figure 8.4.

Newly released containers are deployed to `/containers/stable`. We have the policy that all users should "live at `HEAD`", i.e., they should verify their work can be performed with the latest container. To that end, a convenience symlink[78] is provided that always points to the `latest` stable container image. Users are therefore able to adjust their scripts to always load `/containers/stable/latest` to continuously perform work with the latest container. Furthermore, this symlink also allows administrators to perform easy rollbacks in the unlikely event that a new release contains a severe error which slipped by verification. In case of bigger changes, the last container fulfilling a certain criteria is sometimes tagged via symlink as well – in the example here, it is the last container with Python `3.7`. These tags are removed once they are not used anymore.

Alongside every container, all essential information needed to rebuild it from scratch is also preserved under `/containers/dna`. These archives only average a couple of megabytes in size and could therefore be distributed alongside the results the container was used to produce. Older containers that are not actively used are moved to a slower storage medium that is still reachable for exploratory purposes via `/containers/archive`. So far, no released stable images are discarded. Finally, `/containers/testing` contains images with pending changes to the environment pending evaluation. The container build process is described in detail in Section 8.3.

For external users, and as general convenience, stable visionary containers are made publicly available at:

`https://container.bioai.eu/`

[76]Compared to Electronic Vision(s) software repositories as, for instance, described in Chapter 6.
[77]Some hardware-related workflows use a different "ASIC" container that is also discussed below in Section 8.2.3.
[78]Symbolic Link, also: soft link

```
/containers
├ archive ⟶ /path/to/slow/storage
├ dna
│ ├ ...
│ ├ 2020-11-19_1.tar.gz (1.9M)
│ ├ 2020-12-15_1.tar.gz (1.9M)
│ └ 2020-12-15_2.tar.gz (1.9M)
├ overlays
│ ├ 2020-05-28_buster_texlive.img (3.4G)
│ ├ 2020-05-28_stretch_texlive.img (5.0G)
│ └ texlive_latest ⟶ 2020-05-28_buster_texlive.img (3.4G)
├ stable
│ ├ ...
│ ├ 2020-11-19_1.img (7.0G)
│ ├ 2020-12-15_1.img (7.3G)
│ ├ 2020-12-15_2.img (7.4G)
│ ├ latest ⟶ ./2020-12-15_2.img
│ ├ latest-py37 ⟶ ./2020-04-23_1.img
│ ├ ...
│ ├ asic_2020-06-17_1.img (201M)
│ ├ asic_2020-07-02_1.img (515M)
│ ├ asic_2020-07-10_1.img (515M)
│ └ asic_latest ⟶ ./asic_2020-07-10_1.img
└ testing
  ├ ...
  ├ c12858p1_2020-11-16_1.img (6.6G)
  ├ c13681p3_2021-02-11_1.img (7.2G)
  └ c13719p1_2021-02-14_1.img (7.2G)
```

Figure 8.4: Overview of deployed container's NFS folder structure. All container related files are available via `/containers`. Whenever new changes to our software dependency structure have been thoroughly tested an merged, a new `stable` container is built and made available under `/containers/stable`. Containers are indexed by date. A convenience symlink is automatically updated to point to the `latest` container image built so that user scripts do not have to be constantly adjusted. Special purpose ASIC-containers are deployed alongside in the same manner. When performing breaking updates that are known to affect some setups, we provide custom tags (e.g., `last-py37`). Testing containers are made available in a separate folder and indexed by Gerrit change number/patch level. Refer to the text for details.

It is a simple listing of all recently built container iterations and supports direct download. Assuming one has enough storage space, this eliminates the need to set up custom installations on local computers such as laptops when on the go. Ideally, users have to obtain singularity and they are good to go.

Avoiding Downtime via Snapshots Sometimes a new container release might disrupt user workflow. A prominent example are updates to compilers or core dependency libraries (such as `boost`[79]), typically requiring users to configure and build their software stack anew from scratch. These types of container updates are announced in advance in order to keep the surprise level to a minimum. Still, sometimes it might be inconvenient for users to perform these more "maintenance-related" tasks immediately. Even though new releases are thoroughly tested beforehand (see below), users might also encounter unexpected errors with their personal workflow. In both cases, users can simply switch to an older container version in case they encounter errors with their personal workflow. In case of actual errors, most problems can be tracked down fast by analyzing the difference between the working and faulty container iteration.

Having all container iterations available to users is a *crucial improvement* to pre-container days at Electronic Vision(s), where updates to the environment were infrequent and affected all users at the same time. If some users were adversely affected by such an update, they were *immediately* forced to either work around it on their own or sit idle as their problem got fixed. Now, they simply have to temporarily adjust the path to the container image they are using and can continue working. Downtime is minimal.

Reproducible one-off Projects Typically, when working on one-shot projects, e.g., publications or (hardware-)demos for conferences, users set up an environment once, make sure everything is working and then commence the project. Afterwards, usually a non-zero effort is made to document the software setup, but typically only on the immediate software components, not the environment as a whole. If, at a later point, the project gets revisited, the environment has most certainly changed: Software packages could be either missing, updated in a backward-incompatible way or – in worst case – slightly differ in behavior (due to bugfixes or different configuration). In any case, reproducing the *exact* behavior requires a significant amount of time and effort. When working with a containerized environment, this overhead reduces significantly. As long as authors note which container version and which top-level commits (see Section 4.4.1) they used in each software repository, their experiments remain reproducible, irrespective of how the visionary container gets updated.

Furthermore, completed projects can be turned into automated Jenkins-jobs that run on a nightly or weekly schedule. Examples for projects verified in a nightly routine presented in this thesis include TTFS[80] experiments on BrainScaleS-1 (see Section 12.3.3), NSEM[81] experiments performed on HICANN-DLS[82] [Spilger, 2018] and nightly calibrations for

[79]`https://www.boost.org` (visited on 2021-02-05)

[80]Time-To-First-Spike

[81]Neuromorphic Spike-Based Expectation Maximization, [Breitwieser, 2015]

[82]HICANN Dreieck Ludwighafen Süd: successor to HICANN chip and based on the technology test chip route65 which inspired the reference to BAB65, [Aamir et al., 2018; Friedmann et al., 2017]

`calix` [Weis, 2020]. These jobs are then used to verify that changes to the software stack or environment do not introduce undesired alterations in behavior. This is detailed in Section 8.3.

All in all, containers allow for a *more flexible environment update policy*. Through tests, differences in behavior are detected early so that they can be fixed before affecting users. In time-critical situations, environments can effectively be “frozen” so that breaking changes have no effect.

Provided Environments The container contains a set of disjoint software environments, roughly corresponding to the set of `visionary-` Spack-packages (described in Section 8.1.4). They are implemented as SCIF[83]-based software environments, which Singularity supports. Users can select which `app` to execute a singularity call in via the `--app` CLI-argument. For example, a work-session related to the BrainScaleS-2 software stack (see Chapter 6) could be started via:

```
$ singularity shell --app dls /container/stable/latest
```

Most `visionary-` Spack-packages are deployed as a standalone `app` of the same name. For convenience, all `apps` are also provided without `visionary-` prefix.

Additionally, many `apps` are bundled in two versions: With and without `-nodev` suffix. With suffix, they only contain software packages of their corresponding Spack package, i.e., build/runtime dependencies required to execute programs in the given environment. Without suffix, they *additionally* contain everything from `visionary-dev-tools` compatible with the original environment. For example, if the original environment only support Python 2 then all development tools will also be configured to support Python 2 (or not be included). All related Jenkins-jobs execute in the `-nodev` variant of the `app`. This is done to ensure that `visionary-dev-tools` provides no actual dependencies, but only development tools (hence the name).

Overall, the container provides:

`dev-tools`	Corresponds to the `visionary-dev-tools` Spack package. It is the default unspecific working environment that contains all typical development tools and programs one would expect from a terminal-focused desktop distribution, such as editors, compilers, linters, static code analyzers, plotting tools, etc.
`dls-core`	Corresponds to the `visionary-dls-core` Spack package. It includes all dependencies of the core BrainScaleS-2-software stack as described in Chapter 6. All software tests in Jenkins are executed in this `app`. As it is only serves verification purposes, it is not intended for interactive development and contains no utilities. Hence, there is no `-nodev` variant. Users are advised to use `dls` instead.

[83]SCIentific Filesystem, [Sochat, 2018]

`dls` (`-nodev`)	Corresponds to the `visionary-dls` Spack package. Includes all dependencies of *all* experiments actively developed on BrainScaleS-2 that are verified via Jenkins (in `dls-nodev`). For obvious reasons, `dls-core` is included, as well as all analytics-related software packages needed for experiment execution and plotting of results.
`simulation` (`-nodev`)	Corresponds to the `visionary-simulation` Spack package. Includes simulators and helper libraries typically used for neuroscientific simulations and their analysis within Electronic Vision(s). Jenkins jobs for `sbs`[84] are executed using only these dependencies (in `simulation-nodev`).
`spikey` (`-nodev`)	Corresponds to the `visionary-spikey` Spack package. Includes all dependencies of the legacy Spikey software stack.
`wafer` (`-nodev`)	Corresponds to the `visionary-wafer` Spack package. Includes all dependencies of the BrainScaleS-1 software stack and *all* experiments actively developed on BrainScaleS-1 that are verified via Jenkins (in `wafer-nodev`). The split into `wafer` and `wafer-core` in the same manner as the `dls` stack is planned but has not happened yet.

Read-Only Overlays Additionally, in order to keep the size of containers at a manageable level, some software packages – those that are rather easy to install but large in storage requirements and change at glacial speeds – are only provided via read-only overlays.

They are provided in `/containers/overlays/`. In the same manner as writable overlays mentioned above, they are mapped "over" the base container via OverlayFS and record any filesystem level modifications to the original container image, i.e., a set of additionally installed system-packages that do not interact with Spack-based packages. As most changes between container releases occur in the Spack-install, overlays can be created once and continue to work with newer container releases. The overlay only needs to be recreated if the container base image gets updated or the set of installed system packages from container and overlay start to overlap. As shown in Figure 8.4, we currently provide TeX Live[85] for both Debian Stretch and Debian Buster-based containers.[86] Therefore, depending on the Debian-release the container is based[87] on, users have to select the matching overlay. A work-session permitting the compilation of LaTeXdocuments can hence be invoked via:

[84]Spike-Based Sampling – a library for fast Neural Sampling, [Breitwieser et al., 2020; Breitwieser, 2015]
[85]`https://www.tug.org/texlive/` (visited on 2021-02-09)
[86]`https://wiki.debian.org/DebianReleases` (visited on 2021-02-09)
[87]Containers older than 2021-05-28 are based on Debian Stretch, newer ones are based on Debian Buster.

```
$ singularity shell --app dls
↪ --overlay /containers/overlays/2020-05-28_buster_texlive.img
↪ /container/stable/latest
```

As the example shows, there is support to combine apps with overlays. One potential downside is that Singularity, at the time of writing, only supports a single overlay. However, as there is currently no need to combine overlays, this is not a limitation.

ASIC-Container Even though the original design goal was to have all software environments in a single container, sometimes hard rules have to be bent. In addition to the default visionary container, we distribute an ASIC-specific container. It serves only one purpose: Supply an environment to run a series of hardware development tools provided by companies such as Cadence Design Systems, Inc.,[88] Synopsys technology[89] or Xilinx, Inc..[90] These include iMPACT,[91] Vivado Design Suite,[92] and PetaLinux.[93] Typically, these tools are supported to run in a CentOS-based environment with a couple of system packages installed. However, the ASIC-cluster, that is physically separated from the Electronic Vision(s)-cluster, did not run a recent enough CentOS-version, a problem quite typical in HPC-environments (see Section 4.2). A first attempt was made to integrate the required ASIC-environment into the visionary container, but since the visionary default container is Debian-based, a split turned out to be the less time-consuming option. Since ASIC tools often underlie licensing restrictions, container images containing them or some hard to install support libraries cannot be made publicly available.

Hence, as shown in Figure 8.4, ASIC-containers are built (via `yashchiki`[94], see Section 8.3) and deployed alongside regular visionary container, but evolve at a glacial speed due to their limited scope. At the time of writing, the latest ASIC-container consists of a CentOS 7 base image with 94 additional system packages and three Python-based linters (installed via Anaconda[95]). Essentially, they represent a system upgrade for systems that are not yet at the required OS[96] level to run ASIC tools, a typical application for lightweight containers. Due to their limited scope and the aforementioned licensing issues, ASIC-containers are not made available publicly.

Overhead induced by Singularity Spawning a container introduces overhead in process execution. We estimate this overhead by performing timing analysis using `hyperfine`.[97] The results are shown in Listing 4. We see that wrapping a with Singularity

[88] `https://www.cadence.com` (visited on 2021-02-08)
[89] `https://www.synopsys.com` (visited on 2021-02-16)
[90] `https://www.xilinx.com/` (visited on 2021-02-08)
[91] `https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/pim_c_overview.htm` (visited on 2021-02-08)
[92] `https://www.xilinx.com/products/design-tools/vivado.html` (visited on 2021-02-08)
[93] `https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html` (visited on 2021-02-08)
[94] from Russian, ящики, meaning boxes or “Schachtel” in German, [Vision(s), 2021]
[95] Anaconda Software Distribution, [CA2016]
[96] Operating System
[97] `https://github.com/sharkdp/hyperfine` (visited on 2021-04-11)

```
Benchmark #1: /bin/sleep 0.1
  Time (μ ± σ):    104.4 ms ±   1.6 ms [User:  0.0 ms, System:  0.0 ms]
  Range (min/max): 101.1 ms / 110.9 ms  1000 runs

Benchmark #2: singularity exec --app dls /containers/stable/latest
 ↪  /bin/sleep 0.1
  Time (μ ± σ):    322.8 ms ±  16.0 ms [User: 43.2 ms, System: 45.9 ms]
  Range (min/max): 278.6 ms / 375.6 ms  1000 runs

Benchmark #3: <in-container> /bin/sleep 0.1
  Time (μ ± σ):    104.4 ms ±   1.6 ms [User:  0.0 ms, System:  0.0 ms]
  Range (min/max): 101.1 ms / 110.0 ms  1000 runs
```

Listing 4: Estimating computational overhead of Singularity invocation using a simple benchmark. We perform a simple sleep for 100 ms in three ways: On its own (#1), with container invocation (#2) and spawning the container and performing the whole benchmark in an already started container (#3). Each command is executed 5 times in order to "warm up" the system. This is only relevant for the Singularity based calls because it ensures the image file is cached. Furthermore, we allocated the compute node in exclusive mode in order to eliminate deviations from other user jobs executing at the same time.
We see that executing within a newly spawned container adds a slight delay of about (220 ± 16) ms. About half of which is actually used to facilitate containerization in user and kernel space. While noticeable in direct comparison, it does not make the system feel unresponsive. As shown in benchmark #3, once users have entered the container, there is no additional overhead compared to execution outside.

command adds a slight delay of a few hundred milliseconds, typically (220 ± 16) ms. While above the threshold of human perception it is still too low to cause irritation [Doherty et al., 2015]. This is important when wrapping cluster job submission commands in Singularity calls (see Section 9.2.1).

However, Listing 4 benchmark #3, also illustrates that once we are inside a container there is no measurable difference in execution time compared to outside. This is expected given its technical implementation (cf. Section 8.2.2). Users are therefore advised to not wrap fast-executing commands or binaries with a Singularity-call if they are expected to be executed several times a second.[98] Instead, interactive sessions should be spawned via `shell` subcommand (cf. Figure 8.3).

8.2.4 Singularity-related Problems and corresponding Solutions

For most intents and purposes, Singularity was ready-to-use as a container runtime. However, we identified the following issues:

[98]In Chapter 9, we will still use this technique to provide transparent access to cluster job transmission via `clusterize` (cf. Section 9.2). However, the execution time of a single command far exceeds the overhead introduced by Singularity which – along with the other benefits – makes it acceptable to use.

Static Configuration Singularity is configured via a set of static configuration files, typically residing under `/etc/singularity`.[99] Configuration options range from which namespaces to use for process isolation, over automatic bind-mounts from host filesystem into containers to fine-grained control over network interfaces or how GPUs are attached to containers. One drawback is that this configuration is static, i.e., it applies to all containers executed on the host. In a distributed setting, where many hosts are booted from the same NFS-tree, this can lead to problems because there is no way to dynamically adjust settings like bind-mounts depending on the host's environment.

Consider the following example: Some folders only exist if a corresponding service has been started previously. On nodes where the service has been started, it should be available from within the container, hence a set of folders needs to be bind-mounted into the container. If these folders are hard-coded into the configuration file, however, Singularity will throw an error on all nodes where the paths do not exist. Another example is customizing bind-mounts depending on the location of the binary executed within the container (e.g., bind-mounting different folders containing support libraries from the host system to the same folder within the container). Both examples are needed when operating disjoint production and testing environments of our cluster workload manager deployment of Slurm (see Section 9.1). The solution is the aforementioned `clusterize`-tool, which dynamically wraps the call to singularity and conditionally generates a set of CLI-arguments. It is discussed further in Section 9.2.

Container Build Time of SIF-format SIF [Godlove, 2019] offers more security-related features by integrating hashes of all components, enabling more fine grained control about which user is allowed to execute what container image and integrating with security-related kernel features such as SELinux, AppArmor and seccomp within the container. Furthermore, it the option to have a modifiable overlay directly integrated into the image. However, at the time of writing, we have not yet found a compelling reason to switch towards it: Building SquashFS-based containers with Singularity `2.x` proved to be faster because in Singularity `3.x` the whole sandbox is copied prior to image creation, which adds a significant time penalty when building larger images with many small files. Furthermore, another show-stopping bug was the fact that all files in SIF-formatted containers created from sandboxes used to be owned by `root`.[100] Though the bug has been fixed in the meantime, the container build routine has not been revisited since.

Nesting via Trusted Containers In general, launching "into" a container is an all-or-nothing move. And rightfully so: Once a process is isolated in a container, it should remain contained, especially if we isolate persistent services running third-party code that is only partially trusted. However, as always, every rule has its exceptions. The exception here is the fact that we moved the complete work space of users *into* into the visionary container. All tasks that users previously performed on the frontend are now performed within the container and therefore have to be ensured to be operational.

The first use-case is user-workflows that span multiple environments. Figure 4.1 sketches an example for this type of workflow. If users are unable to "leave" the container

[99] `https://sylabs.io/guides/3.7/admin-guide/configfiles.html` (visited on 2021-02-02)
[100] `https://github.com/hpcng/singularity/issues/2860` (visited on 2021-02-01)

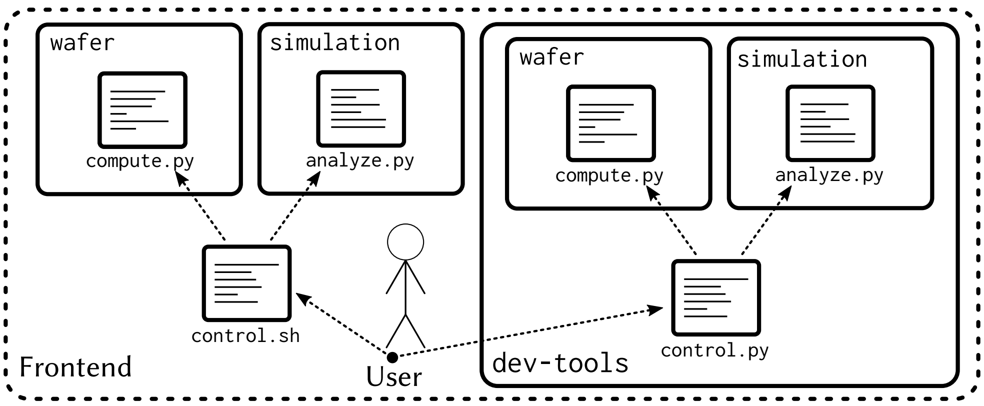


Figure 8.5: Example for nested container workflows where data generation happens in a different environment than analysis and plotting. Different environment are distinguished via boxes. For various reasons (age, supported versions of dependencies, etc.), these environments can be mutually incompatible.
Left: Without the ability to nest containers, a workflow involving several environments has to be controlled one level of abstraction above the jobs performing the actual work, i.e., the front-end. This causes a lot of friction because the front-end machine's environment is very bare-bones by design, requiring users to write their control logic in very crude shell-scripts without any helper utilities.
Right: With the controlling process already spawned inside a containerized environment, the user has much more freedom writing the actual control loop, saving time employing utilities. Only the data generation is off-loaded into a different environment, possibly in a different container. In particular, nested containers are required to provide easy access to job submission binaries that are executed in a different environment, as detailed in Chapter 9 and Section 9.2. Refer to the text for details.

– which acts their current working environment – to offload some steps of their workflow into another environment, they are forced to write their control loop in the next higher layer of abstraction, i.e., the cluster front-end. The cluster front-end's own environment is very bare-bones by design – after all, users are supposed to enter the visionary containers to access the full plethora of tools. Hence, all users reliant to perform work in several mutually incompatible software environments would be forced to write very crude shell scripts to execute their job steps.

Another example where nested containers are necessary involves deployed utility binaries, such as our cluster deployment (discussed in detail in Chapter 9). Here, binaries are built and linked against a specific container image (and the libraries therein). Since the visionary container image and the cluster environment should not be required to be updated in lockstep, deployed binaries should continue to "just work" even though the container image, i.e., the workspace environment, gets updated to contain potentially now-incompatible libraries.

To alleviate this issue, we introduced a new concept: *trusted containers*. These are containers that have been vetted by administrators not to contain any potentially harmful binaries. For obvious reasons, any container uploaded by users would be untrusted. At the Electronic Vision(s) cluster, any deployed container image under `/containers` (see Figure 8.4) is trusted. A trusted container differs from an untrusted one in the fact that it does not relinquish all its capabilities upon invocation:

- Untrusted containers mount the `root`-filesystem with `NOSUID` option which prevents any `setuid`-related activity. As mentioned above, Singularity requires `starter-suid` to acquire `root` permissions for container setup-related operations.
  Trusted containers omit the `NOSUID`-mount option.
- In order to prevent the contained process from *ever* gaining new privileges, untrusted containers `PR_SET_NO_NEW_PRIVS`[101] is set which is inherited by any child processes and cannot be unset.
  Trusted containers omit setting `PR_SET_NO_NEW_PRIVS`.
- The bounding capabilities set,[102] i.e., the ever-decreasing set of capabilities a process can possibly attain when `execve`-ing,[103] is empty in untrusted containers by default. As of `v3.0`, Singularity offers options[104] for untrusted container to retain some capabilities such as `CAP_NET_RAW` which is needed by `ping`[105] to access `RAW` and `PACKET` sockets.
  Trusted containers can have an adjustable set of capabilities remaining in the bounding set. Please note that these capabilities are not in effect when the process is spawned, but have to be attained by executing a `setuid` binary within the (trusted) container environment.

To users, the only difference when dealing with trusted containers is is a restriction in bind-mount options. Otherwise, they could replace configuration files of `setuid`-binaries such as `sudo`.[106] On the Electronic Vision(s) cluster, users typically do not need to perform bind-mounts themselves, but rather have them done via transparently wrapped binaries via `clusterize` (see Section 9.2). Since the kernel imposes a limit of 32 nested user namespaces,[107] this is also the maximum nesting depth supported by this approach.

A first prototype implementation[108] is currently employed at the Electronic Vision(s) cluster. However, some security concerns currently prevent pushing these changes upstream. For example, the current implementation *might* be susceptible to TOC/TUO[109] attacks when wrongly configured. TOC/TUO is a race-condition introduced by the time delay between a program checking the state of some external resource and then using it. In the easiest example, an attacker switches out a symlink to point to another file between check and usage (i.e., reading its contents), thereby causing unintended behavior (in the worst case gaining root-rights). The protection against this class of vulnerabilities is to open the file, perform all checks *while* the file is opened and then continue operating on the opened file *descriptors*[110] *only*. As the Singularity codebase was not expected to

[101] `https://man7.org/linux/man-pages/man2/prctl.2.html` (visited on 2021-02-10)
[102] `https://man7.org/linux/man-pages/man7/capabilities.7.html` (visited on 2021-02-10)
[103] `https://man7.org/linux/man-pages/man2/execve.2.html` (visited on 2021-02-10)
[104] `https://sylabs.io/guides/3.0/user-guide/security_options.html` (visited on 2021-02-10)
[105] `https://man7.org/linux/man-pages/man8/ping.8.html` (visited on 2021-02-10)
[106] su "do", `https://www.sudo.ws/` (visited on 2021-02-10)
[107] `https://man7.org/linux/man-pages/man7/user_namespaces.7.html` (visited on 2021-05-03)
[108] `https://github.com/hpcng/singularity/pull/2729` (visited on 2021-02-10)
[109] Time-Of-Check to Time-Of-Use
[110] In Linux, the opened file descriptors of any process can be found under `/proc/<pid>/fd/<id>`, where ids `0`, `1` and `2` typically correspond `stdin`, `stdout` and `stdout` while all further correspond to opened files, sockets, fifos, etc.

run with potentially elevated rights, susceptibility to TOC/TUO was no concern at the time. Hence, proper implementation potentially requires further changes to the upstream codebase.

These concerns either need to be investigated in more detail or replaced with a more conservative feature-set, outlined below. Due to the fact that the Electronic Vision(s)-cluster is not publicly accessible and all its participants (numbering in the double digit regime) are vetted personally, these security aspects are less of a concern: Typically we want to prevent users from accidentally shooting themselves (or others) in the foot, but do not expect users – once they have successfully logged in – to be actively malicious. However, this is not the case at most HPC-sites which is why these potential security vulnerabilities have to be eliminated in the long run. A rewrite of the extensions as a plugin is planned but did not yet commence. It would need to include a switch from a blacklist prohibiting certain bind-mounts to a whitelist of permitted bind-mount points for safety reasons. Additionally, overlays and other possible ways of binding untrusted (i.e., user-generated) files into the container need to vetted prior inclusion in trusted containers. In case of overlays, they need underlay the same restrictions as trusted containers, i.e., they are only eligible for inclusion when loaded from trusted paths.

A different approach would be to use the `--fakeroot`[111] feature introduced in Singularity `v3.3`. Here, the user namespace[112] is used to map `UID 0` (i.e., `root`) inside the container to a different `UID` outside of the container. This means that *inside* the container `root` rights can be attained via means such as `setuid`, but outside of it all actions are performed as a non-`root` user. As long as the mountpoint containing other container images remains reachable from withing the container, launching another container might be facilitated this way. However, same as the other `root`-less alternatives discussed in Section 4.4.3, they ignore the fact that loop devices still require `root`-rights to be mounted. Ergo, mounting loop devices would still need to be performed by an external service with `root`-rights running outside the container-stack. We could avoid the Docker-like daemon by unpacking images. The container is rather large with many small files. Unpacked, its roughly 2 000 000 files take up 35 GiB, i.e., unpacked containers take up more storage by a factor of five. Hence, unpacking is not an option.

To summarize: Our approach to nested containers is working and a vital component to facilitate ease of use for distinct software environments. There are no fundamental technical hurdles to contribute this approach back upstream with a sound security concept. The only reason it has not been completed so far is a lack of developer time.

Trusted Containers in Python: `veer` Spike-Based Sampling – a library for fast Neural Sampling [Breitwieser et al., 2020; Breitwieser, 2015], supports performing simulation steps in a different subprocess to ensure all compute resources are freed afterwards and no unclaimed objects continue to consume CPU cycles and/or memory. This procedure was

[111]`https://sylabs.io/guides/3.3/user-guide/fakeroot.html` (visited on 2021-02-10)
[112]`https://man7.org/linux/man-pages/man7/user_namespaces.7.html` (visited on 2021-02-11)

necessary because, in the past, some[113] simulators[114] had[115] problems[116] with[117] repeated[118] runs.[119]

Using trusted containers, it was extended to allow performing compute steps in another container. Because of its general usefulness, the functionality was extracted into `veer` [Breitwieser, 2020]. A trivial hello-wold example can be found in Listing 5. `veer` works by replacing the function with a helper construct that spawns a new Python process (optionally in a new container). The function arguments are then pickled and send via local TCP[120]-socket. In the subprocess, the wrapped function is imported from the original model and executed. After execution, the return values are sent back over the same socket, again pickled.

```python
import os
import subprocess
import veer

@veer.in_container(image="container.img",
                   app="special-environment")
def foobar():
    return subprocess.check_output(["lsb_release", "-a"]).decode("utf-8")

if __name__ == "__main__":
    print("Main process:")
    print(subprocess.check_output(["lsb_release", "-a"]).decode("utf-8"))
    print("Child process:")
    print(foobar())
```

Listing 5: Trivial example for running single Python functions in a different container with `veer` via a simple decorator. If host and container image are of different Linux distributions, executing `lsb_release -a` inside the function will yield a different output.

8.3 Container Build Process Automation: yashchiki

After introducing our dependency management in Section 8.1 and how it is deployed for and used by users in Section 8.2, this section describes `yashchiki`,[121] the solution developed throughout this thesis to handle building of visionary containers in a developer

[113] https://github.com/NeuralEnsemble/PyNN/issues/44 (visited on 2021-02-10)
[114] https://github.com/NeuralEnsemble/PyNN/issues/45 (visited on 2021-02-10)
[115] https://github.com/NeuralEnsemble/PyNN/issues/156 (visited on 2021-02-10)
[116] https://github.com/NeuralEnsemble/PyNN/issues/217 (visited on 2021-02-10)
[117] https://github.com/NeuralEnsemble/PyNN/issues/225 (visited on 2021-02-10)
[118] https://github.com/NeuralEnsemble/PyNN/issues/487 (visited on 2021-02-10)
[119] https://github.com/NeuralEnsemble/PyNN/issues/499 (visited on 2021-02-10)
[120] Transmission Control Protocol, [RFC793]
[121] From Russian, ящики, meaning boxes or “Schachtel” in German. Name coined by Dr. Eric Müller.

friendly way. Users are able to submit updates to the containers via Gerrit (see Section 7.2). yashchiki takes care of building, verifying and deploying containers. The build is controlled by a set of keywords in Gerrit comments. By employing several layers of caching, we are able to reduce the build time of a single container from over 25 hours to under 4 hours. Currently, yashchiki is set up as a layered set of bash-scripts, controlling the build of both Spack deployment and encompassing Singularity container.

8.3.1 Container Structure

The most important parts of the internal container structure are shown in Figure 8.6. In accordance to [Quinlan et al., 2004] – and to avoid any potential conflicts with the Debian base install – all custom software components for the visionary container are located beneath /opt.[122] The Spack deployment is located at /opt/spack, with all installed software components installed with hashed-prefixes under /opt/spack/opt/spack (as described in Section 8.1). /opt/spack_views then contains sets of "virtual" combined installations, i.e., Spack views. As explained in Section 8.1, a Spack view corresponds to a "virtual" installation of several packages to the same prefix, however, each file is merely a symbolic link into its corresponding Spack install. There is one view for each environment provided.

The main entry point for the container is via Singularity commands exec or shell. The optional --app <app> parameter causes a corresponding environment setup script residing below /scif to be sourced. At time of writing, /scif is treated as implementation detail of Singularity and not interacted with by yashchiki in a way. The environment script in the container app then ensures that the virtual file tree of the corresponding Spack view under /opt/spack_views/visionary-<app> is integrated into the environment as if it was part of the regular root-tree at /.

The spack views themselves reconstruct the folder structure within the corresponding visionary- Spack package and all its dependencies. All files are then symbolic links to their hash-prefixed Spack deployment under /opt/spack/opt/spack. This means that the naming structure within spack views is relatively stable between visionary container releases (file renaming within packages notwithstanding), whereas prefix-hashes for packages change as soon as one dependency changes.

More low-level tools can be loaded by sourcing corresponding scripts from /opt/init:

```
$ source /opt/init/modules.sh
  (or)
$ source /opt/init/spack.sh
```

They include Spack itself and the GNU modules-support. Spack is useful when creating custom environments by manually loading packages on the command line or creating

[122]https://www.pathname.com/fhs/pub/fhs-2.3.html#OPTADDONAPPLICATIONSOFTWAREPACKAGES (visited on 2021-02-15)

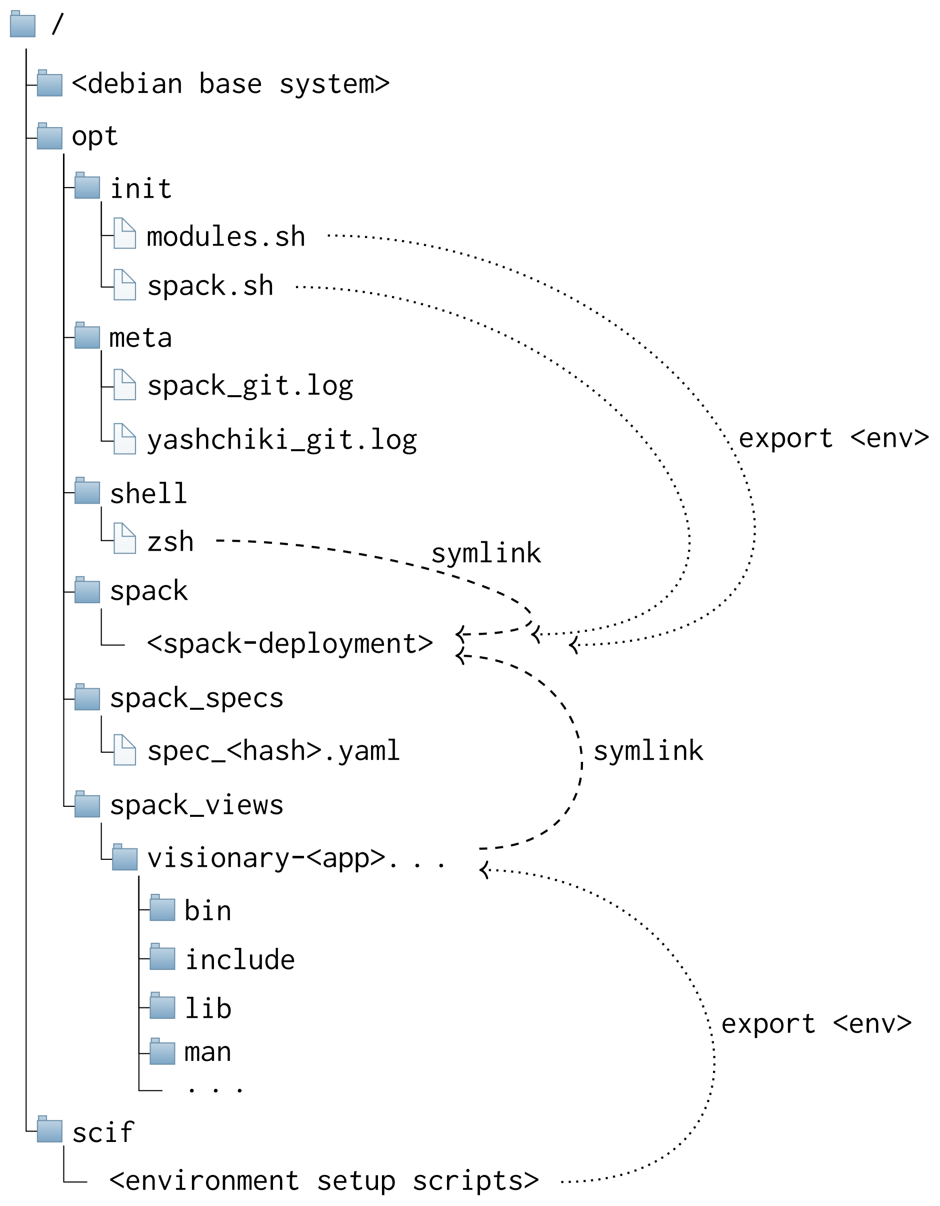


Figure 8.6: Overview of visionary container file structure. Besides a base Debian-system, all further software packages reside in `/opt`. Here `/opt/spack` contains the actual installed software packages in a Spack deployment. `/opt/spack_views` then consists of a set of “virtual” combined installations where each file is a symbolic link into the corresponding Spack install. There is one of these so-called Spack views for each environment provided. Under `/scif`, Singularity stores scripts for each app that map apps to their corresponding Spack view by integrating them into the environment. More low-level tools can be loaded by sourcing corresponding scripts from `/opt/init`. Please refer to the text for further details.

GNU modules-based load scripts,[123] as is done in Section 9.1 for automated cluster deployment. Furthermore, GNU modules are important to access nightly deployments of all visionary software stacks (such as the BrainScaleS-2-stack described in Chapter 6)

[123]Note that due to the fact that all packages within spack are installed by hash-dependent prefixes, these manually created load scripts might have to be regenerated for every container iteration.

which are a great alternative to self-compilation for experimenters that are not performing low-level software work. Please note that even though the nightly deployed software stacks are pre-compiled, they still rely on the software environment within visionary containers to execute.

Custom shells for working within the container can also be provided. They can be specified instead of the default (bash) like so:

```
$ singularity shell -S /opt/shell/<shell> /containers/stable/latest
```

Same as the convenience scripts provided in /opt/init, they are auto-generated on container creation to link to the specific Spack deployment. Currently, there is support for Spack-built zsh.[124]

Finally, for documentation purposes, the container includes all fully concretized Spack specs built into the container under /opt/spack_specs as well as the git log of both Spack and yashchiki repositories at the time of building under /opt/meta. This information makes it easier to trace container state and debug problems after the fact. Since this information is enough to re-create the container from scratch, it is deployed along with each container image under /container/dna, see Figure 8.4.

8.3.2 Update Schedule

Due to the time intensive build process, the visionary container is updated on an as-needed basis in a rolling release schedule. As explained in Section 8.1, different software environments are tracked via Spack visionary- meta-packages that are made available to users in different container-apps. Any changes to environments are submitted as changesets to the Spack-repository: These include new dependencies, updated versions of software packages, and merges of upstream[125] changes into the local visionary branch. Any changes to container layout, provided apps, system dependencies or version pinnings are submitted to the yashchiki-repository.

Due to the fact that new images take on the order of hours to be built, it is infeasible to trigger automated Jenkins builds for every submitted changeset. Instead, users are advised to group several changesets together. If there are dependencies between changes from both repositories, they can be expressed via Depends-On: notation (discussed in Section 7.2.1) as usual. Jenkins builds of new, so-called *testing* containers can be triggered by commenting BUILD_THIS with a set optional parameters[126] in the Gerrit toplevel changeset in either repository. During the build process, yashchiki then identifies all changesets that are being built and notifies each individually by commenting the deployment path of the testing container. This is useful especially in the case of several disjoint changes being built into the same container. Each change might be of interest to different users that

[124] Z SHell, https://zsh.org (visited on 2021-02-16)
[125] https://github.com/spack/spack (visited on 2021-02-15)
[126] A full list of optional parameters can be found at Section 8.3.7.

might otherwise not be aware of the testing container containing “their” particular change. Alternatively, container builds can also be triggered manually in Jenkins.

As shown in Figure 8.4, new testing containers are deployed to `/containers/testing`. The naming scheme is composed of the toplevel changeset and patchset number, date and current iteration number.

At the time of writing, we build in a virtual machine with 16 logical cores of an `Intel Xeon CPU E5-2660 v4 @ 2.00GHz` with 64 GiB of RAM[127] evenly distributed across two Jenkins executors. Given additional compute resources, it is trivial to extend `yashchiki` to more executors.

8.3.3 Buildcaches: Improving Time till Deployment

For performance reasons, `yashchiki` implements its own buildcache. While Spack does feature its own buildcache[128] implementation, it was evaluated to be infeasible for `yashchiki`. As described in Section 8.1, Spack makes extensive use of `RPATH`s, i.e., hard-coded paths in binaries and libraries where other shared objects files are to find in order to make them environment-independent when executed. Since these paths are absolute, `RPATH`s are replaced by placeholder tokens upon addition to the buildcache. When extracting, Spack again search and replaces these tokens with the new target location. In total, *every* file gets parsed at least twice, in Python, i.e., thanks to the GIL[129] in a single thread and therefore with abysmal performance.

Leveraging the fact that `yashchiki` always builds in the same location, there is *no need to rewrite* all files when moving in or out of the buildcache. All packages in all container image iterations will always reside under `/opt/spack/opt/spack`.[130] Hence, it is safe to archive packages directly if we only ever extract them to the exact same location in another image. Packages are stored as compressed archives identified by their computed hashes (cf. Section 8.1). This is safe because as soon as a package is modified by a changeset – be it a new version added or a new configuration variant – its hash changes and therefore also the hash of all packages depending on it. Furthermore, it ensures that we can be sure that if a given package hash is in the cache, it has previously been part of a successful container build and was included along with all its dependencies: We do not need to perform an explicit check. However, there are utility scripts to ensure buildcache integrity. So far, we have not yet encountered an invalid buildcache state.

The only notion of care needs to be exercised when a Spack package gets modified in a way that does not alter either its variant or version information or its dependencies. A prime example would be to modify options passed to the compiler during the configure step. While in this case the content and behavior of the package does change, the Spack-hash

[127]Random-Access Memory

[128]`https://spack.readthedocs.io/en/latest/binary_caches.html` (visited on 2021-02-17)

[129]Global Interpreter Lock, `https://wiki.python.org/moin/GlobalInterpreterLock` (visited on 2021-02-17)

[130]If the Spack deploy location was to change, this of course would require one container build from scratch.

does not and so a new yashchiki build would happily use the already existing package and not rebuild it. Hence, it is considered bad practice – especially build options should be reflected by variants – and falls into the category of human error, that is discovered immediately upon evaluation. In case the modification cannot be avoided in any way, the package has to be manually deleted from the cache.

Since packages are completely independent of each other, we can compress/extract as many in parallel as our compute resources effectively allow for. Extracting all relevant packages from buildcache takes 20–30 min, whereas previously Spack's internal buildcache took several hours. As a side-note, at the time of writing, Spack's verification of installed packages takes an order of magnitude longer than the buildcache extraction: On average around 4 hours. This is one of the largest time sinks during the container build process, as is discussed below.

8.3.4 Debugging Container Builds

Typically, when users submit new changes to the container they add or adjust the set of packages within each provided environment. If packages are already present in Spack they contain dependency information from upstream. Hence, if any incompatibilities are detected during concretization, the build will fail early, i.e., in the order of minutes instead of hours (refer Section 8.3.4). Users can look up what concretization errors were encountered, adjust their changeset accordingly and resubmit.

If changes are more involved, e.g., new packages are being added to the Spack repository that are not present upstream or their build behavior is adjusted, concretization can be fine but then a package fails to build. In this case yashchiki provides means to keep all packages built up until that point and – after adjusting Spack package definition – resume the build where it left of. Alternatively, it is also possible to debug building of Spack packages within an older iteration of the container by "chaining" the local Spack checkout ontop the container-provided one (see Section 8.1.4).

Finally, as all technologies involved such as Spack or Singularity are still actively being developed, we sometimes have to make adjustments to the full container building pipeline. Whenever we merge upstream changes from Spack (see Section 8.1.4), we possibly have to adapt to upstream changes, be it in the conventions of how packages are specified or data formats such as spec files we rely upon. This is one of the areas in which the container build process is still a bit too involved: Like above, we could apply the debugging loop as for single packages, but depending on the type of problem we are attempting to trace the feedback loop slows to a crawl rather quick. Especially problems that involve the interaction of yashchiki and Spack can be involved and hard to track down with debug output alone.

Instead, we want a way to interactively debug the container as it is being built. Currently, this is only possible by manually ssh-ing into the build server and executing each build step manually. The fact that, for technical reasons, the container is built within a nested set of shell scripts (see Section 8.3.5), does not help to simplify matters. It requires root access and extensive knowledge of the build environment, excluding many volunteers from helping. Nevertheless, it is an option of last resort.

8.3.5 Container Image Build Stages

The container build process is separated into several distinct phases. Unless otherwise noted, all phases are performed in sequence. We distinguish between two main types of container builds: *stable* container builds that should never fail and *testing* builds of newly proposed changesets in Gerrit. As discussed above, testing builds are triggered via Gerrit or alternatively directly via Jenkins, whereas stable builds can be triggered via the corresponding Jenkins-job only. Newly built stable containers are announced in the corresponding Mattermost channel (cf. Figure 7.1).

Error handling and failed caches If any build step fails, we abort the build immediately to avoid dealing with ill-defined state. In this case, an error handling step is executed, preserving any Spack packages that have already been built successfully in a so-called *failed cache*. The location of this failed cache is included in the error message commented in the corresponding Gerrit changes, giving users the ability to select failed caches from specific patch levels/build iterations. Please note that these failed caches are never used for stable container builds. In order to ensure a deterministic build, a new stable build should always build everything added since the last stable build. For this reason, the "official" buildcache is updated after stable container builds only.

Environment Validation The first step after job startup is to validate the given configuration. Since Jenkins jobs are mostly controlled via environment variables, we ensure that the environment is in a valid state. If we encounter invalid state, for example if `yashchiki` is instructed to build a stable container from a Gerrit changeset, we abort the build immediately.

Furthermore, we perform additional set up as specified by key-value pairs extracted from the triggering Gerrit comment (see Section 8.3.7). These range from adjusting the verbosity-level over cache-related settings to specifying which state to check out for the Spack repository. The latter is meant as an alternative to commit-specified `Depends-On:` to distinguish the case in which changesets are built together but do not actually depend on each other in a functional sense.

For testing builds, we check if any failed buildcaches exist from previous builds of the same changeset. If so, we use the latest failed buildcache instead of the default buildcache for this build. As described below, when triggering a build, users have options to specify which buildcache to use or even disable the failed-buildcache feature altogether.

Checkout of Spack Stable builds always check out the current `HEAD` of the `visionary` branch in the Spack repository. Testing builds are more flexible. The commit to check out can be specified in the following ways, ordered by priority:

i) Environment variables set directly when manually triggering the job in Jenkins.

ii) Parameters specified in the triggering Gerrit comment (described in detail below).

iii) Depends-On specified in the triggering commit's message, indicating a Gerrit change number to check out.

Fail early: Concretization In order to not waste time with builds that are doomed to fail, all visionary- packages are concretized beforehand. If any package should fail to concretize, the build is aborted *immediately*. In that case a Jenkins artifact[131] is created that contains the concretizing log, including why it failed. Since visionary- packages have many dependencies (see Section 8.1.4), concretizing them takes several minutes. Therefore, all packages are concretized in parallel using the 8 cores available to each executor. The result is then stored to be reused several times in later steps. In the finished container, this information is made available at /opt/spack_specs.

Fetching Archives All software archives required for building are downloaded. In order to not strain remote mirrors by downloading all packages for all builds, we extend Spack's download cache to be persistent across build jobs. Prior to fetching, all packages from the persistent download cached are hard linked into the Spack repository. Then all missing archives are fetched. Afterwards, all new additions are copied back into persistent storage.

Utility Deployment If we build a stable container, the current set of utilities is deployed to /opt/containers/utils. Each utility contains an auto-generated header to reference the commit they were built from as well as deployment date. Available utilities include cleaning helper that can remove testing containers and build remnants that are not needed anymore, i.e., from changesets that are merged or abandoned. Additionally, there are tools to extract a full build cache or dna (essential information to re-create the container) from an existing image. The full list can be found in Section 8.3.8.

Build Sandbox The container is first built in a sandbox. To that end, a Singularity recipe is created dynamically. Besides definitions for all apps, it includes instructions to perform the following steps:

- Prepare a Debian-based Docker image.
- Install all direct dependencies of Spack.
- Extract all packages already present in the build cache.

  Here we can reuse the concretized information generated earlier to quickly identify all required hashes.
- Discover extracted packages via Spack.
- Perform Spack installation of any packages not present in the buildcache.

[131] An artifact is a file that gets stored per build.

- In parallel to steps above: Install system dependencies.

  Due to the fact that Spack does not make use of existing system packages unless explicitly instructed to do so, we can safely perform the last step in parallel. System dependencies are installed from Debian repositories and typically constitute standalone tools that no package from Spack interacts with.
- Checkout, build and install a modified version of Singularity to include modifications described in Section 8.2.4. At the time of writing, we build all available software for the oldest CPU architecture used in the cluster: Sandybridge.[132] This ensures that no unsupported instructions are used in the generated binaries and container images can be deployed cluster-wide.

  Because `root` permissions are needed to create `setuid` files required for container startup (refer to Section 8.2.2), Singularity currently remains the only package build from source not managed by Spack in the visionary container. However, it is planned to deploy Singularity via Spack eventually.
- Finally, apply some system-level modifications to the final sandbox. At the time writing, they include modifying low-level system header files to honor custom preprocessor macros. Modifications are stored as separate patch-files in the `yashchiki` repository and applied in order. Additional modifications can therefore directly be added as new patch files.

Please note that the container build itself needs to run with `root`-rights which we permit via specifically tailored `sudo`-rules. They allow the Jenkins user to become `root` for the build process. Within the container, all Spack-related operations are then performed as a newly created `spack` user.

Create Image from Sandbox Once the sandbox is created, it is compressed into a single SquashFS image. As explained in Section 8.2.4, we did not yet switch to SIF-format for performance reasons.

Update Build Cache In case we are building a stable container, we update the buildcache via one of the just deployed utility scripts. It avoids copying (and compressing) all packages by first identifying which package hashes are not yet contained in the buildcache. Then, only the difference is transfered.

As mentioned above, we emphasize that only updating the “official” buildcache on stable builds ensures a deterministic build process. Failed caches, while useful in iterative debugging of build problems, are never used for stable container builds.

Verification of built Container Image As discussed in Chapter 7, every software repository defines Jenkins jobs that verify its functionality with regards to new software changes. The same principle is applied to containers. Therefore, the new container image is verified by triggering a set of Jenkins jobs. Each job is executed with their respective “stable” `HEAD` inside the new container image. For stable container builds these are expected to

[132]`http://ark.intel.com/products/codename/29900/Sandy-Bridge` (visited on 2021-04-10)

always succeed. For *testing containers* it offers the crucial advantage of *spotting potentially breaking changes* ahead of time.

Clean-Up After *any* build – successful or not – the workspace is cleaned so that build artifacts and temporary files do not pile up. The cleanup includes any temporary build data as well. Merely download- and buildcache are exempt. If any build problem is more severe, it should be debugged in a local Spack deployment, possibly within an older iteration of the visionary container so that its Spack instance can be chained (refer Section 8.1.4).

ASIC container The ASIC container is built in much the same way as the regular visionary container, but omits all Spack-related steps. As shown in Figure 8.4, it is deployed in the same locations as visionary containers for both stable and testing builds, respectively.

8.3.6 Pinning Versions

As a workaround to problems in the concretizer (that we reported[133] upstream), yashchiki implements measures of pinning versions for large quantities of packages. While Spack does have functionality to pin versions of packages deployment-wide, we needed to be able to pin versions for each visionary- package independently: As described in Section 8.1.4, our environments are split in whether they support Python version 2 or 3. When support for Python 3 was integrated into Spack, the current implementation of the concretizer was unable to compute the last version of packages to still support Python 2 because it would assign versions to packages too greedily. Hence, all packages for environments depending on Python 2 had to be manually pinned. Pinned version information are stored in package-specific files annotated as specs. During container build, if present, they are added dynamically to the concretization calls.

8.3.7 Full list of supported Triggers in Gerrit Commits

The full list of Gerrit comment triggers supported by yashchiki is as follows:

`BUILD_THIS`	Start a build with this change as toplevel.
`WITHOUT_FAILED_CACHE`	Do not attempt to locate the latest failed cache created during a previous build attempt of the same changeset, but use the “regular” buildcache instead. The user can also supply `WITH_CACHE_NAME=<name>` to specify a different build cache to be used for this build.
`WITH_CACHE_NAME=<name>`	Use a specific buildcache on conviz (the container building machine) instead of the default one. Failed buildcaches are valid targets.

[133]https://github.com/spack/spack/issues/12431 (visited on 2021-02-18)

`WITH_SPACK_{CHANGE,REFSPEC}`	Since oftentimes `yashchiki` and Spack changes are tested together but have no real dependency on one another, previously the `Depends-On` mechanism in the commit message was “misused” to build a container with a specific Spack and `yashchiki` changeset. Hence, `yashchiki` provides the options to specify: • `WITH_SPACK_CHANGE=<change-num>` to use the latest patch set of the given Spack changeset for the build. • `WITH_SPACK_REFSPEC=<refspec>` to specify a complete Spack-repository refspec[134] that is to be used for this build to have full control over which changeset/patch level to build. These take priority over commit-specified `Depends-On:` and are mutually exclusive with Jenkins-specified build parameters since each build gets either triggered manually in jenkins or via Gerrit.
`WITH_DEBUG`	Enable debug output.

8.3.8 Utility scripts for maintenance

There are several utilities deployed alongside containers (see Figure 8.4) that are mainly of interest for system administrators.

`access.sh`
: Displays which container files actually get used by reading the `atime`, i.e., access time, file attribute. Since the `/containers` mount point is an `ext4`[135]-partition mounted with the `relatime` option, a `cron`[136]-job periodically updates the `mtime` attribute to be newer than the `atime` so that the next file access will trigger an update of `atime`. This helps to identify which containers are still under active usage and which can confidently be archived or deleted without users complaining.

`check_build_cache_integrity.py`
: Verifies that a given build cache contains valid archives that can be extracted.

`clean_testing_build_remnants.sh`
: Uses `gerrit.sh` to query its database about the status of all changes that currently have failed build caches. Requires the environment variable `GERRIT_USERNAME` to be set. By default it presents which changesets can be safely cleaned up, either because they are merged or because they are abandoned. If supplied with an extra `clean` argument, build artifacts are actively deleted.

[134] A full refspec would be: `refs/changes/<change-num[-2:]>/<change-num>/<patch-level>`
[135] `https://www.kernel.org/doc/Documentation/filesystems/ext4.txt` (visited on 2021-03-25)
[136] `https://pubs.opengroup.org/onlinepubs/9699919799/` (visited on 2021-03-25)

`dump_cache.sh`
: Can be used to create a build cache from all packages currently installed in a given container. It is used during the regular container build flow described above.

`extract_dna.sh`
: Can be used to extract all essential information needed to recreate a container from an image. It is used during the regular container build flow described above.

`gerrit.sh`
: Provides helper functions for other scripts to interact with Gerrit.

`merged.sh`
: Same as `clean_testing_build_remnants.sh`, this script checks the state of all Gerrit changes belonging to testing containers, optionally deleting them to conserve space on the `/containers` mountpoint.

`public.sh`
: Parses all stable containers and checks for blacklisted packages

Packing up the Cluster: Slurm in Containers 9

After introducing a manageable way to deal with different sets of evolving software dependencies, we extend this approach to the whole compute cluster. During this thesis, the existing cluster installation (described in [Müller, 2014]) was revamped from the ground up to allow for better debug and testing capabilities.

Already, the deployed cluster management and scheduling framework, Slurm,[1] was extended via plugins to allow for more comfortable allocation of hardware resources (cf. Section 9.1.2). These plugins interfaced with our software stack and therefore introduced the same software dependencies by transitivity. Previously, these dependencies had been fixed once (at deployment) and kept static. If the set of dependencies changed this could create the same kind of “friction” that ultimately led to the introduction of visionary containers (cf. Section 8.2.3).

Furthermore, any changes to these plugins had to be tested “live”, i.e., on the production system. Testing in production usually has detrimental effects, especially on jobs submitted by other users, and is rightfully frowned upon, prohibiting frequent updates to the codebase. Alternatively, there could be completely separate setups for compute and control nodes. They would allow testing changes in isolation, but would, over time, slowly diverge from the production system, increasing the chances for changes to still break production despite being tested.

Both problems are addressed by the new approach introduced here: The new installation is completely encapsulated within the same visionary container (cf. Section 8.2.3) that is used by users. Additionally, we use available container-based means of abstraction to run *several cluster instances* in parallel: One production system that should always be available plus several testing deployments. Users can switch between clusters by executing a single line of code.[2] Of course, this is most useful to developers as users are rarely involved in testing updates to the cluster infrastructure. The important point is: Most code can be tested as is and then moved to production *without any changes*. This grants more freedom to developers while testing and reduces the overall downtime of the system, because changes “just work” when pushed to production.

[1]Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management, [Yoo et al., 2003]

[2]This single line of code loads one of the provided GNU modules.

9.1 Slurm Deployment

Slurm[3] is a scheduling solution widely used in HPC-clusters. Benchmarks, while low in numbers, indicate that Slurm is sufficient to schedule a large number of long running jobs [Georgiou et al., 2013]. At Electronic Vision(s), we can informally confirm this. It consists of several distributed services to do its task. On each cluster node, a local `slurmd` instance is responsible for spawning, managing and even terminating jobs. All `slurmd` instances communicate via RPCs[4] with a central `slurmctld` daemon, responsible for coordinating compute nodes, scheduling jobs and interacting with users. Furthermore, `slurmctld` connects to `slurmdbd`, the database daemon keeping track of relevant usage metrics in its SQL[5]-database.

Users interact via a set of deployed binaries communicating with `slurmctld` via RPC. These include `srun` to submit interactive jobs, `sbatch` to submit batch scripts or `squeue` to request queue information.

All RPC-based communication between services is secured via MUNGE.[6] Using a shared secret across all nodes that is only readable by `root`, it allows sending encrypted messages across the network that include user information. Therefore, it prevents users from impersonating each other. Communication with the `munged` daemon, running on every node, is facilitated via socket. For the same reason, `quiggeldy`, discussed in Chapter 10, makes use of MUNGE as well.

9.1.1 Dedicated Cluster-Controller: `slurmviz`

In contrast to the previous deployment (described in [Müller, 2014]), the central Slurm installation was moved during this thesis from frontend to a special-purpose VM[7]: `slurmviz`. This increases reliability as load on the frontend will have no detrimental effect on cluster performance. Running Debian stretch,[8] `slurmviz` hosts all central Slurm-related services: `slurmctld` and `slurmdbd`. For obvious reasons, it does *not* serve as a compute node and hence does not run `slurmd`. The `slurmviz`-VM is hosted on `libertine`, one of two VM executor hosts in the Electronic Vision(s) infrastructure.

A key difference to the previous deployment is the ability to have several fully-distinct Slurm deployments side-by-side, each potentially executing in a different container. At time of writing, there are four distinct Slurm deployments:

`slurm-skretch`	The production system, running in a Debian stretch based host system.

[3]Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management, [Yoo et al., 2003]

[4]Remote Procedure Callss

[5]Structured Query Language

[6]MUNGE Uid 'N' Gid Emporium, `https://dun.github.io/munge/` (visited on 2021-03-31)

[7]Virtual Machine

[8]`https://www.debian.org/releases/stretch/` (visited on 2021-04-07)

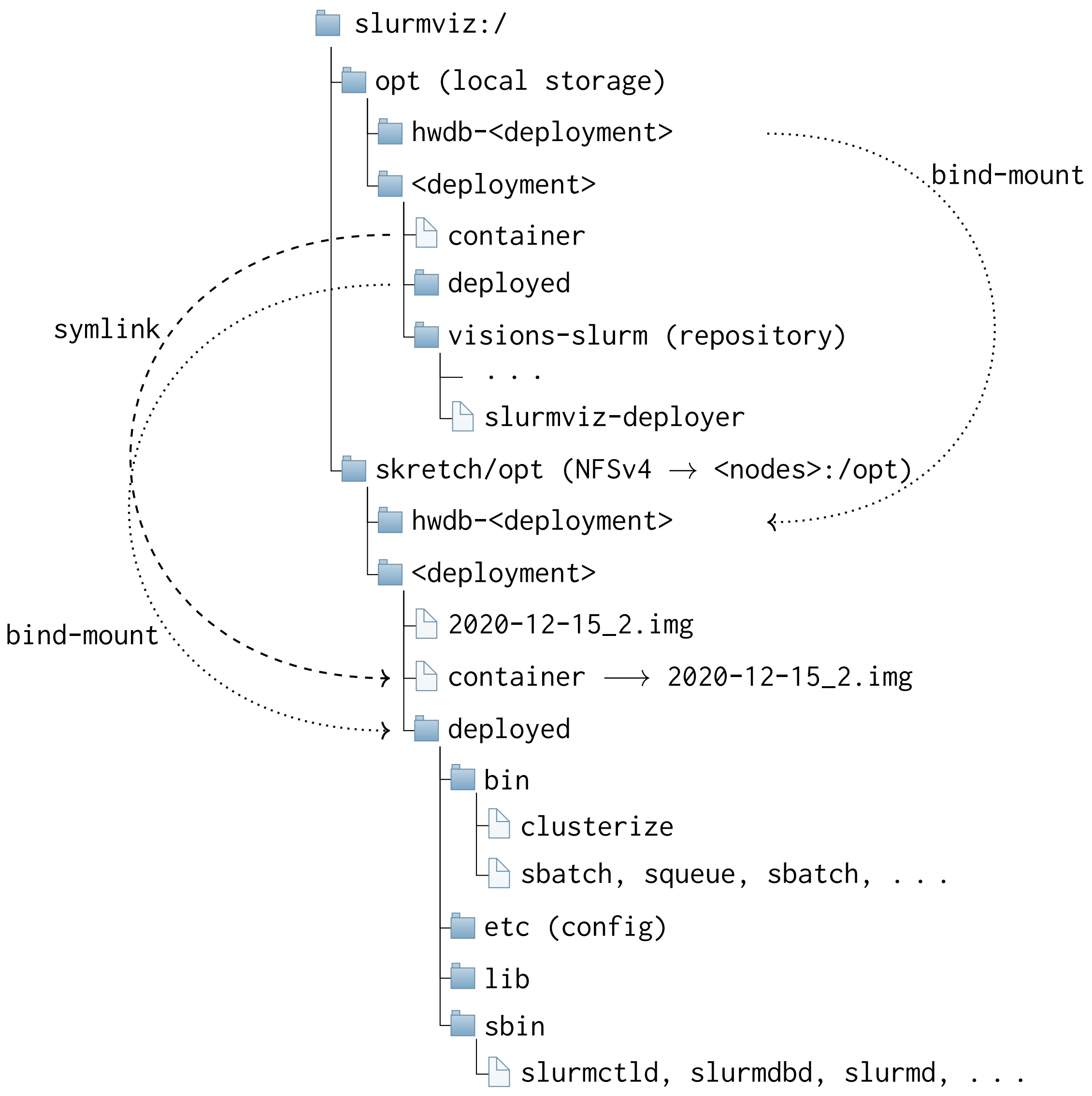


Figure 9.1: Overview of deployment organization on `slurmviz`. Each Slurm deployment is built on local storage (under `/opt`) and deployed to the NFS boot volume of cluster nodes (i.e., `/skretch` is mounted as `/`). Both operations are performed by a custom build-script: `slurmviz-deployer`. To reduce dependency on other parts of infrastructure, the container image used to build each deployment is explicitly copied to its NFS folder and symlinked. On the frontend, symlinks to `clusterize` that are loaded via modules emulate the deployment and allow switching between deployments by exchanging the loaded module (see Section 9.2). This allows testing new container images prior to adjusting the production system without adjusting configuration or user scripts, enabling greater reliability. Refer to the text for further details.

`slurm-skretch-cmauch-testing`	The testing setup primarily used by Christian Mauch to test and verify additions to the Slurm-installation.
`slurm-skretch-obreitwi-testing`	The testing setup primarily used by Oliver Breitwieser, the author of this thesis, to test and verify

	additions to the Slurm-installation.
`slurm-skretch-testing`	A general testing deployment that can be used by other interested parties after consultation.

There are templated `systemd`[9] service-files that allow fine grained control over services. For example, while production scheduler is restarted via:

```
systemctl restart slurmctld@skretch
```

The general testing deployment is just as easily restarted via:

```
systemctl restart slurmctld@skretch-testing
```

This is extended to cluster nodes. Here, we can move a node from production to testing by first draining[10] in the production setup and then issuing the following commands on the node in question:

```
root@<node>$ systemctl stop slurmd@skretch
root@<node>$ systemctl start slurmd@skretch-testing
```

Afterwards, the node will become available within the testing cluster. To reiterate: This will run a *different* build of `slurmd` in a different container environment. In theory – if testing does not involve consumption of any compute resources – both `slurmd` can also be run in parallel. To summarize: All deployments can be managed in much the same way, irrespective of production or testing.

The organizational structure for each deployment is outlined in Figure 9.1. All cluster nodes are booted via PXE[11] from a read-only NFS mount point, serving as the root of their filesystem. On `slurmviz`, this mount is accessible via `/skretch`.[12] The file structure under `/opt` is meant to resemble `/skretch/opt` as closely as needed. Relative to its root folder (`/` and `/skretch` respectively) we ensure:

- Each deployment is built and executed in a unique container (cf. Section 8.2.3), expected to reside at `/opt/<deployment>/container`.
- `/opt/<deployment>/deployed` contains all deployed Slurm binaries and related files.
- `/opt/<deployment>/deployed/etc` contains the Slurm configuration, tracked in a `git`-repository.[13] One sub-folder, `deployment-specific`, contains the only configuration difference between setups. These include used ports on `slurmviz`, database names and licenses. In testing deployments, these files are not checked in.

[9] systemd System and Service Manager, `https://cgit.freedesktop.org/systemd/systemd/tree/README` (visited on 2021-01-29)
[10] Marking it as unavailable and waiting until all already running jobs have terminated.
[11] Preboot Execution Environment
[12] Indicating that this is the stretch-based ~~k~~cluster-mount.
[13] `https://gerrit.bioai.eu/gitweb?p=config-slurm.git;a=summary` (visited on 2021-04-08)

Slurm is built from a local repository[14] checkout in `/opt/<deployment>/visions-slurm` using a custom build script: `slurmviz-deployer`. Originally provided by Kai Husmann, it was extended during this thesis to be container-aware. Written in `bash`, it automatically loads the corresponding container from its deployment and, utilizing Spack (cf. Section 8.1), generates a dynamic module file with the latest versions of Slurm's dependencies available. The module file is cached under `/run/<depoyment>/current_modules.sh`; deleting it retriggers its generation. In particular, this means that there is no Slurm-app in the container, but only a corresponding meta-package tracking dependencies. The whole build and deploy process is then executed within the container. All binary components are compiled with `RPATH`ss enabled. Similar to how Spack uses them (cf. Section 8.1), they hard-code absolute paths to library dependencies. Therefore, when executing any Slurm binary, it will link to the correct libraries as long as the whole deployment is mapped into the container. The environment does not need to be modified. Furthermore, upon successful deployment, `slurmviz-deployer` updates meta-branches in the corresponding `git`-repository that allow other software deployments (in particular the BrainScaleS-1 software stack) to easily identify the current software state of the cluster when performing nightly builds/tests.

At runtime, all deployed Slurm-services are started via templated `systemd` service files, as described above. They wrap their corresponding Slurm binary via `clusterize`, a custom Singularity wrapper developed during this thesis, that is deployed as part of all binaries. `clusterize` then ensures the correct Singularity configuration at runtime (discussed in Section 9.2).

In tandem with every Slurm deployment, we deploy `hwdb`. It is required by the `nmpm_custom_resource` Slurm plugin, explained in Section 9.1.2, in order to retrieve information about deployed hardware systems. A convenient build script is provided,[15] building `hwdb` in the same container as the deployment. It needs to be re-run whenever a new version of `hwdb` or a new container image is deployed. There is a full manual written, among other topics, on how to update the container image.[16]

Containers and User Jobs Since jobs get executed in whatever process environment the node executor daemons, `slurmd`, live in, user jobs will always start running in the container Slurm is currently deployed in. In order to reduce friction, no `app` is loaded in the container which intentionally results in a bare-bone environment. Users are therefore advised to explicitly run their job steps in a container of their choosing. As outlined in Section 9.2, `slurmd` is ran with increased capabilities to allow for maximum flexibility. In theory, even other container solutions could be used (cf. Section 4.4.3). At the time of writing, though, there is no need for any other container-based solution to be deployed on the Electronic Vision(s) cluster.

[14] `https://github.com/electronicvisions/visions-slurm` (visited on 2021-04-08)
[15] `https://github.com/electronicvisions/visions-slurm/blob/master/visionary-utils/hwdb/rebuild.sh` (visited on 2021-04-08)
[16] `https://openproject.bioai.eu/projects/symap2ic/wiki/slurmadmin#update-to--rebuild-for-new-container` (visited on 2021-04-08)

Munge As another layer of precaution we want production and testing deployments not to share MUNGE-secrets. Therefore, if there is misconfiguration and services from different deployments interact, they will not be able to decode each others message and not perform any operation. For ease of implementation and maintenance, all testing deployments share their MUNGE secret because usually only one testing deployment is evaluated at a time. Due to technical limitations, MUNGE enforces the location of key-files to be hard-coded into the binary at compile time. A secondary munge installation is therefore provided under `/opt/munge-testing` (in both local and NFS-mount) with a different secret. `clusterize` then automatically bind-mounts the correct MUNGE socket into the process environment, ensuring the correct secret is used for communication.

Remaining Differences: Ports and Databases While most configuration settings are shared between Slurm deployments, some differences do remain, either due to technical limitations or another layer of precaution. All differences are tracked by a `deployment-specific` subfolder in each configuration repository.[17] Its files are sourced from the "regular" configuration files and only contain settings that differ between deployments: Used ports and database credentials.[18]

The first difference are port numbers. There can only be one service listening to any one port, hence in order to support several deployments running concurrently, especially on `slurmviz`, they need to listen to different ports.

Additionally, for obvious reasons, we want to have separate relational databases for production and testing environments. Therefore, testing any change will *never* result in data loss. Both databases can be kept in sync by a simple duplication prior to testing, but in most cases administrators merely have to ensure that all users involved in testing exist in the corresponding database.

9.1.2 Streamlined Hardware Allocation: `nmpm_custom_resource`

Historically, access to BrainScaleS-1 hardware was facilitated via Slurm licenses. Users would allocate a license and then execute their script. If someone else had already allocated a given hardware resource, the license would be in use and the job would be delayed until it was free. Unfortunately, there was no verification that a given user job would connect to the actual resources it allocated, leading to user jobs influencing each other through unintended human error. Hence, a more robust solution as devised that ensured compute nodes to only communicate with resources they had actually allocated via dynamic firewall rules, while still being easy to use for users.

The solution, `nmpm_custom_resource`,[19] is a time- and sanity-preserving component of our cluster: A custom Slurm plugin, originally tasked with handling access to BrainScaleS-1

[17] `https://gerrit.bioai.eu/gitweb?p=config-slurm.git;a=summary` (visited on 2021-04-08)

[18] The databases are only accessible from within `slurmviz` and used to differentiate between deployments. Hence, it is acceptable to track them via a repository.

[19] `https://github.com/electronicvisions/visions-slurm/blob/visions-slurm-20.02.3/src/plugins/job_submit/nmpm_custom_resource/job_submit_nmpm_custom_resource.c` (visited on 2021-04-05)

hardware. It allows users to request which abstract hardware resources they intent to use – via `--wafer <wafer-id>` and `--fpga <fpga-id>` CLI arguments – when scheduling their job. `nmpm_custom_resource` then queries a local installation of `hwdb` in order to identify which cluster resources have to be allocated for the job and acquires the corresponding licenses. For BrainScaleS-1 (cf. Section 3.1), it is able to identify all FPGA modules that need to be initialized based on the resources a user has requested. Furthermore, it actively tracks the initialization status of all FPGAs in order to minimize startup time by only performing initializations on "dirty" FPGAs. Finally, it defines environment variables that are used in pro- and epilogue scripts. These scripts add firewall rules to allow communication between allocated FPGAs and compute node. This is done to prevent other users from accidentally sending commands to wafer modules they have *not* allocated.

Using different Slurm deployments allows testing and verifying these firewall configuration scripts on the same node with the same configuration, code and hardware. This greatly decreases the amount of time needed to testing and eliminates many potential sources of errors caused by incongruences between testing and production environment. Overall, day-to-day operations are disrupted less.

`nmpm_custom_resource` was written and is maintained primarily by Christian Mauch. It has been adapted to handle access to single BrainScaleS-2 cube-setups via the same syntax.

9.1.3 Ensuring Interactive Capacities: `cerberus`

In every cluster environment there are different kinds of jobs: On one hand there is batch processing of vast amounts of jobs, processed one after the other. They should be finished in a timely manner but nobody is actively sitting in front of terminal, waiting for them to finish. On the other hand users sometimes need to work interactively, especially when debugging or setting up a processing pipeline. Here, it is crucial that a modify/compile/run loop is executed as fast possible. In summary: There needs to be a delicate balance between batch-processing and interactivity.

We solve the problem by introducing a new `interactive` cluster partition.[20] As the name suggests, it is intended for quickly executing interactive jobs that users actively wait for in their terminal. The aim is that this queue always has some free computing capacities at its disposal, unless the cluster is completely at capacity. It therefore includes most compute nodes present in the cluster and has the highest scheduling priority among partitions. A custom Slurm plugin was developed during this thesis: `cerberus`.[21] Its task is to maintain a hard limit of jobs that a user can have present within a partition *in total*.

[20]Slurm has the concept of distinct partitions with distinct configurations such as time limit constraints. Jobs are submitted to one particular partition and then scheduled within, for the most part independently of other partitions. The only exception are overlapping node configurations between partitions. In case of a conflict jobs in the partition with higher priority are more likely to get scheduled first.

[21]`https://github.com/electronicvisions/visions-slurm/blob/master/src/plugins/job_submit/cerberus/job_submit_cerberus.c` (visited on 2021-04-09)

At the time of writing, `cerberus` is configured to allow up to three concurrent jobs per user in the `interactive` partition. Any additional jobs are terminated immediately, thereby preventing users from submitting large quantities of batched jobs (for which there are other dedicated partitions). We therefore manage to strike a balance: Batch computation is supported in all but one partition while interactive jobs are executed almost immediately in most cases.

9.2 Deploying Binaries Cluster-wide: `clusterize`

During the set up of the current Electronic Vision(s) cluster iteration, one problem became apparent: Non-static configuration of Singularity based on external factors. As outlined in Section 8.2, singularity can be configured in two ways. First, there is a system wide configuration file that holds static configuration applied to all Singularity-calls. Second, users are able to augment their Singularity calls via CLI arguments. These include bind-mount directives or which app to use inside the container. If bind-mount targets do not exists, however, the Singularity call will fail altogether. This, of course, is a desirable outcome in most cases. When trying to run binaries within certain containers across different sets of nodes (`slurmviz`, compute nodes, frontend), the process call typically has to be augmented.

We need a way to probe the system we are currently executing on and adjust Singularity's configuration accordingly. To solve this problem, `clusterize` was developed during this thesis. Written in `bash` and AWK,[22] it identifies which Slurm deployment it was started in, i.e., its physical location, and derives which container image to load according to convention outlined in Figure 9.1.

Inside the container, all deployment-specific folders are mapped to their default system location. Hence, while running, all deployments appear to be their regular system-installed counterpart. This includes configuration, logs and runtime libraries. Furthermore, it provides the necessary bind mounts for `iptables`,[23] MUNGE, `mysql`,[24] `nscd`,[25] and `sudo`. The command line is generated dynamically, providing declarative functions such as

```
add_if_exists <source> [<target> [<alternative>]]
```

which emits the necessary CLI arguments to bind-mount `<source>` as `<target>` (if provided) within the container if `<source>` exists on the host, alternatively falling back to mounting `<alternative>` (if provided) to `<target>`. Using declarative style ensures that future maintainers can easily add additional bind-mounts.

Furthermore, when run as root – which is only the case when starting `slurmd` on compute nodes – we provide `--allow-setuid`. This allows for compute jobs to spawn additional

[22] AWK Programming Language, [Aho et al., 1987]

[23] Administration Tool for IPv4 Packet Filtering and NAT, `https://linux.die.net/man/8/iptables` (visited on 2021-04-08)

[24] `https://www.mysql.com/` (visited on 2021-04-08)

[25] Name Service Cache Daemon, `https://linux.die.net/man/8/nscd` (visited on 2021-04-08)

containers more easily. Please note that this only works for the first container spawned, all other containers will have their capabilities removed unless they are “trusted” (see Section 8.2.4).

Preserving Environment Singularity allows augmenting the spawned process’ environment by defining environment variables prefixed with `SINGUALRITYENV_<name>`. Within the container, the prefix is stripped, leaving only `<name>` defined in the container No environment variables starting with `SINGULARITYENV_` remain in the process environment. For the typical Singularity use case, e.g., one user starts a container on their local machine, this is not a problem.

On the frontend, however, where each call to Slurm-binaries is a Singularity-wrapped (for details see Section 9.2.1), effectively “consuming” all `SINGULARITYENV`-variables. Hence, they will not reach any Singularity-calls within user jobs. This is a problem for users because it is simply not practical to rebuild ones environment with *every* `srun`-call.

The solution is to have a two-step invocation of the wrapped process. Upon first invocation, all Singularity-related environment variables are prefixed with `SINGULARITYENV_CLUSTERIZEENV_`. They are effectively wrapped twice. This means `SINGULARITYENV_FOOBAR` becomes `SINGULARITYENV_CLUSTERIZEENV_SINGULARITYENV_FOOBAR`. `clusterize` then calls itself *within* the container and strips `CLUSTERIZEENV_`-prefixes from environment variables. Afterwards, the wrapped process is executed as expected, but in an environment that still contains Singularity-related variables.

This is also beneficial when handling `PATH` and `LD_LIBRARY_PATH`. Because Singularity is typically used to wrap a whole application along with its dependencies, it modifies both variables upon invocation in order to reduce environment related conflicts for typical users. Unfortunately, since we use Singularity to provide the complete system environment, we want modifications to both variables to “survive” job submission onto the cluster. Therefore, both `PATH` as well as `LD_LIBRARY_PATH` are preserved in the same manner. They also appear within the container as they were when `clusterize` was called in order to allow interactive job submissions to Slurm via `srun`, as users expect.

This process is fully transparent to users. While it does incur the typical 200 ms overhead induced by Singularity (cf. Listing 4), this is acceptable because the typical runtime of each Slurm-binary exceeds it. In any case, the benefit of being able to run Slurm-binaries irrespective of the current environment, far outweighs the cost.

Wrap via Symlink Another feature, that is essential for deploying to our frontend node (see Section 9.2.1), is the ability to wrap binaries via symlinking. `clusterize` is able to detect that it has been called via a symlink and will execute a binary of the same name within the container. This means that users do not even notice they are performing a singularity call. They are simply executing a binary. This is the basis for allowing users to switch easily between Slurm deployments, as shown in Listing 6.

Configuration Options By default, `clusterize` determines container image and `app` depending on which host it is executed and in which Slurm deployment. Of course, image and `app` to use can be overwritten via `-c`/`-a` CLI arguments. Alternatively, the environment variables `CLUSTERIZE_CONTAINER`/`CLUSTERIZE_APP` can be set. In that case only location-dependent bind-mounts will be performed. If run with `-v` or if `CLUSTERIZE_VERBOSE` is defined in the environment, `clusterize` will print all steps it performs including the final Singularity command line.

Furthermore, setting `CLUSTERIZE_NO_CLEAN_SENV` will cause `clusterize` to *not* clean the original `SINGULARITYENV_`-prefixed environment variables prior to invoking its Singularity call. This is usually not the desired behavior because `clusterize` aims to be transparent. In particular, this means that these Singularity-related environment variables *will* directly affect `clusterize`'s Singularity-call, rather than wrapping it transparently.

9.2.1 Deployment on Frontend via transparent Wrapping

As per usual, users interact with the cluster environment from the frontend. The challenge now is to provide a possibility for users to switch between Slurm deployments. This is important now for testing but was of equal importance during the transition period from Slurm running on the frontend machine (`helvetica`, described in [Müller, 2014]) to `slurmviz`.

For its implementation there were a few points to consider:

- Users should be able to submit jobs to clusters as they used to, i.e., there should not be any grave idiosyncrasies apart from the usage of Singularity *within* jobs.
- Job submission should work the same way whether or not users just logged into the frontend or are submitting jobs from a session within a visionary container.
- Typically, the deployment rate of Slurm is much slower than the rate of new visionary containers. Hence, we want the solution to be capable of handling situations in which Slurm binaries are executed in an older container than the user session.
- Switching between cluster deployments should be possible and only need one line of code, i.e., a swap-out of GNU modules. Otherwise, there should be no difference in the job submission process for all cluster deployments.

For this, the author chose to use the wrapping-via-symlink feature of `clusterize` (cf. Section 9.2). First, we ensure the expected directory structure (cf. Figure 9.1) by mounting each Slurm deployment via bind mounts to `/opt`. Then, we create a module for each Slurm deployment. It consists of a folder with symlinks to the `clusterize` executable of the corresponding deployment. `clusterize` then executes the binary with the same name as the symlink within the deployment's container image. If the user already is within a Singularity session (in a container provided via `yashchiki`, cf. Section 8.3), we rely on the custom feature of trusted containers – also introduced in this thesis (cf. Section 8.2.4) – to nest singularity calls. In order to make this concept more robust, we link all Slurm binaries via `RPATH`. Therefore, they are able to reliably find their dependency libraries irrespective of any environment modifications users performed. As shown in Section 8.2.3, this solution introduces an acceptable amount of overhead and does not disturb the users' workflow.

```
1 $ module load slurm-singularity/current
2 $ ls -l $(which srun)
3 /wang/environment/cluster/slurm-skretch/bin/srun ->
  ↪  /opt/slurm-skretch/deployed/bin/clusterize
4 $ module swap slurm-singularity/obreitwi-testing
5 $ ls -l $(which srun)
6 /wang/environment/cluster/slurm-skretch-obreitwi-testing/bin/srun ->
  ↪  /opt/slurm-skretch-obreitwi-testing/deployed/bin/clusterize
7 $ module unload slurm-singularity
8 $ ls -l $(which srun)
9 /usr/local/bin/srun -> /opt/slurm-skretch/deployed/bin/clusterize
```

Listing 6: Demonstration of switching between cluster deployments via GNU modules. Loading a `module` modifies the users' environment `PATH` to include another folder that contains symlinks to `clusterize`. The names of symlinks correspond to all provided Slurm binaries (shown here: `srun`). `clusterize` then transparently executes the binary of the same name in its deployment's container image. Switching between cluster environments corresponds to swapping out the loaded `slurm-singularity` module. Afterwards, `srun` (and all binaries) point to `clusterize` in the testing environment which correspondingly executes the testing Slurm binary in the testing container image. As there is only one deployment mapped into the container at any given time, there is no possibility of binaries or libraries from different deployments to influence each other. As shown in lines 7ff, transitioning from the previous deployment to the new one corresponded to simply replacing binaries with symlinks to the new production deployment's `clusterize`.

The solution is demonstrated in Listing 6. To users, it appears as if they are executing Slurm binaries directly, but depending on which module is loaded they are communicating with completely different instances. Switching between deployments indeed requires one line of code. This also made the transition from the old deployment on `helvetica` to `slurmviz` trivial: Users could switch to/from the new deployment and test their setups by loading/unloading the `slurm-singularity` module. Finally, once we were confident in the stability of `slurmviz`, all old Slurm binaries were simply replaced by symlinks to `clusterize` in the production deployment.

Avoid con{ges,ten}tion via Micro-Scheduling: `quiggeldy` 10

10.1 The Case for interactive analog Hardware

Due to the eponymous layer of abstraction in digital hardware, it does not matter on what substrate a calculation is performed: Either a particular hardware realization supports a given logical operation in a deterministic fashion, or not. Any exception is considered a bug [Pratt, 1995]. Of course, no guarantees about runtime or memory requirements are made, which is why compute-intensive tasks are offloaded to HPC-clusters with more specialized hardware.

Fundamentally, users are able to run, tweak and verify their (possibly down-scaled) compute models "locally" in an immediate feedback loop prior to submitting a job to the cluster scheduler. The cluster is only used for bulk data generation such as long running parameter sweeps or extensive model training. With analog hardware, however, this is not yet the case. Here, fixed pattern variations and imperfections in the substrate are not abstracted away, but rather exposed directly to the user to exploit or avoid. As explained in Section 3.2, there are facilities to calibrate analog parameters, however, the target regime of these varies so much from use-case to use-case that a general abstraction, hiding calibration from all users and pretending the analog substrate is perfectly deterministic, would be too limiting. The real-world consequence of this is that experimenters typically pick and choose a setup for their experiments and stick with it, simply because parameters would need to be adjusted slightly when switching setups.

As long as there are more available setups than experiments to be performed, work can go along smoothly. Users have their personal setup either sitting on their desk or attached to the Electronic Vision(s)-cluster, allocating it via Slurm: They can run an iterative exploratory REPL[1] or long-running parameter sweeps. Nobody is blocking anyone else.

As soon as there are more experimenters (or experiments to be performed) than setups, the situation changes. Now hardware setups have to be accessible via the cluster and all people have to schedule compute jobs to interact with them. As soon as one person has allocated a given setup, all others are blocked from accessing it and have to wait. This especially means that while one person is sweeping parameters, others cannot verify recent modifications to their models in a quick manner. The lack of an immediate feedback

[1]Read-Eval-Print-Loop

loop is detrimental especially for new users as it steepens the learning curve. Also between experienced experimenters sharing a setup this has led to crude agreements such as a general ban on parameter sweeps during daytime.

Finally, for specific types of experiments – such as deep learning applications as described in Chapter 13 – there are downtimes in hardware usage: If a model is evaluated or trained in the loop, any allocated hardware setup is idle while parameter updates are computed on the host computer in between experiment-steps. In case of parameter sweeps, several iterations could share one hardware setup concurrently, again increasing experiment throughput per chip. Of course, this depends heavily on the runtime difference between inference on the accelerated chip and the comparably slower update routines on the host.

An immediate solution to the accessibility problem that comes to mind is to have a set of "debugging" hardware setups. Users could test and verify their experiments on those setups and then perform the "actual" experiments elsewhere. However, besides removing perfectly good setups from the compute pool this is infeasible as each setup typically requires its own calibration, placing yet another burden on the user to keep track of.

Another solution would be to force users to separate their job scripts into those parts that need direct immediate hardware access and those performing setup, parameter updates or analysis. Different parts could then be scheduled as job steps with different permissions and Slurm would handle hardware access. This is infeasible because of two reasons: First, and most importantly, it requires too much involvement on the side of users. As this approach is tedious and error-prone, the simpler solution to just allocate hardware for the complete job is far too tempting. Secondly, even if all users did separate their jobs perfectly, there is still the problem of time scales. Slurm is designed for scheduling long running compute jobs as efficiently as possible. It operates on the order of seconds, i.e., upon scheduling, given free resources, it takes a few seconds for a new job to be spawned. However, due to the speedup factor of ~1000, single experiment-steps can run on the order of tens to a few hundred milliseconds realtime. Ergo, any performance gains would be absorbed by delays in scheduling. We need a dedicated solution: `quiggeldy`.[2]

Design Goals

i) Provide *immediate feedback* – on the order of seconds – for users accessing hardware setups to enable tight feedback loops when interactively debugging experiment behavior. Achieve this without exclusive access to the setup except for the most low-level scenarios. Other users should not be locked out.

ii) Allow for *continuous hardware usage.* In particular, users should not have to schedule parameter sweeps verbally between each other.

iii) In some cases, exclusive ownership of a setup is needed, e.g., for advanced debugging. Hence, we need to provide a pathway for `quiggeldy` to be *optionally bypassed for exclusive ownership.* Of course, this should be the exception.

[2]Name derived from "Quick-Queue" implementation.

iv) Be *transparent*: Once users have written their experiment script, it should not have to be modified in order to be run with or without quiggeldy. As a caveat, we are permitted to impose conceptual constraints on experiment designs that might make existing experiment scripts incompatible to be run with quiggeldy without modifications. However, these constraints are not limiting functionality in any way but rather help enforce good design practices, such as snapshotting and explicit tracking of state. They are explained in Section 10.4.

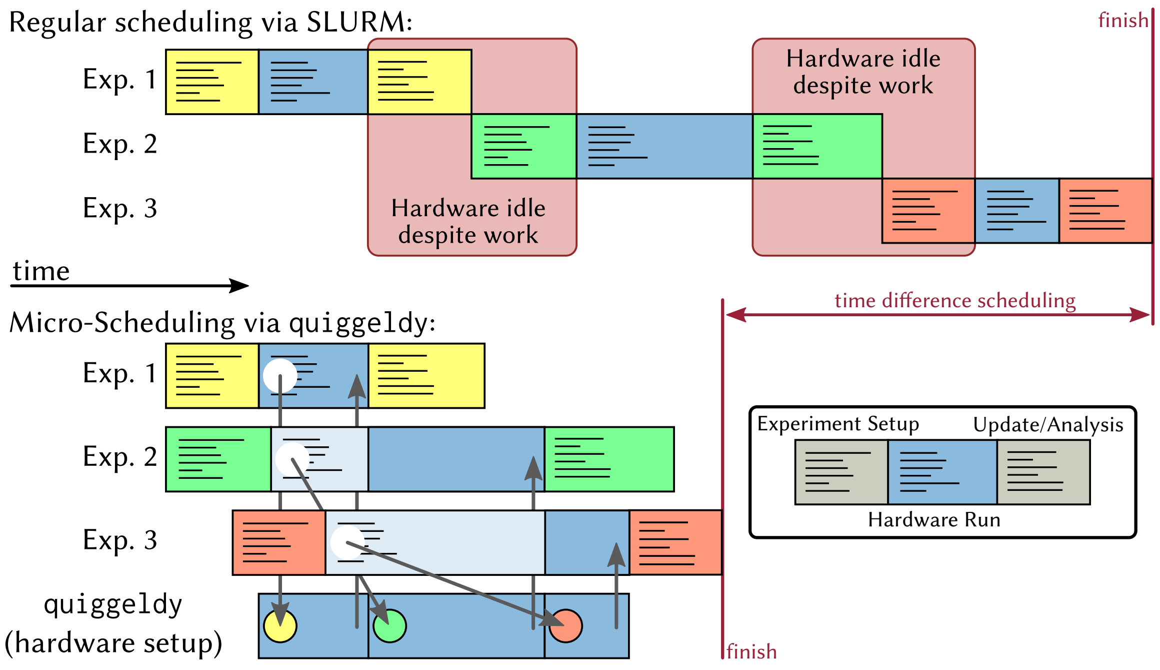

Regular scheduling via SLURM:
Exp. 1
Exp. 2
Exp. 3
Hardware idle despite work
Hardware idle despite work
finish
time
time difference scheduling
Micro-Scheduling via quiggeldy:
Exp. 1
Exp. 2
Exp. 3
quiggeldy (hardware setup)
finish
Experiment Setup
Update/Analysis
Hardware Run

Figure 10.1: Core principle of quiggeldy operation.
Bottom-Right: Every experiment executed on hardware can roughly be separated in three phases: Setup phase (the experiment is defined by creating the pbmem on the host computer), hardware run (the pbmem executed on the hardware) and analysis (the response from hardware is evaluated on the host again). These three phases could be run in the loop while training. Only the execution phase requires exclusive access to the hardware. Please note that relative run times are not drawn to scale and depend heavily on the nature of the experiment.
Top: Hardware usage of three experiments without using quiggeldy. Because hardware resources need to be allocated for the whole compute job, other experiments have to wait for the whole job to finish – including analysis. This causes times in which hardware resources are idle (indicated in red) despite work (i.e., experiment-steps) actively waiting to be processed.
Bottom: Hardware usage of the same three experiments with hardware access gated via quiggeldy. Instead of establishing direct connection to the hardware setup, each job connects to quiggeldy instead. Exposing the same interface as a regular hardware backend, quiggeldy takes care of running the supplied pbmem and sends the response back to the corresponding compute job. Only the compiled set of FPGA words is sent to quiggeldy for execution (indicated by the circle). Hence, experiment execution is completely transparent to users, but setup and analysis are performed without active hardware allocation, resulting in higher experiment throughput. In this example all three experiments only contain one step and finish executing after roughly half the time compared to the regular example. If experiments contained several steps they would run interleaved. No modifications of experiment scripts are needed to switch between quiggeldy and direct access. However, there are some conceptual changes to consider in terms of reinitialization (discussed in Section 10.4). Refer to the text for details.

10.2 Core Principles

quiggeldy is a transparent wrapper around hardware experiment execution. This is illustrated in Figure 10.1. Users appear to be directly connected to the hardware backend, executing their experiments "locally", i.e., as if they had exclusive access to the hardware resource. Instead, their experiment-steps are transferred to a remote site where the quiggeldy-daemon is running. It schedules all submitted experiment-steps in a round-robin fashion with steps from other users, i.e., every user has equal opportunity for their experiments to be run. Round-robin here means that each user gets allotted an (adjustable) specific time slot in which experiment-steps that are currently pending execution are executed one-after-the other. Once there are no steps left to execute or the time-limit is reached, execution shifts to the next user in line. As much computation as possible is still conducted on the user side (see in Section 10.3). This includes experiment setup and construction in a (potentially) slow language as well as post-experiment read-out of results, analysis and parameter updates. By not having hardware units allocated for these phases, quiggeldy allows for continuous hardware usage as experiment-steps by one user can be executed while experiment-steps of other users are analyzed.

By employing a reinitialization-mechanism (see Section 10.4), the hardware is semi-automatically initialized into a state that the user experiment expects. Within the Electronic Vision(s)-cluster, quiggeldy daemons act as regular users, i.e., they have to properly allocate cluster resources in order to be permitted hardware access.

quiggeldy's implementation is separated into hardware-specific experiment execution and a generic micro-scheduling framework that can be re-used in other scenarios. It makes great use of template metaprogramming[3] to infer all involved data types and generates all involved data structures via macros, so that the scheduling part implemented in lib-rcf is completely separated from the functional part in hxcomm that executes on hardware. Hence, code modifications in lib-rcf are only needed to adjust scheduling functionality. By making use of concurrency wherever possible, especially for accepting experiment submissions and delivery of results, it is ensured that the hardware backend is continuously executing experiments. A first prototype of quiggeldy was developed within haldls (discussed in Section 10.5.1) that unveiled some technical difficulties. It was hence decided to reorganize part of the software stack with regards to connection handling, described in Section 10.5.2. The final implementation is then discussed in Section 10.5.3, along with configuration options (see Section 10.5.4) and utilities (see Section 10.5.5).

While the transition to quiggeldy being the default method of access to hardware resources is still ongoing, it has been deployed successfully in several scenarios: Training deep learning models on the BrainScaleS-2 Mobile[4] during the final phase of the competition "Innovationswettbewerb Künstliche Intelligenz" was conducted via quiggeldy, discussed in Section 10.7.1. A first interactive live-demo using quiggeldy was conducted during NICE 2021. Here, more than 60 users in total conducted experiments interactively on eight hardware units (see Section 10.7.2).

[3]Template metaprogramming can be seen as a pure functional DSL on types evaluated at compile time.
[4]BrainScaleS-2 Mobile Analog Neuromorphic Hardware System, [Stradmann et al., 2021]

Communication in `quiggeldy` is handled via RCF, an inter-process communication library that has been used in the past [Husmann, 2012] and is also used in `flange`. As a result, it supports both synchronous and asynchronous modes of experiment submission. At time of writing, `quiggeldy` defaults to synchronous mode to be consistent with all other connections, i.e., experiments get executed in lock-step, one by one. However, as long as experiment-steps have no dependencies on one another (e.g., in a parameters sweep), they can also be submitted simultaneously in an asynchronous manner. Internally, `quiggeldy` tags each submitted experiment-step with a sequence number, so that they are executed in the order they are submitted and not silently lost. But, users can also enable "out-of-order" execution in which case experiment-steps will be executed in the order they arrive. This mainly benefits pipelining experiments for a single user. `grenade`'s execution model enables automatic tracking of dependencies in the compute-graph. Identified independent parts can then be dispatched asynchronously to the chip as soon as all required dependencies are available, increasing experiment throughput even more.

Within the Electronic Vision(s) cluster, `quiggeldy` is integrated into Slurm via a custom plugin: `hagen-daas`[5], discussed in Section 10.9. Users simply specify which hardware setup they intend to use by a CLI argument similar to the previously used solutions (Slurm licenses). *That is it.* `Hagen-daas` then ensures that a `quiggeldy` daemon governing the given hardware setup is up and running when the user job is scheduled and provides information in the job's environment as to how to connect to it. As will be explained in Section 10.5.4, `quiggeldy` supports periodically releasing its Slurm license, reallocating it again once needed. This allows for compute jobs that require exclusive hardware access to run undisturbed amidst an active `quiggeldy` deployment. An example for this includes the nightly calibration jobs performed by `calix` that currently do not use the reinitialization mechanism (cf. Section 10.4) and simply assume hardware state to be consistent from one experiment-step execution to the next. However, there are no hard technical hurdles preventing adoption to `quiggeldy` in `calix`, it is merely a question restructuring the code. This would even allow for calibration while parameter sweeps are conducted.

10.3 Integration into BrainScaleS-2 Software Stack

For expressivity and maintainability reasons, experimenters are advised to specify their models in a high-level description that focusses on core model functionality rather than implementation details. This allows for separation of concerns via proper layering and eases the burden of entry for future maintainers: A modeler taking over existing experiment code can identify their predecessor's intent more clearly, whereas lower layers in the stack are free to optimize their implementation.

As described in Chapter 6 and illustrated in Figure 6.1, experiments on BrainScaleS-2 by non-expert users are specified in PyNN or `hxtorch`. Expert users might also opt to skip the highest levels of abstraction and access the hardware "directly" via CoCos or even

[5]Howto Avoid Grabbing Emulators Nightlong – Dls As A Service

fisch-Registers. In any case, the description is translated from high-level Python scripts over stadls-pbmems[6] with haldls-CoCos into fisch-pbmems and finally into streams of UT-messages in hxcomm. These UT-messages are then sent by sctrltp via Ethernet to the FPGA and executed. quiggeldy is implemented at the hxcomm-level as "yet another Connection-class" to transparently wrap execution of these lowest-level representations of experiment data on the user-side, i.e., raw FPGA words.

By introducing the concept of connection handles in hxcomm, all connection-types are exposed as simple tokens that symbolize an actively opened and held connection for as long as the token exists. This is typically known as RAII[7]-style. All layers above hxcomm should not interact with them directly but rather pass them to objects or functions in hxcomm or their wrapping counterparts in higher layers. An example for such a function is execute_messages(connection, messages) which takes a connection handle and a sequence of UT-messages, executes them via the given connection and returns a sequence of response UT-messages. Both fisch and stadls provide a run(connection, pbmem)-function that allows execution of pbmems by wrapping execute_messages(). The highest layers either expose a slightly different pynn.run() function or merely provide operations that are then mapped to executed pbmems in case of hxtorch. In any case, since all connections abide to the same API, user code does not have to differentiate between them: It can be executed on exclusively allocated hardware directly, in an accompanying co-simulation or remotely via quiggeldy. Furthermore, hxcomm provides an aptly named get_connection_from_env() that allocates a connection based on the current environment. Hence, in most cases all layers above hxcomm simply create a connection handle from get_connection_from_env() and just pass it along whenever they need to interact with the backend. The concept of handles in hxcomm is described in greater detail in Section 10.5.2. Overall, it allows for all user scripts to be run via quiggeldy by simply setting some environment variables described in Section 10.5.4.

Another reason to implement quiggeldy in hxcomm is experiment throughput. While quiggeldy does make great use of concurrency to handle retrieval, execution and delivery of experiment-steps, it still requires a certain amount of compute resources (in terms of CPU and memory) to en- and decode higher-level representations to/from UT-messages. By implementing quiggeldy in hxcomm, we deliberately limit ourselves to moving uninspected[8] blobs of data around. This avoids any slow-downs that are potentially induced by costly-to-decode pbmems or other abstractions such as fisch-tickets in higher layers.

More importantly, however, implementing quiggeldy in hxcomm makes it completely *independent* from higher level representations. That means quiggeldy does *not need to be redeployed*, unless there are fundamental changes to hxcomm or the layers below, namely sctrltp and lib-rcf. Developers are free to introduce new or modify existing abstractions in higher layers (such as CoCos) without worrying about quiggeldy compatibility. They can immediately be tested and verified using existing deployments.

[6]PlayBack MEMory programs

[7]Resource Allocation Is Initialization

[8]In reality, submitted data is implicitly checked to contain valid UT-messages. Furthermore, responses from hardware are checked for certain error-indicating messages that might require action. More details in Section 10.5.3.

10.4 Reinitialization to enforce Structure in larger Experiments

`quiggeldy` is designed to be as unobtrusive as possible. As explained above (and in detail in Section 10.9), in the fully integrated use-case experimenters merely adjust one CLI argument to submit their jobs to `quiggeldy`, the rest is performed "automagically". However, there is one big difference between using `quiggeldy` and having exclusive access to hardware: Other people. In the exclusive case, an experimenter can be sure that between executing two experiment-steps, the hardware remains in the exact same state and configuration. With `quiggeldy`, other experiments might be scheduled between two steps that leave hardware in a different state. Hence, we need a way to ensure a certain configuration for experiment-steps upon execution: Reinitialization.

A typical experiment consists of several steps: At the beginning of the experiment, the chip is initialized and configuration applied. Afterwards, external input is optionally fed into the chip and any selected observables – be it spikes, membrane potential or correlation measurements – are recorded and transferred back to the user. Due to technical constraints[9] the maximum size of playback memory in the FPGA is fixed: In the current iteration it amounts to 32 MB with each instruction taking up 8 B.[10] Single experiments executed on hardware therefore cannot grow arbitrarily large and have to be split into several steps, often separating configuration and actual execution. Also, some forms of learning experiments might be conducted "in the loop" and therefore require several experiment-steps interleaved with parameter adjustments performed on the host machine. Overall, we can conclude that a typical user script features several experiment-steps.

Experiment-steps can broadly be divided into two categories: Those that apply configuration and those that generate results. With the introduction of `quiggeldy`, it obviously makes no sense to execute configuration steps if hardware access shifts to another user right after. We would just waste hardware time. Instead, configuration experiment-steps are registered as reinit data. `quiggeldy` tracks every connected user script as a session. Each session has a separate stack of reinit-`pbmems` converted to UT-messages associated with it. As the name implies, they are used to re-initialize hardware right before results-producing experiment-steps are performed. If hardware access shifts to another session between two steps the reinit is re-applied to ensure hardware configuration to be in the state the step expects. Otherwise the reinit is not applied in order to save hardware time. Please note that while one user can have several sessions connected in parallel – for example by running several scripts connected to the same hardware – `quiggeldy` considers users, *not* sessions, when scheduling in a round-robin fashion. Especially, this means that users cannot connect multiple times to increase their share of executed experiment-steps. However, each session supports its own stack of reinit-`pbmems`.

Furthermore, when registering reinit, users can decide whether or not to *enforce* it. When enforced, a reinit is executed prior to the first run in addition to when access is temporarily

[9]There is no support yet to continuously stream `pbmems` into the FPGA memory and read back results during an experiment. From a technical standpoint, it could be implemented for existing chips.

[10]Personal correspondence with Dr. Vitali Karasenko.

lost: It acts as initialization *and* re-initialization, just as described above. When not enforced, the reinit merely acts as a checkpoint: The user indicates what state the hardware is expected to be in. In this case reinit data is not immediately transferred to the daemon but only uploaded once actually needed. This allows for checkpoints to be updated locally in rapid succession without transferring large amounts of unneeded data. A prime example would be a learning experiment that uses the on-chip PPU to train one or more epochs before the current state of the network is read out to be logged and augmented. While the host computer is evaluating the read back data and possibly adjusting hyper-parameters, the hardware is free to run other experiments. Hence, we need to register reinit data that, at the very least, writes the current weight-matrix. However, this should only be done if hardware actually was in use by someone else in the meantime. Writing the full weight matrix after every epoch would be wasteful.

In order to not force users to recreate their complete configuration every time they perform even the smallest parameter update, quiggeldy supports the notion of a stack of reinit-pbmems. Each entry in the stack is a single pbmem. Upon reinitialization the stack is applied in the order it was created (i.e., from bottom to top). Please note that, as mentioned above, all pbmems are converted to raw sequences of UT-messages on the client-side in the same manner as regular experiment-steps. All entries in the stack can be updated independently of each other without changing their order of application. For example, the first stack entry could be a generic chip initialization, the second entry static configuration that does not change throughout the experiment and the third entry only setting those parameters that are learnt. During experiment, only the third entry would need to be constantly updated. Due its relatively small size, this is performed in a timely manner.

User scripts need to be functionally equivalent whether or not quiggeldy is used. To that end, in the non-quiggeldy case an enforced reinit decays to a regular execution on hardware at time of registration. Non-enforced reinits are completely ignored since no hardware state needs to be restored.

Enforcing reinitialization has additional benefits: Since all settings not explicitly set are effectively in a semi-random state upon experiment execution, users are forced to express what settings in hardware they rely on via reinit mechanisms. This makes their scripts – or the abstractions they use – more expressive and reliable. Furthermore, when copying code snippets around (potentially from other experiments) during development, users cannot be sure that these snippets do not implicitly rely on bits of configuration set at some distant code point. For experiments designed with reinitialization, the state is fully determined by the stack of reinit-pbmems registered and, hence, more easily identified in code. Overall, this increases code quality in experiment descriptions.

For future generations of hardware, reinitialization could be completely automated, but requires support in hardware. Currently, it is only possible to read out part of the complete digital state in hardware. Some settings can only be written, but not read back. While there are other ways to verify their correctness, if it was possible to read out the *complete* digital state in hardware, quiggeldy could simply read out and store the digital chip state when switching sessions. Of course, due to the analog nature of the substrate, only the digital chip state would be able to be preserved in any case, but resetting all analog observables to a known state at the beginning of an experiment-step is already best practice at the time of writing. However, allowing for complete digital read out of the full chip comes at a cost of both on-chip resources as well as development time. Its implementation must be carefully gauged. Until then, manual reinitialization provides the same functionality while encouraging users to be more aware of which hardware parameters their experiments actually depend on.

10.5 Implementation

`quiggeldy` is developed primarily in C++ and runs on the server side as a dedicated daemon. On the client side `quiggeldy` acts as a regular (newly implemented) connection. The concept of reinitialization (cf. Section 10.4) is exposed to users via `ReinitStackEntry`-objects that are wrapped to Python. They are the only way users actively interact with the concept of `quiggeldy` presented here.

This section gives an overview over `quiggeldy`'s technical implementation details and demonstrates how template metaprogramming helps maintainability.

10.5.1 First implementation in `haldls`

A first prototype of `quiggeldy` was developed[11] in `haldls` for HICANN-DLS. It could already be deployed via `hagen-daas` integrated in Slurm, however, only in a testing deployment. One technical difference was the method by which a given hardware setup was allocated: Since early test boards were attached to specific compute nodes via USB,[12] the first implementation's daemon allocated `gress`[13] and had to be run on the setup-specific node. For comparison, the current implementation uses Slurm licenses for managing access to HICANN-X[14] chips. Since HICANN-X's FPGA is reachable via Ethernet from any compute node, there are no hard limits as to where a given `quiggeldy` daemon needs to run.

The first prototype helped identify key issues in the software stack's architecture, most prominently that connections to different backends were not abstracted away in `hxcomm` but "spilled over" into layers above. The reason to implement `quiggeldy` in `haldls` in the first place, was the fact that `stadls` encapsulated the concept of experiment execution via its `ExperimentControl` class. It has since been removed while addressing these issues (cf. Section 10.5.2).

Furthermore, the first implementation did not yet feature the reinitialization mechanism described in Section 10.4 and enforced each experiment to be fully contained in itself.

[11]`https://gerrit.bioai.eu/c/haldls/+/3325` (visited on 2021-03-25)
[12]Universal Serial Bus
[13]Generic RESources
[14]Short Form of HICANN-DLS-SR-HX, [Schemmel et al., 2020]

For various technical reasons, experimenters were sometimes forced to perform several experiment-steps. Some functionality on early test chips was adversely affected if a pbmem was actively being executed and not all chip features were accessible[15] via pbmem-based software abstractions yet. Hence, experiments needed to be split up into several steps. Here, it became apparent that reconfiguring the whole chip for even the smallest experiment-step is infeasible, giving birth to the idea of reinitialization that only reconfigures the whole chip when needed. Nevertheless, the general merit of quiggeldy's approach could be showcased.

10.5.2 Prerequisites in hxcomm: Connections as Handles

quiggeldy is designed to be as transparent as possible in order to not be "yet another thing" an experimenter has to worry about when writing scripts. Hence, an experiment script – once written with the concept of reinitialization (Section 10.4) in mind – should not have to be adjusted whether or not it is run via quiggeldy. In order to achieve this, all connections had to support the same API so that they could be used interchangeably. For weakly typed languages like Python, in which most user experiments are written, this would already have been possible since in Python everything is an object, the capabilities of which are only evaluated at runtime. But, since we expose the API to a strongly typed language, i.e., C++, the precise connection type needs to be known at compile time. This was no problem as long as there was a single hardware backend to execute on. For HICANN-X a co-development workflow was introduced [Grübl et al., 2020]. This lead to the introduction of SimConnection that connected via flange to a SystemVeriolog-based simulation of the chip, next to the already existing ARQConnection connecting to the real backend.

With more than one distinct connection type, different parts of software – such as hardware tests – were compiled into several binaries: One for every connection type. This meant that the user had to choose which connection to use by selecting which binary to run.

Furthermore, connections could be freely interacted with. They supported adding single words and committing, i.e., sending, them at arbitrary points in time. This method of access is needed for some hxcomm-internal tests, but is not necessary in upper layers which are concerned with executing whole experiments.

Therefore, the connection interface in hxcomm was reworked[16] in several major ways: First, connections are effectively turned into handles, unmodifiable tokens from which users can only retrieve read-only information. For all layers above hxcomm, the only way to execute anything on the corresponding backend is to pass both the connection handle and *a sequence* of UT messages, i.e., FPGA words, to execute_messages(). This function was originally implemented in fisch but moved to hxcomm. It transfers the supplied sequence

[15]Example for unsupported features at the time: Setting DAC configuration directly via SPI and not via FPGA instruction from pbmem: https://gerrit.bioai.eu/gitweb?p=frickel-dls.git;a=blob;f=src/frickel-dls/dacs.cpp;h=1e28445ea04868f829cdea0abb4b294c395c7ebf;hb=HEAD;js=1#l26 (visited on 2021-04-09)

[16]https://gerrit.bioai.eu/c/hxcomm/+/10315 (visited on 2021-03-26)

to the backend and retrieves responses until a `halt`-response is observed[17] and returns a sequence of response FPGA words. This effectively means that, conceptually, the smallest unit of execution moves from single words to a whole sequence that represents a coherent experiment-step. With this interface change we are merely enforcing a best practice that was already in place before, thereby ensuring that the choice of connection will not affect experiment results on a fundamental level.

Second, the next step is to make use of fairly recent advances in the C++ standard, namely `C++17` [ISO, 2017] and onward: Sum-types. At runtime, sum-types hold one of a select few many types, as opposed to product-types that hold several types at once.[18] The sum-type implementation in C++ is called `variant`.[19] Sum-types are somewhat similar to C-style `unions`,[20] but where a `union` is a reinterpretation of the same bits in memory as several types (typically `floats`, `ints` or other PODs[21]), sum-types are fully tracked by the type system and can hence only be accessed via a `visit` helper function.[22] We therefore introduce an aptly named `ConnectionVariant` that is a `variant` over available connections. Additionally, as in most cases we can infer what connection to use from the environment, we introduce a `get_connection_from_env()` function that returns a `ConnectionVariant` holding whatever connection type was inferred and initialized. All code can then simply be formulated in terms of `ConnectionVariant`.[23] The actual connection type is then evaluated at runtime when given to `execute_messages()`. As all functions dealing with connections,[24] it is implemented in a templated fashion so that, at compile time, code will be generated for whatever connection it was called with. The template can also be specialized for different connection types, including the full function signature which is inferred via templated helper structs that can be specialized for each connection as well.

Within `hxcomm`, the older interactive behavior of connections can be accessed via a `Stream` helper class,[25] as shown in Listing 7. The `Stream` object behaves just like the pre-rework connections did, e.g., allowing to send single FPGA words. Also, it can be constructed from `ConnectionVariant`. As described above, not all connections support the full `Stream` interface. A particular exception is the new `QuiggeldyConnection` which will be introduced in Section 10.5.3. All connections are tagged to indicate developer intent on whether or not a connection is supposed to support the full `Stream` interface. By using template metaprogramming, we can then automatically filter `ConnectionVariant`

[17]The FPGA can alternatively emit a timeout notification, indicating an error in the program (such as an omitted `halt`-instruction).

[18]The simplest example for product-types are tuples, however, any `class` with more than one member variable can be considered a product-type.

[19]`https://en.cppreference.com/w/cpp/utility/variant` (visited on 2021-03-26)

[20]`https://en.cppreference.com/w/cpp/language/union` (visited on 2021-03-26)

[21]Plain Old Data types

[22]`https://en.cppreference.com/w/cpp/utility/variant/visit` (visited on 2021-03-26)

[23]Of course, explicit connection types are still supported.

[24]This includes `execute_message()`'s equivalents in `fisch` and `stadls` named `run()`, extending `execute_message()`'s functionality to `pbmems`.

[25]Technically, the upper C++ layers could make use of the `Stream`-interface; prohibiting it, while possible, causes more overhead than it is worth. Not using the `Stream`-interface in upper C++ layers is hence enforced in code review (cf. Section 7.2). It is not exposed to Python at all.

```cpp
using namespace hxcomm;
using namespace hxcomm::vx;
using namespace hxcomm::vx::instruction;
/* connection creation omitted */
auto stream = Stream{connection};
// manually add halt instruction
stream.add(UTMessageToFPGA<system::Loopback>(system::Loopback::halt));
stream.commit();
stream.run_until_halt();
auto const responses = stream.receive_all();
```

Listing 7: Sending and executing a simple halt instruction on a connection supporting the full interface via `Stream` helper class.

to a `ConnectionFullStreamInterfaceVariant` which is used in all hardware tests that require precise access to connections via the full `Stream`-interface. Furthermore, we introduce helper structures which verify that all connections tagged as supporting the full `Stream` interface actually do so at compile time. At failure, they give more helpful error message than if a connection was simply "misused" in other parts of the code, saving developers time in the future. They effectively represent C++-concepts[26] which were not yet available at the time of writing.

Finally, Python bindings for all connections (including the variant) were adjusted to mimic their RAII-nature on the C++ side in pyhxcomm. This means that a connection is held for as long as its corresponding connection handle is created – and released once it's destroyed. Furthermore, `get_connection_from_env()`'s functionality is exposed via `ManagedConnection` so that, when it comes to connection handling all user code only needs to contain Listing 8. These `pyhxcomm::Handle<ConnectionType>` bindings are

```python
# ... (Create PlaybackMemoryProgram -> pbmem) ...
with pyhxcomm.ManagedConnection() as connection:
    # Connection is only valid within context.
    pystadls.run(connection, pbmem)
# The connection is already deallocated.
# All results implicitly stored in pbmem are still valid.
```

Listing 8: Simple user script in Python that executes a single pbmem via `stadls.run()` and works with all `hxcomm` connection backends.

templated over the wrapped connection type so that bindings for new connections can be easily generated. This means that bindings for additional connection backends can easily be generated.

[26] `https://en.cppreference.com/w/cpp/language/constraints` (visited on 2021-05-03)

10.5.3 Implementation in C++ via Template Metaprogramming

As depicted in Figure 10.2, the implementation of `quiggeldy` is split into two parts:

i) a scheduler that receives and distributes work from multiple users to a worker
ii) a worker that actually performs whatever task is implemented

We hence separate implementation of the scheduler in `rcf-extensions` from the task that is being executed, namely submitting experiment-steps via a wrapped connection in `hxcomm`.

Worker-Interface The scheduler is templated over a generic `Worker` type that has to adhere to a rather simple interface, shown in Listing 9. Through introspection and

```cpp
using namespace std;

class MyWorker
{
  // Acquire all resources to begin execution.
  void setup();

  // Map encoded user data to a user and session id if verified.
  // Otherwise, return empty optional to indicate invalid data.
  optional<pair<MyUser, MySession>> verify_user(string const&);

  // Execute the given unit of work (i.e., experiment-step)
  MyReturnType work(MyWorkParameters const& work);

  // Perform a reinit with the given data.
  void perform_reinit(MyReinitData const& reinit);

  // Release all resouces acquired during setup().
  void teardown();
};
```

Listing 9: Interface any `Worker`-type wrapped by `RoundRobinReinitScheduler` should adhere to. Through introspection, all types prefixed with “My” are extracted at compile time and automatically inserted in all interface signatures.

template metaprogramming, all relevant type information is extracted at compile time and automatically inserted into both RCF-interface as well as all helper structures involved in `rcf-extensions`. This allows for easier refactoring since there is a single source of truth and no type information from `hxcomm` bleed into `rcf-extensions::`-namespace. The RCF-interface, as the name suggests, is an implementation detail required by RCF. It defines the function signatures which can be called on the server side by a client. We provide a `RRWR_GENERATE(MyWorker,MyScheduler)`-macro that will generate a `MyScheduler_t`-scheduler class wrapping `MyWorker`, an `I_MyScheduler` RCF-interface supporting type

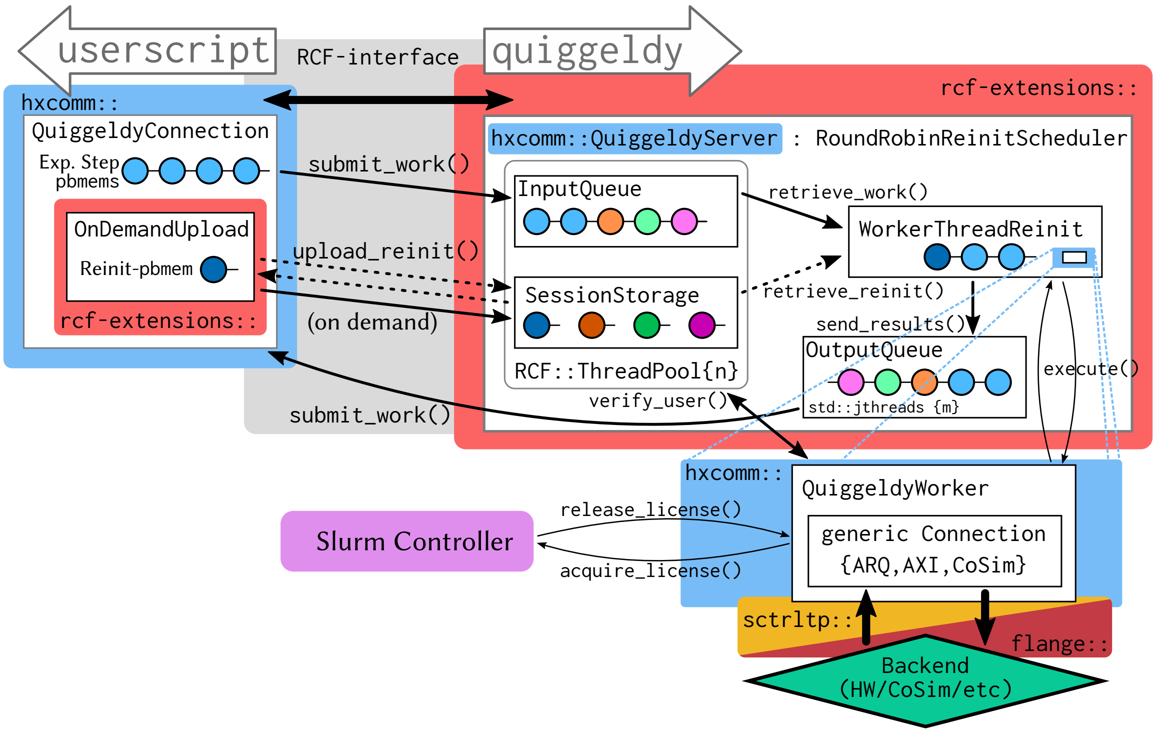


Figure 10.2: Overview of quiggeldy implementation: In user code, a QuiggeldyConnection is instantiated that communicates to quiggeldy via an auto-generated RCF-interface. Core functionality of quiggeldy is provided by a generalized scheduler that allocates work (circles) in a round-robin fashion and supports performing a reinit (darker shade) upon switching of sessions. It is defined in rcf-extensions, but extended with hxcomm-specific query options (not shown here) in hxcomm::QuiggeldyServer. Submitted experiment-steps are received and stored asynchronously in an InputQueue. Reinit-pbmems are uploaded on-demand to SessionStorage, transferred once needed via three way handshake. Besides managing the state of all reinit-pbmems, SessionStorage also tracks connection-count per user session, freeing up held reinit-resources as clients disconnect or time out. WorkerThreadReinit then concurrently retrieves work from InputQueue and reinits from SessionStorage if needed. Actual application specific tasks, i.e., performing reinit and computation, are then forwarded to a wrapped QuiggeldyWorker, the application context implemented in hxcomm. WorkerThreadReinit merely instructs the worker to perform tasks, such as resource allocation (setup), resource release (teardown), reinit (perform_reinit) and, most importantly, experiment execution (work). This separates abstract scheduling concept from specific application. On the application side QuiggeldyWorker executes work by wrapping any other connection available in hxcomm: Depicted here is direct Ethernet-based communication to an FPGA via sctrltp or RCF-based communication to a Co-Simulation using flange, but direct access via special memory regions as in the case of AXIConnection on the BrainScaleS-2 Mobile system (cf. Figure 3.8a) is also possible. Upon setup and teardown QuiggeldyWorker communicates with the Slurm controller to acquire or release the license governing the backend it is using. Furthermore, it provides methods to verify user information set in QuiggeldyConnection. WorkerThreadReinit pushes any results to the OutputQueue which delivers them concurrently back to the user script. Clients can submit work either synchronously or asynchronously in which case a running sequence number ensures that work is executed in order. As indicated by n and m, the number of threads handling incoming connections as well as delivering results is configurable.

information extracted from `MyWorker` and all utility classes needed for performing on-demand reinit uploads discussed below. In case the RCF-interface needs to be extended with application-specific functionality there are various other macros provided that allow for more fine-grained interface definition in “application-space”. This is especially needed when providing read-only access to the worker via visitors, discussed below.

Access Control Since the `verify_user(string const&)`-method of the worker-interface returns an optional-value to indicate an authentication result (cf. Listing 9), `quiggeldy` is easily extendible towards access control. Again, this access control is implemented in application domain, i.e., `hxcomm`. At the time of writing, we just verify proper MUNGE authentication (see below), but more advanced scenarios would be possible. For example, one setup could be allocated for a given publication but still be shared between all authors via `quiggeldy`. All other users would then receive an error when attempting to connect.

Client-side: `QuiggeldyConnection` On the client side, connections are established through `QuiggeldyConnection` that exposes the same interface as all other connections present in `hxcomm` (cf. Section 10.5.2). Hence, it makes no difference for users which connection their experiment code is executed on. Each instance generates a UUID[27] that identifies it to the server. This allows session tracking across multiple concurrent RCF-connections that are needed to support asynchronous experiment execution but also on-demand reinit upload (discussed below) during synchronous experiment execution.

`StreamRC`-Interface As introduced in Section 10.5.2, within `hxcomm` more fine-grained access to connections is provided via the `Stream`-interface for those connections that support it. Similarly, access to connections that support the reinit-based approach presented in this thesis in the form of `quiggeldy` is achieved by another helper class: `StreamRC`.[28] It exposes both synchronous and asynchronous submission as well as access to the reinit stack. In particular, the specialization of `execute_messages` for `QuiggeldyConnection` is done via `StreamRC`. Same as `Stream`, attempting to construct an instance of `StreamRC` with a connection not supporting it will lead to a helpful error message at compile time. Furthermore, it is meant as implementation utility to facilitate integration of asynchronous dispatch in other software layers (cf. Section 10.6).

`QuiggeldyFuture` Calls to asynchronous dispatch routines return a `QuiggeldyFuture` which tracks all state of the asynchronous call. RCF requires the client making the call to live for as long as the `RCF::Future` that in itself represents the delayed computation. `QuiggeldyFuture` supports all typical future operations, i.e., checking if the result is available or blocking until it is, optionally with a timeout.

[27]Universally Unique IDentifier
[28]`Stream`-interace with for Remote exeCution (pronounced “streamers”)

User-Verification via MUNGE User sessions are authorized via MUNGE, the same solution used by Slurm. On the user-side, session information is encoded using a locally available MUNGE-socket. It uses a shared secret to symmetrically encode/decode messages along with user and group id information that regular users cannot influence. The server can then group all experiment-steps submitted by one user in a single queue. Scheduling then selects one user queue after the other in a round robin fashion, ensuring each user is able to submit the same number of experiment-steps. We therefore prevent one user from getting an experiment throughput advantage by opening several connections. The implementation is extendible towards other scheduling strategies. For example, if the need arises, we could use recent average runtime of experiment-step per user and choose the next experiment-step based on that. Runtime per experiment-step is already being tracked and logged for statistical purposes. MUNGE-support can be toggled both at compile[29] and runtime. Of course, if it is enabled on the server, any client without compiled-in MUNGE support will not be able to connect and submit work.

Reinit Users track their configuration state via reinit-`pbmem`. These should be able to support rapid client-side updates while only transferring them to the server when needed. RCF does not support the server calling methods on the clients without a completely separate RCF-interface in the opposite direction, including an RCF-server running on the client.

Therefore, we implement on demand uploads via a single RCF-interface with a procedure similar to a three way handshake in TCP. Its core idea is demonstrated in Figure 10.3. On the client-side, user code informs `OnDemandUpload` of new or updated reinit data. This causes a new reinit id to be generated which identifies this particular instance of reinit data. The only requirement for the reinit id is that it is different between upload events. At the time of writing it is implemented as a random number drawn from a globally available random number generator[30] which is sufficient for the small number of distinct sets of reinit data we expect. Other implementations, such as hashing the supplied reinit data would be viable as well.

After generating the reinit id the server is immediately notified (corresponding to the `SYN`-step in a three way handshake). Once the `notify()`-call returns (corresponding to `SYN/ACK`), control is returned to user code that had been blocking up until this point. The last step (`ACK`) is performed in a separate thread that calls a designated `pending()`-function. The server defers this call until it actually requires its corresponding reinit data by storing its `RCF::RemoteCallContext`. Once `pending()` completes with a return value indicating an upload request, the client will commence uploading.

In order to make the overall system more robust, `OnDemandUpload` uses a loop to execute `notify`→`pending`→`upload`. This allows for restarts of the server not to affect correctness of the reinit state as the reinit data will be uploaded again immediately. At most, currently submitted experiment-steps might fail, but typically user-side code already has counter-measures in place to deal with potential connection errors. This is also useful during a situation requiring continuous execution such as a live-demo (see Section 10.7.2)

[29]Disabling MUNGE at compile time eliminates MUNGE-related libraries from the dependency list.
[30]`https://en.cppreference.com/w/cpp/numeric/random/random_device` (visited on 2021-04-01)

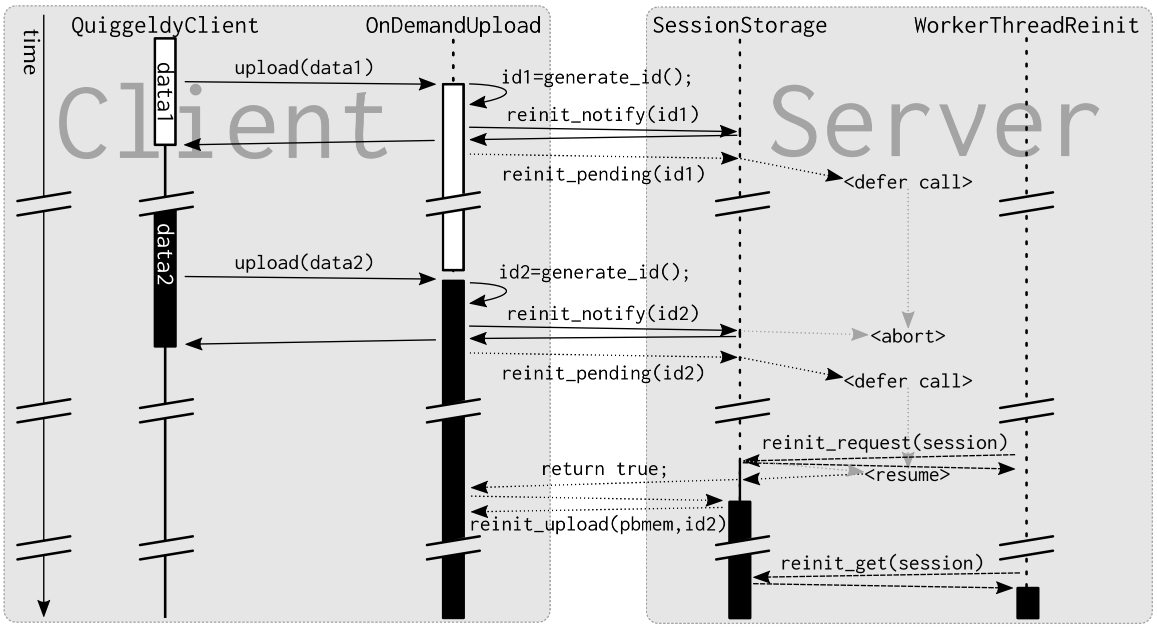


Figure 10.3: Simplified workflow schematic of on-demand upload for reinit-pbmems via three way handshake. As soon as user code registers new reinit data, a new reinit id is generated and the server is notified. In order to avoid race conditions, execution in user code (solid arrows) blocks until the server acknowledges the notification by completing the `notify()` call. `OnDemandUpload` then issues a concurrent `pending()` call (dotted arrows) that is deferred on the server until reinit data is actually needed. Hence, updated reinit data only causes the notification to be updated while no potentially large upload is performed yet. As soon as `WorkerThreadReinit` (dashed arrows) requests a reinit – done at session switch – the `pending()` call is commenced with a return value indicating that the upload should be performed. `OnDemandUpload` then calls the final `upload()`-routine. When `WorkerThreadReinit` needs the reinit data, it is retrieved from `SessionStorage` and can be used immediately. Please note that all calls from `OnDemandUpload` to `SessionStorage` are run through `RoundRobinReinitScheduler` that implements the RCF-interface server-side. This mechanism is unique for each user session, i.e., for each `QuiggeldyConnection` instance.

or surveillance of sensory data, where single failed steps need to be tolerated and not immediately terminate the whole experiment pipeline.

Reinit-Stack As discussed above, the implementation in `rcf-extensions` is templated over whatever data type is used to perform reinitializations. In order to allow for more flexibility on the user side, we choose in `hxcomm` for reinit data to be a sequence of pbmems transpiled to UT-messages. New instances of `ReinitStackEntry`, created from a connection handle, are pushed on top of the stack that is itself held within the connection. Adhering to RAII, destroying a `ReinitStackEntry` will remove it from the stack, throwing an error if it was not on top. Each `ReinitStackEntry` holds a single `pbmem` that can be updated via a setter at any time. Similarly named structures are implemented in `fisch` and `haldls` to provide reinit functionality for all levels of abstractions. Due to the indeterminism of garbage-collection[31] in Python, users are advised to manually call

[31]An object that goes out-of-scope or gets deleted is not guaranteed to be immediately destroyed as is the case in languages with more explicit memory management such as C++ or Rust.

entry.pop() on ReinitStackEntrys once they are not needed anymore. Of course, for connection handles that do not support reinits, only enforced reinits are immediately applied, all others discarded (as discussed above).

Order of Execution in single Experiments A single QuiggeldyConnection assigns each executed experiment-step an ever-increasing sequence number that is used both for logging purposed as well as ensuring that asynchronously submitted steps are executed in the order they were submitted in and no steps are lost. This is important when scheduling dependent executions via grenade where some intermediate results might be required earlier than others. However, users can also specify that their experiment-steps are executed out-of-order, i.e., in whatever order they arrive on the server side. If *one user* uses multiple instances of QuiggeldyConnection in an asynchronous way, i.e., there is more than one separate experiment for a single user, quiggeldy will attempt to sort submitted work in a way that minimizes the number of required reinits for the user. Especially, this means that, for a single user, different experiment streams are *not* scheduled in a round-robin fashion as between users. All functions related to work-retrieval are templated over the actual sorting implementation, allowing for an easy drop-in replacement should the need arise. Overall, the order of execution is a per-user setting. Scheduling between different users is always round-robin.

Extending RCF-Interface to retrieve information from Worker For some use cases, it is important to retrieve information from the server side. Examples include whether or not MUNGE-authentication is enabled, version information with which software state quiggeldy was built and, maybe most importantly, which hardware resource is actually wrapped by the daemon. Especially the latter is very important in layers such as hxtorch in order to automatically load the latest nightly calibration for the given chip.

In order to still keep scheduling concept and application strictly separated, RoundRobinReinitScheduler has support to retrieve read-only information from the wrapped worker object via the Visitor pattern. Listing 10 shows two protected functions

```cpp
template <typename VisitorT>
auto visit_worker_const(VisitorT visit) const;

template <typename VisitorT>
auto visit_set_up_worker_const(VisitorT visit);
```

Listing 10: Protected functions provided by RoundRobinReinitScheduler. They can be used in derived classes to to access read-only attributes of the worker object.

provided by RoundRobinReinitScheuler that take a visitor[32] which is called with a const reference of the worker (i.e., visitors cannot modify the worker). In the latter case the worker object is ensured to be set up – which for QuiggeldyWorker means it has an active

[32]In some languages this pattern is also known as applying a lens, even though a lens is usually a combination of a getter and a setter whereas here we only have the getter part.

connection.

In application-specific code, we can then derive `QuiggeldyServer` from `RoundRobin-ReinitScheduler` and use the protected functions to implement information retrieval conveniently. For example, acquiring the aforementioned unique hardware identifier can be implemented with two lines of code, as shown in Listing 11. New methods need to be

```
std::string get_unique_identifier(std::optional<std::string> hwdb_path)
{
    return parent_t::visit_set_up_worker_const(
        [&hwdb_path](auto const& worker) {
                return worker.get_unique_identifier(hwdb_path);
            });
}
```

Listing 11: Example for the implementation of information retrieval using the protected helper functions from Listing 10. Here, `parent_t` is a simple type-alias to `RoundRobinReinitScheduler`. Deriving a server class from `RoundRobinReinitScheduler` and extending the RCF-interface can both be done in application code, i.e., `hxcomm`. No code adjustments in `rcf-extensions` are necessary.

added to the RCF-interface and `QuiggeldyServer` explicitly bound to it at runtime, again for RCF-related technical reasons. A convenience `bind_to_interface<interface>()` is provided, allowing the derived class to bind itself to the extended RCF-interface. Due to the helper macros mentioned above, the complete implementation can be performed in `hxcomm` only, therefore allowing for rapid functionality extensions.

Serialization RCF serializes all types involved in RCF-interfaces with its own serialization framework: SF.[33] The BrainScaleS-2 software stack uses `cereal` for serialization. The first iteration of serialization was implemented in a very bare-bones, straightforward way: We implemented a simple translational layer by providing a templated `translate_sf_cereal`-function that converts any "cerializable" type to a binary representation (optionally endianness-portable) that is then sent via SF as is. On the remote site, the reverse operation was performed. As shown in Section 10.8, this naive implementation added some overhead, but allows transmission between different target architectures. If the same implementation is used on both ends, there is a faster option implemented that just sends raw binary data without serialization, as shown later in Figure 10.7. The remaining serialization and transmission delay, which will always be small but non-zero, will effectively be amortized when submitting experiment-steps in asynchronous fashion.

Testing via Mock-Workers The split implementation allows for verifying the scheduling concept independent from its implementation. For this, a `WaitingWorker` is provided that merely keeps track for which session it is currently set up. It asserts both proper ordering of observed sequence numbers as well as proper reinitialization and throws an exception if it detects any inconsistencies. In order to identify potential race-conditions both reinit and regular workloads are simulated via `sleep()`s that range up to the order

[33]Serialization Framework of RCF

of milliseconds.

There are tests for all different components: Round-robin scheduler (without reinit), round-robin scheduler with reinitialization and on-demand uploading. These are executed as part of lib-rcf's integration tests as described in Section 7.1.

Robustness Since quiggeldy should be as transparent as possible to users, emphasis during design was put on robustness. To that end, quiggeldy has several features that ensure users can continue to work despite errors: Each hardware response is concurrently scanned for timeout notifications that indicate an FPGA error. If any such notification is detected in-between pushing the current response to OutputQueue and retrieving the next request from InputQueue, we forcibly reconnect to the selected backend[34] so that the next experiment-step will have a "fresh" connection without error-state.

So far during deployments (cf. Section 10.7), we did not encounter any segmentation faults or other crash-inducing bugs, but as features are added, so may be bugs. Therefore, as already mentioned above, the reinitialization mechanism is implemented in such a way as to allow for server-side restarts. To users this appears as any other hardware error, e.g., timeout notifications. Clients will immediately restore their reinit state and continue executing, as long as they have measures in place against potential such hardware errors. This would even allow for in-place updating while experiments are running, as long as low-level layouts of FPGA messages in hxcomm do not change. However, this is not advised as general best practice.

10.5.4 Configuration

Environment When creating a connection from environment, hxcomm checks the following environment variables in regards to quiggeldy:

QUIGGELDY_ENABLED	Indicates the user's intent to use a quiggeldy-based connection. If defined *and* non-zero, hxcomm will attempt to create a QuiggeldyConnection and terminate if not successful.
QUIGGELDY_IP	Should contain the remote IP of the host where the quiggeldy daemon is running.
QUIGGELDY_PORT	Should contain the remote port on which the quiggeldy daemon is listening.
QUIGGELDY_USER_NON_MUNGE	In case the remote quiggeldy daemon does not enforce user authentication via MUNGE, this environment variable can be set specify as which user to identify. This is useful in situations where several QuiggeldyConnections are to

[34]In most cases we will detect only timeout notifications when connecting to actual hardware via ARQConnection or AXIConnection.

be run by one user with proper round-robin scheduling. An example would be the NICE 2021 hands-on tutorial (see Section 10.7.2) where all compute jobs of participants were run by one system user.

Additionally, on the server side there are other environment variables influencing behavior:

`QUIGGELDY_LOGEVEL` Set `quiggeldy`'s loglevel. Should be set to one of `trace`, `debug`, `info`, `warn` or `error` (see `--loglevel` in Appendix B.1 for an explanation). This is especially useful in combination with `wriggeldy` (see Section 10.5.5).

`QUIGGELDY_TIMEOUT` Specifies the number of seconds until `quiggeldy` terminates itself when started via `wriggeldy` (see Section 10.5.5).

Command Line Arguments At the time of writing `quiggeldy`'s binary itself is configured via a set of CLI arguments. Here we present only a subset that showcase important functionality, the full list can be found in Appendix B.1.

`--mock-mode`
: `quiggeldy` features a mock-mode.[35] When enabled, `quiggeldy` accepts connections as usual, but does not connect to the "real" hardware backend. Instead, empty responses are returned. This is useful to troubleshoot connectivity issues without blocking any physical hardware resources. It is used in `hxcomm` software tests to ensure principle functionality.

`-r / --release <seconds>`
: Sets the number of seconds between seconds between releases of Slurm allocations. This allows other jobs that require exclusive hardware access to be scheduled. A value of zero causes `quiggeldy` to immediately release the Slurm allocation as soon as there no experiments pending execution.

`-t / --timeout <s>`
: Especially when combined with `hagen-daas` (which is introduced in Section 10.9), `quiggeldy` daemons should not idly waste compute resources that are not needed. Hence, the number of seconds after which `quiggeldy` shuts itself down after being idle can be specified. If set to zero, `quiggeldy` will *not* terminate itself when idling. The timeout should be chosen larger than any conceivable update period for a user job. In case of shutdown, `hagen-daas` will reactivate `quiggeldy` if a new user job for the same hardware resource is scheduled. See Section 10.9 for details.

`-u / --user-period-ms <ms>`
: `quiggeldy` supports the concept of user periods. It is the minimal amount of milliseconds that a given user has access to the hardware before `quiggeldy` switches to another user, if there are any. By default, this feature is disabled, but it might be useful

[35]Not to be confused with the mock-mode in `hxtorch` that emulates hardware behavior.

in future scenarios. One such scenario involves several users performing parameter sweeps for an experiment with a reinit that takes a relatively long time compared to executed experiment-steps. If experiment-steps are submitted asynchronously, `quiggeldy` can execute several experiment-steps for one user in rapid succession for one reinit prior to switching to another user. In this case, switching after executing a single experiment would lead to a decrease in performance. However, in the default case we expect relatively short reinit durations compared to actual runtime of experiment-steps.

`-v / --version`
: Due do to the "living at HEAD" paradigm employed at Electronic Vision(s), semantic versioning appears rather pointless. Instead, `quiggeldy` notes down the state of `hxcomm` and all dependency repositories at the time of compilation, i.e., which `git`-commit `HEAD` points to and if the working area is "dirty" (i.e., modified). This information, as well as the compilation date, can be requested via this switch. As explained in Section 10.5.5, versioning information can also be obtained from remote running `quiggeldy` daemons via `viggeldy`.

10.5.5 Utilities

In addition to `quiggeldy` itself, there are some utility binaries deployed alongside. Some utilities are accessible via both verbose long name and a shorter one that can be quickly tab-completed on the terminal.

`quiggeldy_mock_client`
: In conjunction with `quiggeldy`'s mock-mode (described above), the mock client helps diagnosing connectivity problems. At the time of writing, its functionality is limited to establishing a connection an sending an empty sequence of FPGA words to which it expects an empty response.

`viggeldy` $\longrightarrow$ `quiggeldy_query_version`
: At compile time, the current version information, i.e., toplevel commit hash and title of all dependencies, are compiled into `quiggeldy`. In order to access this information from remote users can execute `viggeldy`. This allows users to identify *exactly* what code is running on the remote site and helps diagnosing problems.

`wriggeldy` $\longrightarrow$ `wrap_with_quiggeldy`
: When prefixing an ordinary executable or user script with `wriggeldy`, a local `quiggeldy` instance is started that extracts hardware backend information from the environment in the same manner a user script would. Then it adjusts its process environment so that the then spawned wrapped process connects to this `quiggeldy` instance. The reasoning for this is simple: Whenever a new concept is introduced, it needs to be tested thoroughly. Also, part of debugging any problem is eliminating alternatives by testing them in isolation. Hence, `wriggeldy` can be used to not only verify that a given user script is capable of being executed via `quiggeldy`, but also

to verify code modifications to `quiggeldy` itself. By binding to a random unused port, we avoid conflicts with any other services running on the same machine. The launched `quiggeldy` instance is configured to terminate after being idle for a few seconds. This can be adjusted via the `$QUIGGELDY_TIMEOUT` environment variable.

10.6 Integration into other Layers

In order for `quiggeldy` to work more seamlessly, i.e., invisible to the user, it is integrated into higher layers of the software stack. The end goal for `quiggeldy` is to be completely invisible to the experimenters specifying their scripts. Instead, upper layers should both ensure reinitialization and make use of `quiggeldy`'s asynchronous submission of experiment-steps wherever possible. Some of these efforts are presented here.

10.6.1 `hxtorch`

Since `hxtorch`, at the time of writing, strictly operates in non-spiking HAGEN[36]-mode[37] tracking necessary chip state is rather straight forward. All operations involving hardware executions are already self-contained. The only amount of state they require is a loaded configuration. Hence, we simply register the loaded calibration `pbmem` as reinit data.[38] This ensures the chip is always calibrated when performing MACs.[39]

In the future, when larger models and inputs are being processed, this mechanism could be extended to add additional reinit steps that ensure the correct weight matrix is loaded while the input data is chunked into pieces that fit into FPGA memory and submitted asynchronously for execution. Going even further, this could be expanded to a general streaming mode of operation that is already supported by the FPGA.[40] Here, received experiment-steps would be immediately streamed to the FPGA as soon as they are available, possibly interleaved with reinit data upon a switch of users.

10.6.2 `PyNN.brainscales2`

As described in Chapter 6, PyNN provides its own set of abstractions such as `Populations` connected via `Projections`. Users call `pynn.setup()` to initialize the backend, then proceed to create the networks. The model is then run by calling `pynn.run(duration)`. As shown in Figure 6.1, the long term goal is to have all hardware execution be performed by `grenade`. At the time of writing, however, network execution is split into two parts: First, a `pbmem` is generated and executed in PyNN`.brainscales2` directly. It contains

[36]Heidelberg AnaloG Evolvable neural Network
[37]Support for spiking-mode is planned.
[38]`https://gerrit.bioai.eu/c/hxtorch/+/12677` (visited on 2021-04-03)
[39]Multiply-ACcumulate operations
[40]Personal correspondence with Dr. Vitali Karasenko.

static configuration information derived from calibration data with potential expert user adjustments that grenade is not yet supporting. Then control is handed over to grenade which performs the actual simulation run.

Rudimentary integration with quiggeldy is straight-forward[41]: As it yields no results to have the configuration run in isolation, it is registered as enforced (i.e., mandatory) reinit step. Hence, when grenade – which currently is not aware of quiggeldy – performs the actual execution, the chip is configured correctly. Of course, this means that the reinitialization acts more like another step of the experiment. Since the reinit data will be refreshed prior to every run(), it will never be skipped even though the configuration might not have changed between two different run()-invocations. Please note that this is *not* inherently caused by using quiggeldy, but rather due to PyNN.brainscales2 still being in a stage of development where feature-completeness is more important than premature optimization.

In the future, it is planned for grenade to be aware of the reinitialization concept so that it can register the initial configuration itself. This way, all experiment execution would be handled handled by grenade.

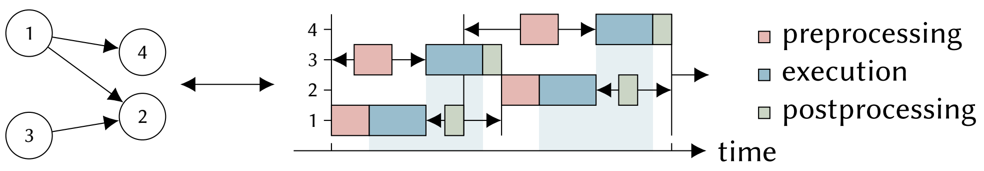


Figure 10.4: One of grenade's features is JIT execution (right) of a dependency-graph (left) consisting of four execution instances. Each execution step that has all its dependencies available can be scheduled asynchronously via quiggeldy. Adapted from: [Spilger et al., 2020, Fig. 5].

10.6.3 Outlook: grenade

As mentioned above, grenade, which also deals with scheduling tasks on the level of single models, is not yet integrated into the quiggeldy workflow. As shown in Figure 10.4, its main task is generating a data flow and compute graph of the model and identifying which operations are best scheduled in what order in order to maximize experiment throughput (or, conversely, minimize experiment runtime). In the future, this will involve distributing operations of a single model across multiple chips. Using quiggeldy's asynchronous capabilities, distributing across different chips is straightforward to implement.

10.7 Deployments

During its development, quiggeldy has been deployed several times in order to test-drive its reliability.

[41]https://gerrit.bioai.eu/c/pynn-brainscales/+/13998 (visited on 2021-04-03)

10.7.1 | Remote Execution on different Instruction Set Architecture

In the final phase of the competition BMBF[42] Pilotinnovationswettbewerb "Energieeffizientes KI-System", training and verifying the model on the BrainScaleS-2 Mobile (see Section 13.3) was undertaken via `quiggeldy`. The on-chip processor on the Zynq Ultrascale+ [Xilinx, 2019] is an ARM64[43]-based SoC[44] and therefore incompatible to the `x86-64` architecture deployed on regular Electronic Vision(s) cluster nodes. Hence, an ARM64-based `visionary`-package (using the same principles as outlined in Sections 8.1.4, 8.2.3 and 8.3) was cross-compiled[45] to provide the necessary runtime environment.

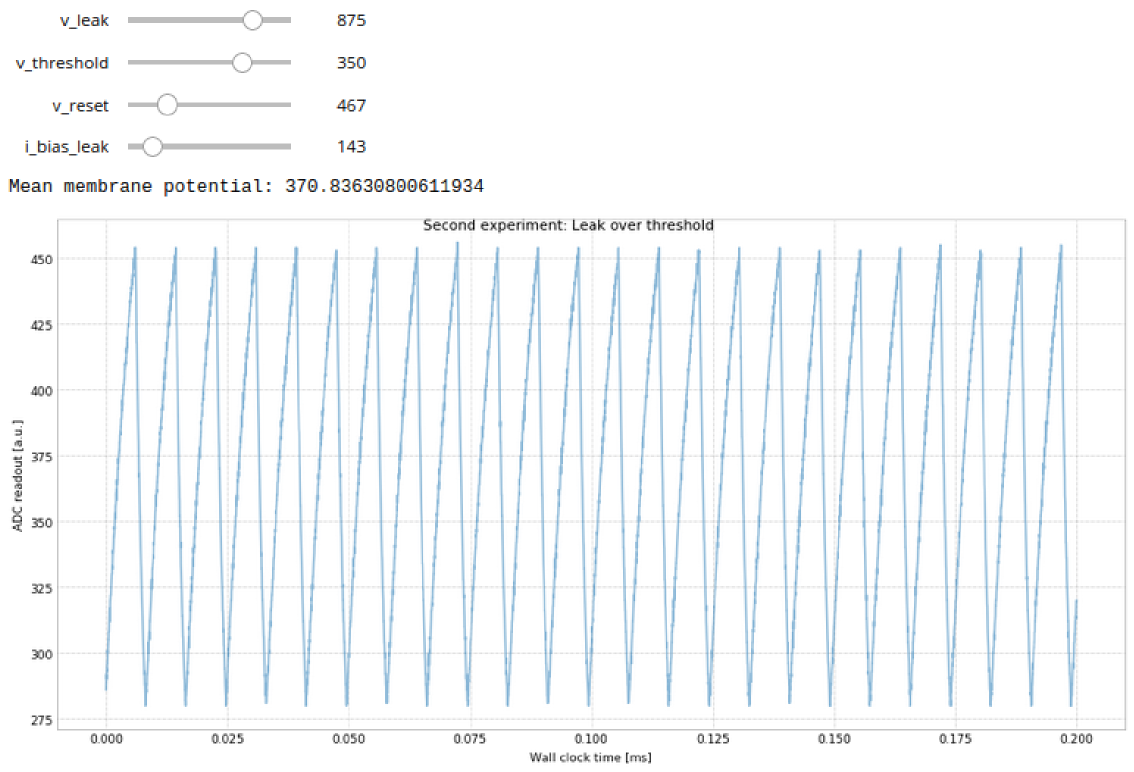
In order to avoid executing the whole compute stack (Chapter 6) – including Python scripts in the topmost layer – on the embedded co-processor, `quiggeldy` was used to effectively split the stack in two: On the one hand, the C++ part, i.e., everything up until `hxcomm`, was deployed without Python-bindings to the SoC, compiled for ARM64-architecture. On the other, the complete software stack, compiled for `x86-64` was kept in users' workspaces. At the most they needed some additional Gerrit-changes (see Section 7.2) related to BrainScaleS-2 Mobile's ARM64-nature. `quiggeldy` was then used to execute experiment-steps on BrainScaleS-2 Mobile. These included both training and evaluation.

It was straightforward to extend `quiggeldy`'s functionality to wrapping the newly introduced `AXIConnection`.[46] In order to limit the vulnerability surface to errors somewhat, `quiggeldy` was restricted to single-user mode only, same as the regular non-`quiggeldy` usage in the Electronic Vision(s)-cluster. Users negotiated with each other in order to access BrainScaleS-2 Mobile-setups. Please note that this restriction has since been lifted and multi-user access to BrainScaleS-2 Mobile-setups is supported. A major benefit of using `quiggeldy` was that it allowed users to continue working in their regular home-directories and not move their workspace partially onto the resource-limited SoC.

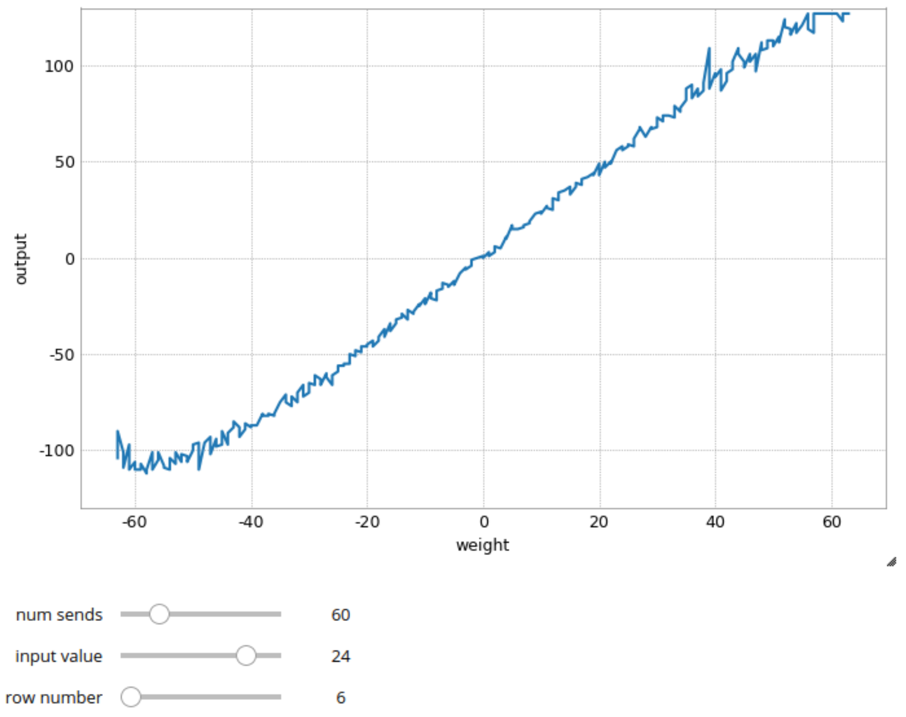
During the competition, executing experiments across ISA-boundary was possible without any major obstacles. Occasionally – about once per night of continuous sweeping – we observed low-probability serialization flukes on the client side regarding single steps that did not affect the daemon running on the SoC. They could be easily mitigated by executing the corresponding step again and are pending a more thorough investigation, especially in regards to different serialization implementations (cf. Section 10.8).

Please refer to Chapter 13 for more details.

[42]Federal Ministry of Education and Research (*Bundesministerium für Bildung und Forschung*)
[43]Advanced RISC Machines
[44]System on a Chip
[45]Thanks in particular to Dr. Eric Müller who "volunteered" countless hours to iron out ARM64-related build problems.
[46]`https://gerrit.bioai.eu/c/hxcomm/+/13034` (visited on 2021-04-04)

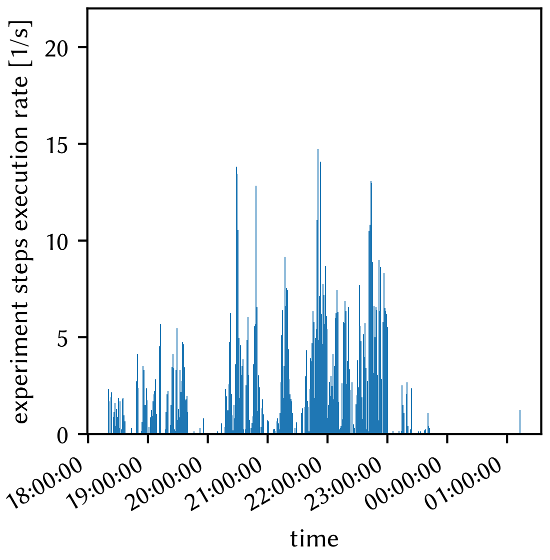

```
v_leak        875
v_threshold   350
v_reset       467
i_bias_leak   143
Mean membrane potential: 370.83630800611934
```

(a) Screen shot of interactive demo of a single spiking neuron during BrainScaleS-2 hands-on tutorial at NICE 2021. Users can vary leak, reset and threshold potential as well as leakage conductance via sliders. The resulting plot then shows the generated membrane voltage trace as read out by the on-chip ADC. The demo was set up by Johannes Weis.

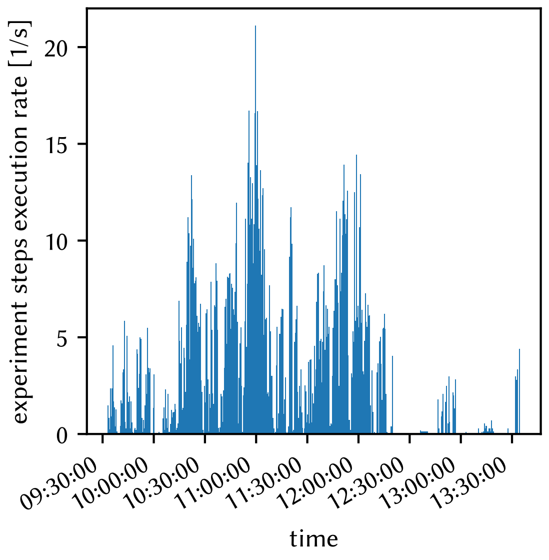

```
num sends      60
input value    24
row number     6
```

(b) Screen shot of interactive demo of a non-spiking, i.e., HAGEN-model, MAC operation during BrainScaleS-2 hands-on tutorial at NICE 2021. Users can vary the input value, the number of repetitions with which it is being sent and the chip row whose results to display via sliders. The weight matrix is set up in a way to feature all possible weights from −64 to 64. The demo was set up by Arne Emmel.

Figure 10.5: Two examples of interactive user demos, both spiking and non-spiking. Once users adjust the sliders the underlying Python code performs an emulation in hardware and displays the new results almost instantly, the whole pipeline including rendering executes in under a second.



(a) First slot at 16th March 2021.



(b) Second slot at 18th March 2021.

Figure 10.6: Rate of executed experiment-steps via `quiggeldy` during the BrainScaleS-2 hands-on tutorial at NICE 2021. Experiments were distributed among 8 hardware setups. In total there were 86 077 experiment-steps executed.

10.7.2 Interactive hands-on Tutorial at NICE2021

During the NICE 2021,[47] a hands-on tutorial showcasing the capabilities of the BrainScaleS-2 platform was conducted.[48] It featured two time slots of 3 hours and was attended by 70–80 people in total, 66 of whom executed at least one experiment. Eight hardware setups were provided during the demo. The number of users therefore greatly outweighed the number of hardware setups.

In the past, at similar live-demo occasions, Slurm was used to schedule hardware access via short running jobs. This had the same issues as outlined Section 10.1 and overall was inconvenient to use, because Slurm jobs could only be executed as a whole script. Therefore `quiggeldy` was deployed to facilitate concurrent hardware access.

Users were able to use a set of provided Jupyter[49] notebooks, i.e., interactive web applets that allow for execution of Python code. Each notebook was spawned inside a compute job that was configured to connect to a specific hardware setup via environmental settings (see Section 10.5.4). Users were distributed pseudo-randomly but fixed across hardware setups so that their assigned hardware setup would not change even if they needed to restart their notebook kernel. Access to each hardware setup was managed by one `quiggeldy` instance. All `quiggeldy` instances ran on the same cluster node and were configured by providing different command line arguments (cf. Appendix B.1). Notebooks included both spiking as well as non-spiking examples, as seen in Figure 10.5. Users were able to experiment with

[47] Neuro-Inspired Computational Elements Conference 2021

[48] Live-recordings are available at: `https://youtu.be/_WiFvEpKNuI`, `https://youtu.be/KN-xhNiuAV4` and `https://youtu.be/XAlLdLtMWEI` (all visited 2021-05-03).

[49] Project Jupyter – Julia Python and R, `https://jupyter.org` (visited on 2021-04-04)

parameter settings via sliders and get immediate feedback from hardware. All integrations to other layers described in Section 10.7 were in place. This meant that reinitialization (cf. Section 10.4) could be handled completely transparent.

Figure 10.6 shows the time distribution of executed jobs in form of a histogram. As shown in the live-recordings, the tutorial was conducted in phases: First a tutor would explain a given experiment and then encourage each member of the audience to try for themselves until it was time to move on to the next experiment-step. We can see this pattern in the job counts being clustered at certain times. While first experiments (as shown in Figure 10.5) mainly involved adjusting parameters and running single experiment-steps, later experiments would also perform small training examples by executing several steps in the loop. In total users performed 86 077 experiment-steps. Despite having up to ten tutors present to intervene should problems arise, users did not report any issues once entering their notebooks. In particular, there were no problems with users executing both spiking and non-spiking demos on the same setup, seemingly at the same time. Overall, user feedback was very positive. Hence, the hands-on tutorial can be regarded a huge success thanks to all people involved. Everything ran smoothly. Regarding `quiggeldy`, it was an important step to showcase the soundness and stability of this approach. It is a crucial first step to providing a pleasant and easy-to-use non-expert user experience.

After the hands-on tutorial concluded all but one HICANN-X setup were inserted back into the regular cluster. The last remaining setup was then set up to service all notebooks, should any participant still be interested in going through the examples at a later time. This setup was also used to conduct experiments during “Girl’s Day 2021”[50] without any problems. At this point, `quiggeldy` had already been running continuously for more than 30 days without error. A similar approach could be used when implementing access via `EBRAINS` Colab.[51]

10.8 | Overhead-Evaluation

We evaluate the potential overhead `quiggeldy` introduces by using a set of simple toy examples. As test case we multiply a full 100×100 matrix with a vector of length 100 and vary the batch size between 1 and 10^5. We use `hxtorch.matmul()`, i.e., HAGEN-MAC (cf. Section 3.2.2), to perform the operation. `hxtorch` as support for batched matrix multiplication and so the whole operation is fitted into a single `pbmem`. Figure 10.7 shows the timing results. The experiment protocol is executed twice in the same job: The first pass is done connecting directly via `hxcomm::vx::ARQConnection` (orange line). Afterwards, in the same compute job, we execute the same script prefixed with `wriggeldy` (cf. Section 10.5.5) so that execution happens via local `quiggeldy` that internally uses an `hxcomm::vx::ARQConnection` (blue line). We therefore ensure that the only difference between both runs is the detour over `quiggeldy`, otherwise the execution happens on the same node with the same conditions. Since we use a local `quiggeldy` there are no other

[50] `https://archive.is/l1bIu` (visited on 2021-04-23)
[51] Colaboratory

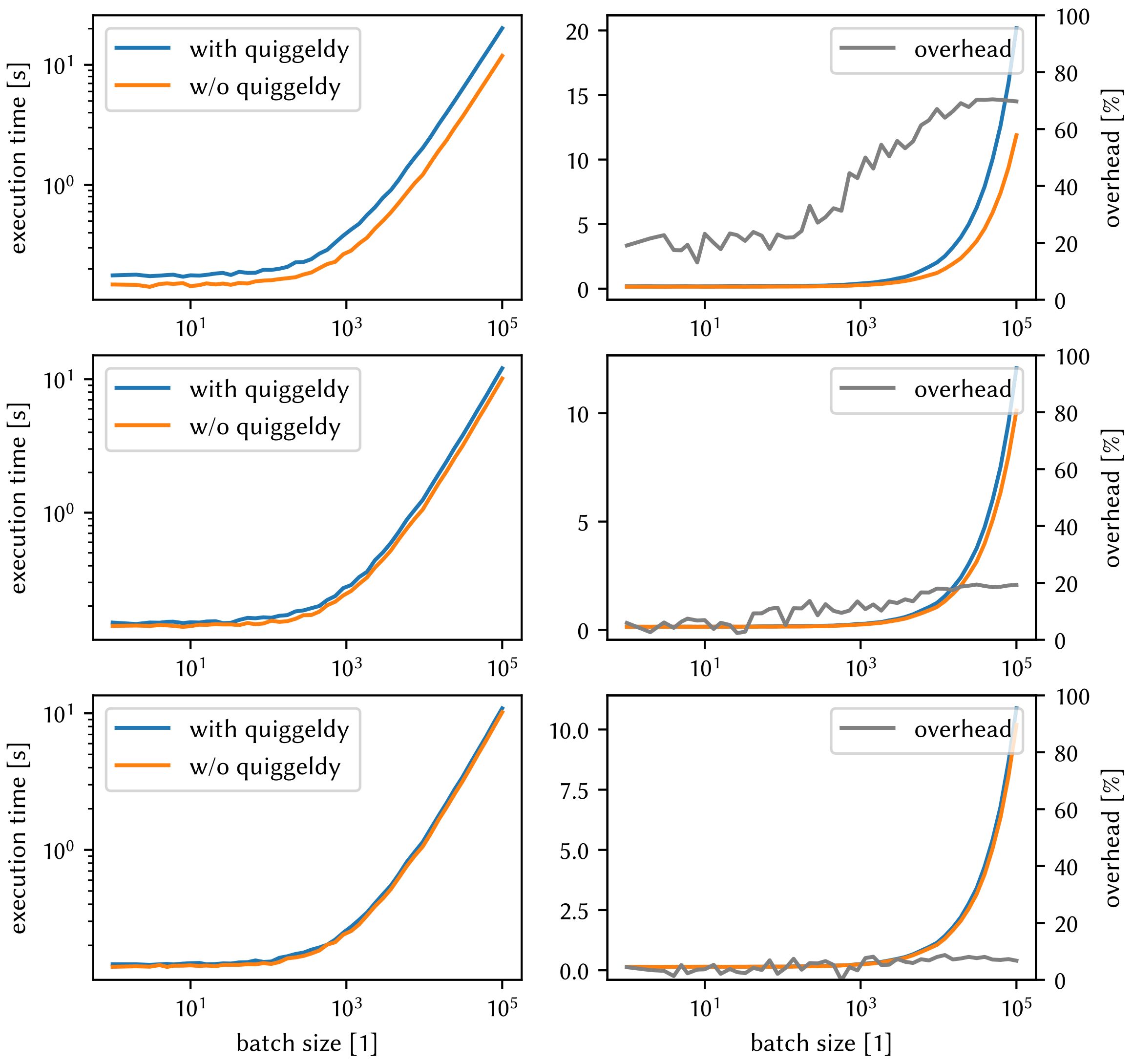


Figure 10.7: Overhead evaluation for `quiggeldy` in non-spiking mode with different types of serialization. As test case we multiply a full 100 × 100 matrix with a vector of length 100 and vary the batch size. Each batch size is executed 10 times and the minimum taken. The only difference between both executions is a wrapping with `wriggeldy`. The same data is shown with different scaling: *left* (logarithmic) and *right* (linear).
Top: Naïve straightforward implementation using `cereal` to fill an SF buffer. We notice a measurable increase in execution time of about 20 %. As we increase batch size, corresponding to the absolute size of the experiment-step being executed, we observe the overhead increasing steadily to more than 70 %.
Middle: Improved serialization implementation relying on existing methods to perform fast vector serialization provided by SF while relying on `cereal` for time-tracking related data structures. This reduces relative overhead for large payloads by almost a factor of four to (18.7 ± 0.6) %.
Bottom: Sending raw bits because object representations on both client and server side are identical and only comprised of PODs. The relative overhead never exceeds 9 % and averages around (7.5 ± 0.6) % for larger payloads. Ultimately, we see that the overhead is within acceptable margins, but further optimization is bound to reduce it even further.

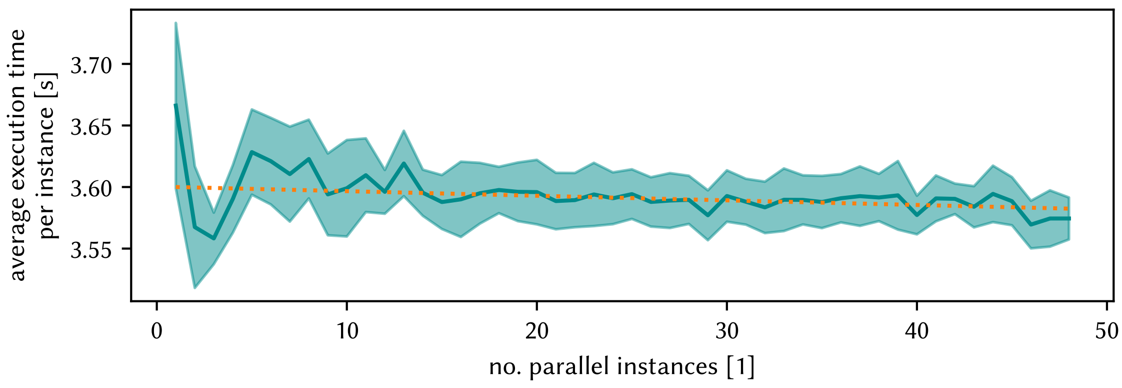


Figure 10.8: Evaluating hardware utilization via quiggeldy. We execute a varying number of parallel instances of the stadls hardware test suite and measure total execution time. Each instance registers as a different user and executes a total of 46 distinct hardware tests in sequence. We repeat each measurement 10 times and calculate mean (solid line) and standard deviation (shaded area). The whole benchmark is wrapped with wriggeldy (cf. Section 10.5.5), so that a single quiggeldy instance is used for the whole measurement. In total, 540 960 experiment steps are scheduled and executed. Since the largest average execution time is observed for a single executor and then declines sharply, we see that any potential overhead introduced by transmission delays (cf. Figure 10.7) is amortized. This is confirmed by fitting a straight line to the average execution times (dotted, orange), resulting in a slope of $-3.73 \cdot 10^{-4}$ s/no. instances, i.e., only very slightly decreasing. Please note that standalone execution time for stadls hardware tests is (9.55 ± 0.26) s (averaged over 10 runs), i.e., almost a factor of three longer than execution via quiggeldy at (3.59 ± 0.03) s.

users present. Each batch size is executed 10 times and the minimum taken to reduce the influence of side effects. We only measure the execution time of `hxtorch.matmul()`. The overhead is then calculated as percentage increase in complete execution time.

First, we evaluate the naïve implementation that uses the existing cereal implementation to fill an SF-buffer that is then sent over the wire (cf. Figure 10.7, Top Row). Here, we notice a small but measurable increase in execution time. For small batch sizes it amounts to about 20 % which then increases to more than 70 % as the batch size increases. Evaluating timing information from quiggeldy's logs, we identified the serialization routine to indeed be the major contributor to execution delay. The most likely explanation is the naïve implementation not pre-allocating one large buffer but rather one for each object. We tried to account for this by increasing the pre-allocated size of the SF-buffer, but to no avail. The relative overhead stays at (68.9 ± 2.1) %.

Using a successively more low-level approach, we first improved serialization by explicitly using faster serialization methods for vectors already present in SF (cf. Figure 10.7, Middle Row). This reduces relative overhead for large payloads by almost a factor of four to (18.7 ± 0.6) %. Going one step further we move to explicitly exchanging copies of the binary data structures (cf. Figure 10.7, Bottom Row). Since we transfer lists of FPGA words, they are implemented as an `std::vector` over a `std::variant` of `hxcomm::UTMessages` that are essentially PODs. In particular, there are no internal pointers or references to be taken care of. The relative overhead never exceeds 9 % and averages around (7.5 ± 0.6) % for larger payloads. Already now, it is hardly noticed during day-to-day operations. Future optimization will bring this number down even further. For example, the solution presented here contains at least two unnecessary copy-operations of data (one at client and one at server-side) which could be eliminated by using more specialized data structures.

Furthermore, as soon as hxtorch/grenade switch to asynchronous submissions, the delay can effectively be hidden: As all input and output operations are performed concurrently (cf. Figure 10.2) to the main thread executing on hardware, serialization delays will be amortized. We investigate this by executing varying numbers of (spiking) stadls hardware tests in parallel. All instances are using the same quiggeldy instance to access hardware. While hardware tests are typically shorter than "real" experiment steps, this makes them more susceptible to any introduced delays. We measure the runtime for all tests, as seen in Figure 10.8. As we can see, the execution time per instance stays more or less constant for larger instances. Also, since the execution time for a single instance is the highest, we conclude that we are able to amortize any potential overheads introduced by the current serialization implementation due to quiggeldy's concurrent design. Please note that standalone execution time for stadls hardware tests is (9.52 ± 0.22) s (averaged over 10 runs), i.e., almost a factor of three longer than execution via quiggeldy at (3.59 ± 0.03) s. This is due to the fact that in standalone execution mode every test has to establish its own connection to hardware that entails delays, whereas via quiggeldy only a single connection to hardware is opened.

Overall, we have demonstrated that, while the current serialization implementation in quiggeldy does add a slight overhead (that will decrease further with more optimization), we can effectively amortize it via asynchronous dispatching due to quiggeldy's concurrent design.

10.9 Transparent Cluster-Integration: hagen-daas

After introducing quiggeldy in the this chapter, the remaining challenge is its integration into the Electronic Vision(s) cluster setup presented in Chapter 9. The straightforward solution would be to have one quiggeldy daemon permanently running for every available hardware setup, as was the case during the hands-on tutorial at NICE 2021 (see Section 10.7.2). This solution, however, would permanently tie up computing resources that might not always be required. The compute resources allocated to quiggeldy daemons running on certain nodes as system services would have to be permanently removed from Slurm's configuration and could only be re-allocated with additional overhead. For example, if some hardware setups are not used for parameter sweeps overnight, the compute resources could not be used to perform nightly validation tests (cf. Chapter 7). As a form of mitigation, one might envision a plan to have quiggeldy daemon run as permanent cluster jobs, so that their compute resources could be at least freed manually. However, this would place an additional administrative burden on the cluster admins to ensure certain jobs are almost always running, for example after a node reboot.

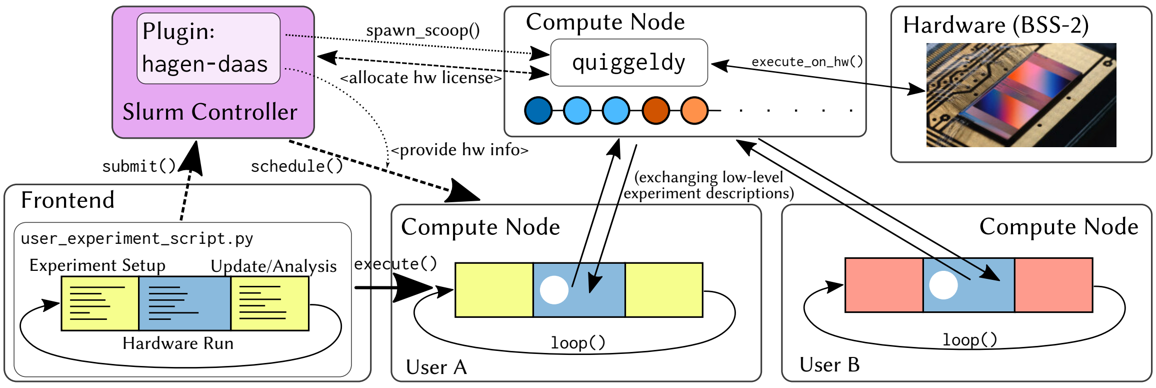


Figure 10.9: Overview of hagen-daas workflow: Users submit their experiment scripts for execution on Slurm-provided compute resources. Via a custom --daas CLI argument, they are able to select which hardware resource they want to execute on. hagen-daas then ensures that a scoop, i.e., a quiggeldy daemon instance, is running when the user job is scheduled to execute. If the scoop is delayed while starting up, hagen-daas actively reschedules user jobs until the scoop is ready. The scoop can be started on a variety of hosts from which it is able to establish connection to hardware. hagen-daas modifies the job's environment to provide this connect information (see Section 10.5.4) so that hxcomm will automatically use a QuiggeldyConnection (cf. Section 10.5.2). The user job then commences execution on whatever node was provided. During execution all experiment-steps are forwarded to quiggeldy in the lowest form of logical representation which schedules it in round-robin fashion with potential experiment-steps from other users as shown in Figure 10.2. This allows for both a tight feedback-loop for users who need immediate feedback while still allowing others to schedule long-running parameter sweeps without need for consulting another.

For this reason, a Slurm plugin[52] was developed: hagen-daas[53], written in C. It provides the integration between Slurm, scheduling jobs on relatively slow timescales with a runtime of seconds to minutes, and quiggeldy that schedules single experiment-steps with a runtime of hagen-daas ensures that submitted jobs automatically connect to a corresponding quiggeldy instance, launching it if necessary. Same as quiggeldy, it aims to be as transparent to the user as possible.

Previously, when not using hagen-daas, users request hardware resources by allocating a corresponding Slurm license. A user job will not be scheduled if another job already holds the license. Therefore, only one user can access any hardware resource at the same time. In practice, Slurm has support for multiple instances of the same license (to model a limited amount of software licenses that can be used concurrently), but without any form of arbitration including reinitialization (cf. Section 10.4) multiple users cannot connect to the same hardware setup. We therefore limit all licenses to have a total count of one. The Slurm-plugin nmpm_custom_resource (cf. Section 9.1.2), developed at Electronic Vision(s) primarily to handle access to BrainScaleS-1 hardware, allows users to request hardware resources via CLI arguments when scheduling their job. These arguments are then, among

[52]https://gerrit.bioai.eu/c/visions-slurm/+/3933 (visited on 2021-04-05)
[53]Howto Avoid Grabbing Emulators Nightlong – Dls As A Service

other[54] operations, translated to a set of acquired licenses.

The work principles of hagen-daas are visualized in Figure 10.9. It operates in much the same way as nmpm_custom_resource, the only difference for users is a different CLI argument: --daas <id>. The <id> can be a generic identifier for each hardware setup. At the time of writing, it corresponds to the previous license naming scheme W<wafer-id>F<fpga-id> in order to ease transition. In the long run, we plan to use the same syntax to request Slurm-resources and for querying the hwdb. The choice of identifier does not affect hagen-daas in any way.

In terms of hagen-daas nomenclature, any software daemon that governs hardware access is called a *scoop*. Whenever a user job is about to be scheduled, hagen-daas's primary goal is to check if a scoop for the given daas id is already running. If so, the job may be scheduled regularly. If not, the scoop is launched as yet another regular Slurm job while the user job is rescheduled. Once the scoop is set up, the user job is launched as well. Within the scoop, quiggeldy then proceeds to acquire the necessary licenses once the first user job connects. As quiggeldy is configured to terminate itself once being idle for too long, scoop jobs run in a special Slurm partition without runtime restrictions.

Furthermore, as stated in Section 10.2, quiggeldy will release its hardware allocation periodically. This will allow other jobs that need exclusive access to run. They can still be submitted using the regular API described above. Currently, we have no automatic checkpointing in the Electronic Vision(s) cluster-setup that would allow pausing and resuming compute jobs at any time. Therefore, allocating a setup exclusively while jobs are actively executing jobs via quiggeldy will cause them to stall, i.e., they will continue to consume compute resources while waiting for quiggeldy to continue executing. Therefore, it will be infeasible to block a setup for several hours while it is actively being used. The envisioned use-case is, for example, to perform nightly calibrations – taking on the order of single digit minutes – while parameter sweeps are being executed otherwise. Any long term exclusive access should be discussed with other users – as was the established practice prior to quiggeldy.

There are automated build scripts provided[55] that ease deploying an updated version of quiggeldy into the Electronic Vision(s) cluster environment (cf. Chapter 9).

In case of an unexpected restart of the central Slurm control daemon, slurmctld, hagen-daas is able to reconstruct its information about running scoop jobs. Therefore, there are no special precautions that need to be undertaken in case of restarting cluster services.

[54]Other features include tracking the initialization status of wafer reticles and setting up dynamic firewall rules so that the corresponding compute node is able to communicate with the FPGAs governing access to wafer reticles.

[55]https://github.com/electronicvisions/visions-slurm/blob/master/visionary-utils/quiggeldy/rebuild.sh (visited on 2021-04-08)

11 Remaining Challenges for Deployment

As with any software related deployment that is actively being used and developed, there are always new challenges to tackle. Some of those remaining challenges and ideas are listed here.

Integrating `waf` and Spack Currently, we use Spack to track and build all software dependencies, but use `waf` to build our software stacks. For historical reasons,[1] we track dependencies between different software layers in `waf`. It would be desirable to integrate our `waf`-based projects in to Spack. There already is rudimentary support for `waf`-based projects in Spack.[2] However, Spack is not yet aware of our dependency tracking mechanism.

This integration would increase options for users as it would allow our software stack to be easily deployable via Spack at remote sites. Users would not need to download our full container, but had the option to instruct Spack to build all needed software components for them.

While not a priority now, it will become one as soon as potential mobile versions of BrainScaleS-based hardware become available that would not be hosted at Electronic Vision(s) but at remote sites not attached to our cluster.[3]

Trusted Containers in Upstream-Singularity The concept of nested containers is essential when allowing users to transition seamlessly between compute environments. Our realization of this concept via trusted containers (cf. Section 8.2.4) is of potential interest to other HPC sites. In order to ensure a proper security audit, our working prototype will have to be evaluated and modified in some aspects: The default should be a whitelist of allowed bind-mount locations rather than a blacklist. Plus, potential security vulnerabilities such as TOC/TUO should be investigated more rigorously and, where needed, alleviated.

Pick & Choose Visionary Containers Each visionary container is one large packed image. This reduces overall storage requirements because, thanks to Spack's DAG-based concretization mechanism (cf. Section 8.1.1), dependencies present in several environments only need to be built and installed once. However, it would also be convenient to have each

[1]Our `waf`-based dependency tracking predates Spack by several years.

[2]`https://github.com/spack/spack/blob/a8ccb8e116a1abfaa1d7ecb4e84f9f14ebab12ad/lib/spack/spack/build_systems/waf.py` (visited on 2021-04-10)

[3]The BrainScaleS-2 Mobile system marks an important step in this direction, following USB-Spikey deployment.

compute environment available as distinct overlay images. When building test containers, only those images that are affected by changes get updated, cutting down on build time. The full container could still be recovered by combining all overlays.

Currently, Singularity lacks the concept of supporting more than one overlay. Hence, the functionality would have to be provided as a plugin. Furthermore, `yashchiki` (cf. Section 8.3) would need to be extended to handle several distinct but concurrent container builds.

This would be especially helpful towards users developing code on their own machines prior to submitting jobs on the cluster. They could download and use smaller and more focussed images to do initial verification work.

Improved Distribution of Visionary Containers Within the Electronic Vision(s) cluster all visionary container images are served from an SSD[4]-backed NFS mount point. This is perfectly acceptable within our cluster, as transmissions delays are low, bandwidth high and we have additional mitigation strategies such as NFS-caching. Upon first usage, a given image file is mounted as loop device and both Linux kernel and NFS ensure subsequent accesses are fast. Using deployment strategies such as `eStargz` [Tokunaga, 2020], visionary containers could be made available over the internet and in cloud operations. Because of the seek-able `.tar.gz` format, this would allow for transmitting only those parts of a compressed large container that are actually used at the remote side.

Spack Environments Ever since the solution described in this thesis was implemented, Spack is actively being developed and receives new features. One such feature is the concept of *environments*. Each environment is essentially a set of packages concretized together. Judging from feature description, environments could replace and enhance functionality of `visionary-` meta-packages (cf. Section 8.1.4).

Streamlining `yashchiki` to increase Turn-Around As described in Section 8.3, `yashchiki` is a more than sufficient solution to build containers. Reducing build times from more than a day to a few hours saves one order of magnitude of time already. However, as always, things could be improved further. At the time of writing, `yashchiki` is rather tightly integrated into the VM it is executed on: `conviz`. On `conviz` we provide two Jenkins-executors that are available to produce images. In some cases, this is not enough; especially when investigating the effects of several updated/added dependencies. By decoupling the currently rather tight integration we allow `yashchiki` to be more easily run on more build nodes. The current mitigation strategy is to combine several changes into one container build, but – given additional compute resources – it would be more desirable to build larger quantities of more fine grained images.

For this, it would be desirable to modify `yashchiki` to be a deployable binary that could be executed anywhere where `root`-permissions[5] are available. Currently, for historic reasons, all interactions with the host-system are defined in the Jenkins-job's `Jenkinsfile` and one central `bash`-header. However, in the future it would be much more desirable

[4]Solid State Disk

[5]These are necessary to build the container, cf. Section 8.3.5.

to have all dependencies and interactions with the surrounding system tracked via explicit configuration in one central location. Currently configuration involves, among other things, file systems, temporary spaces, and `cgroups` to contain runaway build processes. For this, it would probably be beneficial to switch from `bash` to `go`, the language Singularity is written in as well. The added type safety and directly exposed system calls would increase development speed by detecting bugs at compile time. Going one step further, `yashchiki` could even interface with Singularity directly (e.g., via a plugin) to have more fine grained control over the container build process. Since the current `bash` implementation is fast enough already in terms of execution speed, we expect no substantial performance increase. The `yashchiki` binary could then be deployed *anywhere*, including build nodes, personal computers or even the cloud and would not depend on certain system utility programs to be available. After adjusting configuration, building visionary containers would be straightforward, albeit time consuming given the host system.

Snapshotting `yashchiki` / Improving Spack's Performance In the current implementation of `yashchiki`, the largest time sinks in terms of container build time is Spack validating the extracted components from build caches as well as building the final views. While there are efforts to improve Spack's performance, due to its Python-based implementation we are not expecting to see huge jumps in the near future. Therefore, ideally, every container build stage (cf. Section 8.3.5) should be snapshotted and kept so that any failed build can be resumed at the latest stage that was executed correctly. Debugging some intricate build problems is rather involved and requires direct access to `conviz`, which is not a stable long-term solution. Snapshotting would allow developers to download and investigate snapshots of failed builds in an interactive fashion. In the same vein as the previous paragraph, extending `yashchiki` to a full runtime would make this easier to implement. Whether these snapshots are done at the filesystem level via `btrfs`[6] or a user level snapshotting solution akin to `restic`[7] is an implementation detail best discussed then.

Building for several Architectures In order to achieve compatibility with all compute nodes we execute on, we build all available software for the oldest CPU architecture used in our cluster. At the time of writing, this is Sandybridge.[8] We therefore ensure that no unsupported instructions are used in the generated binaries, allowing container images to be deployed cluster-wide. However, it would also be desirable to have builds for several architectures within a single image, each in its own Spack view. This would create separate views (remember that these are just structured symlinks) of each container `app` for all distinct architectures we target. By probing the host CPU for its supported instruction sets at container startup, we can dynamically load the correct view into the process environment. This would ensure optimal execution performance on newer compute nodes while still maintaining execution compatibility for older nodes. Of course, all packages that are built for several architectures would increase the container size.

[6]`https://btrfs.wiki.kernel.org/` (visited on 2021-04-10)
[7]`https://restic.net/` (visited on 2021-04-10)
[8]`http://ark.intel.com/products/codename/29900/Sandy-Bridge` (visited on 2021-04-10)

`quiggeldy`: The next Steps In the current iteration, `quiggeldy` is configured via a mixture of CLI arguments and environment variables (cf. Section 10.5.4). After leaving the prototyping stage, it would be a natural next step to have the option to provide a configuration file instead. Also, as identified in Section 10.8, serialization still adds a percentage overhead in the single digits. A more sophisticated solution, for example based on `sctrltp` that is actually zero-copy and already used to communicate with the FPGA, can decrease the overhead even further. Finally, `quiggeldy`'s scheduling algorithm could be extended. We already compute and log runtime statistics for executed experiment steps. By taking these into account, we could achieve an even more balanced time-allocation distribution among users.

Using `quiggeldy` for a completely integrated Colaboratory One of the goals of `EBRAINS` and HBP[9] in SGA3[10] is accessibility of all neuromorphic system via a centralized Colaboratory,[11] based on JupyterHub.[12] It allows interactive execution of Python-based Jupyter notebooks in the browser and facilitates exchange and collaboration of high level scripts by not too tech-savvy people and scientists. The NICE 2021 hands-on tutorial demonstrated a crucial first step in this direction. It also used JupyterHub and users were able to get immediate feedback from BrainScaleS-2-based hardware via `quiggeldy`.

Extending this setup to a more permanent solution, while still requiring a lot of effort,[13] would be relatively straightforward. Previous attempts relied on Colab-users allocating hardware setups exclusively via Slurm. Once `quiggeldy` is integrated, Colab users can seamlessly share setups amongst themselves and even regular cluster users – without actively noticing.

[9]Human Brain Project

[10]Human Brain Project Specific Grant Agreement 3

[11]`https://lab.ebrains.eu` (visited on 2021-04-11)

[12]`https://jupyter.org/hub` (visited on 2021-04-11)

[13]Python-based interactive web installations are notoriously difficult to properly track in terms of dependencies.

Science is made up of so many things that appear obvious after they are explained.

— Frank Herbert, Dune

III

Robust Learning Strategies in Neuromorphic Hardware

Neuromorphic Learning with Time-to-first-Spike Coding 12

This chapter presents work that has been done in close collaboration with Julian Göltz as part of his master thesis [Göltz, 2019] which the author had the pleasure to supervise. It has since been extended primarily in collaboration with Laura Kriener and reported in [Göltz et al., 2021] that is also presented here. For full details on the author's contributions, please refer to Appendix A.

In biology, fast decision making on the order of $\sim$100 ms is paramount and heavily influenced by precise spike-times [Thorpe et al., 1996; Thorpe et al., 2001]. Following our overview of deep learning with and without SNNs[1] in Chapter 2, this chapter introduces a novel method of applying deep learning to multi-layer SNNs. Instead of encoding state transmitted by units within the network in rates [Schmitt et al., 2017; Petrovici et al., 2017a], it operates on single spikes in a Time-To-First-Spike (TTFS) coding scheme [Thorpe et al., 2001]. In particular, it does not rely on some form of surrogate gradients [Zenke et al., 2018; Neftci et al., 2019] or other internal state such as synaptic input [Wunderlich et al., 2020]. The only input needed for the algorithm to update model parameters are spike-times. While certainly missing its immediate counterpart in biology, it is a prime candidate for application to analog neuromorphic substrates where tracking of analog internal state variables can typically only be performed for a subset of units (cf. Figure 12.1).

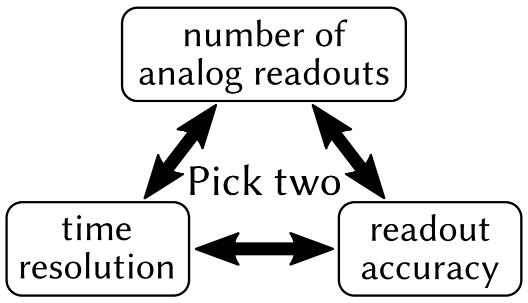


Figure 12.1: The problem with analog readout of state in analog neuromorphic hardware. While it is possible to integrate dedicated circuitry to record observables of particular interest, those will consume chip area and thus reduce the number of available units in total. Hence, algorithms that only rely on spike-times read back from hardware have an advantage in this regard.

Another aspect to consider is sparsity. [Mostafa, 2017] propose to encode feature salience in spike timing (cf. later Figure 12.3c,d). In terms of image-based input, one could consider each pixel represented by an input spike: The brighter the pixel, the earlier the spike. A real-valued input is effectively translated into a spike delay. Classification happens in much the same way: The first neuron to spike in the output layer represents its classification.

[1]Spiking Neural Networks

During training the network is encouraged to spike as early as possible. This lets important information propagate fast through the network, decreasing the time-to-solution significantly.

Only a few spikes (compared to rate-coding) are needed for input processing. Assuming that energy consumption of a neuromorphic device is somewhat correlated with the amount of spikes transmitted and processed,[2] TTFS allows for energy-efficient and fast inference.

Outline

Section 12.1 introduces the basis for TTFS coding and learning with IF[3] neurons. Section 12.2 then extends the idea towards LIF[4] neurons and derives exact update rules suitable for deep learning for two ratios of time constants. Section 12.3 details classification results for several datasets in software (Section 12.3.1) and on neuromorphic hardware (BrainScaleS-2,[5] cf. Section 12.3.2, and BrainScaleS-1,[6] cf. Section 12.3.3), concluding with an investigation of TTFS robustness to hardware-induced distortions (cf. Section 12.3.4).

12.1 Background: Learning with IF-neurons

[Mostafa, 2017] introduce learning with TTFS for IF neurons and CuBa[7]-synapses. Their membrane dynamics u are given by

$$C_\mathrm{m}\, \dot{u}(t) = \sum_i w_i \sum_{t_i} \theta(t - t_i) \exp\left(-\frac{t - t_i}{\tau_\mathrm{s}}\right) , \tag{12.1}$$

with membrane capacitance C_m, pre-synaptic weights w_i and incoming spike-times t_i, synaptic time constant τ_s and θ the Heaviside step function.[8] Here, the first sum runs over all input neurons, whereas the second sum runs over all input spikes of a given input neuron. Once the membrane potential crosses the fixed threshold ϑ, a spike is emitted and the neuron's potential reset to V_reset. Typically, the neuron enters a refractory period of τ_ref in which the membrane potential is held fixed. Since the approach presented in this chapter aims to use as few spikes as possible, most neurons spike once or less.

Since IF neurons act as capacitors without leakage, the injected currents add independently:

$$u(t) = \sum_i w_i \sum_{t_i} \theta(t - t_i) \left[1 - \exp\left(-\frac{t - t_i}{\tau_\mathrm{s}}\right)\right] \tag{12.2}$$

[2]Please note that this effect is *negligible* in case of BrainScaleS-1 or BrainScaleS-2. Here, we predominantly save energy by minimizing the time-to-solution at a generally low power-intake.

[3]Integrate-and-Fire

[4]Leaky-Integrate-and-Fire

[5]BrainScaleS-2 Analog Neuromorphic Hardware System, [Schemmel et al., 2017; Schemmel et al., 2020]

[6]BrainScaleS-1 Wafer-Scale Mixed-Signal Accelerated Neuromorphic System, [Schemmel et al., 2008; Schemmel et al., 2010]

[7]Current-Based

[8]$\theta(x) = 1$ for $x > 0$ and 0 otherwise.

We now aim to calculate the output spike-time T, i.e., the precise moment when

$$u(T) = \vartheta\,. \tag{12.3}$$

Defining the causal set $\mathcal{C}$, i.e., the set containing all pre-synaptic spikes a neuron receives prior to spiking,

$$\mathcal{C} = \{i \mid t_i < T\} \tag{12.4}$$

we can rewrite Equation (12.2) to be

$$\vartheta = \sum_{i\in\mathcal{C}} w_i \left[1 - \exp\left(-\frac{T - t_i}{\tau_\mathrm{s}}\right)\right]. \tag{12.5}$$

By solving for T we get

$$\frac{T}{\tau_\mathrm{s}} = \ln\left[\frac{\sum_{i\in\mathcal{C}} w_i \exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)}{\sum_{i\in\mathcal{C}} w_i - \vartheta}\right] \tag{12.6}$$

We therefore get a closed-form input-output relation for each neuron in a feed-forward network to which we can apply regular backpropagation. If neurons do not spike their spike-times are assumed to be ∞ and their gradients vanish.

For ease of computation, [Mostafa, 2017] use an exponential mapping for spike-times

$$T \longmapsto z_T := \exp\left(\frac{T}{\tau_\mathrm{s}}\right) \qquad t_i \longmapsto z_i := \exp\left(\frac{z_i}{\tau_\mathrm{s}}\right) \tag{12.7}$$

so that we arrive at the following partial derivatives

$$\begin{aligned}\frac{\partial z_T}{\partial w_i} &= \frac{z_i - z_T}{\sum_j w_j - \vartheta}\mathbb{1}_{i\in\mathcal{C}}\\ \frac{\partial z_T}{\partial z_i} &= \frac{w_i}{\sum_j w_j - \vartheta}\mathbb{1}_{i\in\mathcal{C}}.\end{aligned} \tag{12.8}$$

Here, $\mathbb{1}_{i\in\mathcal{C}}$ is equal to 1 iff $i \in \mathcal{C}$, i.e., $t_i < T$, and 0 otherwise.

We use the following loss function

$$\mathcal{L}_\mathrm{IF}[\mathbf{z}^{(N)}, n^*] = \ln\left[\sum_n \exp\left(-\left[z_n^{(N)} - z_{n^*}^{(N)}\right]\right)\right] + \alpha \sum_{\substack{\text{all}\\\text{neurons}}} \max\left(0, \vartheta - \sum_i w_i\right) \tag{12.9}$$

where $z_n^{(N)}$ denote the transformed output spikes of the output layer and n^* denotes the index of the correct label. The second term in Equation (12.9) ensures that all neurons in the network have a chance to spike by increasing all input weights to neurons that are quiet. α balances between the two terms.

Together with L2 weight normalization and gradient normalization, [Mostafa, 2017] then

Figure 12.2: 100 example patterns from MNIST test dataset [LeCun et al., 1998].

train the model in a feed-forward network to classify MNIST,[9] one of the staple benchmarks in machine learning. It consists of 60 000 + 10 000 (training + test) 28×28 grayscale images of handwritten data (cf. Figure 12.2). The model reaches accuracies of up to 97.55 % (cf. Table 12.1). One of the key results is that the network classifies its input, i.e., an output neuron spikes, after roughly 3.0 % (784–800–10) or 9.4 % (784–400–400–10) of all hidden neurons have spiked once. Hence, it is very spike efficient while classifying rapidly after roughly 1–3 τ_s from stimulus onset.

Network	Training Input	Training Accuracy	Test Accuracy
784–800–10	non-noisy	99.987 %	97.20 %
784–800–10	noisy	99.995 %	97.55 %
784–400–400–10	non-noisy	99.969 %	96.92 %
784–400–400–10	noisy	99.745 %	97.14 %

Table 12.1: Classification results reported by [Mostafa, 2017] when learning with IF-neurons using Equations (12.6), (12.8) and (12.9) for two different architectures and potentially noised input. Numbers indicate the number of neurons present in each layer. The Input layer is $28 \times 28 = 784$ images wide. Non-noisy and noisy input were trained separately. Noise was applied as spike-delay, drawn from the positive part of a zero-mean Gaussian distribution with τ_s variance. Adapted from: [Mostafa, 2017, Table 1].

12.2 Extending to LIF-neurons

In this section we extend [Mostafa, 2017] to LIF neurons [Brunel et al., 2007] – a widely-used dynamic model of spiking neurons with realistic integration behavior [Rauch et al., 2003; Gerstner et al., 2009; Teeter et al., 2018] – and derive closed-form solutions for the gradients of output spikes with respect to individual input spike-times and weights.

[9]MNIST Database, `http://yann.lecun.com/exdb/mnist/` (visited 2021-04-10), [LeCun et al., 1998]

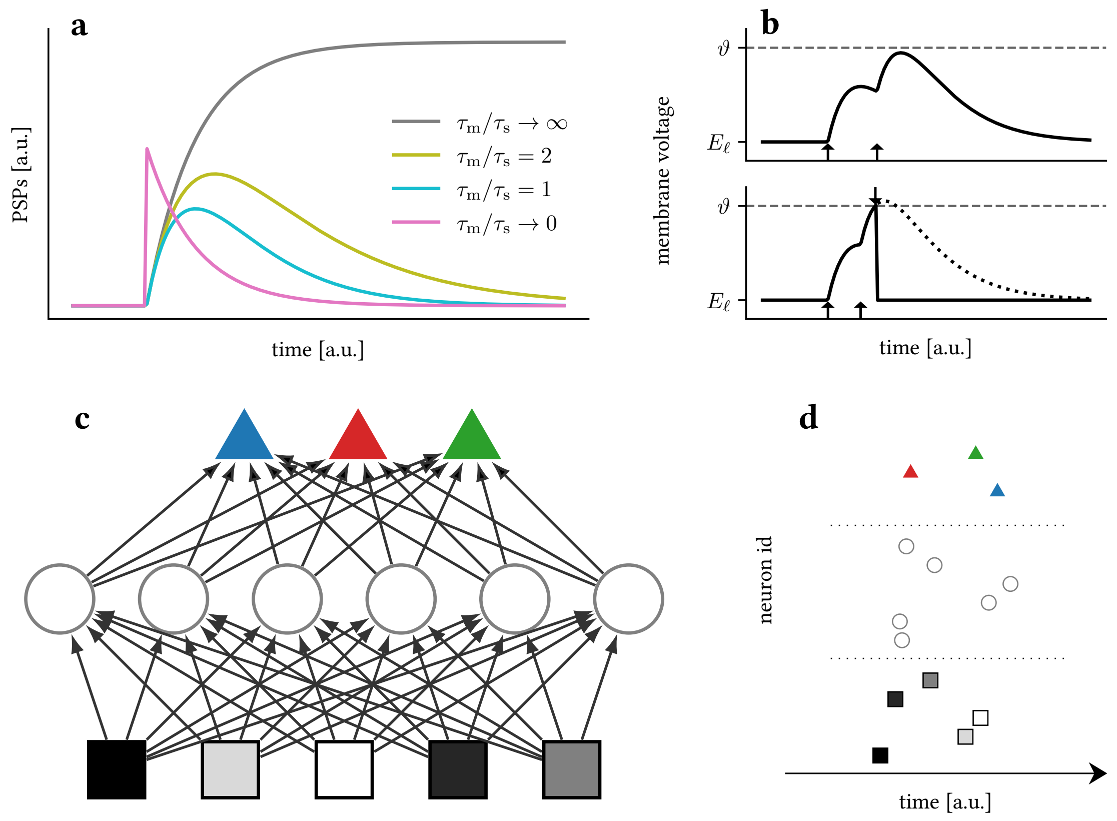


Figure 12.3: Overview over TTFS coding and learning.
Top Row: Single neurons.
(a) PSP shapes for different ratios of time constants τ_s and τ_m. Neurons gradually "forget" prior input due to finite time constants. Please note that the yellow line ($\tau_m/\tau_s = 2$) also describes the case $\tau_m/\tau_s = 1/2$ due to the symmetry in Equation (12.14).
(b) One key challenge of this finite memory arises when small variations of the synaptic weights result in disappearing/appearing output spikes, which elicits a discontinuity in the function describing output spike timing.
Bottom Row: Application to feed-forward hierarchical networks.
(c) Network structure. The geometric shape of the neurons represents their respective types (input □, hidden ○, label △). The shading of the input neurons is a representation of the corresponding data, such as pixel brightness (■, . . . , ■, . . . , □). The color of the label neurons represents their respective class (▲, ▲, ▲).
(d) TTFS coding exemplified in a raster plot. As an example of input encoding, the brightness of an input pixel is encoded in the lateness of a spike. Note that in this chapter, TTFS coding simultaneously refers to two individual aspects, namely the input-to-spike-time conversion and the determination of the inferred class by the identity of the first label neuron to fire (▲).
Adapted from: [Göltz et al., 2021, Figure 1].

12.2.1 | Closed Form Solution for LIF Neurons

In order to to extend the concept from Section 12.1 from IF to LIF, we need to extend Equation (12.1) by a leakage term with reversal potential E_ℓ and conductance g_ℓ:

$$C_\mathrm{m}\,\dot{u}(t) = g_\ell\,(E_\ell - u(t)) + \sum_i w_i \sum_{t_i} \theta(t-t_i)\exp\left(-\frac{t-t_i}{\tau_\mathrm{s}}\right) \tag{12.10}$$

Please note that C_m and g_ℓ implicitly define the membrane time constant τ_m:

$$\tau_\mathrm{m} = \frac{C_\mathrm{m}}{g_\ell} \tag{12.11}$$

Since E_ℓ is an arbitrary reference point, we set it to zero without loss of generality. We immediately notice that the homogeneous part in Equation (12.10) is given by an exponential with time-constant τ_m. Hence, inserting $u(t) = V(t)\,\exp(-\frac{t}{\tau_\mathrm{m}})$ into Equation (12.10) yields

$$\begin{aligned}
\tau_\mathrm{m}\left[\dot{V}\,e^{-\frac{t}{\tau_\mathrm{m}}} - \frac{1}{\tau_\mathrm{m}}V\,e^{-\frac{t}{\tau_\mathrm{m}}}\right] &= -V\,e^{-\frac{t}{\tau_\mathrm{m}}} + \frac{1}{g_\ell}\sum_i w_i \sum_{t_i}\theta(t-t_i)\,e^{-\frac{t-t_i}{\tau_\mathrm{s}}} \\
\iff \qquad \tau_\mathrm{m}\,\dot{V} &= \frac{1}{g_\ell}\sum_i w_i \sum_{t_i}\theta(t-t_i)\,e^{\frac{t_i}{\tau_\mathrm{s}}}\,e^{-\left(\frac{1}{\tau_\mathrm{s}}-\frac{1}{\tau_\mathrm{m}}\right)t} \\
\implies \qquad V(t) &= \frac{1}{g_\ell\,\tau_\mathrm{m}}\sum_i w_i \sum_{t_i}\theta(t-t_i)\,e^{\frac{t_i}{\tau_\mathrm{s}}}\int_{t_i}^{t}\mathrm{d}t'\,e^{-\left(\frac{1}{\tau_\mathrm{s}}-\frac{1}{\tau_\mathrm{m}}\right)t'} \\
&= \frac{1}{C_\mathrm{m}}\frac{\tau_\mathrm{m}\tau_\mathrm{s}}{\tau_\mathrm{m}-\tau_\mathrm{s}}\sum_i w_i \sum_{t_i}\theta(t-t_i)\,\left[e^{\frac{t_i}{\tau_\mathrm{m}}} - e^{-\left(\frac{t-t_i}{\tau_\mathrm{s}}-\frac{t}{\tau_\mathrm{m}}\right)}\right]
\end{aligned} \tag{12.12}$$

where we also set $u(0) = V(0) = 0$ as initial condition. For the membrane potential we therefore get

$$u(t) = \frac{1}{C_\mathrm{m}}\frac{\tau_\mathrm{m}\tau_\mathrm{s}}{\tau_\mathrm{m}-\tau_\mathrm{s}}\sum_i w_i \sum_{t_i}\theta(t-t_i)\,\left[\exp\left(-\frac{t-t_i}{\tau_\mathrm{m}}\right) - \exp\left(-\frac{t-t_i}{\tau_\mathrm{s}}\right)\right] \tag{12.13}$$

Same as before, we now attempt to find the output spike-time $u(T) = \vartheta$:

$$\vartheta = \frac{1}{C_\mathrm{m}}\frac{\tau_\mathrm{m}\tau_\mathrm{s}}{\tau_\mathrm{m}-\tau_\mathrm{s}}\sum_{i\in C} w_i\,\left[\exp\left(-\frac{T-t_i}{\tau_\mathrm{m}}\right) - \exp\left(-\frac{T-t_i}{\tau_\mathrm{s}}\right)\right] \tag{12.14}$$

We see that when taking the limit $\tau_\mathrm{m} \longrightarrow \infty$ we indeed recover Equation (12.5). Unfortunately Equation (12.14) has no general closed form solutions due to the fact that T appears in two exponentials with different time constants. However, we can still derive solutions for particular ratios of time constants.

12.2.2 Deriving a Learning Rule for Case #1: ${}^{\tau_\mathrm{m}}/_{\tau_\mathrm{s}} = 1$

If both time constants are equal (cf. Figure 12.3a), we can apply l'Hôpital's rule in the limit $\tau_\mathrm{m} \longrightarrow \tau_\mathrm{s}$ to find Equation (12.14) becoming a sum over α-functions:

$$\vartheta = \frac{1}{C_\mathrm{m}} \sum_{i \in \mathcal{C}} w_i \cdot (T - t_i) \, \exp\left(- \frac{T - t_i}{\tau_\mathrm{s}} \right) \tag{12.15}$$

This can be rewritten as

$$g_\ell \, \vartheta \, \exp\left(\frac{T}{\tau_\mathrm{s}} \right) = \underbrace{\sum_{i \in \mathcal{C}} w_i \exp\left(\frac{t_i}{1 \cdot \tau_\mathrm{s}} \right)}_{=: a_1} \frac{T}{\tau_\mathrm{s}} - \underbrace{\sum_{i \in \mathcal{C}} w_i \frac{t_i}{\tau_\mathrm{s}} \exp\left(\frac{t_i}{\tau_\mathrm{s}} \right)}_{=: b} =: -y \tag{12.16}$$

$$a_n := \sum_{i \in \mathcal{C}} w_i \exp\left(\frac{t_i}{n \cdot \tau_\mathrm{s}} \right) \tag{12.17}$$

Were we introduced convenience definitions a_n, b and y. Our goal is now to bring Equation (12.16) into the form

$$h e^h = z \tag{12.18}$$

which is solved by the differentiable Lambert W function $h = \mathcal{W}(z)$. Inserting $\frac{T}{\tau_\mathrm{s}} = \frac{b}{a_1} - \frac{y}{a_1}$ into Equation (12.16) gives:

$$\begin{aligned} g_\ell \vartheta \exp\left(\frac{b}{a_1} \right) \exp\left(- \frac{y}{a_1} \right) &= -y \\ \Longleftrightarrow \quad \underbrace{\frac{y}{a_1}}_{=: h} \exp\left(\frac{y}{a_1} \right) &= \underbrace{- \frac{g_\ell \vartheta}{a_1} \exp\left(\frac{b}{a_1} \right)}_{=: z} \end{aligned} \tag{12.19}$$

Using Equation (12.18) we can rewrite this to

$$\frac{T}{\tau_\mathrm{s}} = \frac{b}{a_1} - \mathcal{W}\left[- \frac{g_\ell \vartheta}{a_1} \exp\left(\frac{b}{a_1} \right) \right] \tag{12.20}$$

which is the input-output relation we were looking for.

Choice of branch As shown in Figure 12.4c, the defining relation for the Lambert W function is not bijective. Hence, we need to identify which real branch to use,[10] defined by $h \lesseqgtr -1$ (cf. Figure 12.4d). These correspond to two threshold crossings in case when spiking: one early, one late (cf. Figure 12.4a). We need to select the early crossing. For this, we first limit our discussion to scenario of a single input spike able to elicit an output

[10] Additionally, the Lambert W function has infinitely many imaginary branches, but they are of no concern for us here.

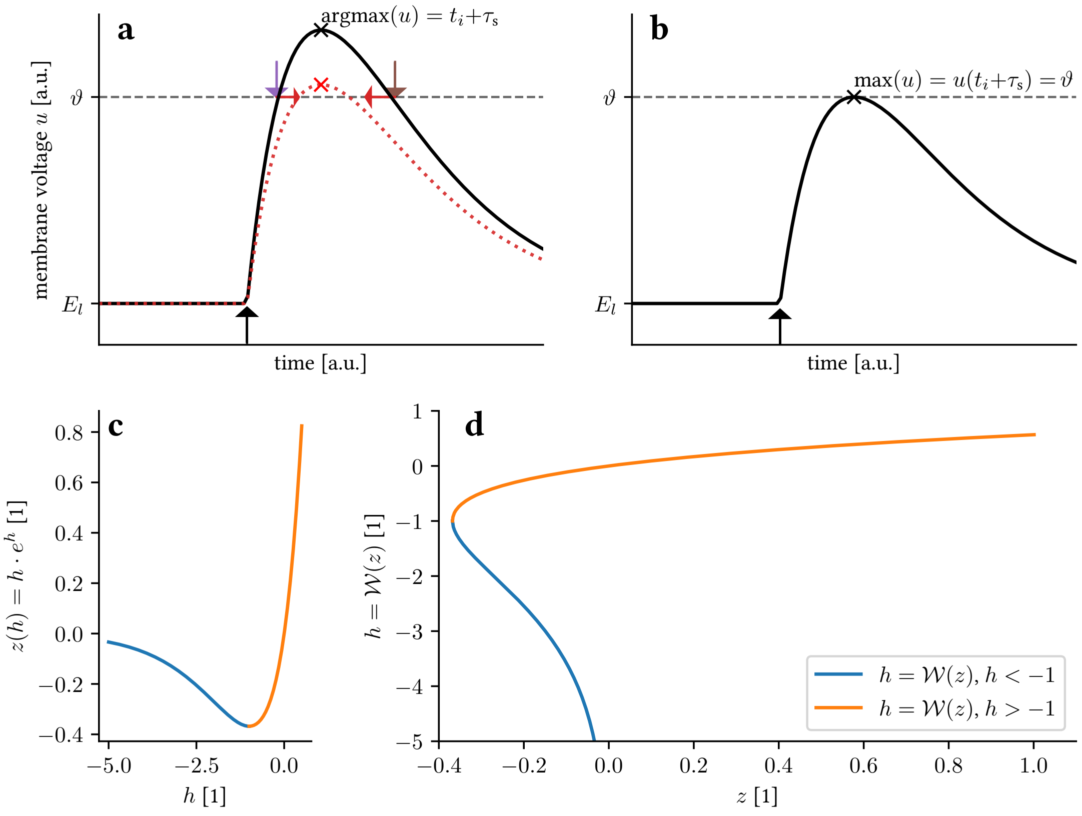


Figure 12.4: **(a)** Membrane dynamics for one strong input spike at t_i (upward arrow) with two threshold crossings due to leak pullback (earlier violet, later brown). The change induced by a reduction of in input weight is shown in red. **(b)** Edge case without crossing and exactly one time where $V(t) = \vartheta$. **(c)** Defining relation for the Lambert W function $\mathcal{W}$ for real values, evidently not an injective map. **(d)** Distinguishing between $h \lessgtr -1$ allows to define the inverse function of *(c)*, the Lambert W function $\mathcal{W}$.
Adapted from: [Göltz et al., 2021, Figure A].

spike. In this case the definition for h reduces to:

$$h = \frac{y}{a_1} = \frac{b}{a_1} - \frac{T}{\tau_s} \xrightarrow{\text{single spike}} h = \frac{t_i - T}{\tau_s} \tag{12.21}$$

Since $\tau_m = \tau_s$, we know the maximum PSP[11] occurs at $t_i + \tau_s$, hence the output spike can occur *at most* then (cf. Figure 12.4b), resulting in:

$$\begin{aligned} & T \leq t_i + \tau_s \\ \iff \quad & -1 \leq \frac{t_i - T}{\tau_s} = h \end{aligned} \tag{12.22}$$

We therefore need to select the branch with $h \geq -1$. In the general case of several input spikes, we can argument in a similar way. Since we have an output spike, the

[11]Post-Synaptic-Potential

net sum in a_1 and b (cf. Equation (12.16)) must be positive. Therefore their ratio in Equation (12.21) is positive as well. Furthermore, we observe that, because of T's negative sign in Equation (12.21), the larger the output spike-time the smaller h. Hence, in order to select the earlier, i.e., smaller, output spike-time we need to select the branch with larger h which is $h > -1$ (cf. Figure 12.4d).

Derivatives Now we are equipped to calculate the gradients required for learning. We start by calculating the total derivative for the output spike from Equation (12.20):

$$\begin{aligned}\frac{\mathrm{d}T}{\tau_\mathrm{s}} = &\left[-\frac{b}{a_1^2} - \mathcal{W}'(z)\,\frac{\partial z}{\partial a_1}\right] \underbrace{\left(\frac{\partial a_1}{\partial t_i}\,\mathrm{d}t_i + \frac{\partial a_1}{\partial w_i}\,\mathrm{d}w_i\right)}_{=\mathrm{d}a_1} \\ &+ \left[\frac{1}{a_1} - \mathcal{W}'(z)\,\frac{\partial z}{\partial b}\right] \underbrace{\left(\frac{\partial b}{\partial t_i}\,\mathrm{d}t_i + \frac{\partial b}{\partial w_i}\,\mathrm{d}w_i\right)}_{=\mathrm{d}b}\end{aligned} \tag{12.23}$$

We calculate derivatives of all convenience definitions:

$$\frac{\partial z}{\partial a_1} = -\frac{z}{a_1}\left(1 + \frac{b}{a_1}\right) \qquad \frac{\partial z}{\partial b} = \frac{z}{a_1} \tag{12.24}$$

$$\frac{\partial a_n}{\partial t_i}(\mathbf{w},\mathbf{t}) = \mathbb{1}_{i\in\mathcal{C}}\,\frac{w_i}{n\cdot\tau_\mathrm{s}}\exp\left(\frac{t_i}{n\cdot\tau_\mathrm{s}}\right) \qquad \frac{\partial b}{\partial t_i}(\mathbf{w},\mathbf{t}) = \mathbb{1}_{i\in\mathcal{C}}\,\frac{w_i}{\tau_\mathrm{s}}\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\left[1 + \frac{t_i}{\tau_\mathrm{s}}\right] \tag{12.25}$$

$$\frac{\partial a_n}{\partial w_i}(\mathbf{w},\mathbf{t}) = \mathbb{1}_{i\in\mathcal{C}}\,\exp\left(\frac{t_i}{n\cdot\tau_\mathrm{s}}\right) \qquad \frac{\partial b}{\partial w_i}(\mathbf{w},\mathbf{t}) = \mathbb{1}_{i\in\mathcal{C}}\,\frac{t_i}{\tau_\mathrm{s}}\,\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right) \tag{12.26}$$

Expanding $\mathrm{d}a_1$ and $\mathrm{d}b$, while inserting Equations (12.25) and (12.26) into Equation (12.23), gives for all spike-times that are part of the causal set:

$$\begin{aligned}\frac{\mathrm{d}T}{\tau_\mathrm{s}} = \frac{1}{a_1}\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\Bigg\{&\left[-\frac{b}{a_1} + \mathcal{W}'(z)\,z\left(1 + \frac{b}{a_1}\right)\right]\left(\frac{w_i}{\tau_\mathrm{s}}\,\mathrm{d}t_i + \mathrm{d}w_i\right) \\ &+ \left[1 - \mathcal{W}'(z)\,z\right]\left(\frac{w_i}{\tau_\mathrm{s}}\left[1 + \frac{t_i}{\tau_\mathrm{s}}\right]\mathrm{d}t_i + \frac{t_i}{\tau_\mathrm{s}}\,\mathrm{d}w_i\right)\Bigg\}\end{aligned} \tag{12.27}$$

Multiplying out and summing up, we arrive at:

$$\frac{1}{\tau_\mathrm{s}}\frac{\partial T}{\partial t_i}(\mathbf{w},\mathbf{t}) = \frac{1}{a_1}\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\frac{w_i}{\tau_\mathrm{s}}\left[1 + \left(\frac{t_i}{\tau_\mathrm{s}} - \frac{b}{a_1}\right)(1 - z\mathcal{W}'(z))\right] \tag{12.28}$$

$$\frac{1}{\tau_\mathrm{s}}\frac{\partial T}{\partial w_i}(\mathbf{w},\mathbf{t}) = \frac{1}{a_1}\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\left[z\mathcal{W}'(z) + \left(\frac{t_i}{\tau_\mathrm{s}} - \frac{b}{a_1}\right)(1 - z\mathcal{W}'(z))\right] \tag{12.29}$$

In order to be more robust to hardware variations (cf. later Figure 12.13d), a crucial step is now to recognize and replace all occurrences of the original output spike-time on the right hand sides in Equations (12.28) and (12.29). This allows gradients to be more adaptive if spike-times read back from a neuromorphic backend are subject to distortions (cf. Figure 12.13). For this, we use Equation (12.18) to calculate the derivative of the Lambert W function:

$$\begin{aligned}
&& \mathcal{W}(z)e^{\mathcal{W}(z)} &= z && \Bigg|\frac{\partial}{\partial z} \\
\Longleftrightarrow && \mathcal{W}'(z)\, e^{\mathcal{W}(z)}\,[1+\mathcal{W}(z)\,] &= 1 \\
\Longleftrightarrow && z\,\mathcal{W}'(z) &= \frac{\mathcal{W}(z)}{1+\mathcal{W}(z)} && (12.30)
\end{aligned}$$

Inserting Equation (12.30) into Equations (12.28) and (12.29) yields the final derivatives of Equation (12.20), again for input spikes from the causal set.

$$\frac{\partial T}{\partial t_i}(\mathbf{w},\mathbf{t},T) = -\frac{w_i}{a_1}\frac{1}{\mathcal{W}(z)+1}\,\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\,(T-t_i-\tau_\mathrm{s}) \tag{12.31}$$

$$\frac{\partial T}{\partial w_i}(\mathbf{w},\mathbf{t},T) = -\frac{\tau_\mathrm{s}}{a_1}\frac{1}{\mathcal{W}(z)+1}\,\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\,(T-t_i) \tag{12.32}$$

These can be integrated into any backpropagation learning scheme (cf. Equation (2.14)) as gradient for LIF neurons with equal membrane and synaptic time constants.

12.2.3 | Deriving a Learning Rule for Case #2: $\tau_\mathrm{m}/\tau_\mathrm{s} = 2$

Since Equation (12.14) is symmetric in both τ_m and τ_s, the following derivation holds for both $\tau_\mathrm{m}/\tau_\mathrm{s} = 2$ and $\tau_\mathrm{m}/\tau_\mathrm{s} = 1/2$ (cf. Figure 12.3a). We start out by inserting $\tau_\mathrm{m} = 2\tau_\mathrm{s}$ in Equation (12.14) and expanding:

$$g_\ell\vartheta = \exp\left(-\frac{T}{2\tau_\mathrm{s}}\right)\underbrace{\sum_{i\in\mathcal{C}} w_i\,\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right)}_{=a_2} - \exp\left(-\frac{T}{\tau_\mathrm{s}}\right)\underbrace{\sum_{i\in\mathcal{C}} w_i\,\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)}_{=a_1} \tag{12.33}$$

Reordering leads to

$$0 = -a_1\left[\exp\left(-\frac{T}{2\tau_\mathrm{s}}\right)\right]^2 + a_2\,\exp\left(-\frac{T}{2\tau_\mathrm{s}}\right) - g_\ell\vartheta \quad . \tag{12.34}$$

From the way Equation (12.34) is written, we can immediately deduce that it is a quadratic polynomial in $\exp(-T/2\tau_\mathrm{s})$. Thus, solutions are:

$$\begin{aligned}
&& \exp\left(-\frac{T}{2\tau_\mathrm{s}}\right) &= \frac{a_2 \pm \sqrt{a_2^2-4a_1g_\ell\vartheta}}{2a_1} && (12.35) \\
\Longrightarrow && \frac{T}{\tau_\mathrm{s}} &= 2\,\ln\left(\frac{2a_1}{a_2+\sqrt{a_2^2-4a_1g_\ell\vartheta}}\right) && (12.36)
\end{aligned}$$

Choice of branch Same as above, we are presented with two solution branches: One where the membrane voltage approaches the threshold from below and then again from above when decaying back to the leak potential (cf. Figure 12.4a,b). Obviously, we need to identify the earlier branch. Inspecting Equation (12.36), we see that, since the logarithm is monotonic, we need to select the larger denominator to select the earlier (i.e., smaller) T. Hence, we need to select the $+$-branch in Equation (12.35).

Derivatives After identifying the closed-form solution and its correct branch, we need to calculate its derivatives in order to apply backpropagation. We define for brevity

$$x := \sqrt{a_2^2 - 4a_1 g_\ell \vartheta} \tag{12.37}$$

Its partial derivatives are

$$\frac{\partial x}{\partial a_1} = -\frac{2g_\ell\vartheta}{x} \qquad \frac{\partial x}{\partial a_2} = \frac{a_2}{x} \tag{12.38}$$

Together with Equations (12.25) and (12.26), the total derivative for $i \in \mathcal{C}$ becomes

$$\begin{aligned}\frac{\mathrm{d}T}{\tau_\mathrm{s}} &= \frac{a_2+x}{a_1}\left\{\left[\frac{2}{a_2+x} - \frac{2a_1}{(a_2+x)^2}\frac{\partial x}{\partial a_1}\right]\mathrm{d}a_1 - \left[\frac{2a_1}{(a_2+x)^2}\left(1+\frac{\partial x}{\partial a_2}\right)\right]\mathrm{d}a_2\right\} \\ &= 2\left\{\left[\frac{1}{a_1} + \frac{2g_\ell\vartheta}{(a_2+x)x}\right]\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\left(\frac{w_i}{\tau_\mathrm{s}}\mathrm{d}t_i + \mathrm{d}w_i\right)\right. \\ &\qquad \left. - \underbrace{\left[\frac{1}{(a_2+x)}\left(1+\frac{a_2}{x}\right)\right]}_{=\frac{1}{x}}\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right)\left(\frac{w_i}{2\tau_\mathrm{s}}\mathrm{d}t_i + \mathrm{d}w_i\right)\right\}\end{aligned} \tag{12.39}$$

Multiplying out, we arrive at the partial derivatives:

$$\frac{1}{\tau_\mathrm{s}}\frac{\partial T}{\partial t_i}(\mathbf{w},\mathbf{t}) = \frac{w_i}{\tau_\mathrm{s}}\left\{2\left[\frac{1}{a_1} + \frac{2g_\ell\vartheta}{(a_2+x)x}\right]\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right) - \frac{1}{x}\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right)\right\} \tag{12.40}$$

$$\frac{1}{\tau_\mathrm{s}}\frac{\partial T}{\partial w_i}(\mathbf{w},\mathbf{t}) = 2\left\{\left[\frac{1}{a_1} + \frac{2g_\ell\vartheta}{(a_2+x)x}\right]\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right) - \frac{1}{x}\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right)\right\} \tag{12.41}$$

Same as before, we can increase robustness (cf. later Figure 12.13d) by reinserting the original definition for T, i.e., Equation (12.36):

$$\frac{\partial T}{\partial t_i}(\mathbf{w},\mathbf{t},T) = w_i\left\{\frac{2}{a_1}\left[1 + \frac{g_\ell\vartheta}{x}\exp\left(\frac{T}{2\tau_\mathrm{s}}\right)\right]\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right) - \frac{a_1}{x}\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right)\right\} \tag{12.42}$$

$$\frac{\partial T}{\partial w_i}(\mathbf{w},\mathbf{t},T) = 2\tau_\mathrm{s}\left\{\frac{1}{a_1}\left[1 + \frac{g_\ell\vartheta}{x}\exp\left(\frac{T}{2\tau_\mathrm{s}}\right)\right]\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right) - \frac{a_1}{x}\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right)\right\} \tag{12.43}$$

These are the final derivatives required for learning.

12.2.4 TTFS-based Learning in deep Networks

After deriving exact expressions for the gradient for two distinct ratios of time constants, we can now apply it to a deep feed-forward network with N layers (cf. Figure 12.3c). By N layers we explicitly do *not* count the input layer. We denote layer affiliation by a bracketed superscript, e.g., $t_j^{(l)}$ denotes the spike-time of the jth neuron in the lth layer whereas $w_{ki}^{(l)}$ denotes the weight projecting from the ith neuron in layer $(l-1)$ to the kth neuron in layer l. In other words, the projection direction of a weight is read from right to left. Here, the input layer corresponds to the 0th layer.

Loss function Applying the backpropagation algorithm [Linnainmaa, 1970; Rumelhart et al., 1986], i.e., Equation (2.14), requires the loss function to be differentiable with respect to both synaptic weights as well as spike-times. We define our learning goal similar to [Mostafa, 2017] in that the first output spike in the last (label-) layer determines the network's classification decision. Our aim is to maximize the temporal difference between spike-times from the correct label neuron and all others. We therefore define the following loss function

$$\begin{aligned} \mathcal{L}[\mathbf{t}^{(N)}, n^*] &= \text{dist}\left(t_{n^*}^{(N)}, t_{n\neq n^*}^{(N)}\right) \\ &= \ln\left[\sum_n \exp\left(-\frac{t_n^{(N)} - t_{n^*}^{(N)}}{\xi\tau_\text{s}}\right)\right] \end{aligned} \tag{12.44}$$

where $\mathbf{t}^{(N)}$ denotes the vector of label spike-times $t_n^{(N)}$, n^* the index of the correct label and $\xi \in \mathbb{R}^+$ is a scaling parameter. Please note that setting the scaling factor ξ to 1 does *not* recover the original loss function Equation (12.9). Because Equation (12.44) does not involve two stages of exponentiation it is numerically more stable.[12] Furthermore, Equation (12.9) is not time-invariant as shifting all spikes by a fixed amount δt *does* affect its absolute value:

$$\begin{aligned} \mathcal{L}_\text{IF}[\left(e^{\mathbf{t}+\delta t}\right)^{(N)}, n^*] &= \ln\left[\sum_n \exp\left(-\left[z_n^{(N)}\, e^{\delta t} - z_{n^*}^{(N)}\, e^{\delta t}\right]\right)\right] \\ &= \ln\left[\sum_n \exp\left(-\left[z_n^{(N)} - z_{n^*}^{(N)}\right]\, e^{\delta t}\right)\right] \\ &\leq \ln\left[\sum_n \exp\left(-\Delta t_{n^*}^\text{worst}\, e^{\delta t}\right)\right] \\ &= \ln\left[n_\text{label}\, \exp\left(-\Delta t_{n^*}^\text{worst}\, e^{\delta t}\right)\right] \\ &= \ln\left(n_\text{label}\right) - e^{\delta t}\, \Delta t_{n^*}^\text{worst} \end{aligned} \tag{12.45}$$

[12] Two stages of exponentiation are rather unstable, e.g., $\exp(\exp(6)) \approx 1.61 \cdot 10^{175}$.

Here, $\Delta t_{n^*}^{\mathrm{worst}}$ denotes the worst differentiated label as an upper bound for the objective function:

$$\Delta t_{n^*}^{\mathrm{worst}} := \min_n \left[z_n^{(N)} - z_{n^*}^{(N)} \right] \tag{12.46}$$

Despite a label being wrongly classified ($\Delta t_{n^*}^{\mathrm{worst}} < 0$), we see that by shifting to earlier spike-times $\delta t < 0$, we can *still* decrease the upper bound of the objective function. On the other hand, once a label is classified correctly ($\Delta t_{n^*}^{\mathrm{worst}} > 0$), the upper bound can be decreased by simply delaying all spike-times. This influences learning in deeper networks: The energy landscape for deeper layers (with larger absolute spike-times) is different from shallow networks with less layers. It is immediately obvious that shifting spike-times in Equation (12.44) by a similar spike delay δt cancels out and does *not* affect its value.

Hence, we consider defining the learning task on difference in spike-times to be the more "natural" and robust choice. [Mostafa, 2017] use an extra reference neuron connected via trainable weights to all neurons in the network acting as both bias and time reference when training MNIST. As shown later in Section 12.3, with our approach this is only necessary if the dataset explicitly requires it.[13]

By taking the gradient of Equation (12.44), we obtain synaptic plasticity rules. For the label layer we have:

$$\Delta w_{ni}^{(N)} \propto -\frac{\partial L[\mathbf{t}^{(N)}, n^*]}{\partial w_{ni}^{(N)}} = -\frac{\partial L[\mathbf{t}^{(N)}, n^*]}{\partial t_n^{(N)}} \frac{\partial t_n^{(N)}}{\partial w_{ni}^{(N)}} \tag{12.47}$$

For all deeper layers ($l \neq N$), we can apply the chain rule

$$\Delta w_{ki}^{(l)} \propto -\frac{\partial L[\mathbf{t}^{(N)}, n^*]}{\partial w_{ki}^{(l)}} = -\delta_k^{(l)} \frac{\partial t_k^{(l)}}{\partial w_{ki}^{(l)}} \tag{12.48}$$

where we have defined a propagated error term $\delta_k^{(l)}$ that is recursively calculated over all neurons in the next layer.

$$\delta_k^{(l)} := \frac{\partial L[\mathbf{t}^{(N)}, n^*]}{\partial t_k^{(l)}} = \sum_j \delta_j^{(l+1)} \frac{\partial t_j^{(l+1)}}{\partial t_k^{(l)}} \tag{12.49}$$

In the following, we derive the full multi-layer update rules for $\tau_{\mathrm{m}} = \tau_{\mathrm{s}}$, which is the case we implemented in predominantly hardware. However, derivation for ${}^{\tau_{\mathrm{m}}}\!/_{\tau_{\mathrm{s}}} = 2$ can be performed analogously. Rewriting Equations (12.31) and (12.32) in a layer-wise setting,

[13]In particular, we solve MNIST without bias spikes whereas [Mostafa, 2017] use them. For a different dataset, Yin-Yang, that will be introduced later, we do need a bias spike because input dimensionality is low. It does *not* serve as time reference.

the derivatives of the spike-time for a neuron k in arbitrary layer l are

$$\frac{\partial t_k^{(l)}}{\partial t_i^{(l-1)}}(\mathbf{w}^{(l)}, \mathbf{t}^{(l-1)}, \mathbf{t}^{(l)}) = -\frac{w_{ki}^{(l)}}{a_1} \exp\left(\frac{t_i^{(l-1)}}{\tau_\mathrm{s}}\right) \frac{1}{\mathcal{W}(z)+1} \frac{t_k^{(l)} - t_i^{(l-1)} - \tau_\mathrm{s}}{\tau_\mathrm{s}} \tag{12.50}$$

$$\frac{\partial t_k^{(l)}}{\partial w_{ki}^{(l)}}(\mathbf{w}^{(l)}, \mathbf{t}^{(l-1)}, \mathbf{t}^{(l)}) = -\frac{\tau_\mathrm{s}}{a_1} \exp\left(\frac{t_i^{(l-1)}}{\tau_\mathrm{s}}\right) \frac{1}{\mathcal{W}(z)+1} \frac{t_k^{(l)} - t_i^{(l-1)}}{\tau_\mathrm{s}} \tag{12.51}$$

Please note that a_1 and z's dependency on $\mathbf{t}^{(l-1)}$ and $\mathbf{w}^{(l)}$ was omitted for brevity. Inserting Equations (12.49) to (12.51) into Equations (12.47) and (12.48) yields a synaptic learning rule which implements exact error backpropagation on spike-times. In order to resemble the standard error backpropagation algorithm for ANNs,[14] we can rewrite to

$$\boldsymbol{\delta}^{(N)} = \frac{\partial L}{\partial \boldsymbol{t}^{(N)}} \tag{12.52}$$

$$\boldsymbol{\delta}^{(l-1)} = \left(\widehat{\boldsymbol{B}}^{(l)} - 1\right) \odot \boldsymbol{\rho}^{(l-1)} \odot \left(\boldsymbol{w}^{(l),T} \boldsymbol{\delta}^{(l)}\right) \tag{12.53}$$

$$\Delta \boldsymbol{w}^{(l)} = -\eta\, \tau_\mathrm{s} \left(\boldsymbol{\delta}^{(l)} \boldsymbol{\rho}^{(l-1),T}\right) \odot \widehat{\boldsymbol{B}}^{(l)} \tag{12.54}$$

where $\odot$ is the element-wise product, the T-superscript denotes the transpose of a matrix and $\boldsymbol{\delta}^{(l-1)}$ is a vector containing the backpropagated errors of layer $(l-1)$. η is the learning rate. The individual elements of the tensors above are given by:

$$\rho_i^{(l)} = -\frac{1}{a_1} \exp\left(\frac{t_i^{(l)}}{\tau_\mathrm{s}}\right) \frac{1}{\mathcal{W}(z)+1} \frac{1}{\tau_\mathrm{s}} \tag{12.55}$$

$$\widehat{B}_{ki}^{(l)} = \frac{t_k^{(l)} - t_i^{(l-1)}}{\tau_\mathrm{s}} \tag{12.56}$$

Input Conversion As stated above, real-valued input data x has to be converted into a TTFS coding. We therefore define an interval of earliest and latest possible input spike-times and map linearly:

$$\left[\min_{x\in\text{input}}(x),\ \max_{x\in\text{input}}(x)\right] \longmapsto \left[t_\text{early},\ t_\text{late}\right] \tag{12.57}$$

Bias spikes, if needed, are implemented as an additional input connected to all neurons. They occur at the same time t_bias for each sample of image data. Their weights are subject to the same kind of plasticity Equation (12.54).

Regularization Learning is augmented by several regularization techniques. First, an additional bias term is added to the loss function Equation (12.44) that prefers the correct

[14] Artificial Neural Networks

label neuron to spike as early as possible.

$$\mathcal{L}_{\text{early}}[\mathbf{t}^{(N)}, n^*] = \mathcal{L}[\mathbf{t}^{(N)}, n^*] + \alpha \left[\exp\left(\frac{t_{n^*}^{(N)}}{\beta\,\tau_{\text{s}}}\right) - 1\right] \tag{12.58}$$

where $\alpha, \beta \in \mathbb{R}^+$ are scaling hyperparameters.

Overfitting is reduced by applying Gaussian noise to input spike-times. This is especially important when training on images such as the MNIST dataset.

In order to compensate for vanishing denominators in Equation (12.54), we only allow weight updates Δw within a given range. Any weight update exceeding this range is set to zero.

Finally, we control for the portion of non-spiking neurons in each layer. As soon as it hits a pre-determined limit, we boost (i.e., increase) the input weights to all silent neurons. In case several layers exceed the threshold, only the first of such will be boosted in order not to overwhelm the network. Boosting ramps up exponentially if a layer exceeds the threshold for multiple epochs in a row.

Implementation All TTFS related experiments are implemented as custom modules for `PyTorch`.[15] Mostly written in Python,[16] it employs a custom C++[17] implementation to improve the speed with which Lambert W can be calculated on GPUs.[18]

The `PyTorch` module implements layers of LIF neurons with custom forward and backward-pass implementations. In the forward pass – needed in software-only experiments – spike-times are calculated via Equations (12.20) and (12.36). Here, we automatically determine causal set by calculating all possible sets in parallel and then choosing the earliest output time. While memory-intensive, this method proved faster than integrating the original differential equations Equation (12.10), which is mathematically equivalent. In hardware-in-the-loop settings similar to [Schmitt et al., 2017], spike-times are obviously returned by the neuromorphic emulator and need not be determined in software.

The backward pass then implements Equation (12.54). As optimizer we use Adam[19] (cf. Section 2.1).

12.2.5 | Learning with CoBa Synapses

An early version of the TTFS algorithm presented here was implemented on BrainScaleS-1 (cf. Section 12.3.3). One crucial difference to BrainScaleS-2 is the CoBa[20] synapse model. Compared to CuBa synapses, where each spike applies a fixed amount of charge onto the membrane, CoBa synapses are more biologically realistic in that each spike "type"

[15]Python-Implementation of Lua-library `torch`, [Paszke et al., 2019]
[16]Python Programming Language, [Rossum, 2000]
[17]C++ Programming Language, [ISO, 2017]
[18]Graphics Processing Units
[19]ADAptive Moment estimation, [Kingma et al., 2014]
[20]Conductance-Based

(inhibitory, excitatory) increases the conductance to its reversal potential. For a detailed motivation and history about CoBa synapses and their difference to CuBa, we refer to the literature: [Dayan et al., 2001; Gerstner et al., 2002; Petrovici, 2016]. Here, we limit our discussion to their functional differences.

$$C_\mathrm{m}\, \dot{u}(t) = g_\ell \left(E_\ell - u(t)\right) + I_\mathrm{syn}(t) \tag{12.59}$$

$$\begin{aligned} I_\mathrm{syn}(t) = {} & [E^\mathrm{rev}_\mathrm{exc} - u(t)] \underbrace{\sum_{i \in \mathrm{exc}} w_i \sum_{t_i} \theta(t - t_i)\, \exp\left(-\frac{t - t_i}{\tau_\mathrm{s}}\right)}_{=:g_\mathrm{exc}(t)} \\ & + [E^\mathrm{rev}_\mathrm{inh} - u(t)] \underbrace{\sum_{i \in \mathrm{inh}} w_i \sum_{t_i} \theta(t - t_i)\, \exp\left(-\frac{t - t_i}{\tau_\mathrm{s}}\right)}_{=:g_\mathrm{inh}(t)} \end{aligned} \tag{12.60}$$

Which differs from Equation (12.10) by its differently shaped synaptic input current. Here, $E^\mathrm{rev}_\mathrm{exc}$ and $E^\mathrm{rev}_\mathrm{inh}$ are the excitatory and inhibitory reversal potentials. We have omitted that both synapse types can have different synaptic time constants. Also note that synaptic weights w_i represent a different physical quantity and hence differ in dimensions, as their names suggest: For CuBa weights are currents (typically given in nA) and for CoBa conductances (typically given in nS). We define the total synaptic conductance

$$g_\mathrm{tot}(t) := g_\ell + g_\mathrm{exc}(t) + g_\mathrm{inh}(t) \tag{12.61}$$

and divide both sides of Equation (12.59) by it. As before we set E_ℓ to zero and reorder to achieve:

$$\begin{aligned} \frac{C_\mathrm{m}}{g_\mathrm{tot}(t)}\, \dot{u}(t) = -u(t) & + \frac{E^\mathrm{rev}_\mathrm{exc}}{g_\mathrm{tot}(t)} \sum_{i \in \mathrm{exc}} w_i \sum_{t_i} \theta(t - t_i)\, \exp\left(-\frac{t - t_i}{\tau_\mathrm{s}}\right) \\ & + \frac{E^\mathrm{rev}_\mathrm{inh}}{g_\mathrm{tot}(t)} \sum_{i \in \mathrm{inh}} w_i \sum_{t_i} \theta(t - t_i)\, \exp\left(-\frac{t - t_i}{\tau_\mathrm{s}}\right) \end{aligned} \tag{12.62}$$

At the time of writing, no closed-form solution for membrane dynamics of LIF neurons with CoBa synapses are known, much less an expression for the first spike-time given sufficient input spikes. We therefore need to approximate CoBa dynamics, which is not uncommon when training spiking neural networks [Neftci et al., 2019]. Comparing Equation (12.62) and Equation (12.10), we observe a strong resemblance. The only glaring differences are the time dependent conductance in the pre-factors and a separation into excitatory and inhibitory branches.

We make the assumption that spike-times for the CoBa model can be predicted reasonably well by using a CuBa model and scaling synaptic weights by a fixed weight-scale factor (WSF) $\alpha_{\mathrm{CoBa}\to\mathrm{CuBa}}$. This is not to be confused with the concept of a weight sum cost in [Mostafa, 2017]. Formally, we can express our assumptions as finding $\alpha_{\mathrm{CoBa}\to\mathrm{CuBa}}$ such that:

$$\frac{E_{\text{exc}}^{\text{rev}}}{g_{\text{tot}}(t)} \approx \frac{\alpha_{\text{CoBa}\to\text{CuBa}}}{g_\ell} \approx \frac{E_{\text{inh}}^{\text{rev}}}{g_{\text{tot}}(t)} \quad \text{and} \quad g_{\text{tot}}(t) \approx g_\ell \implies \frac{C_{\text{m}}}{g_{\text{tot}}(t)} \approx \tau_{\text{m}} \tag{12.63}$$

This effectively maps weights

$$w_{\text{CuBa}} \longmapsto \tilde{w}_{\text{CuBa}} = \alpha_{\text{CoBa}\to\text{CuBa}}\, w_{\text{CoBa}} \tag{12.64}$$

Its dimension is that of a voltage:

$$[\alpha_{\text{CoBa}\to\text{CuBa}}] = \left[\frac{w_{\text{CuBa}}}{w_{\text{CoBa}}}\right] = \left[\frac{\text{A}}{\text{A}/\text{V}}\right] = \text{V} \tag{12.65}$$

For applications in hardware, the $\alpha_{\text{CoBa}\to\text{CuBa}}$ is determined experimentally by simulating or emulating network dynamics. By inserting spike-times and $\tilde{w}_i$ into Equations (12.16) and (12.34), we find for $\alpha_{\text{CoBa}\to\text{CuBa}}$:

$$\text{case \#1 } (\tau_{\text{m}} = \tau_{\text{s}}) \qquad \alpha_{\text{CoBa}\to\text{CuBa}} = \frac{\vartheta\, g_\ell \, \exp\left(\frac{T}{\tau_{\text{s}}}\right)}{a_1 \frac{T}{\tau_{\text{s}}} - b} \tag{12.66}$$

$$\text{case \#2 } (\frac{\tau_{\text{m}}}{\tau_{\text{s}}} = 2) \qquad \alpha_{\text{CoBa}\to\text{CuBa}} = \frac{\vartheta\, g_\ell}{a_2 \, \exp\left(\frac{T}{2\,\tau_{\text{s}}}\right) - a_1 \, \exp\left(\frac{T}{\tau_{\text{s}}}\right)} \tag{12.67}$$

During training, we keep $\alpha_{\text{CoBa}\to\text{CuBa}}$ fixed. Please refer to Section 12.3.3 for results.

12.3 | Results

This section details results of applying the presented TTFS framework to various backends. A summary is found in Table 12.2.

Overview of Datasets We apply the learning algorithm presented here is applied to several different datasets. The obvious first choice is MNIST for better comparability to [Mostafa, 2017]. But using MNIST has some downsides: First, it consists of images composed of 28×28 pixels. Therefore, it requires at least 784 input channels for neurons in the first layer which exceeds the capabilities of some early prototype chips.[21] Furthermore, it is fairly easily separable with shallow architectures. A simple linear classifier is able to achieve 88 % classification accuracy [LeCun et al., 1998].

Additionally, a new classification task was envisioned by Laura Kriener and Julian Göltz: Yin-Yang [Kriener et al., 2021]. In contrast to MNIST's high-dimensional, fairly disjoint input patterns, it features a more low-dimensional and "continuous" approach, shown in Figure 12.5a. Each data sample consists of a pair of coordinates $(x, y) \in [0, 1]^2$. For robustness, and in order to exemplify the inherent symmetry present in the Yin-Yang

[21]For example, the first HICANN-DLS[22] prototype chip features 32 Neurons with 128 synpatic input lines.

dataset	**hidden neurons**	**test accuracy** [%]	**train accuracy** [%]
Yin-Yang			
in SW	120	95.9 ± 0.7	96.3 ± 0.7
on BrainScaleS-2	120	95.0 ± 0.9	95.3 ± 0.7
MNIST			
in SW	350	97.1 ± 0.1	99.6 ± 0.1
in SW ($\tau_\mathrm{s} = 2\tau_\mathrm{m}$)	350	97.2 ± 0.1	99.7 ± 0.1
MNIST 16×16			
in SW	246	97.4 ± 0.2	99.2 ± 0.1
on BrainScaleS-2	246	96.9 ± 0.1	98.2 ± 0.1

Table 12.2: Summary of TTFS results. Accuracies are given as mean value and standard deviation. For comparison, the Yin-Yang dataset is classified by a (non-spiking) linear classifier with (64.3 ± 0.2) % test accuracy, while a (non-spiking, not particularly optimized) ANN with 120 hidden neurons achieves (98.7 ± 0.3) %. In case of MNIST, a $784 - 350 - 10$ fully connected ANN manages to reach a test accuracy of (98.2 ± 0.1) %. Adapted from: [Göltz et al., 2021, Table 2].

symbol, each data sample is augmented by its mirrored coordinates $(1 - x, 1 - y)$. The dataset features three aptly named labels: *Yin*, *Yang* and *Dot*. It was specifically designed to not be linearly separable: Shallow classifiers reach around (64.3 ± 0.2) % accuracy, whereas an ANN with one hidden layer of 120 hidden units achieves (98.7 ± 0.3) %. It is because of this large gap that Yin-Yang can serve as an expressive test for the TTFS algorithm presented here. Finally, due to its relatively small input size (4 input spikes and an additional bias spike not part of the dataset) a complete deep network can be mapped comfortably onto one HICANN-X[23] chip.

12.3.1 Software-Only Simulations

Yin-Yang Figure 12.5 shows classification of the aforementioned Yin-Yang dataset in a network with a 5–120–3 structure employing Equation (12.54). There are five input neurons: $(x,\, y)$ coordinates and their “mirrors” mapped to the spike-timing interval via Equation (12.57) as well as a bias spike inserted in the middle of the input interval. This is done to reduce the maximum possible distance between input spikes to the hidden layer, facilitating learning. A single hidden layer holds 120 neurons, followed by three label neurons indicating the network’s decision.

The training procedure is illustrated via exemplary voltage traces of all three label neurons in Figure 12.5e (the examples are marked in Figure 12.5a). Please note that these are *not* used during training as the TTFS algorithm presented here operates on spike-times only. While all three label neurons spike roughly at the same time at the beginning of training (due to randomized weights), we observe a clear separation after training, as

[23]Short Form of HICANN-DLS-SR-HX, [Schemmel et al., 2020]

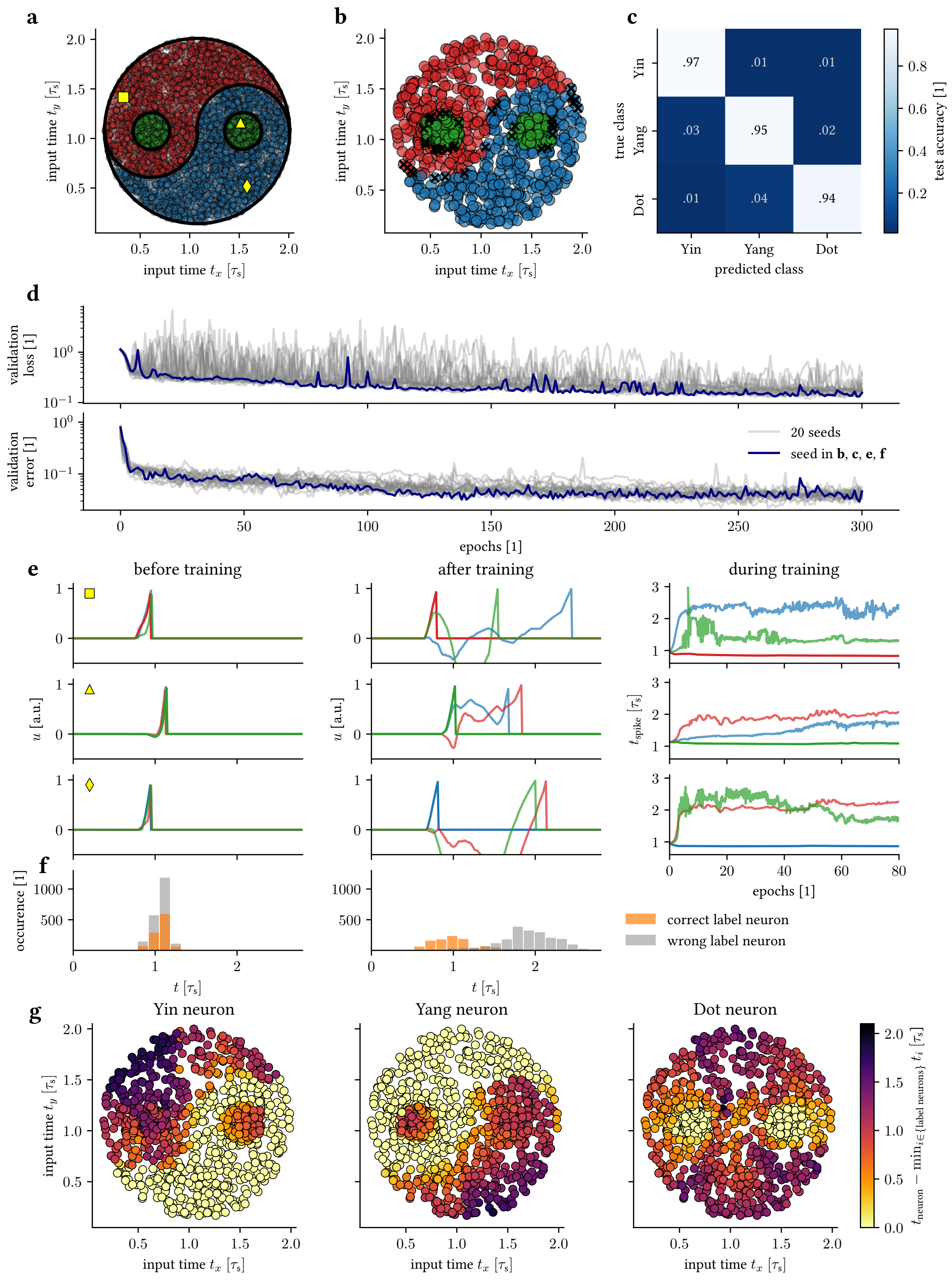


Figure 12.5: (Caption on next page.)

Figure 12.5: Classification of the Yin-Yang data purely in software.
(a) Illustration of the Yin-Yang dataset [Kriener et al., 2021]. Samples are separated into three classes, Yin (●), Yang (●) and Dot (●). Yellow symbols (□, △, ◊) mark samples for which the training process is illustrated in (b). Input times t_x and t_y correspond to spike-times associated with x and y coordinates of samples (cf. Equation (12.57)). Additionally, there are "mirrored" input spike sources that emit spikes at $t_{\text{late}} - (t_x - t_{\text{early}})$ and $t_{\text{late}} - (t_y - t_{\text{early}})$, respectively.
(b) Classification result on the test set (1000 samples). Color of each sample indicates which class was determined by the trained network. Wrongly classified samples (marked with black X) all lie very close to the border between classes.
(c) Confusion matrix for the test set after training.
(d) Training progress (validation loss as given in Equation (12.58) and error rate) over 300 epochs for 20 training runs with random initializations (gray). The run shown in panels (b), (c), (e) and (f) is plotted in blue.
(e) Training mechanism for three exemplary data samples from (a). In first three rows, left and middle columns depict voltage dynamics in the label layer before and after training for 300 epochs, respectively. Voltage traces of all three label neurons are color-coded according to their corresponding class in (a). Before training, random initialization of weights causes label neurons to show similar voltage traces and almost indistinguishable spike-times. After training there is a clear separation between spike-times of the correct label neuron and all others: The correct neuron spikes first. Evolution of label spike-times during training is shown in the right column over first 80 epochs.
(f) Spike histograms over all training samples. TTFS induces a clear separation between spike-times of correct and wrong label neurons.
(g) Spike-times of Yin, Yang and Dot neurons for all test samples after training. For each sample, spike-times are measured from the first label layer spike. Bright yellow denotes zero difference, i.e., correct classification due to the respective label neuron spiking first. High classification accuracy after training is reflected in these bright yellow areas resembling shapes of Yin, Yang and Dot areas. Adapted from: [Göltz et al., 2021, Figure 2]

shown in Figure 12.5f. For input data that is firmly located within each class (i.e., not on the border between two labels), separation is achieved after relatively few epochs (less than ten).

After 300 epochs the network is able to classify the test dataset with $(95.9 \pm 0.7)\,\%$ accuracy, estimated from 20 different random seeds. As shown in Figure 12.5b, we see that the misclassifications happen on the edge between labels. Figure 12.5g confirms this: Roughly speaking, we see an increase in spike-time difference for off-label neurons (i.e., label neurons not coding for the currently presented class) the further we are from the decision boundary. However, as we approach the decision boundaries, a sharp decline in spike-time differences at the edges between labels becomes apparent.

MNIST Next we classify MNIST in order to study the scalability of our approach to larger and higher-dimensional datasets, as seen in Figure 12.6. Here, we use a 784–350–10 network structure with pixel intensity translated to input spike-times (cf. Equation (12.57)). In order to aid generalization, Gaussian noise was applied to spike-times, but, in contrast to Yin-Yang, no bias spikes were used because input spikes are distributed more evenly across the input interval. This renders a bias spike unnecessary.

Using 10 different initial random seeds, we estimate the test accuracy after training at $(97.1 \pm 0.1)\,\%$. For comparison, this is almost on par with the best estimate from [Mostafa, 2017] at 97.55 % (cf. Table 12.1) who used a considerably larger network with more than

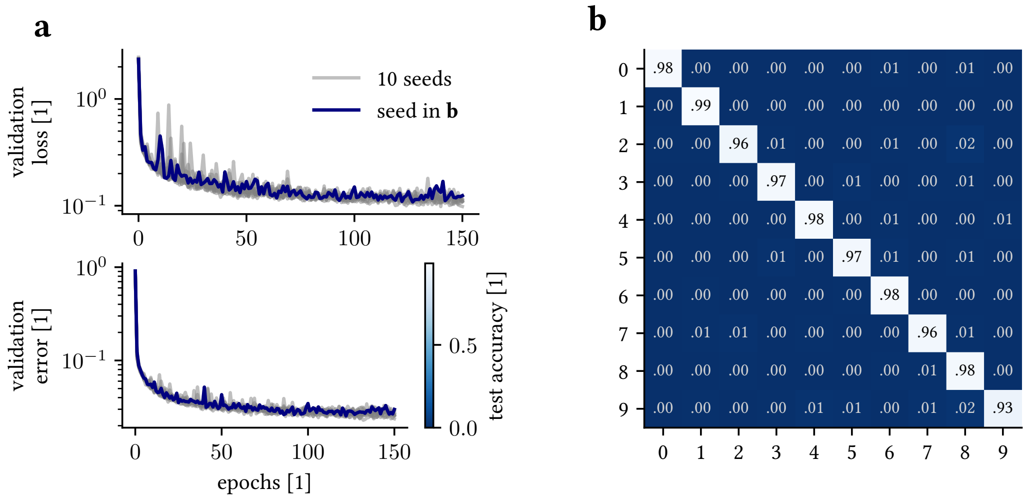


Figure 12.6: Classification of the MNIST dataset purely in software.
(a) Training progress of a network over 150 epochs for 10 different random initializations. The run drawn in blue is the one which produced the results in (b).
(b) Confusion matrix for test dataset after training. Adapted from: [Göltz et al., 2021, Figure 3]

double the amount of hidden non-forgetting IF neurons. In the same vein, [Kheradpisheh et al., 2019] report 97.4 % test accuracy using 400 hidden non-leaky IF neurons with a slightly different approach for calculating gradients. [Comsa et al., 2020], using 340 hidden units supported by a regular spike grid and extensive hyperparameter optimization, report a maximum test accuracy of 97.96 %. It should be noted that frameworks that rely on single spike-times, such as the one presented in this chapter, seem to have an intrinsic advantage in terms of accuracy over rate-based approaches. For comparison, [Esser et al., 2015] report 92.7 % test accuracy using 512 neurons while [Tavanaei et al., 2019] need 1000 hidden neurons to achieve 96.6 % test accuracy. Of course, using single spike-times comes at the cost of additional complexity when calculating gradients.

12.3.2 | Application to BrainScaleS-2

As shown in Section 12.3.1, time-to-solution is typically influenced by inherent time constants (τ_s and τ_m) as well as the depth of the network. Given biologically relevant timescales, this leads to classification results on the order of milliseconds. Since neuromorphic systems such as BrainScaleS[24] operate at an intrinsic acceleration factor of 10^3 to 10^4, we can perform the same computations within microseconds.

Executing on an (accelerated) neuromorphic mixed-signal substrate comes at at price: distortions induced by the analog substrate that the learning scheme has to deal with. These include spike-time jitter, limited weight range and resolution as well as variability in set neuron parameter.

Hardware is trained in-the-loop [Schmitt et al., 2017; Kungl et al., 2019; Cramer et al.,

[24]BrainScaleS Mixed-Signal Accelerated Neuromorphic Systems, [Schemmel et al., 2008; Schemmel et al., 2010; Schemmel et al., 2017; Schemmel et al., 2020]

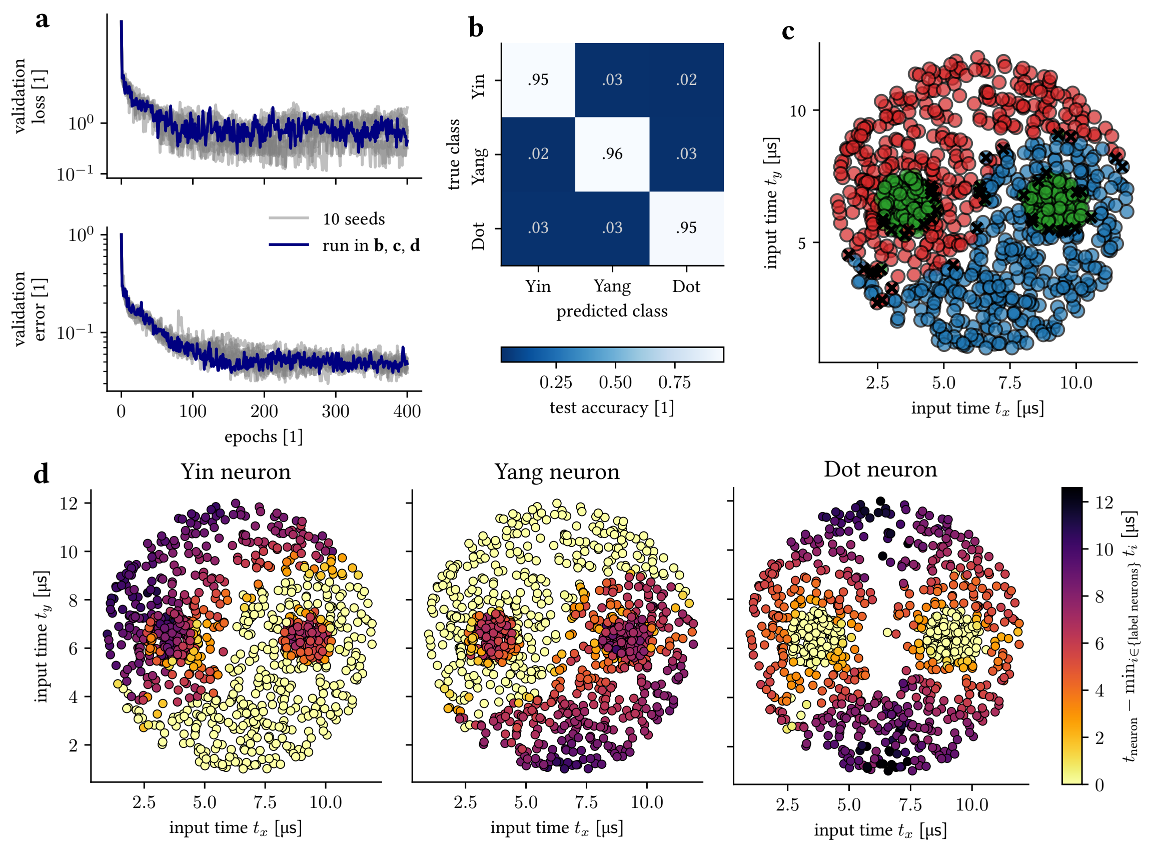


Figure 12.7: Classification of YinYang-Dataset on on BrainScaleS-2.
(a) Training progress over 200 epochs for 11 different random initializations. The run drawn in blue also produced the results shown in panel (b-d).
(b) Confusion matrix for the test set after training.
(c) Classification result on the test set. For each input sample the color indicates the class determined by the trained network. Wrong classifications are marked with a black X. The wrongly classified samples all lie very close to the border between two classes.
(d) Separation of label spike-times (cf. Figure 12.5e). For each of the label neurons, bright yellow dots represent data samples for which it was the first to spike, thereby assigning them its class. As already observed software simulations, the bright yellow areas align well with the shapes of Yin, Yang and Dot areas of the dataset. Adapted from: [Göltz et al., 2021, Figure 4].

2020]: The forward pass from Section 12.3.1 is replaced by a run on hardware. Spike-times emitted by the neuromorphic substrate are fed back into the PyTorch-based optimizing loop. In order to speed up learning, input data is fed to the hardware in a mini-batch scheme, with enough delay between consecutive images to ensure decay of membrane potentials. Weight updates are then averaged over gradients for the whole mini-batch. Since hardware parameters are set as unit-less DAC[25] values, they have to be calibrated and matched to their network counterparts. In particular, we matched PSP shapes and spike-times predicted by a software forward-pass. This is a common procedure when mapping any type of network or learning algorithm to hardware. In order to ensure our implicit assumption that each neuron only spikes once, we set a long-enough refractory

[25]Digital-to-Analog Converter

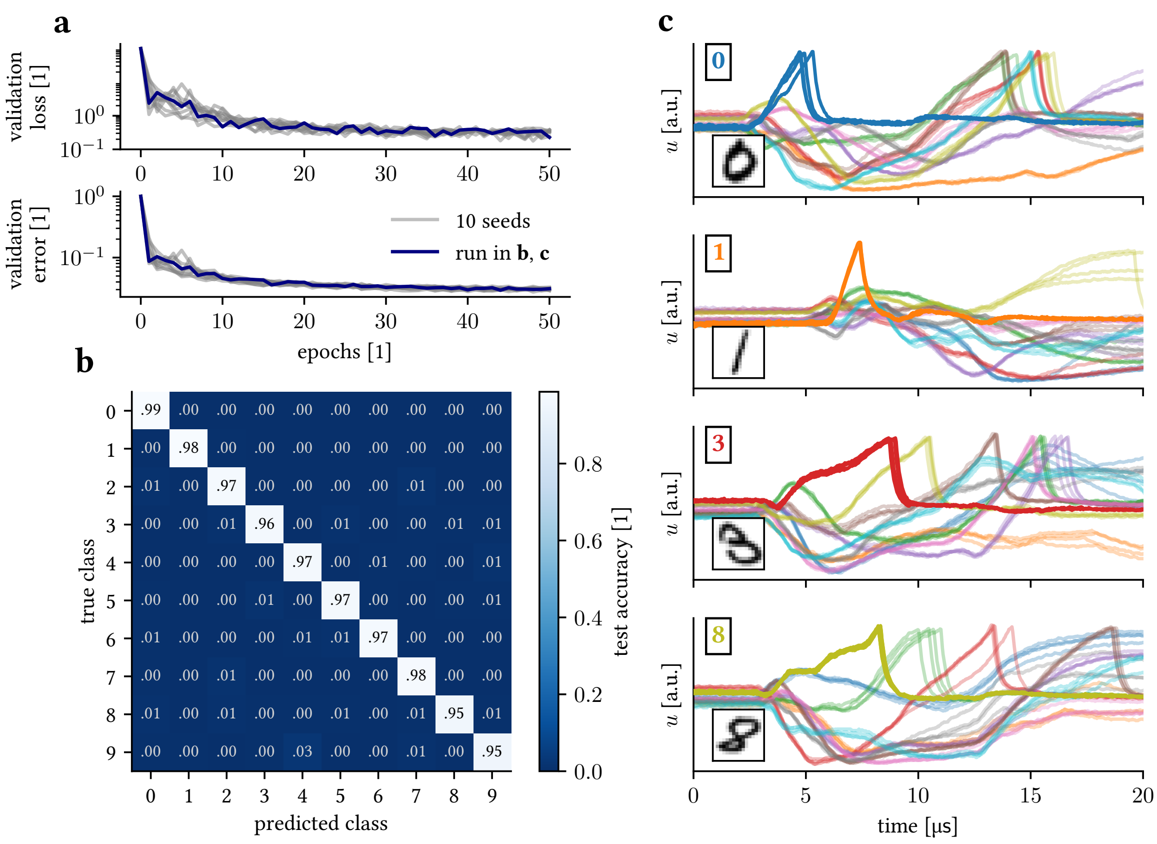


Figure 12.8: Classification of the MNIST dataset on BrainScaleS-2.
(a) Evolution of training over 50 epochs for 10 different random initializations. The run drawn in blue is the one which produced the results shown in panel (g) and (h).
(b) Confusion matrix for the test set after training.
(c) Exemplary membrane voltage traces on BrainScaleS-2 after training. Each panel shows color-coded voltage traces of four label neurons for one input that was presented repeatedly to the network (inlays show the input and its correct class). Each trace was recorded four times to point out the trial-to-trial variations. Due to limitations on simultaneous membrane readouts, each voltage trace was recorded in a different run. Adapted from: [Göltz et al., 2021, Figure 5].

period for each neuron so that any potential second spike is delayed sufficiently and cannot affect classification. As neurons forget on the time-scale of τ_s, setting a refractory period $\tau_\mathrm{ref} > \tau_\mathrm{s}$ proved to be sufficient. During training, we keep "shadow"-weights at full resolution in the backward pass and discretize them to BrainScaleS-2's 6 bit weight resolution when writing them back to hardware [Hubara et al., 2017].

Yin-Yang We train the classify the Yin-Yang dataset with a 5–120–3 network in-the-loop on BrainScaleS-2. For technical reasons, five inputs were not enough to provide sufficient stimulus to the network. This was due to a combination of limited weight range (that is set globally) and neuron variability. In order to mitigate this, each input was quintupled, resulting in an effective input layer of 25 sources. Further input copies could be added to increase stimulus even further. This has the additional benefit of averaging over some fixed-pattern variability because more circuitry is used. Figure 12.7 shows classification results. The system is able to quickly differentiate between all presented patterns, resulting in an overall test accuracy of (93.8 ± 0.4) %, estimated from 10 runs with different seeds. This is almost on par with software simulations (cf. Section 12.3.1). Misclassification still only happens on label-borders (cf. Figure 12.7c). Also, we see that label neurons might even cease to spike for input far from its label boundary, indicated by the slightly different dot-patterns in Figure 12.7d. We attribute the remaining gap in accuracy to the inherent trial-to-trial variations in hardware, as indicated later by Figure 12.8c.

MNIST We use BrainScaleS-2 to classify MNIST. Due to the constraints in chip dimensions (cf. Section 3.2.1), we are limited to a total of 256 simultaneous inputs. Please note that in HICANN-X, the prototype chip used, a single synaptic column consists of 256 synaptic circuits but only supports 128 arbitrarily signed inputs. This is because, for a single connection, two synapses are needed to facilitate either excitatory or inhibitory effect. Both are needed because a given weight might change sign during training.[26] In order to achieve 256 simultaneous inputs, we interconnect two neighboring neuron circuits from top and bottom half of the chip to effectively form a larger neuron.

Since MNIST originally has $28 \times 28 = 784$ input dimensions, we use a down-sampled version of the dataset with 16×16 pixels resolution. The overall network structure is 256–246–10, utilizing all neuron circuits on the HICANN-X. Results are shown in Figure 12.8. From ten different initializations we estimate the test accuracy to be (96.9 ± 0.1) %. As with Yin-Yang, we observe that the test accuracy is slightly below its simulated counterpart, which achieves (97.1 ± 0.1) % on down-sampled MNIST with the same network structure. The most likely explanation for the observed difference in accuracy is once again hardware variability. This is exemplified in Figure 12.8c: Despite setting the same network parameters, we record slightly different voltage traces resulting in slightly different spike-times.

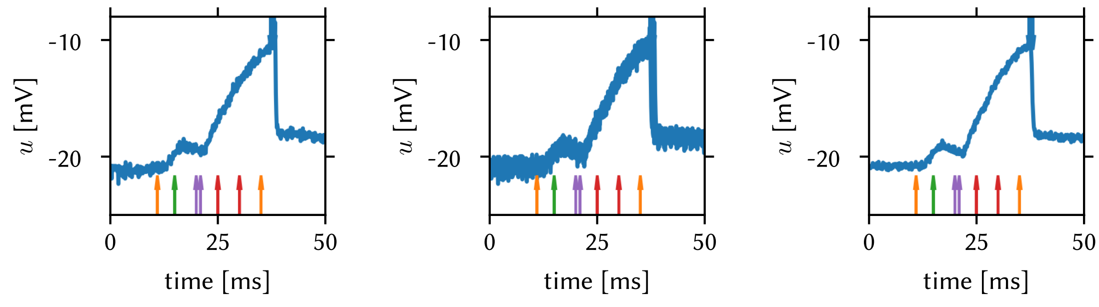
Resource Efficiency Leveraging BrainScaleS-2's speed-up factor of 10^3, which transforms biologically plausible time-constants on the order of milliseconds to microseconds in realtime, we are able to perform rather fast inference: As indicated in Figure 12.8c, the network typically reaches a classification result (i.e., a label neuron spikes) in less than 10 µs realtime. Overall, classifying the full MNIST test dataset comprised of 10 000 images takes a total of (0.968 ± 0.006) s realtime. This includes transmission to the controlling FPGA,[27] emulation and returning the classification results to the host computer. Thorough investigation reveals that actual on-chip inference takes about 480 ms, allowing for up to 20 800 classifications per second (when ignoring software overhead on the host computer). Using the on-chip power measuring capabilities,[28] we use a source-meter to estimate energy-consumption of all chip components needed for spike generation and processing on the cube-setup to be 175 mW. This includes high-speed communication links (approx. 60 mW), digital periphery with its clocking infrastructure (approx. 80 mW) and biasing of analog circuits (approx. 35 mW). In particular, this does not include power-consumption

[26]This is the same constraint affecting input size later in Chapter 13.
[27]Field-Programmable Gate Array
[28]Described in Section 3.2.3, but also applying to the cube-setup.

of the support-hardware, including the controlling FPGA, because it is not optimized for low-powered operation and easily replaceable. Only taking into account time actually spent on-chip, we estimate the energy consumption to be 8.4 µJ per classification.

The time per classification on-chip is rather high at 48 µs. This is due to the fact that, in the current implementation, we need to allow for enough time to pass for all membrane activity to cease prior to sending the next image. A different technique is employed by [Cramer et al., 2020]. They perform a "manual" reset of all membrane potentials via PPUs,[29] allowing for classification times down to 11.8 µs per image. In all presented experiments, typical time-to-solution is 1–1.5 τ_s. With $\tau_s \approx 6$ µs,[30] classification times of approximately 10 µs seem achievable.[31] This speed-up of almost factor 5 is bought by powering up the PPUs during inference,[32] adding about 20 mW to the power budget. In that case energy costs per classification come down to approximately 2 µJ.

These energy measurements are to be considered preliminary and are only meant to give a ballpark estimate. For an overview and comparison with other implementations please refer to Table 12.3.



(a) A single voltage trace. (b) All 20 traces. (c) Average over all 20 traces.

Figure 12.9: Voltage traces on the BrainScaleS-1. Spike-times are given in biological units (i.e., 10^4 times slower than realtime).
Due to readout noise there are variations when recording a voltage trace on hardware. Here, 20 repetitions of the same setup are recorded in sequence in one run. Input spikes are displayed as arrows from below. Output spikes are displayed as blue arrows from above. Readout noise is reduced by averaging. There is no averaging for the output spikes, all recorded spikes are shown, i.e., outliers are visible. Adapted from: [Göltz, 2019, Figure 6.8].

12.3.3 Application to BrainScaleS-1

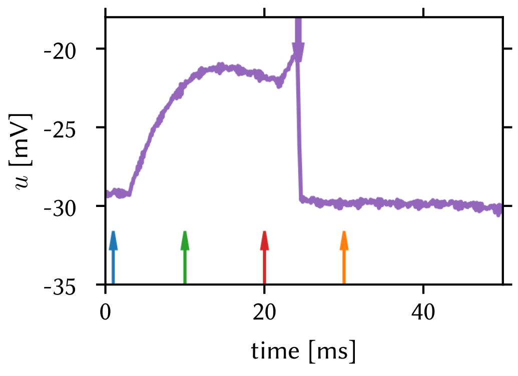
An early version of the TTFS algorithm presented here was implemented on BrainScaleS-1 (cf. Section 3.1), implemented during [Göltz, 2019]. At the time, the current BrainScaleS-2 prototype chip, HICANN-DLS, was too small for suitable application scenarios. As explained in Section 12.2.5, we need to determine the WSF $\alpha_{\mathrm{CoBa}\to\mathrm{CuBa}}$. Because BrainScaleS-1

[29]Plasticity Processing Units
[30]The exact parameters always depend on the chip used and its calibration.
[31]As seen in Figure 12.8c, the correct label neuron always spikes well within the first 10 µs.
[32]In all experiments presented here both PPUs were powered off.

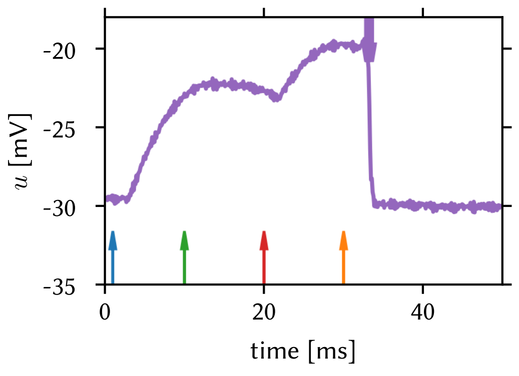
Platform	Type	Technology	Coding	Network Size/Structure	Energy per Classification	Classifications per Second[1]	Test Accuracy	Reference
Nvidia Tesla P100	digital	14 nm	ANN	CNN	852 μJ	125 000	99.2 %	[1][8]
SpiNNaker	digital	130 nm	rate	764–600–500–10	3.3 mJ	91	95.0 %	[2]
True North[2]	digital	28 nm	rate	CNN	0.27 μJ	1000	92.7 %	[3]
True North[2]	digital	28 nm	rate	CNN	108 μJ	1000	99.4 %	[3]
FPGA[3]	digital	45 nm	temporal	784–600–10	—	—	96.8 %	[4]
unnamed (Intel)[4]	digital	10 nm	temporal	236–20	17.1 μJ	6250	89.0 %	[5]
unnamed (Intel)[5]	digital	10 nm	temporal	784–1024–512–10	112.4 μJ	—	98.2 %	[5]
unnamed (Intel)[5]	digital	10 nm	temporal	784–1024–512–10	1.7 μJ	—	97.9 %	[5]
Loihi[6]	digital	14 nm	temporal	1920–10	—	—	96.4 %	[6]
SPOON[7]	digital	28 nm	temporal	CNN	0.3 μJ	8547	97.5 %	[7]
BrainScaleS-2	mixed	65 nm	temporal	256–246–10	8.4 μJ	20 800	96.9 %	[1][9]

[1] Please note that the platforms presented achieve high throughput through different means: Some process a large number of input samples in parallel while others operate on single samples in a sequential but fast manner.
[2] In [Esser et al., 2015] it is stated that "The instrumentation available measures active power for the network in operation and leakage power for the entire chip, which consists of 4096 cores. We report energy numbers as active power plus the fraction of leakage power for the cores in use.". For the first result 5 cores were used, while the second result requires 1920 cores.
[3] IF implementation. No energy or speed measurements reported.
[4] Images preprocessed with 4 5×5 Gabor filters and 3×3 pooling.
[5] No speed measurements reported.
[6] No energy or speed measurements reported. Images were preprocessed with an algorithm described as "using scan-line encoders".
[7] Reported energy values are pre-silicon simulations.
[8] Reference measurement for comparison against current commercial solutions.
[9] Presented in this chapter.

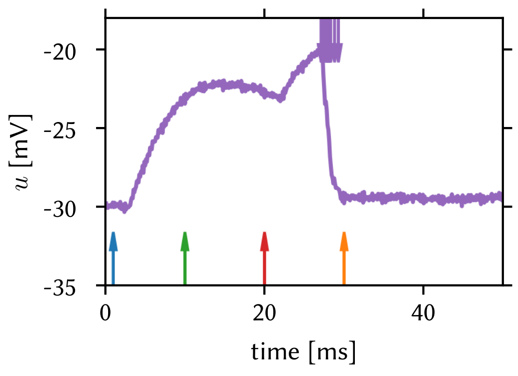
Table 12.3: Literature review for pattern recognition models on neuromorphic back-ends, including results which do not detail certain measurements. **References:** [1]: [Göltz et al., 2021], [2]: [Stromatias et al., 2015], [3]: [Esser et al., 2015], [4]: [Mostafa et al., 2017], [5]: [Chen et al., 2018], [6]: [Lin et al., 2018], [7]: [Frenkel et al., 2020]. Adapted from: [Göltz et al., 2021, Table 1 & SI.F1].



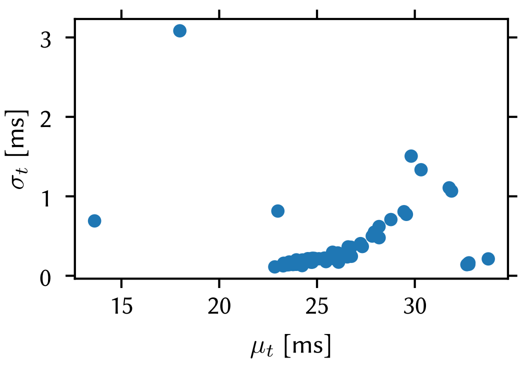
(a) Output spikes happen early, directly after an incoming spike. We observe output spike-time variability is low.



(b) Output spikes happen directly after last input spike. We observe output spike-time variability is low.



(c) Output spikes happen between those of (a) and (b). We observe output spike-time variability is greater than in (a) and (b).



(d) We plot standard deviation σ against mean μ of output spike-times. While there is a positive correlation at first, σ reduces again as seen in (a) to (c). Median of mean spike-times $\mu_t \pm$ interquartile range is $24.9\,\text{ms}^{+2.1\,\text{ms}}_{-0.8\,\text{ms}}$.

Figure 12.10: Analysis of spike-time variation on BrainScaleS-1. Spike-times are given in biological units (i.e., 10^4 times slower than realtime).
Four input spikes arrive at a single neuron with varying threshold voltage, leading to a difference in output spike timing. Depending on when the output neuron fires (immediately after an input spike or some time after) has an influence on trial-to-trial output spike-time variance. Three example traces are shown for illustration. Data is generated from 71 jobs with 100 runs each. Adapted from: [Göltz, 2019, Figure 6.11].

uses FG[33] as parameter storage, there is a considerably larger trial-to-trial variability between experiment runs compared to BrainScaleS-2, i.e., when the hardware is configured anew. Once configured, readout noise can be decreased significantly by averaging over several readout traces (cf. Figure 12.9). As shown in Figure 12.10, spike variability is dependent on actual spike timing. During training, same as for BrainScaleS-2, we keep "shadow"-weights at full resolution in the backward pass and discretize them to BrainScaleS-1's 4 bit weight resolution when writing them back to hardware.

Since the experiments on BrainScaleS-1 predate the inception of Yin-Yang, we used a 7×7 dataset consisting of four simple input patterns for all experiments performed on BrainScaleS-1. The network structure is 49–20–4, i.e., it contains a single hidden layer with 20 neurons. Its mapping structure is visualized in Figure 12.11. We see that while

[33]Floating Gates

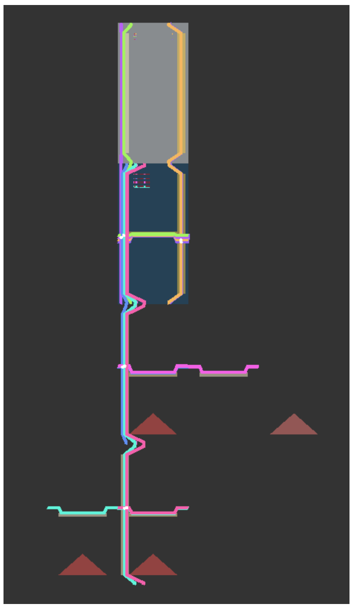


Figure 12.11: Visualisation (from [Boell, 2018; Weidner, 2019]) of the mapping of the network used on BrainScaleS-1 wafer 37. Network structure is 49–20–3 with all 20 hidden neurons on the HICANN shown in dark blue (HICANN 271). The label layer is the topmost area, HICANN 239. The inputs are distributed over 4 HICANNs (322, 323, 299, 301), indicated by the red triangles. The routes (cf. Figure 3.2) between the HICANNs are shown as colored lines. Taken from: [Göltz, 2019, Figure 8.1].

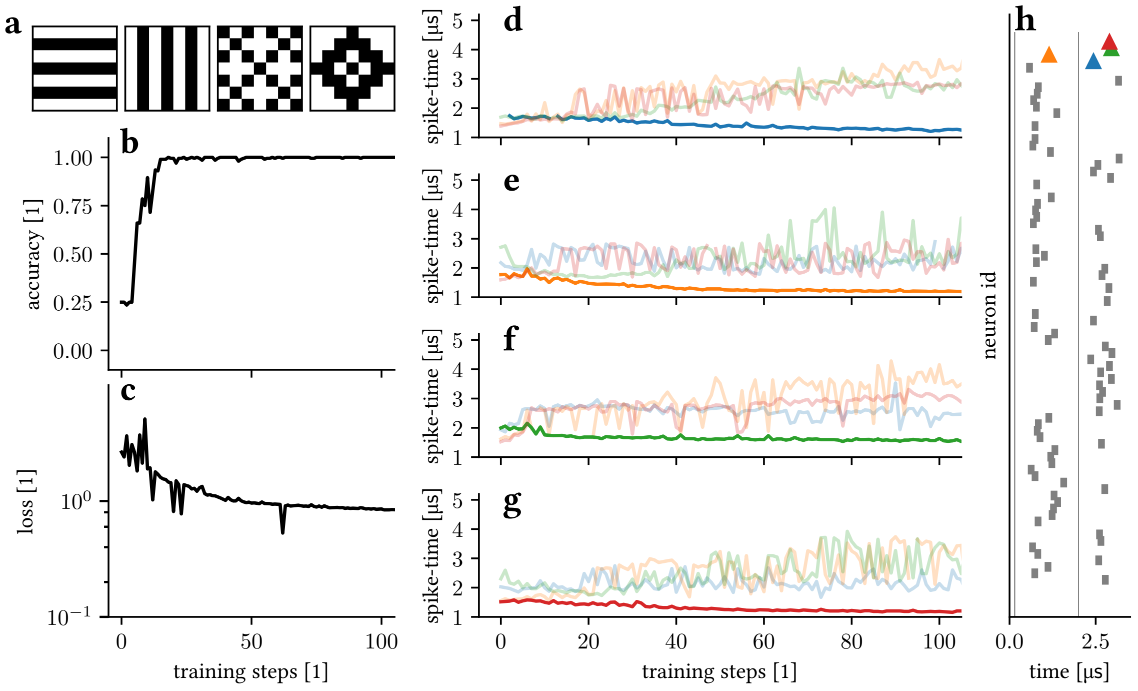


Figure 12.12: Training a spiking network on BrainScaleS-1.
(a) Simple dataset consisting of 4 classes with 7 × 7 input pixels.
(b/c) Accuracy and loss during training of the four pattern dataset.
(d-g) Evolution of spike-times (in realtime) in label layer for the four different patterns. In each, the neuron coding the correct class is shown in full color.
(h) Raster plot for the second pattern (e, correct class ▲) after training.
Adapted from: [Göltz et al., 2021, Figure SI.A1].

the full network can be placed on a single HICANN,[34] we need to send its 49 inputs from several other HICANNs. Classification results are shown in Figure 12.12. Despite increased variability and the need to translate weights via WSF $\alpha_{\mathrm{CoBa}\rightarrow\mathrm{CuBa}}$, the network manages to classify all sample patterns correctly with 4 bit weight resolution. As we can see in Figure 12.12d-g, the network manages to achieve a clear separation of spike-times between the correct label neuron and all others.

12.3.4 Investigating Robustness

In order to investigate further whether the presented TTFS learning and coding scheme is suited for application in neuromorphic substrates, we investigate its robustness towards hardware-induced distortions further. Most forms of physical neuronal substrates share the same forms of variability [Petrovici, 2016]. As discussed in Chapter 3, these include synaptic weights that are limited in both range and resolution as well as all parameters being afflicted by variability: both trial-to-trial as well as fixed-pattern. Learning methods that are inherently reliant on precise parameter values or unable to tolerate small distortions are inadequate for application in physical neuromorphic emulation [Petrovici et al., 2014].

Distortions are studied by applying them to software-simulations that are then trained on the Yin-Yang dataset. The results are presented in Figure 12.13. They include limiting the available weight range or resolution, applying noise to the time constants of each neuron both with and without a general shift that causes deviations from theoretical assumptions. In particular, we need to highlight the importance of re-inserting the output spike-time back into the weight update (cf. Equations (12.31), (12.32), (12.42) and (12.43)), which was was performed in all plots except Figure 12.13d.

In dimensionless weight units, scaled by the inverse threshold, an upper limit of $w_{\mathrm{clip}} \approx 3$ seems to be sufficient for maximum performance (cf. Figure 12.13a). This corresponds to a single spike being able to drive the membrane potential from resting to threshold voltage all on its own. In this case, when discretizing weights in forward-pass we observe almost constant performance up until 5 bit resolution, after which accuracy gradually declines (cf. Figure 12.13b). Weight updates in the backward-pass were still performed at floating point resolution. As shown in Figure 12.13c, the network appears very resilient to variability in synaptic and membrane time constants. In particular, Figure 12.13d highlights the importance of reinserting the output spike-time into the weight updates. The "naïve" variant, Equations (12.28) and (12.29), demonstrates performs considerably less robust.

Going one step further, we investigate the networks' robustness to distortions applied post-training (cf. Figure 12.14). For example, temperature variations can affect an already trained analog neuromorphic system.[35] We use 10 differently seeded MNIST in software with dimensionless parameters $\vartheta = 1$ and $\tau_{\mathrm{s}} = \tau_{\mathrm{m}} = 1$. Afterwards, we shift parameter values (either ϑ or both $\tau_{\mathrm{s}}/\tau_{\mathrm{m}}$) and randomly eliminate[36] a portion of neurons and perform

[34]High Input Count Analog Neural Network

[35]This is investigated in a different context in Section 13.3.5.

[36]Each of the 10 trained networks is subjected to the same 10 differently seeded random-deletion scenarios.

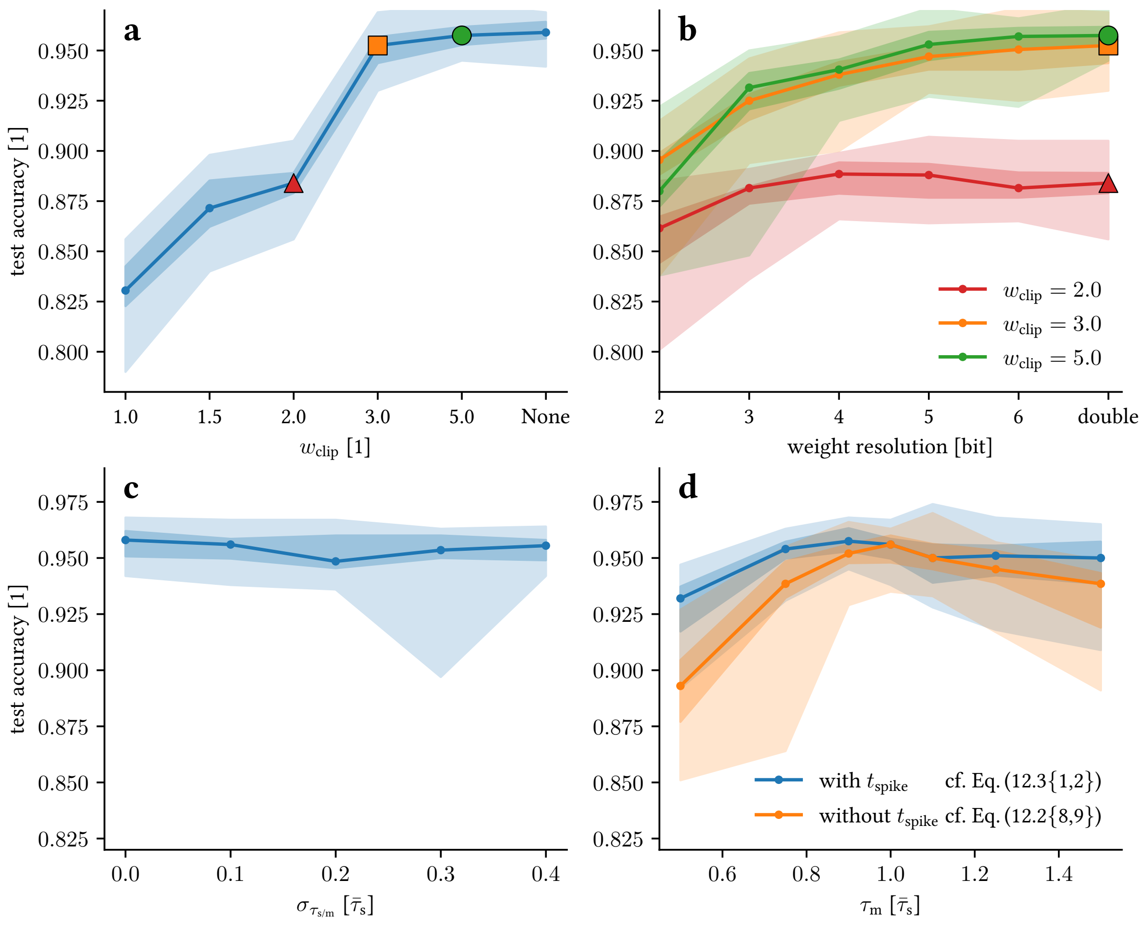


Figure 12.13: Effects of substrate "imperfections". Modeled constraints were added artificially to simulated networks. All panels show median, quartiles, minimum, and maximum of the final test accuracy on the Yin-Yang dataset for 20 different initializations.
(a) Limited weight range. The weights were clipped to the range $[-w_{\text{clip}}, w_{\text{clip}}]$ during training and evaluation. The triangle, square and circle mark the clip values that are used in panel (b).
(b) Limited weight resolution. For the three weight ranges marked in (a) the weight resolution was reduced from a double precision float value down to 2 bits. Here, n-bit precision denotes a setup where the interval $[-w_{\text{clip}}, w_{\text{clip}}]$ is discretized into $2 \cdot 2^n - 1$ samples (n weight bits plus sign).
(c) Time constants with fixed-pattern noise. For these simulations each neuron received a random τ_{s} and τ_{m} independently drawn from the distribution $N(\bar{\tau}_s, \sigma_{\tau_{\text{s/m}}})$. This means that the ratio of time constants was essentially never the one assumed by the learning rule.
(d) Systematic shift between time constants. Here τ_{s} was drawn from $N(\bar{\tau}_{\text{s}}, \sigma_{\tau_{\text{s/m}}})$ while τ_{m} was drawn from $N(\bar{\tau}_{\text{m}}, \sigma_{\tau_{\text{s/m}}})$ for each neuron for varying mean $\bar{\tau}_m$ and fixed $\sigma_{\tau_{\text{s/m}}} = 0.1\bar{\tau}_{\text{s}}$. The orange curve illustrates a training where the backward pass performs "naive" gradient descent, without using explicit information about output spike-times. The blue curve, as all other panels, has the output spike-time as an observable. Adapted from: [Göltz et al., 2021, Figure 6]

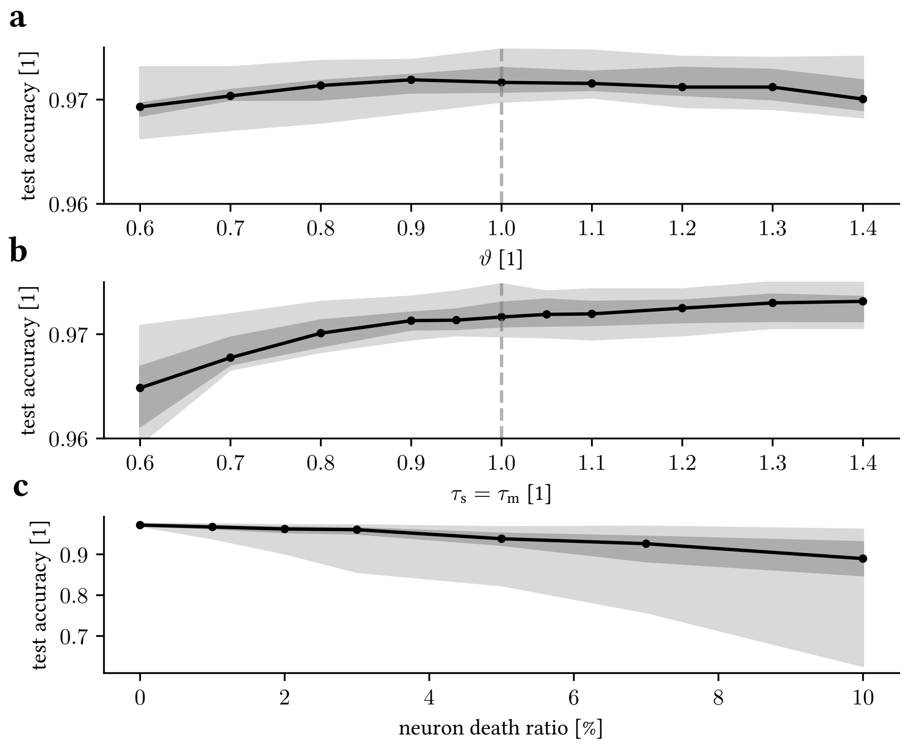


Figure 12.14: Robustness to variations *after* training. All panels show median (black), quartiles (dark gray), and full range (light gray) of results. **(a)** Dependence of test accuracy for evaluation for 10 trained networks with shifted threshold value ϑ. **(b)** Test accuracies for shifts in neuron time constants τ_s and τ_m. **(c)** Influence of random elimination of hidden neurons. For each neuron death ratio, 10 different random sets of hidden neurons were deleted. These ten deletion sets were applied to each of the ten networks from (a) and (b). Adapted from: [Göltz et al., 2021, Figure SI.C1].

inference *without* re-training. While the latter scenario is not that likely to occur in neuromorphic hardware, it is relevant for biological systems. In hardware it could pose as a way to re-purpose some synaptic columns on-chip after training for other functionality in future applications.

Overall, we see that the network is quite resilient to the induced distortions. Even for 5 % elimination rate the second quartile still bounds at 92.3 % accuracy. Interestingly, increasing both time constants appears to have a stabilizing effect, whereas slight variations in the firing threshold are hardly noticeable. In total, these findings suggest that “wrong” spikes, i.e., neurons suddenly becoming able to fire because of a reduced firing threshold or faster time constants, have a far more detrimental effect than delayed[37] spikes from “correct” neurons (i.e., which would also have spiked in the undistorted case).

Overall, our findings suggest that the presented TTFS-based learning paradigm is applicable even if time constants do not match the ratio they were derived for perfectly (up to 10–20 % difference), even after training. This makes it a prime candidate in scenarios where a model is trained once and then constantly used to perform time-critical inference on an analog neuromorphic device.

[37] either via increased threshold or longer time constants

Fast Analog Inference on BrainScaleS-2 13

This chapter presents work that has been done in close collaboration with Arne Emmel as part of his master thesis [Emmel, 2020] which the author had the pleasure to supervise. Core results of its continuation here are also reported in [Stradmann et al., 2021]. For full details on the author's contributions, please refer to Appendix A.

In deep learning, a network's discriminative power, in first-order approximation, is directly proportional to three aspects: the amount of parameters it contains, how large its training dataset is and for how long it is allowed to train [Sejnowski, 2020]. Hence, for current commercial applications, its computational power is directly proportional to the amount of MACs[1] we are able to perform. Additionally, in the future, the majority of these applications will consist of edge applications [Shi et al., 2016; Park et al., 2018; Dongarra et al., 2019; Chen et al., 2019]: Pre-trained models that are deployed onto embedded or mobile edge computing devices close to data storage, since moving vast amounts data is bound to become more expensive in terms of energy and responsiveness than computation itself [Toole, 2019]. Here, it is important to perform fast inference while using as little energy as possible. When taken to the extreme, both goals are only achievable with specialized hardware.

In this chapter, building on Chapter 2, we present one such approach: Using the non-spiking HAGEN[2] mode of BrainScaleS-2[3] (cf. Section 3.2.2), we implement conventional MAC and ReLU[4] operations in a fast analog compute scheme. Developed as a demonstrator for the competition BMBF[5] Pilotinnovationswettbewerb "Energieeffizientes KI-System" (*Energy-efficient AI system*), it uses a CNN[6] to classify ECG[7] data into healthy sinus rhythm and atrial fibrillation (AF). The network is designed to fit completely into the synapse array so that no reconfigurations are necessary during classification. It is the first application deployed onto the new BrainScaleS-2 Mobile[8] system (cf. Section 3.2.3).

Outline

Section 13.1 gives a very short introduction into the topic of Electrocardiography. Section 13.2 then describes how we can use CNNs to detect atrial fibrillation, describing

[1]Multiply-ACcumulate operations
[2]Heidelberg AnaloG Evolvable neural Network
[3]BrainScaleS-2 Analog Neuromorphic Hardware System, [Schemmel et al., 2017; Schemmel et al., 2020]
[4]Rectified Linear Unit
[5]Federal Ministry of Education and Research (*Bundesministerium für Bildung und Forschung*)
[6]Convolutional Neural Network
[7]ElectroCardioGram
[8]BrainScaleS-2 Mobile Analog Neuromorphic Hardware System, [Stradmann et al., 2021]

different intermediate model architectures (cf. Section 13.2.2). Finally, results are presented in Section 13.3, including energy-consumption measurements (cf. Section 13.3.4) and investigations into temperature stability (cf. Section 13.3.5). We conclude with a back-of-the-envelope calculation to estimate feasibility of the current ASIC[9] as sensor pre-trigger (cf. Section 13.3.6).

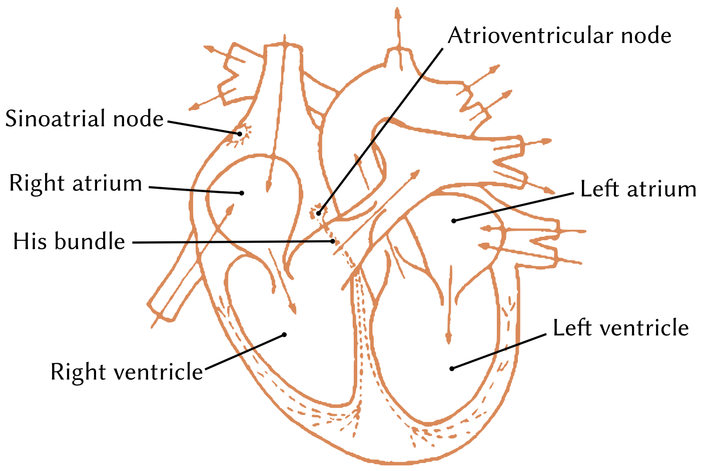


Figure 13.1: Schematic representation of human heart including conduction system. In a healthy heart, electrical signals originate at the sinoatrial node, are conducted through the atria and passed via the atrioventricular node to the ventricles, causing the heartbeat. Adapted from [Rangayyan, 2002].

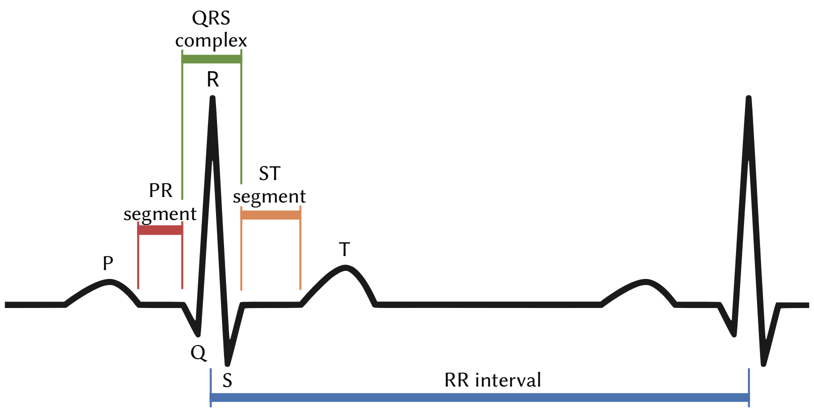


Figure 13.2: The individual components of the electrocardiogram of a simplified sinus rhythm signal. The *QRS* complex, including the *R* peak, is the most prominent and thus defines the beat position. The actual manifestation of the graph depends on individual factors, such as the position and orientation of the heart in the thorax, the physiological characteristics of the components of the heart and the placement of the electrodes on the skin. Missing or altered features can indicate an anomaly and thus a heart disease. In particular, atrial fibrillation can be diagnosed by irregular *RR* intervals and missing *P* wave. Adapted from: [Emmel, 2020, Figure 2.9]

[9] Application-Specific Integrated Circuit

13.1 Poor Man's Introduction to Electrocardiography

We start by giving a basic overview on electrocardiography, i.e., measuring the electrical activity of the heart using surface electrodes on the skin. The resulting ECG is a set of graphs measuring electric potential differences between designated points on the skin over time. It was developed at the beginning of the 20th century and still uses the nomenclature established by Willem Einthoven [Luederitz, 1993]. Besides allowing for easy reading off a person's heart rate, its curves are affected by cardiovascular diseases and abnormalities and can therefore aid in diagnosing them. One of these conditions is atrial fibrillation.

Ideal Shape of ElectroCardioGrams As shown in Figure 13.1, in a healthy heart, the electrical signal responsible for heart contractions (and therefore the heartbeat) originates at the sinoatrial node in its right atrium, travels via gap junctions through the atria and causes first right then left atrium to contract [Rangayyan, 2002]. Visible in ECG as P-wave (cf. Figure 13.2), it is comparatively slow at 80 ms duration and 0.1–0.2 mV amplitude.

Next, the signal passes the atrioventricular node, connecting atria and ventricles. The signal is delayed by about 60–80 ms, referred to as *PR* segment. To protect the ventricles from excessively fast contractions, the atrioventricular node remains refractory for a certain period. As an additional safety measure, the atrioventricular node possesses a slow intrinsic firing rate ensuring a minimum supply of blood in case the electrical conduction system of the atria breaks down. All in all, these mechanisms ensure a heart rate of about 40–300 $^1/_{\text{min}}$ [Guyton et al., 2006].[10]

What follows can be considered the "main component" of every heartbeat: A rapid depolarization produces a sharp *QRS* complex with an amplitude of about 1 mV over a duration of 80 ms. As it is most pronounced, the *R* peak is often used to define the beat position.

Action potentials of ventricular muscle cells typically last 300–350 ms, a comparatively long duration. After the *QRS* complex we first observe the *ST* segment, an isoelectric plateau of 100–120 ms. Subsequently, diffuse relaxation of muscle cells then causes a slow T wave of about 100–160 ms with an absolute amplitude of 0.1–0.3 mV.

Atrial fibrillation AF is the most common serious abnormal heart rhythm, affecting more than 33 million people worldwide [Chung et al., 2020]. It is defined as a "supraventricular tachyarrhythmia characterized by uncoordinated atrial activation with subsequent deterioration of atrial mechanical function" [Fuster et al., 2006]. In other words: The electrical conduction system of the heart is disordered. During AF, the electrical signal stimulating the contraction of ventricles longer originates from the sinoatrial node. Instead, the signal propagates continuously through the atria in a chaotic manner. Whenever passing the atrioventricular node while excitable, the signal enters the ventricles and triggers a contraction. This results in a replacement of P waves (cf. Figure 13.2) by rapid oscillations (or fibrillatory waves), resulting in an irregular heart rhythm. For example,

[10]According to the American Heart Association, for an adult a normal resting heart rate lies in range of 60–100 $^1/_{\text{min}}$, which corresponds to an average inter-heartbeat-interval of 0.6–1.0 s.

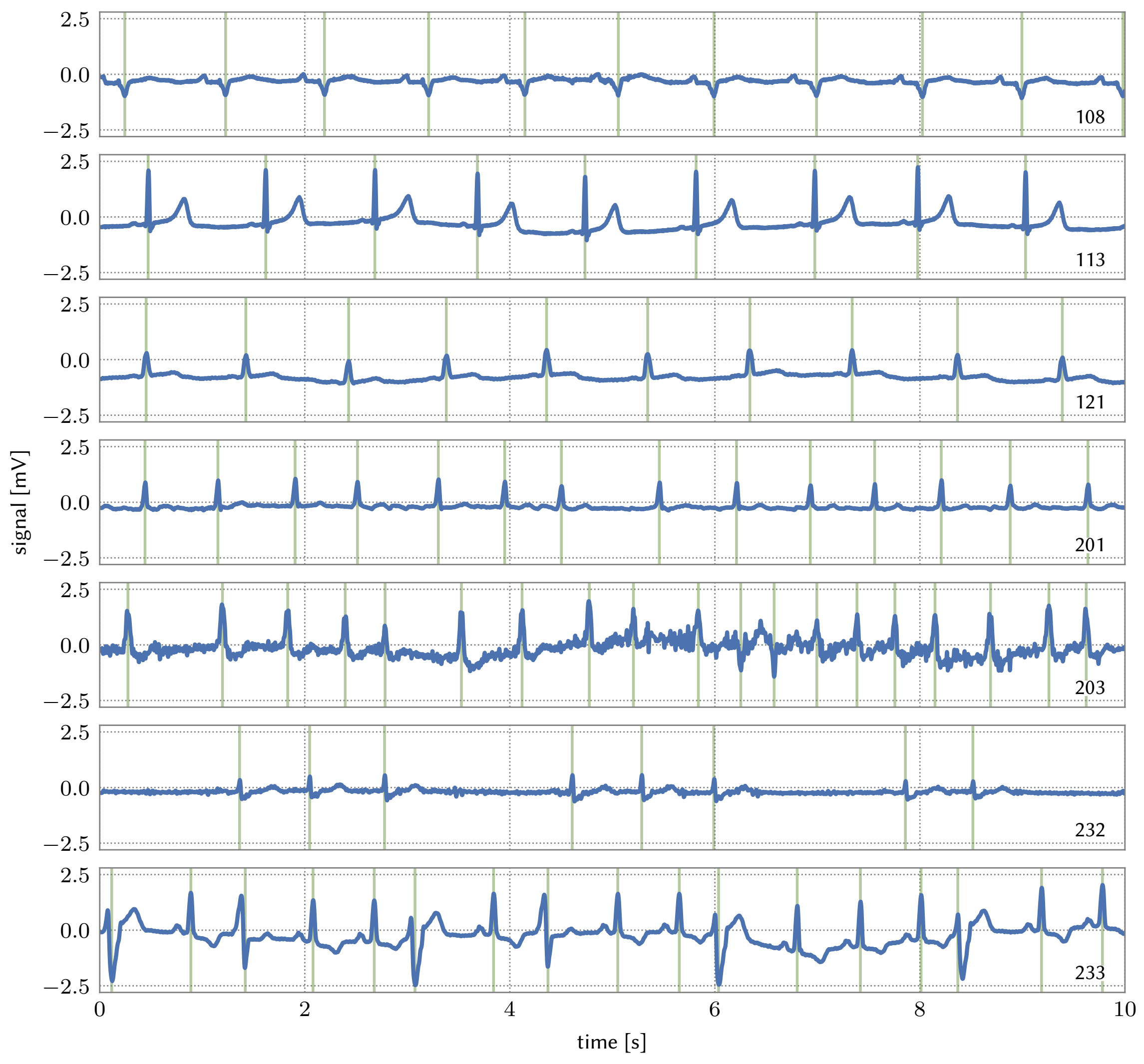


Figure 13.3: Some examples of ECG signals from MIT DB with annotated beat positions. The first three traces (108, 113, 121) show sinus rhythm. Although they are all regular, they differ significantly in shape and amplitude. Trace 201 is an example of AF, the most common arrhythmia, the last three (203, 232, 233) show examples of other arrhythmias. Especially trace 203 is subject to considerable noise. The beat annotations were carried out by several experienced cardiologists who assign each beat to one of 19 different types. Adapted from: [Emmel, 2020, Figure 2.12]

this can be observed in tape 201 in Figure 13.3, where such oscillations and irregular beats occur.

BMBF Dataset The main training dataset was provided as part of the competition Pilotinnovationswettbewerb “Energieeffizientes KI-System” (*Energy-efficient AI system*) [BMBF, 2019]. It contains 16 000 two-channel electrocardiogram recordings, half of them with sinus rhythm and half with atrial fibrillation. They are sampled at 512 Hz and are about 120 s in length. Each sample is a 16 bit unsigned integer value, of which only 12 bit are effectively used. Contestants were instructed to train their machine learning models on the provided training dataset. During final evaluation, each model was evaluated by BMBF referees on unseen training data.

13.2 Classifying ECG with CNNs

Beat-Detection Prior to describing the CNN-based model used to perform AF-detection on sensor data, it should be noted that CNNs are not the only method of solving this problem. For example, [Emmel, 2020] shows that a purely signal processing based approach can also be used to solve the problem at hand sufficiently. Here, we extract heartbeat positions and plot successive heartbeat intervals in a so-called Lorenz plot (cf. Figure 13.4). Using heartbeat intervals as input to a fully connected classifier with 20 hidden units, we achieve 93.6 % true positive and 10.3 % false negative detection rate. However, the beat-detecting preprocessing is rather involved, cannot be performed on the analog neuromorphic substrate and is limited to the task of AF detection. Since our goal is to investigate the general feasibility of BrainScaleS-2 as low-powered accelerator for CNNs, we do not investigate this approach further.

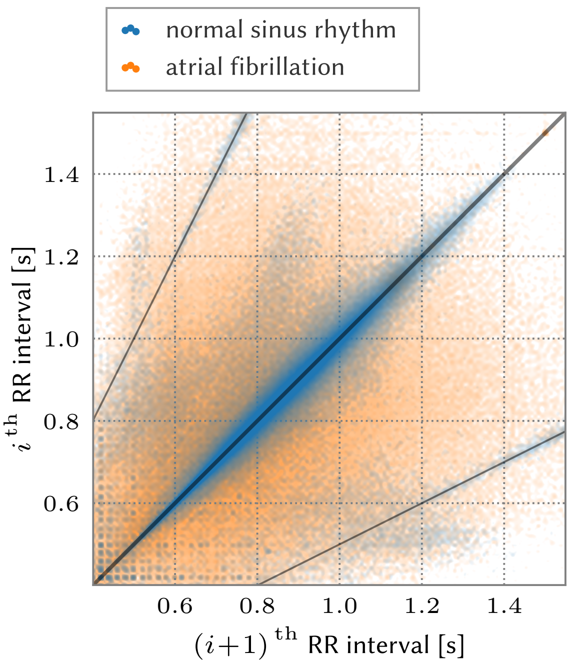


Figure 13.4: Lorenz plot from the detected beat positions in training set of the BMBF competition. Points on the diagonal correspond to two consecutive intervals of equal length. AF is spread over the entire area of the plot, which implies chaotic behavior. Samples with sinus rhythm tend to be concentrated around the main diagonal Further accumulations are observed on the two lines below and above, which correspond to twice the interval and indicates an undetected beat in between. The regular dotted pattern in the lower left is due to power-line interference at 50 Hz. Adapted from: [Emmel, 2020, Figure 5.2].

13.2.1 Preprocessing

As with any real world application in data science, the dataset contains distortions we need to deal with. This can either be done via a dedicated preprocessing step beforehand or built into a preprocessing pipeline and performed online. We opted to do both in that we used preprocessed data for training to conserve time but also supported online preprocessing

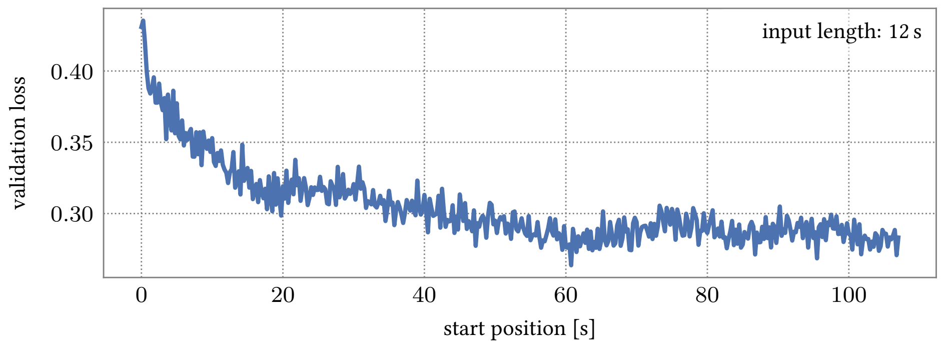


Figure 13.5: Dependence of average loss on starting position of time-slice of 12 s duration. Towards the end of ECG recordings the average loss is lower and therefore precision is significantly higher than at the beginning. This can be explained by increased physiological noise – e.g., patients readjusting sensors – typically occurring directly after recording start. After about 20 s the subject usually seems to be in a resting position and disturbances due to muscle activity become less likely. To be sure, we cut the first 60 s. Adapted from: [Emmel, 2020, Figure 5.9].

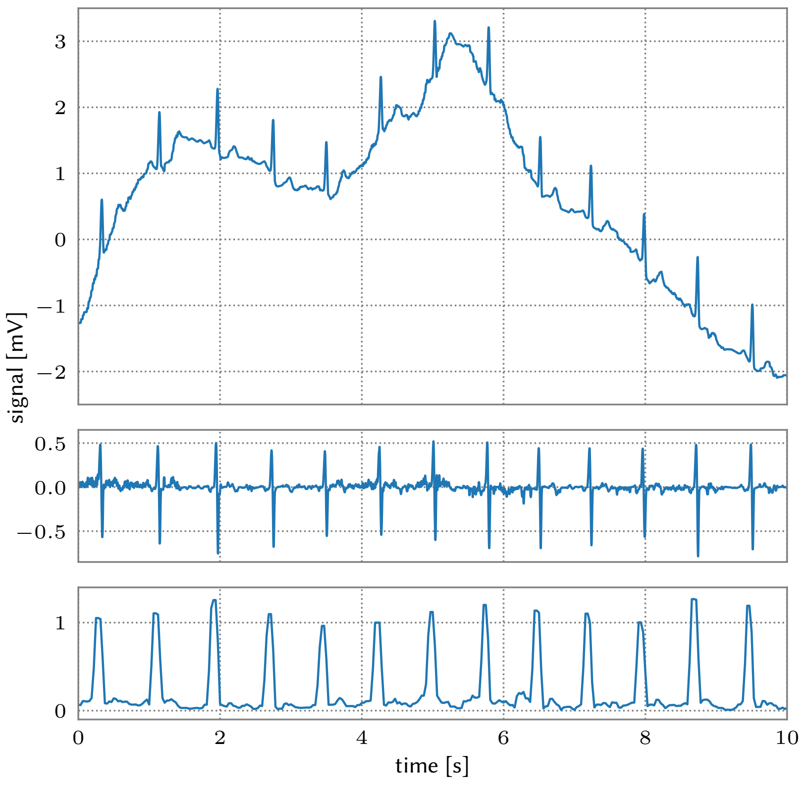


Figure 13.6: Preprocessing steps. **Top:** The raw, i.e., unprocessed, input sample. **Middle:** Signal after taking (discrete) derivative to reduce baseline fluctuations. **Bottom:** Subsequent maximum-minimum-difference pooling reduces the sample rate and provides positive activations, which form the final input signal to the CNN on the ASIC. For training, all preprocessing steps are done beforehand. During final evaluation, preprocessing is performed in the FPGA via FIR filters.
Adapted from: [Stradmann et al., 2021, Figure 8]

via FIR[11] filters on FPGA[12] on unseen test data during final competition evaluation. Functionally, both methods were ensured to be equivalent. Distortions include noise, baseline wandering, changing signal amplitudes and shape abnormalities. Common noise sources are 50 Hz or 60 Hz power line interference, other muscle activity and movements of the electrodes on the patients' skin.

It is important to note that the ECG shape itself is also influenced by a lot of individual factors during measurement. These include the exact position and orientation of the heart in the thorax, the physiological characteristics of the various components of the heart in relation to each other as well as electrode placement on the skin. Therefore, only few signals look as sketched in Figure 13.2. Certain details can appear weaker, inverted or may even be hidden. An illustrative example of this is shown in Figure 13.3, which contains exemplary ECG data from MIT DB.[13] Here, the first three lines correspond to sinus rhythms, but still have rather different shapes. This makes interpretation of electrocardiograms challenging even for experienced cardiologists.

Due to many artifacts occurring within the first minute of data (patients putting on electrodes etc.), we generally[14] skip the first 60 s of data. This was estimated by sweeping the cutoff time while performing inference, as seen in Figure 13.5. Here, we observed increased loss in accuracy up until the chosen cutoff at 60 s.

Preprocessing of the remaining data trace is then shown in Figure 13.6. It consists of a discrete derivate to suppress low frequency, i.e., baseline, oscillations as they are irrelevant for AF detection. Then we perform maximum-minimum-difference pooling: We slide a window of 32 entries across the data stream. In each window we compute the difference between the maximum and minimum entry, which has the effect of producing a strictly non-negative signal. By subsampling (using a stride of 16) we reduce the sampling rate to 32 Hz. Finally, we clip the data within $[57,\ 340]$,[15] rescale and quantize it to fit the 5 bit input-range of BrainScaleS-2.

13.2.2 | Network Structure

Accurate Variant When designing the model, we took an iterative approach. The first model designed is presented in Figure 13.8. Its hyperparameters were chosen such that the full network would fit onto a single chip. In particular, this means that no time- and energy-consuming reconfigurations of the synapse array are necessary during inference. It receives *full* input traces, truncated to 102 s at a subsampled rate of 32 Hz (cf. Section 13.2.1),

[11]Finite Impulse Response

[12]Field-Programmable Gate Array

[13]The Massachusetts Institute of Technology-Beth Israel Hospital Arrhythmia Database (48 records, 30 minutes each), `http://ecg.mit.edu/dbag/evnode3.htm` (visited on 2021-04-23)

[14]By "generally", we mean "whenever using shorter input slices". While we did use the full input trace for the *accurate* model, we skip the first 60 s for all *efficient* model runs. Both models are explained in Section 13.2.2.

[15]Please note that Figure 13.6 shows data in mV, while we describe operations on the numerical bit-values of raw data.

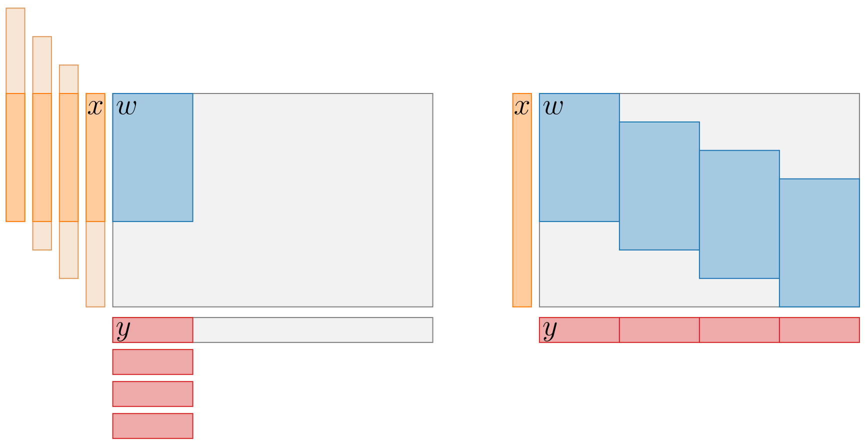


Figure 13.7: Two different variants of the one-dimensional convolution operation on BrainScaleS-2 implemented in `hxtorch`.
Left: The kernel weight matrix w is placed at a fixed position in the synapse array. The input vector x is multiplied by the kernel in several shifted variants to perform the convolution. Since the kernel is always in the same place, fixed-pattern deviations of the analog substrate can be compensated by adjusting individual weights. However, since operations have to be performed in sequence, they take longer.
Right: In order to maximize the usage of the synapse array, the same weight matrix w is rolled out several times on the synapse array. This reduces the number of necessary MAC operations on hardware by a factor determined by how often the kernel fits into the physical dimensions of the chip. Correspondingly, total operation runtime is reduced by the same factor. For small kernels with few channels this makes a significant difference. However, due to fixed-pattern noise in individual components of analog hardware, each parallel operation can incur slight deviations because different analog components perform the same logical operation. These can add up in the final convolution result, resulting in improved runtime at the cost of slightly reduced accuracy. Taken from: [Emmel, 2020, Figure 4.3].

i.e., 3278 samples in total. These may contain aforementioned artefacts, such as patients adjusting sensors.

The network features a single convolutional layer, implemented in the top half of the chip. It computes 14 different features with kernels of size 43. Its weights are duplicated 18 times across the upper half of the synapse array in order to speed up computation, as shown in Figure 13.7. Still, since each input vector contains 3278 samples, several MAC operations are necessary per input sample. Subsequent ReLU and MaxPool operations are performed via both on-chip PPUs,[16] same as bias values for all MAC outputs. The MaxPool-operation replaces every windows of size 36 by its maximum value. This significantly reduce the number of samples per each of the 14 features from 648 to 18. Per input trace, we therefore have total of 252 values after passing the top half of the chip.

The remaining features are thus processed further on the lower chip half in two fully connected linear layers of size 252×127 and 127×2, interspersed by another ReLU operation in PPUs. Because input to the synapse array is limited to 128 entries, we split the input vector to the first linear layer (of size 252) in two parts and send them to different quadrants in the lower half. Correspondingly, the first linear layer's weight matrix split and distributed over both quadrants of the lower synapse array. Both output halves are

[16]Plasticity Processing Units

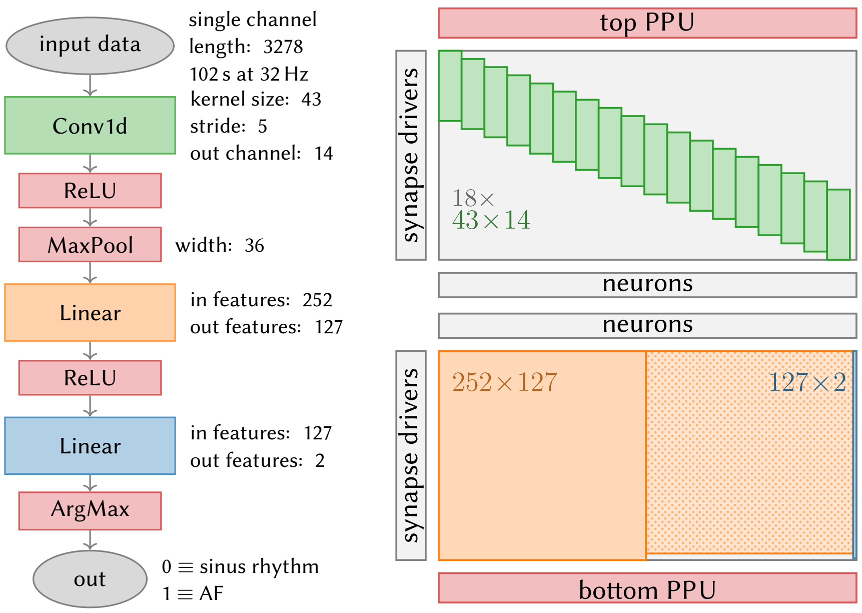


Figure 13.8: Network structure of *accurate* model that served as important step towards the efficient model. **Left:** Layer structure of the used deep convolutional neural network model. **Right:** Corresponding on-chip arrangement. The convolutional layer (green) is processed in the upper synapse array, the identical weight is arranged 18 times on the substrate to enable parallel processing. All ReLUs and MaxPooling after the first layer (red) are performed in digital logic on PPU. The further processing takes place on the lower synapse array with a fully connected layer and 127 hidden neurons (orange). It has to be split and arranged side by side, because it does not fit into the array in one piece. The dotted part of the layer uses a different label bit and is therefore able to receive the second half of inputs at the same time in parallel. The actual classification is then achieved in the last layer (blue) with two neurons on the right, which form the output. Adapted from: [Emmel, 2020, Figure 5.4]

then recombined in PPU prior to applying the interluding ReLU operation. In order to allow computing output of the last layer in parallel, we distinguish input to both matrices in the lower right quadrant by setting different label bits. This causes each matrix' entries to only process their designated input. Finally, whichever label neuron has the largest output determines classification.

Efficient Variant As described below, the accurate model variant reaches more than sufficient results for the BMBF competition. However, the main task is not accuracy, but efficiency while still meeting competition criteria.[17] To that end, a more efficient model iteration was devised. It is depicted in Figure 13.9. We discuss the most significant differences.

First, we reduce the amount of input data we actually consider for classification. As hinted at in Section 13.2.1 and Figure 13.5, using a slice of 13.5 s is more than enough. This reduces input size from 3278 to 432 entries with the same preprocessing.

Next, we tune the convolutional layer to feature a larger kernel (91 entries) but reduce the number of computed output features to 8. Overall, this increases the weight matrix

[17] $\geq$90 % true positive (AF) detection rate and $\leq$20 % false positives (sinus rhythm classified as AF)

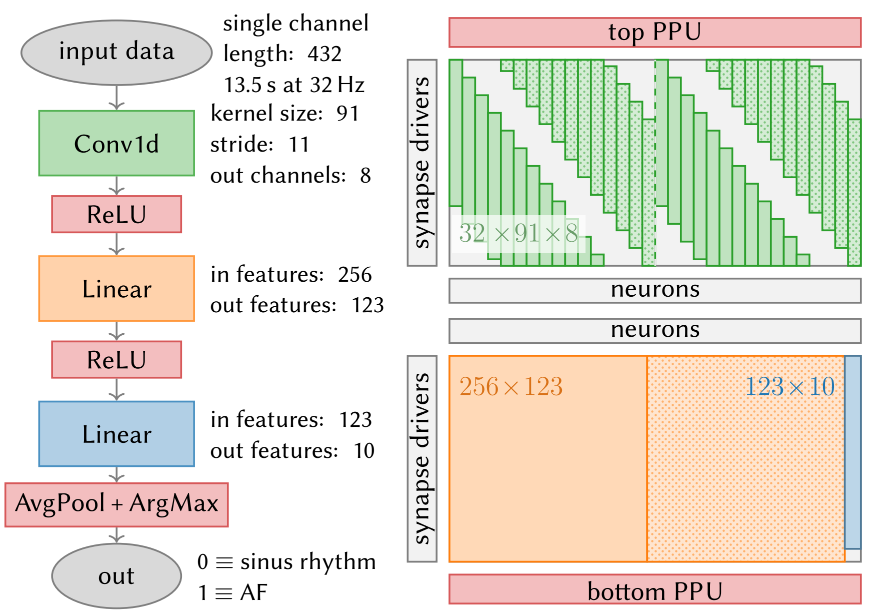


Figure 13.9: Network structure of *efficient* model that was used in BMBF competition.
Left: Layer structure of the used deep convolutional neural network model.
Right: Corresponding on-chip arrangement. The convolutional layer (green) is processed in the upper synapse array. Parallel processing is achieved by rolling out the convolutional weight matrix 32 times on the substrate. Each upper quadrant processes half of the input samples. The dotted areas are configured to compute the process the second half of each quadrants input that is sent with a different label bit. All ReLU operations (red) are performed in digital logic by both PPU. Further processing takes place on the lower synapse array with a fully connected layer and 123 hidden neurons (orange). To ensure efficient use of the substrate, it is divided into two parts and placed side by side. The dotted part of the layer receives the second half of inputs at the same time and is processed in parallel. The actual classification is then achieved in the last layer (blue) with 10 neurons on the right. While trained with Argmax, they are combined into two logical neurons by average pooling during validation to effectively reduce analog noise. Adapted from: [Stradmann et al., 2021, Figure 7].

by about $^1/_6$, but since energy-consumption does not noticeably depend on the portion of non-zero entries in the synapse-array, this is of no further concern. We eliminate the first MaxPool-operation – which is time-consuming because it cannot be performed in the analog core – by increasing the stride so that all output values can be fed to the first linear stage (after ReLU) without subsampling. Same as before, we duplicate the convolutional kernel 32 times to compute all 256 outputs of the first layer in parallel. The input stream is split in half and each half is sent to one of the upper synapse array quadrants. Here, each quadrant's input is split again and the second half is transmitted with a different label bit. This causes the dotted-green parts in Figure 13.9 to process the second half, while the first half is getting processed by the regular green parts, virtually doubling the input stream length of each quadrant to a maximum of 256. This sufficient for the 216 entries each quadrant receives. We are thus able to perform the first layer's convolution *in a single MAC operation.*

The first linear layer is adjusted in size to fully utilize the maximum input bandwidth of 256. Its output dimensions were reduced to 123 in order to allow for a five times redundancy of each label neuron in the second linear layer (correspondingly resized to 123×10). This increases robustness, especially to temperature variations, etc. During training, we performed 50 % dropout [Hinton et al., 2012b] prior to taking the ArgMax *per label* in order to to allow each of the quintupled label neurons to differentiate itself. For inference (and validation) we then replaced dropout and ArgMax with an average pooling over all label neurons of the same class. AvgPool is both faster to compute in PPU and more robust to variations than ArgMax. Same as before, classification is determined by which label has the larger score.

Increasing Training Data Since the efficient variant only uses 13.5 s of input data, we increased available training data by drawing several slices from the original training data. In order to make the network robust to specific starting positions of input slices, we took overlapping slices every 73 samples. This effectively increases the amount of training data by an approximate factor of 21 to 325 500 samples. Of course, we reserved 500 full-length data traces for validation beforehand.

Loss Function For training, we use a conventional cross-entropy loss term

$$\begin{aligned} \mathcal{L}[\mathbf{y}^{(\text{label})},\, n^*] &= -\omega_{n^*}\, \ln\left[\frac{\exp\left(y_{n^*}^{(\text{label})}/\xi\right)}{\sum_n \exp\left(y_n^{(\text{label})}/\xi\right)}\right] \\ &= \omega_{n^*}\left\{-\frac{y_{n^*}^{(\text{label})}}{\xi} + \ln\left[\sum_n \exp\left(\frac{y_n^{(\text{label})}}{\xi}\right)\right]\right\} \end{aligned} \tag{13.1}$$

where n^* is the correct class, $\mathbf{y}^{(\text{label})}$ the vector of outputs of the label layer (i.e., sinus rhythm and AF), ω_n a class-specific weight factor and ξ is a scaling factor. By setting $\xi = 63$ we scale outputs to the range $[-1,\, 1]$ to prevent steep modes in the loss energy landscape, thereby regularizing the gradient. Because the margin for error on true positives is larger than for false positives, we prioritize the correct classification of AF by setting $\omega_{\text{AF}} = {}^3\!/_5$ and $\omega_{\text{sinus rhythm}} = {}^2\!/_5$. While we did experiment with other regularization terms, none were used for final training (i.e., the *efficient* variant).

Learning is then performed in a hardware-in-the-loop setting similar to [Schmitt et al., 2017]: The forward pass is calculated in hardware (or mock-mode, see below) while the reported activations are fed back into the optimizer to perform weight updates. As optimizer we use Adam[18] (cf. Section 2.1).

13.2.3 Mock-Mode

BrainScaleS-2 is not a commercial product (yet), but rather a scientific research platform still under ongoing development. At the start of the efforts described in this chapter, actual hardware prototypes were not yet readily available. Additionally, the first prototype iteration with HAGEN extensions, HICANN-X[19] v1, suffered from an unfortunate hardware

[18]ADAptive Moment estimation, [Kingma et al., 2014]
[19]Short Form of HICANN-DLS-SR-HX, [Schemmel et al., 2020]

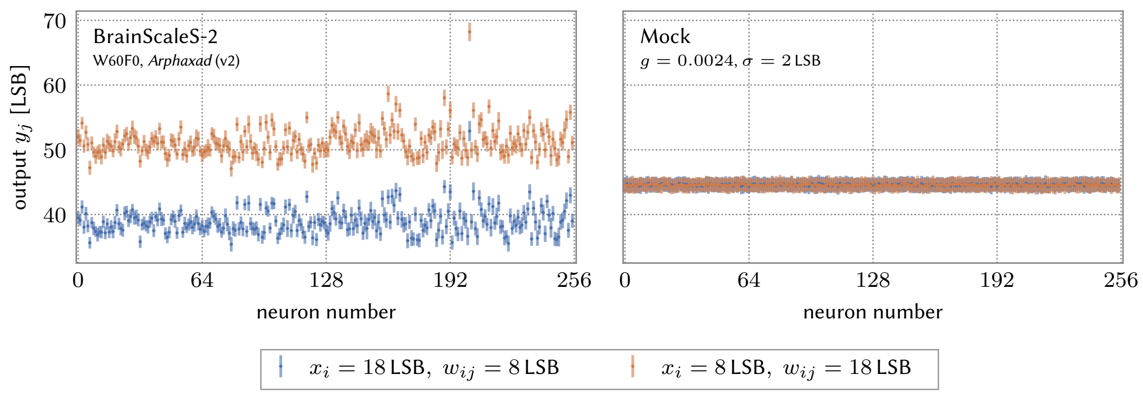


Figure 13.10: Comparison of statistical noise and fixed-pattern deviations between mock mode and execution on BrainScaleS-2. A uniformly filled vector (128 entries) and a uniform 128×128 matrix are multiplied 100 times, the mean output of each neuron and its standard deviation are shown. **Left:** Results of hardware runs with an input of 18 LSB and weight of 8 LSB (blue). Mean standard deviation of neurons is 1.89 LSB, fixed-pattern deviations between neurons are about 5.6 %. Swapping vector and weight values results in significantly higher output (orange). **Right:** the same calculation simulated with the mock mode. No fixed patterns are included, the mean value and the statistical noise are in good agreement with the results on BrainScaleS-2. Swapping the input values has no effect. Adapted from: [Emmel, 2020, Figure 4.6].

bug – detailed in [Weis, 2020] – that increased MAC runtime considerably. The equivalent of a 4.3 s `PyTorch`[20] operation took 2.1 h on HICANN-X v1 [Emmel, 2020]. Lastly, because of its prototypical nature, available resources in terms of HICANN-X v2 chip counts were and will always be dwarfed by conventional compute infrastructure for the foreseeable future. Besides `quiggeldy` (discussed in Chapter 10) as one attempt at mitigating this, a way was needed to prototype modeling ideas in a timely manner without hardware: the mock-mode. It follows the well established route of modelling hardware behavior in software.

The mock-mode is fully integrated into `hxtorch`.[21] It uses the same pre- and post-processing as experiments employed directly on hardware. Experiments can effectively switch between substrates with a single boolean flag. Instead of executing MACs in hardware, they are executed in `PyTorch` but then quantized and clipped to confirm to hardware constraints in HICANN-X. In particular, this means 5 bit vector entries, 6 bit weights and 8 bit (signed) output values. Weight matrices exceeding 128 rows (i.e., the maximum input height of the synapse array) are automatically partitioned into smaller computations and summed up at the end. This is similar to what is done on actual hardware [Spilger et al., 2020; Spilger, 2021]. All quantization operations are only done internally. Weights are stored as floating point “shadow”-copies [Hubara et al., 2017] to ensure compatibility with most `PyTorch` based optimizers.

Individual MAC operations are implemented for a given row i as

$$y_i = \sum_j x_j \cdot w_{ij} \cdot g_{\text{BSS-2}} + \kappa_i \quad \text{with} \quad \kappa_i \sim \mathcal{N}(0, \sigma) \tag{13.2}$$

[20]Python-Implementation of Lua-library `torch`, [Paszke et al., 2019]
[21]`PyTorch` for BrainScaleS-2, [Spilger et al., 2020]

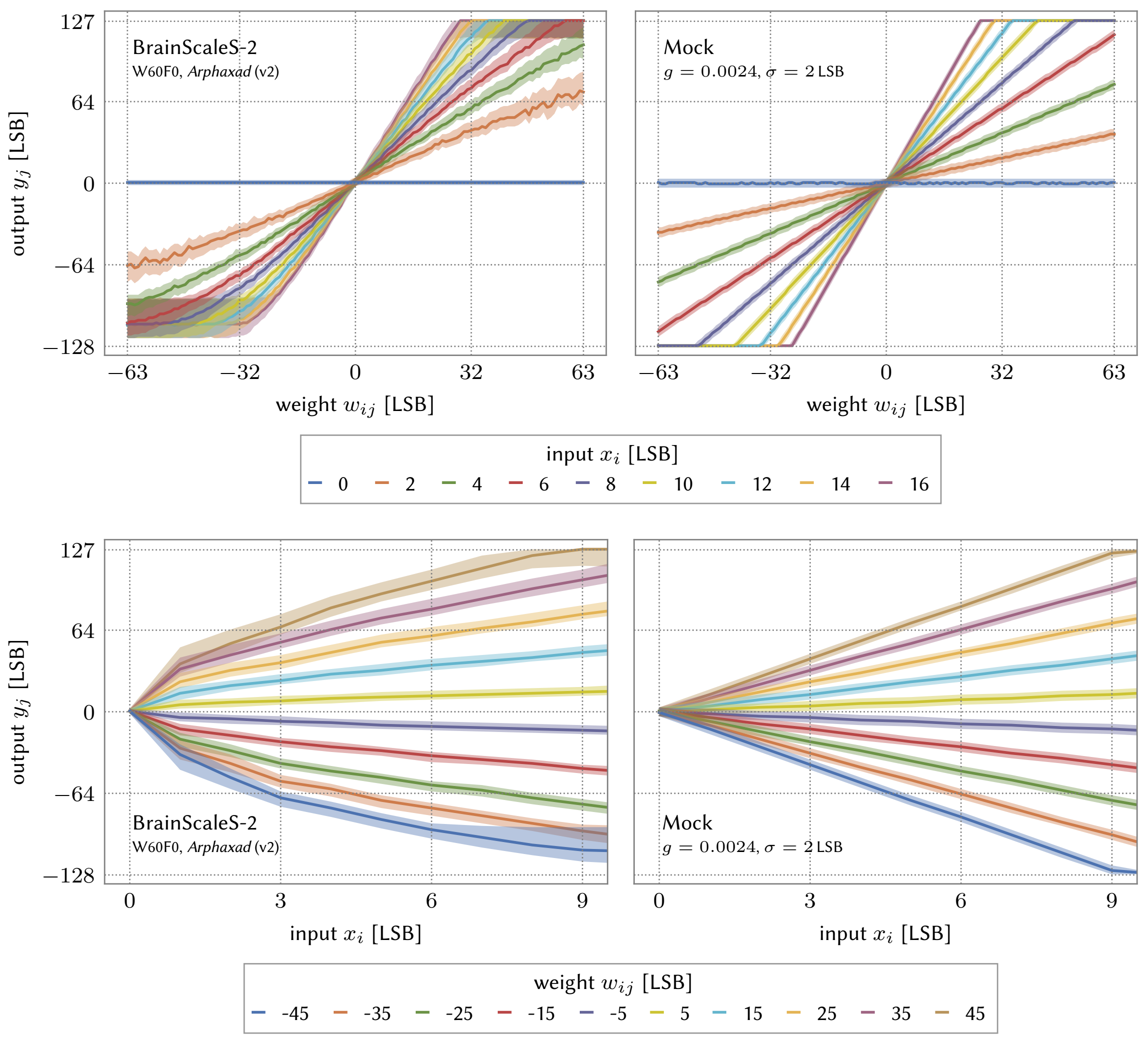


Figure 13.11: Comparison of the deviations in terms of linearity between mock mode and execution on BrainScaleS-2. A uniformly filled vector (128 entries) and a uniform 128 × 128 matrix are multiplied 100 times for each combination of x_j and w_{ij}. The median outputs y_i of all neurons are shown, the colored regions comprise 95 % of the corresponding neuron outputs.
Top: Within the available output range, there is usually a good linear relationship between weight and output value. Slightly more pronounced deviations can be observed for negative weights.
Bottom: Especially for small inputs the relation between input value x_j and output y_i clearly deviates from a linear approximation. This leads to significant differences to the results from the simulation in this value range (right). Adapted from: [Emmel, 2020, Figure 4.5].

where y_i is the final result, x_j an input vector entry, $g_{\text{BSS-2}}$ a gain-factor discussed below and κ_i intrinsic noise in each operation that we assume to be normally distributed with a given width. Both gain-factor $g_{\text{BSS-2}}$ and noise width σ are global parameters set at the beginning of experiment. When running on actual hardware, both parameters are estimated and printed as part of a sanity check during initialization so that corresponding mock-mode simulations can be adjusted.

While MACs operations on hardware are linear for specific weight ranges (cf. Figure 13.11 top left), its outputs are scaled by a gain-factor. It is influenced by several hardware settings, such as settling time until read-out or number of resends of the input vector to boost readout-gain [Weis, 2020]. Typically, hardware settings need to be tuned such that relevant small values are clearly distinguishable from noise while not saturating for large values unless tolerable.

The mock-mode is suitable for pre-training. When going from mock-mode to hardware the network experiences a drop in accuracy due to the unaccounted deviations discussed below. However, after training for a small number of epochs network accuracy recovers almost completely [Emmel, 2020].

Difference between Mock-Mode and BrainScaleS-2 Since the mock-mode was intended to be a fast initial stepping stone when developing models for hardware, we opted for a relatively simple linear model with stochastic noise. On real hardware, however, there are more distortions that need to be taken into account in future development iterations.

Despite calibration [Weis, 2020], neurons still display fixed-pattern noise. This is investigated in Figure 13.10 by multiplying a uniform vector and matrix. Ideally, all results should be the same. While repeated executions only vary with up to 2 LSB statistical error, we observe about 5–10 % fixed-pattern deviations. Fixed-pattern deviations are influenced by a lot of factors, including used chip, calibration settings and input values. Swapping numerical values of vector and weight matrix reveals another divergence: Due to its non-symmetric implementation (vector entries influence pulse length and weight values pulse height, cf. Section 3.2.2), we observe vector matrix MACs to be non-commutative.[22]

Next we investigate overall linearity by varying both vectors and weights in Figure 13.11 and plotting the median over all output rows. Here, we observe that varying weights is linear within the limits of the converting CADC.[23] Merely negative weights show minor deviations (cf. $w_{ij} = \pm 5$ in Figure 13.11, bottom left). Varying input values, however, shows clear deviations from linearity, especially for the smallest input values $< 3\,\mathrm{LSB}$. This non-linear effect is more pronounced for larger weight values.

Accounting for these differences would greatly increase predictive power of the mock-mode, potentially eliminating the need to retrain when moving to hardware. While we opted to fully train on hardware and thereby accounted for deviations during training in the following (cf. Section 13.3), it would be beneficial to incorporate non-linearities and other hardware-inherent properties when calculating the gradient in the backward pass during learning. Despite our results showing that learning is possible, future work should definitely be put into modeling hardware responses to small input values and their influences on gradients during learning.

Initialization We use the Kaiming initialization method [He et al., 2015], which is the default in `PyTorch`. Its key idea is to ensure a constant signal strength throughout the network by taking into account the variance and effective gain of all operations involved. Since BrainScaleS-2 only supports positive 5 bit input, we typically scale the 7 bit positive

[22] In other words: On BrainScaleS-2 we observe $20 \cdot 10 \neq 10 \cdot 20$ (in terms of vector matrix multiplication) due to non-linear effects (cf. Figure 13.10).

[23] Correlation ADC

output branch (post-ReLU) inversely by $g_{\text{scale}} = 4$ ($\simeq 2\,\text{bit}$) so that the total gain factor g_{tot} is

$$g_{\text{tot}} = \frac{g_{\text{scale}} \cdot g_{\varphi}}{g_{\text{BSS-2}}} \tag{13.3}$$

where $g_{\varphi} = \sqrt{2}$ is the gain factor of the ReLU operation. Hence, since we use ReLU activations, we initialize all weights in the l-th layer from a uniform distribution over the interval $[-r^{(l)}, r^{(l)}]$

$$w^{(l)} \sim \mathcal{U}\left(-r^{(l)}, r^{(l)}\right) \quad \text{with} \quad r^{(l)} = g_{\text{tot}} \cdot \sqrt{\frac{3}{n_{\text{in}}^{(l)}}} \tag{13.4}$$

where $n_{\text{in}}^{(l)}$ is the fan-in of the l-th layer, i.e., the number of input dimensions.

13.2.4 Application to BrainScaleS-2 Mobile

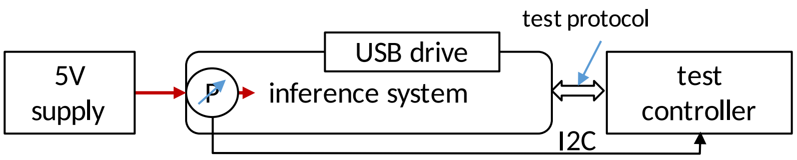


Figure 13.12: Evaluation setup for the BrainScaleS-2 mobile system: Power is supplied using a single 5 V supply, the power delivery of which is measured at the system's input. Test data can optionally be read from and results can be written back to a USB mass storage device. GPIO pins are available for orchestrating different phases of the experiment from an optional external controller. Taken from: [Stradmann et al., 2021, Figure 9].

Software Deployment As discussed in Section 3.2.3, BrainScaleS-2 Mobile is deployed as SoC[24] [Xilinx, 2019]. Besides an embedded FPGA, it also features an ARM64[25]-based quad-core CPU.[26] While this CPU is barely[27] used during standalone inference mode for the BMBF competition (described below), it provides opportunities for future applications to gather remote sensor data or perform various other forms of "conventional" computing during mobile experiment execution. The entire SoC could even serve as host compute node for the attached ASIC.

The SoC features a manufacturer-provided embedded Linux distribution called "Petalinux". Overall, it was required to cross-compile all needed software components to ARM64 architecture. To that end, we provide both a cross-compiler environment on the cluster as well as within Petalinux itself, based on the `visionary-dls-core` Spack[28] package (cf. Section 8.1.4). Since we track all software dependencies via Spack (cf. Section 8.1), this

[24]System on a Chip
[25]Advanced RISC Machines
[26]Central Processing Unit
[27]The ARM64-based embedded CPU is merely used to configure FPGA, PPUs and storage devices at startup.
[28]Supercomputing PACKage manager, [Gamblin et al., 2015]

was straightforward since Spack specs[29] feature explicit architecture annotations.[30]

Training and initial validation was conducted via `quiggeldy` (cf. Chapter 10). As already discussed in Section 10.7.1, we opted to use `quiggeldy` in order to effectively cut the dependency stack in half: The lower level C++[31] libraries up until `hxcomm`[32] are deployed on the SoC, while the upper layers remain in user workspaces off-system. In particular, we avoid transferring and executing Python[33] scripts. This reduces the number of software dependencies that need to be cross-compiled significantly, while providing a more efficient user-experience since all experiment code can be executed in each user's workspace. Only serialized low-level experiment configuration data is exchanged between cluster compute node and SoC. All data for Figures 13.14 to 13.16 was gathered this way.

Facilitating experiment step submission via `quiggeldy` has another benefit when when attaching BrainScaleS-2 Mobile systems to the Electronic Vision(s)[34] compute cluster (cf. Chapter 9). This eliminates the need for another compute node serving as sentinel as `quiggeldy` is able to facilitate multi-user access running directly on SoC. When submitting experiment steps asynchronously, we are able to eliminate Ethernet-induced delays entirely because `quiggeldy` is "directly" attached to the ASIC, i.e., able to feed data to the FPGA via DMA.[35]

Standalone Experiment Execution During final competition evaluation, we switch to standalone inference mode execution, developed in [Spilger, 2021]. The C++ parts of the BrainScaleS-2 software stack are combined into a single standalone binary that is executed directly on SoC. It implements fused `hxtorch`-operations via `grenade`[36]'s graph execution that have a fixed schedule (as opposed to JIT[37] execution on host computers). For obvious reasons, `quiggeldy` is not necessary in this scenario. In standalone mode, control is shifted from the FPGA – that executes a fixed sequence of instructions from its playback memory, filled from the host-computer – directly to both PPUs which then handle all control flow (cf. Figure 3.9). Via quintupled serial communication links to the FPGA, they are able to instruct it to perform data load and store operations via DMA, trigger operations for delivery of input activations from the FPGA to the analog cores, reading out CADCs or perform digital operations that cannot be carried out directly on the analog substrate (e.g., ReLU activations).

Evaluation Procedure During final evaluation, a USB[38] storage device containing previously unseen test data provided by external referees from BMBF is attached to the

[29]Specification of Package Configurations as used by SpackSpecifications of Package Configurations as used by Spack

[30]Of course, not all build processes were compatible with Petalinux out-of-the-box, but could be adapted within finite time in exchange for Dr. Eric Müller's sweat, blood and tears.

[31]C++ Programming Language, [ISO, 2017]

[32]Low-Level Communication With HICANN-X via Hostarq

[33]Python Programming Language, [Rossum, 2000]

[34]Electronic Vision(s) Group at the Kirchhoff-Institute for Physics in Heidelberg

[35]Direct Memory Access

[36]GRaph-based Experiment Notation And Data-flow Execution

[37]Just-In-Time

[38]Universal Serial Bus

system.[39] In order to ensure comparability, all contenders of the competition had to adhere to a common interface for experiment orchestration. It specified a total of four phases, including system initialization, transfer of input data from the USB mass storage device to internal DRAM,[40] actual inference and final back transfer of classification results to the attached storage device. Final measurements were taken for several blocks of 500 ECG traces each. The evaluation setup is depicted Figure 13.12. The included I^{2}C[41] chain is not part of this protocol and can optionally be used for fine-grained power measurements of the sensors (cf. Section 3.2.3). All official measurements for comparing the competition's contenders were conducted with off-system equipment supplied by the external power supply.[42]

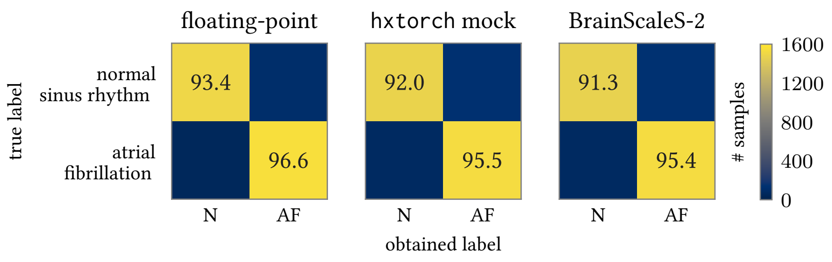


Figure 13.13: Confusion matrices for the *accurate* model variant presented in Figure 13.8. All results significantly exceed the requirements of the BMBF competition, which correspond to recall accuracies of at least 80 % for sinus rhythm and 90 % for AF. Adapted from: [Emmel, 2020, Figure 5.6].

13.3 Results

13.3.1 Accurate Variant: Classification Results

Up until the BrainScaleS-2 Mobile system was commissioned, models were trained on cube setups in the default cluster environment. Here, experiments are executed using the BrainScaleS-2 software stack compiled for `x86-64` and transferred directly to the FPGA via Ethernet (cf. Chapter 6). We train on 15 500 training ECG traces, i.e., all but 500 randomly chosen ECG traces (250 sinus rhythm and 250 AF). As mentioned above, for the accurate model variant we almost operate on the full trace at 102.4 s input length (cf. Figure 13.8).

Classification results for the *accurate* model variant are shown in Figure 13.13. Compared to full-floating point counterpart in `PyTorch`, we observe minor drop (1–2 %) in classification results when going to a mock-mode implementation. Interestingly, the hardware implementation performs only minimally worse than the intermediate mock-mode variant,

[39]As explained below, the system designated #1 in Figures 13.14 and 13.16 was used for final competition evaluation.

[40]Dynamic Random-Access Memory

[41]Inter-Integrated Circuit

[42]The experiment interface has been designed by and final evaluation measurements were taken at the German Research Centre for Artificial Intelligence Kaiserslautern under the direction of Prof. Dr. Hans Dieter Schotten.

despite the latter not capturing all intrinsic hardware distortions. Overall, we reach 95.4 % true positive and 8.7 % false positive rate on hardware with this initial model. The criteria of the BMBF competition are fulfilled.

quantity	**value**	**unit**
mean power consumption: system	5.6	W
mean power consumption: BrainScaleS-2 ASIC	0.7	W
time (500 records)	138	ms
correct classification atrial fibrillation	93.7 ± 0.7	%
wrong classification Sinus Rhythm	14.0 ± 1.0	%
total energy	0.78	J
energy FPGA base board	0.35	J
energy ARM CPU	0.17	J
energy FPGA	0.10	J
energy DRAM (upper limit)	56	mJ
total energy ASIC	96	mJ
energy ASIC IO	32	mJ
energy ASIC analog	31	mJ
energy ASIC digital	33	mJ
total operations in CNN	65.875	MOp
BrainScaleS-2 ASIC processing speed	477	MOp/s
BrainScaleS-2 ASIC energy efficiency	689	MOp/J

Table 13.1: Measured results for *efficient* model variant used for the classification of 500 randomly selected ECG traces that were excluded from training. Adapted from: [Stradmann et al., 2021, Table 1].

Estimating Power Consumption from Number of Operations We evaluate the number of operations needed to classify a single trace. Since the whole model fits into both halves of the synapse array, we do not need to reconfigure the chip during inference, saving a lot of overhead. For the first layer, we need 648 distinct MACs to compute the full output signal. Since we deploy 18 separate copies of the weight matrix we compute that many strides in parallel and therefore cut down the required MACs to merely 36. The remaining computations in conventional compute logic (ReLU and ArgMax) and the lower chip half (two MACs in total for the fully connected linear layers) can be performed in parallel to the next ECG trace being processed in the top half. Their runtime is therefore amortized. This means that, when classifying a whole dataset, on average we require the duration of about 36 MACs for a single trace. Since a single MAC takes about 5 µs [Stradmann et al., 2021], we ideally need 180 µs per trace. While the accurate model was not run on BrainScaleS-2 Mobile, we can still estimate a lower bound of the power consumption of the analog core to be 126 µJ per trace and 63 mJ for the whole validation dataset (cf. Table 13.1). However, given the discrepancy between "ideal" and actual runtime of more than a factor of 55 for the efficient model (as will be explained below), we can correct our estimate to be almost 7 mJ per trace and up to 3.5 J for the whole validation set.

13.3.2 Applicability to other Datasets

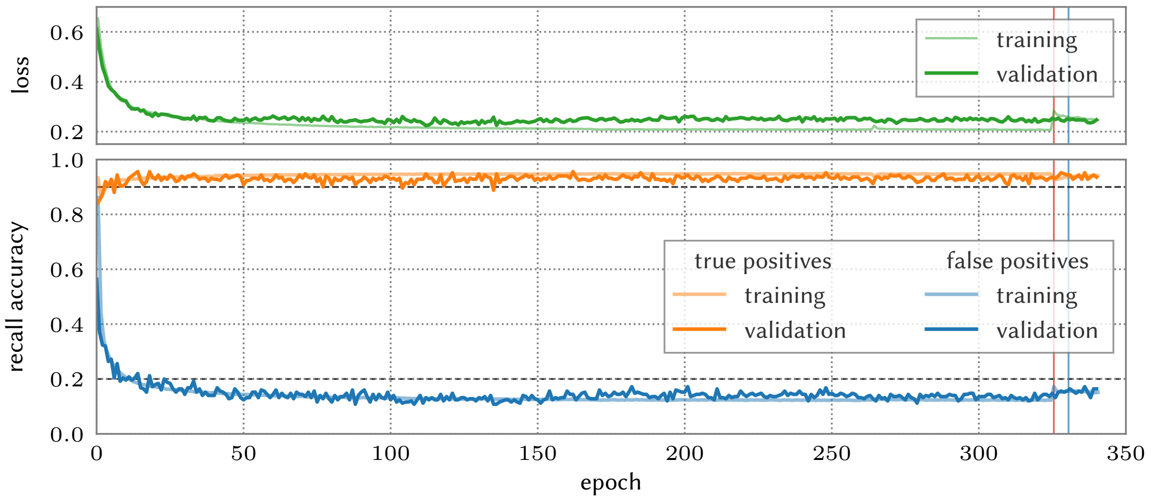
An intermediate model iteration between both models presented here was applied to different datasets, which it was not trained on, namely the PhysioNet Challenge 2017 [Clifford et al., 2017]. The dataset is filtered for recordings that show either sinus rhythm or AF and are at least 12 s in duration to be applicable. Using trigonometric interpolation, all recordings are aligned to a sampling rate of 512 Hz expected by the preprocessing chain. The dataset is not altered further and, in particular, was *not used during training*. Despite this, the model still detects AF with 92.1 % true positive rate while misclassifying 16.8 % of sinus rhythm as AF. These results are below that model's performance on the BMBF dataset (93.6 % true positive and 14.2 % false positive rate, respectively), but still showcase that the model has learned some generally applicable features to detect AF. For further details, see [Emmel, 2020].

13.3.3 Efficient Variant: Classification Results

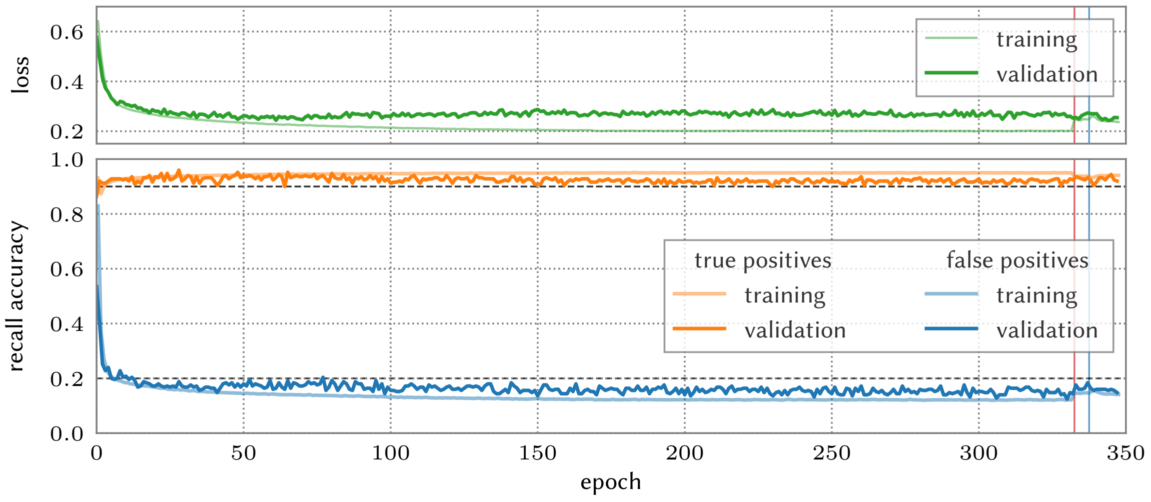
For final competition evaluation, we train the efficient model variant completely from scratch on BrainScaleS-2 Mobile via `quiggeldy`. Using 325 500 training traces (generated via aforementioned slicing techniques, cf. Section 13.2.2), we sweep several hyperparameters to find a good balance between gain factor and readout noise. Final evaluation is performed on three systems. The training process visualized in Figure 13.14. System #1, which was used for final evaluation, has 5 V supply voltage (cf. Figure 13.14a). We investigate the potential effect of different supply voltages with System #2 that was fitted with 12 V supply voltage (cf. Figure 13.14b). Finally, we have system #3 also sporting 5 V supply voltage. After being the first system to successfully complete the standalone evaluation routine within competition criteria under preliminary temperature variations (cf. Figures 13.14c and 13.15), system #3 was kept safe as a fallback and did not participate in further testing.

Each system was trained on its own from scratch. As we can see, all systems manage to fulfill the competition criteria within 10 epochs. However, continuously evaluating early snaphots for longer durations revealed that competition criteria could be violated in rare occasions due to hardware variations (<1 % of all validations, not shown). This can also be seen in Figure 13.14a where the true positive rate sometimes dips below 90 %. We continue training, which does stabilize classification rates at the cost of overfitting that is indicated by the difference in loss observed between training and validation. Therefore, we lower the neuron capacitance, boosting sensitivity to smaller MAC outputs (cf. red vertical line in Figure 13.14), and continue training for only a few epochs. We see that this does indeed close the gap between training and validation loss, all the while increasing true positive recall accuracy at the cost of a slight increase in false positives. In total, this distributes safety margins for both rates more equal.

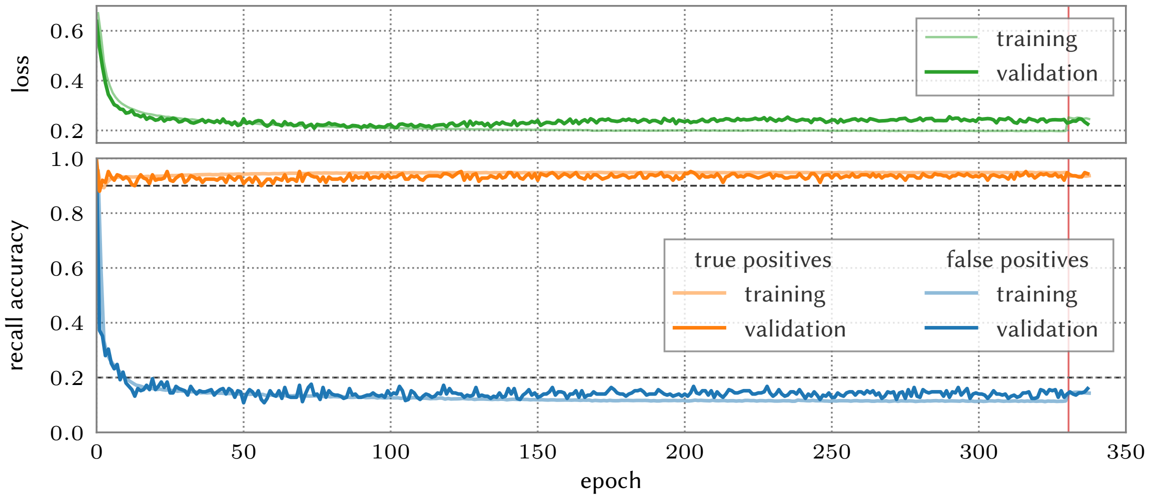
In order to evaluate the effect of 5 V versus 12 V supply voltage, we switch the analog BrainScaleS-2 cores of system #1 and #2 (cf. blue vertical line in Figure 13.14). Throughout this section, we refer to both systems by the chip they carried at the *end of training*. As we



(a) Training progress system #1.



(b) Training progress system #2.



(c) Training progress system #3.

Figure 13.14: Training progress of final model on three BrainScaleS-2 Mobile systems. System #1 was used during final competition evaluation, system #2 is a separate board with 12 V supply voltage and system #3 is identical to system #1. System #1 and #3 have a supply voltage of 5 V. The learning rate was decreased in steps. After successful training, system #3 was kept as a fallback solution. All systems were trained at room temperature, i.e., 18–20 °C. Red and blue vertical lines refer to changes in systems discussed in text.

can see, this has no discernible effect on classification rates. In particular, we emphasize that we did *not need to recalibrate*: By simply loading the corresponding calibrations, generated while still on the other baseboard via `calix`[43] (cf. Chapter 6) at different supply voltages, we did not witness a difference in classification results. This is especially obvious in direct comparison to the change in neuron capacitance immediately prior. We also note that it was possible to perform training on a cube-based setup and maintain classification results when transferring the BrainScaleS-2 ASIC carrier board from cube to mobile setup without further training (not shown). This suggests that there is a negligible effect of baseboard onto the BrainScaleS-2 ASIC as long as supply voltages are stable. We then stop training and evaluate temperature stability, discussed further below.

13.3.4 Power Consumption

As discussed above, the efficient variant is able to process one input trace in a single MAC operation. Because all layers (the convolutional layer in the top half of the chip, plus both fully connected linear layers in the bottom half, cf. Figure 13.9) compute in parallel, we can pipeline the execution of sequential traces as we did for the accurate variant. This reduces the theoretical inference time per trace by a factor of 36, compared to the accurate variant, to 5 µs per trace or 2.5 ms total as a lower bound for the whole validation set.

For actual evaluation, all models are “traced”, i.e., serialized into a fixed binary format that is then loaded by the standalone experiment executor described above. Where possible, operations are fused to eliminate unnecessary conversions [Spilger, 2021]. This also allows to easily gather statistics, for example count the number of MAC operations. We can monitor supply currents of individual components via shunt-based power monitoring ICs[44] [Texas Instruments, 2020]. Here, we optimized the readout process to allow for maximum sampling frequency to allow for accurate energy consumption calculation by integrating consumed power over time.

All measured results when executing are listed in Table 13.1. Overall, we reach $(93.7 \pm 0.7)\,\%$ recall accuracy with a false positive rate of $(14 \pm 1)\,\%$ with the efficient variant. Excluding the other three phases, classifying all 500 samples took 138 ms, i.e., significantly longer than our crude 2.5 ms lower bound by a factor of 55.2. Duration was equally distributed over all involved operations – i.e., control flow, digital computations and data transfer – hinting at the fact that further optimization of the software pipeline could be performed to close the gap towards the theoretical limit somewhat.

For example, we could perform some ReLUs directly in the analog core. The CADCs in each quadrant can be configured independently. From a technical standpoint, it would therefore be possible to adjust both CADCs in the upper quadrants to automatically perform ReLU operations by reading with a strong bias. Insufficient charge on the membrane would then cause CADCs to read a value of zero, eliminating the need for an additional digital operation in FPGA or PPUs. However, this would prevent *all* outputs in the given quadrant

[43]CALIbration Framework for HICANN-X, [Weis, 2020]
[44]Integrated Circuits

from accumulating negative outputs that might be needed as intermediary results if larger MAC are performed in a distributed manner. Furthermore, timing between a CADC result becoming available and PPU reading it back and issuing the next MAC is currently executed in a fixed timing schedule. Here, further investigation could improve timing and therefore throughput. Future chip generations could even provide a feedback line from CADC to PPU so that an interrupt could be triggered once readout completes.

Still, we measure a more or less constant power consumption of 5.6 W during inference for the whole system. Here, only 0.7 W (12.5 % of the total power) are consumed by the BrainScaleS-2 ASIC. Despite not being used during inference and with most power saving features enabled, the ARM64-based CPU still consumed more than 1.7× the energy of the BrainScaleS-2 ASIC. This poses another area of optimization to reduce the energy footprint the system as a whole. Classifying the whole validation dataset with 500 traces took 780 mJ with 96 mJ consumed by the BrainScaleS-2 ASIC. This gives a cost of 192 µJ per trace.

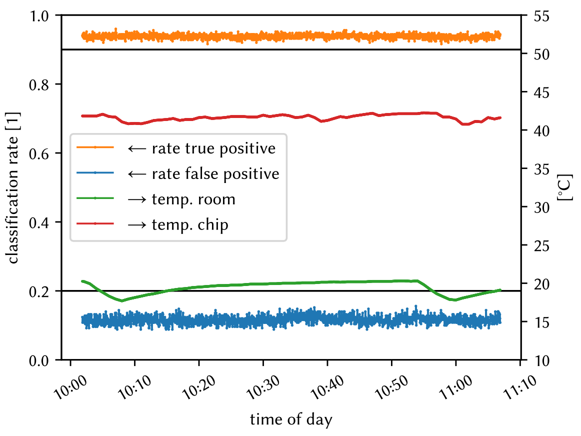
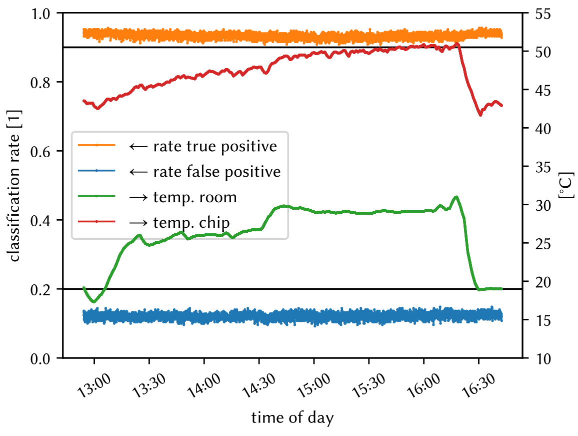


Figure 13.15: Prior to storing system #3 as fallback, we validated its temperature stability by performing constant variations over the course of one hour. Room temperature was not actively controlled but followed regular hysteresis patterns of installed air conditioning. As we can see, system #3 stayed within competition criteria. True positive recall accuracy: (93.8 ± 0.6) %, false positive classification rate: (12.5 ± 0.8) %

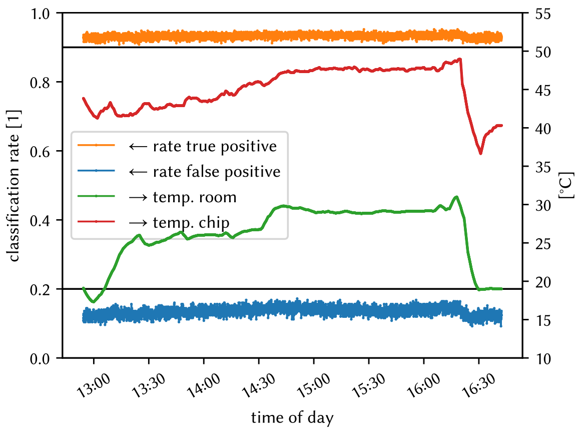
13.3.5 Temperature Stability

Temperature variations afflict the characteristics of all electronic circuitry. For digital hardware, most of these variations can be detected comparatively easy: Either all signals stay within their pre-defined ranges that map them to the digital domain, or not. Typically, it is a hard pass/fail situation. If violated, external parameters, such as supply voltages, can be tweaked to restore functionality. With analog hardware, the case is more complicated. Here, even slight alterations influence analog values which can accumulate to larger effects – it is a continuous spectrum. Even at the same temperature, analog results are slightly fluctuating in terms of performance despite always classifying the same validation data.

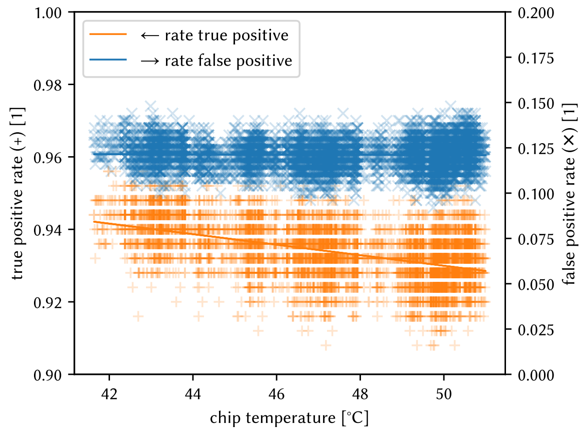
In order to ensure stability of classification results over longer time periods, we ran validation with the unseen portion of data (500 traces set aside earlier, the same we fed the standalone runner) continuously in the loop. For this, we use the same deployment as for training, i.e., not the standalone runner. Overall, we spend (1.937 ± 0.072) s per validation run, i.e., a factor of 14 times slower. The difference in execution speed is due to the



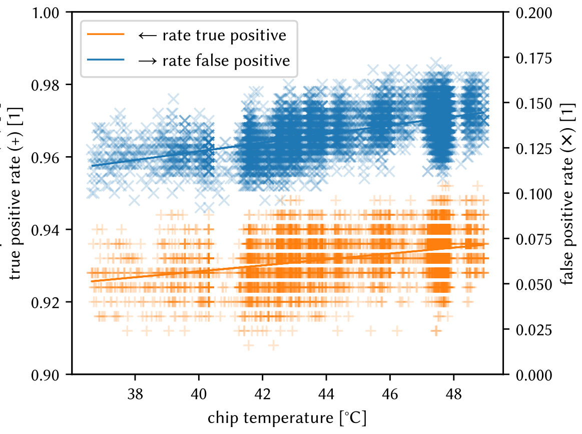
(a) Temperature stability of setup #1.
Overall true positive rate: (93.4 ± 0.8) %
Overall false positive rate: (12.1 ± 0.8) %



(b) Temperature stability of setup #2.
Overall true positive rate: (93.3 ± 0.6) %
Overall false positive rate: (13.5 ± 1.2) %



(c) Inference results at given chip temperature for setup #1.
Slope true positive rate: (−0.1452 ± 0.0030) %/°C
Slope false positive rate: (−0.0055 ± 0.0036) %/°C



(d) Inference results at given chip temperature for setup #2.
Slope true positive rate: (0.0818 ± 0.0025) %/°C
Slope false positive rate: (0.2393 ± 0.0041) %/°C

Figure 13.16: Investigating temperature stability of BrainScaleS-2 Mobile.
Over the course of almost four hours, systems #1/#2 (**left**/**right**) were set up to continuously perform validation on unseen training data while varying room temperature (and thereby chip temperature) by adjusting air conditioning in the lab. Both systems were trained at room temperature, i.e., 18–20 °C. System #1 was used during final competition evaluation, system #2 is a separate board with 12 V supply voltage (system #1 has a supply voltage of 5 V).
(a/b): Inference results over time while varying room and chip temperature. The competition criteria of more than 90 % true and less than 20 % false positive detection rate are marked as solid lines. During the inference run, we gradually increased room temperature by adjusting the air conditioning system, while performing rapid cooling at the end. While we do see slight changes in the classification rates over the whole experiment run, both systems do not violate the competition criteria.
(c/d): To investigate the influence of chip temperature on inference further, we plot classification rates against chip temperature for the whole experiment run. Besides trial-to-trial variability, we observe a slight temperature dependency of classification results via linear fit. We also observe a clear discretization into single data points affecting classification rates due to the small validation set of 250+250 samples. Interestingly, while for system #1 *both* rates decrease with increased temperature, for system #2 we observe an increase. Overall, temperature variations are within tolerable limits.

controlling Python process, transfer of intermediate results back and forth between chip and host, non-fused, i.e., non-pipelined, sequence of operations and writing intermediate results back to disk. This is yet another example showcasing how important streamlined software implementations are when aiming for final high-performance execution. Future work will go into making fused operations the default execution mode for most applications, while still returning intermediate results back to the host to be used in the backward pass for learning. However, in this case we are only interested in stability over time. For this task, the experiment execution rate is sufficient. All setups were analyzed with their final state from Figure 13.14.

For system #3, its continuous inference results over the course of one hour are shown in Figure 13.15. We measure both room temperature at air conditioning blowout as well as the temperature of the BrainScaleS-2 ASIC itself via external temperature sensors. We see that the hysteresis of the air conditioning control system imposes slight temperature variations over the course of one hour. Despite this, the trained model manages to stay within competition criteria over the full duration. System #3 is then excluded from further testing to serve as fallback.

We investigate temperature dependency further by taking active control of target temperatures for the air conditioning. This time, we evaluate system #1 and #2 to see if a difference in supply voltage has any noticeable effect. As shown Figure 13.16, we slowly but steadily increase the air conditioning's target temperature over the course of 1.5 h. After reaching a room temperature of about 28 °C, we keep it fixed for another 1.5 h to wait for the chip temperature to catch up. We see that the chip temperature of system #1 reaches and even crosses 50 °C, whereas system #2 plateaus just below 48 °C. Finally, we set the air conditioning to 30 °C (its maximum value) and observe a maximum chip temperature of 51.0 °C for system #1 and 48.9 °C for system #2. We conclude the test by rapidly cooling down the room, simulating rapid cooling by ventilation through an open window[45] during winter. Temperature tests performed prior to switching BrainScaleS-2 ASICs of system #1 and #2 showed the reversed temperature difference of 2–3 K (not shown here). This suggests that the ASIC in system #1 is intrinsically warmer and the difference not caused by the base board or its placement on the lab table.

We immediately notice that both systems stay within competition criteria throughout the whole competition. From Figures 13.16a and 13.16b we see a slight correlation between classification results and temperature. This is investigated further in Figures 13.16c and 13.16d by plotting both true and false positive rate against chip temperature and performing a linear fit. For system #1 we notice a slight decrease in true positive rate and an even smaller decrease in false positive rate for higher temperatures, as one would expect since training happened at room temperature. Interestingly, for system #2 we see an increase in *both* true and false positive rate at higher temperatures.

Overall, we see that the BrainScaleS-2 ASIC is both stable across large temperature ranges and shows no significant dependence on supply voltage. However, since both chips differ in their slight temperature dependence, further investigation is warranted.

[45] A common practice during the ongoing pandemic.

13.3.6 Back-of-the-Envelope: Standalone Pre-Trigger

Judging from measurements above, we estimate the applicability of a neuromorphic pre-trigger. In this scenario, a standalone BrainScaleS-2 ASIC would wake up periodically, evaluate an ECG trace of 13.5 s and trigger a longer recording that is stored only if AF is detected. Between measurements, the chip is completely powered off and does not consume any power. The controlling FPGA's bitfile would then contain a static `pbmem`[46] that configures the synapse array and hands control over to the PPU. Initializing the digital part and communication infrastructure of the ASIC takes less than 1 ms.[47] By far the longest time period is waiting for the CapMem[48] voltages to settle after setting parameters, which takes 20 ms [Weis, 2020]. All other initialization routines can be performed concurrently to that: Configuring the synapse array can be performed in 1.4 ms [Weis, 2020]. Performing FIR-based preprocessing in the FPGA for a single trace is only limited by DRAM latency and memory bandwidth. Assuming $125\,\mathrm{MHz} \cdot 64\,\mathrm{bit} = 1\,{}^{\mathrm{GB}}\!/_{\mathrm{s}}$ bandwidth preprocessing the data takes 0.222 ms, i.e., negligible. Given the measured values from Table 13.1, classifying a single trace for the competition takes 0.276 ms on average. Since we cannot pipeline all three layers, we take a conservative estimate for the inference of a single trace to be 1 ms until we reach a conclusion about whether to power off the device or continue recording. Adding another millisecond as general overhead we assume a runtime of 22 ms for the complete operation.

We measured the total power consumption of the ASIC to be 0.7 W. A lattice crosslink would suffice in an optimized interface between sensor and chip, expected to consume about 100 mW during operation with an implementation reachable within 1–2 years.[49]

With 0.8 W power consumption for the complete ASIC, we spend 17.6 mJ per inference. A regular CR2032 lithium button cell with 3 V nominal voltage and 230 mA h capacitance would therefore provide enough energy to perform more than 141 000 inferences. Assuming one estimate every two minutes, we would be able to perform inferences for more than 196 days straight.

Ultra-low-power ECG recording devices consume as little as 61.1 µW when taking data recordings, whereas the bulk of energy is spent when performing I/O,[50] e.g., via BT[51] at 1.85 mW [Altini et al., 2011]. Here, recording 13.5 s of data amounts to 0.825 mJ, hardly increasing our energy budget, while streaming the data to a remote location directly takes 25 mJ. This means that one recording plus inference consumes as much energy as streaming out data for 10 s that still needs to be analyzed for significance. Again, we want to emphasize that this the energy consumption for a current prototype chip that has not yet been particularly optimized for low-power operation. In the future, such an optimized version of the BrainScaleS-2 ASIC would be included as an embedded co-processor. Nevertheless, even with the current prototype generation, it is more efficient

[46] PlayBack MEMory program
[47] Personal correspondence with Dr. Vitali Karasenko.
[48] Capacitive Memory
[49] Personal correspondence with Joscha Ilmberger.
[50] Input/Output
[51] Bluetooth

to perform inference on the embedded ASIC than to stream it out. This means that, as soon as we wait more than 10 s between inferences when trying to identify data that is interesting to store, we save energy. In case of the aforementioned two minutes in between inferences, we save more than a factor of 12 in energy compared to streaming out the full data. By increasing the delay between consecutive measurements, energy-savings are increased even further.

Discussion & Conclusion 14

As we discuss in Part I, there is an emerging need for new compute paradigms, in particular those facilitating machine learning tasks. But engaging in such a new computing paradigm, let alone building custom analog neuromorphic hardware from the ground up is a Herculean effort in terms of technical work required. Communicating and controlling custom hardware, especially a neuromorphic substrate running in an accelerated time frame, requires a sophisticated software solution that can be maintained and extended over several generations of scientists (cf. Chapter 6). Unfortunately, scientists are not the first group of people that come to mind when thinking about software quality – and rightfully so: Scientists want to probe ideas and generate knowledge. For most, writing software is a means to an end, just one of many tools in their belt.

To tackle this problem, in Part II we introduce several methods to *facilitate collaborative software development in a scientific environment* employed at Electronic Vision(s).[1] Collaborators of varying skill levels need to be brought up to speed quickly and should not need to jump through too many technical hoops to be productive. On the other hand, contributions should be vetted and provided to others without introducing too much technological friction in terms of hours spent on solving avoidable problems. The long-term goal is for experiment code to be as declarative as possible: Allow experimenters to describe *what* their algorithm or model is about, *not how* it is able to achieve that on a specific neuromorphic emulator.

To that end, we introduce and enhance already present best practices for real world large scale software development. Prior to being integrated into the code base, new changes are reviewed by other developers and automatically verified to work against existing software components (cf. Chapter 7). Nightly tests verify continuous functionality of commissioned hardware platforms, both via dedicated hardware tests and existing experiments. Here, we tightened the integration of all involved components, especially between our central build tool, `symwaf2ic`,[2] and code review.

Addressing the on-boarding issue for new collaborators, one of the *core novel components* of this thesis is a complete solution to track and manage an evolving set of complex software environments in **visionary containers**. For this we use Spack,[3] a relatively novel package manager devised for HPC[4] deployments, that we actively contributed to in order to fit our use-case (cf. Section 8.1). It enables precise tracking of software dependencies

[1]Electronic Vision(s) Group at the Kirchhoff-Institute for Physics in Heidelberg
[2]Electronic Vision(s)-specific fork of `waf`
[3]Supercomputing PACKage manager, [Gamblin et al., 2015]
[4]High-Performance Computing

for distinct and independent environments, such as different hardware platforms and software applications. In particular, it allows us to keep software stacks for older hardware generations at a *known stable* state while not prohibiting other environments from adopting the newest release of a given support library. All environments are embedded into a single container image, dubbed *visionary container*, accessible throughout the cluster and on the web (cf. Section 8.2). It allows users to have the same compute environment whether they work on a private machine, our own compute infrastructure or even remote clusters. *Setting up a work environment and keeping it up to date becomes trivial.* They are extendable to different compute architectures and embedded devices (cf. Chapter 13).

Visionary containers are updated in a rolling release scheme. During this thesis, great effort has been put into providing an easy-to-use automated container build framework, `yashchiki`,[5] that can be easily controlled by users via Gerrit[6] comments (cf. Section 8.3). Once built, every new "testing" container is verified against all available soft- and hardware tests, ensuring confidence in a change not to disrupt day-to-day operations. This is a vast improvement over pre-thesis workflows where there was one environment for *all* software related work and upgrades were slow, painful and all-or-nothing.

Additionally, visionary containers are essential to tackle the problem of reproducibility in science, since dependency structures are often intricate and hard to reproduce [Krafczyk et al., 2021]. By noting down commit states of the locally deployed software stack and visionary container used, we can ensure that a particular software state is executed in *precisely* the same environment. We provide a second safeguard by storing each container's "DNA", i.e., precise build information from which it can be recreated easily. Since it's size ranges in a few mega bytes, it can be easily provided with supplementary paper data. This represents a huge step toward of *reproducible software-aided science*: All involved software dependency components are explicitly tracked.[7] Furthermore, once a given experiment runs in CI,[8] it can be ensured to continue running long after its author has moved on to other endeavors.

We extend the concept of deploying software in visionary containers to our cluster scheduler software (cf. Chapter 9), allowing us to run different deployments in parallel to each other. Changes to our intricate hardware management and cluster set up (cf. Section 9.1 and Chapter 10) can therefore be tested and verified *in place*, without affecting day-to-day operations. To the code there are no differences between testing and production and, hence, no modifications are necessary when transitioning from one to the other.

Going one step further, this thesis introduces **`quiggeldy`**, a micro-scheduler operating at time scales below Slurm[9] (cf. Chapter 10). It is able to interleave experiment-steps submitted from different users to the same physical hardware setup. Via reinitialization (cf. Section 10.4), users are able to explicitly track the state their experiments depend on in

[5]from Russian, ящики, meaning boxes or "Schachtel" in German, [Vision(s), 2021]

[6]Gerrit Code Review, [Harris, 2020]

[7]Of course, for technical reasons this does not include the kernel version, but typical simulation code executes in user-space.

[8]Continuous Integration

[9]Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management, [Yoo et al., 2003]

hardware, leading to an overall better structure in experiment scripts. We measure worst-case overhead of a not particularly optimized serialization implementation in sequential single-user mode to be within single digit percentages. Because `quiggeldy` supports asynchronous experiment-step submission and handles I/O[10] concurrently to execution, we can effectively amortize delays by adapting other software layers – predominantly `grenade`[11] – to make use of this feature. Concurrent submission shows no discernible overhead.

Extensibility was core to the design process, so that both new functionality can be added to the existing implementation in straightforward manner and the concept of round robin micro-scheduling be applied to different tasks. In particular, this allows `quiggeldy` to be extended towards a full-fledged monitoring and control system in the future.

To users, `quiggeldy` aims to be as transparent as possible, only requiring a single different CLI[12] argument when submitting jobs. In the past, limited availability of prototype setups lead to problems such as one experimenter being unable to quickly try an idea or debug an experiment due to another long-running experiment blocking. These problems were “patched” by more or less fixed assignments of chips to experimenters and blocking long-running operations such as parameter-sweeps until night hours. `quiggeldy` represents a proper solution, paramount for overall experiment throughput, proper encapsulation as well as responsiveness of hardware resources even under workload.

In Chapter 11, we discuss several immediate outstanding technical challenges on how the BrainScaleS[13] compute platforms could be improved and extended. Implementing them closes the gap towards a fully streamlined neuromorphic platform, ultimately rivaling commercial compute clouds available today. Of course, we need to be aware of our limitations: The BrainScaleS compute platform is a research endeavor first and foremost. Hence, bulk workforce is limited and needs to be allocated wisely. The presented solutions for managing software development workflows are deployed and in operation at the Electronic Vision(s) cluster, facilitating collaborative science for, on average, more than forty concurrent users.

In Part III we reap the crops of our efforts in Part II: We showcase two novel learning strategies for the BrainScaleS platform, developed using the principles presented above.

First, we present the **Time-To-First-Spike** paradigm (cf. Chapter 12). Here, we train feed-forward networks of LIF[14] neurons with CuBa[15]-synapses to classify a given input as soon as possible, i.e., when the first label neuron spikes. Once trained, these networks can form decisions after only a small subset of hidden neurons were elicited a spike.

The core result we provide is a rigorous analysis of neuro-synaptic dynamics that allow

[10]Input/Output

[11]GRaph-based Experiment Notation And Data-flow Execution

[12]Command-Line Interface

[13]BrainScaleS Mixed-Signal Accelerated Neuromorphic Systems, [Schemmel et al., 2008; Schemmel et al., 2010; Schemmel et al., 2017; Schemmel et al., 2020]

[14]Leaky-Integrate-and-Fire

[15]Current-Based

us to formulate exact input-output relations between afferent spike-times and resulting output spikes for each neuron. From these, we can derive *exact* gradient update rules for two ratios of synaptic and membrane time constants. The framework is adaptable to any loss function that is differentiable with regard to output spike-times. Here, our choice encourages early classification. Training is achieved via standard but extended machine learning frameworks adapted to incorporate LIF-dynamics and update rules.

The TTFS framework is a prime candidate for application on accelerated analog neuromorphic substrates such as the BrainScaleS platform. By only requiring spike-times to perform weight updates, it is light on required communication bandwidth and read out observables for in-the-loop training.

In the data coding scheme of the same name, each real-valued input feature is translated to a corresponding input spike with a corresponding delay. Among the datasets used for evaluation is both MNIST[16] and a novel non-linearly separable Yin-Yang motive [Kriener et al., 2021] that are both easily convertible to the TTFS coding scheme. We compare software-simulations with applications to HICANN-X,[17] the latest prototype generation of BrainScaleS-2,[18] for both datasets. Furthermore, we show an extension to CoBa[19]-based synapses on BrainScaleS-1[20] with a simpler dataset. On BrainScaleS-2, the reduction in accuracy when moving to the analog, possibly variation-afflicted substrate is found to be marginal. HICANN-X manages to classify a given MNIST sample within 10 µs realtime after receiving the first input spike. Ensuring for relaxation between patterns, we are able to classify the complete subsampled MNIST test dataset in less than 1 s (480 ms of which actually spent on chip for inference) with (96.9 ± 0.1) % accuracy, consuming about 8.4 µJ per classification. Using methods from [Cramer et al., 2020], we expect to shorten time per classification from 48 µs to approximately 10 µs.

The next step for TTFS is to extend the approach towards harder problems such as ImageNet [Deng et al., 2009], one of the standard datasets in vision. By either time-multiplexing single layers or connecting several chips, larger networks could be realized on current HICANN-X prototypes. Using existing software infrastructure, this can be implemented completely transparent to the user.

Another avenue to pursue would be to implement on-chip learning in the PPU[21] of BrainScaleS-2. Here, weight update calculations in PPUs need to keep up with the accelerated analog circuitry. Early results indicate that simplified versions retain most of their discriminative power [Göltz et al., 2021]. Another approach would be to cache computationally more expensive function evaluations via LUTs,[22] stored directly within PPU's memory. Here, it will be interesting to investigate the trade-off between limited but

[16]MNIST Database, `http://yann.lecun.com/exdb/mnist/` (visited 2021-04-10), [LeCun et al., 1998]
[17]Short Form of HICANN-DLS-SR-HX, [Schemmel et al., 2020]
[18]BrainScaleS-2 Analog Neuromorphic Hardware System, [Schemmel et al., 2017; Schemmel et al., 2020]
[19]Conductance-Based
[20]BrainScaleS-1 Wafer-Scale Mixed-Signal Accelerated Neuromorphic System, [Schemmel et al., 2008; Schemmel et al., 2010]
[21]Plasticity Processing Unit
[22]Look-Up Tables

fast SRAM[23] directly in the PPU and slower but far larger DRAM[24] accessed via FPGA.[25]

Fundamentally, TTFS update rules only depend on pre- and post-synaptic spike-times as well as an external error signal for which there are potential biological propagation mechanisms [Sacramento et al., 2018; Payeur et al., 2020]. This means that TTFS framework presented here is *not limited* to feed-forward structures, allowing for applications to recurrent and multi-spike paradigms.

To summarize, we successfully demonstrate that our TTFS-based approach operates at a delicate balance point in terms of performance, speed, efficiency and robustness.

The second learning strategy presented in this thesis is not novel because of its methodology, but because of its implementation: Here, we perform **fast analog inference** by employing HICANN-X to classify medical ECG[26] traces into sinus rhythm and atrial fibrillation (AF) using Convolutional Neural Networks. The HAGEN[27] mode performs analog MACs[28] (cf. Section 3.2.2). Using the current HICANN-X prototype, built to evaluate essential feasibility of analog non-spiking computation without disturbing spiking operation, we are already able to perform a full 256×512 MAC every 5 µs, corresponding to 52 GOp/s. Further optimization, both tweaking parameters for the current iteration as well as design improvements for the next generation, could bring processing rates closer to the fundamental limits of the synapse array that is able to handle one event every 8 ns, theoretically allowing for up to 32.8 TOp/s at current prototype size.

The system participated in the BMBF[29] Pilotinnovationswettbewerb “Energieeffizientes KI-System” (*Energy-efficient AI system*). For this competition, systems were rated in terms of energy consumption while having to maintain certain classification criteria.[30]

In order to conserve energy, we developed a model fitting completely into available hardware real estate (cf. Section 13.2.2). Input traces were truncated as much as possible to reduce the number of required operations.

For the competition, a new type of setup was commissioned: BrainScaleS-2 Mobile.[31] It is truly standalone, featuring an embedded FPGA and ARM64[32]-based quad-core-microprocessor [Xilinx, 2019]. The embedded nature of the system with different instruction set posed a technical challenge. Here, the methods developed in Part II were *crucial*: First, by explicitly tracking all dependencies we could immediately identify which software needed to be cross-compiled. Furthermore, `quiggeldy` was necessary to allow for immediate training and inference controlled from a regular cluster compute node. We stress that this streamlined development immensely, allowing for development time to be

[23]Static Random-Access Memory
[24]Dynamic Random-Access Memory
[25]Field-Programmable Gate Array
[26]ElectroCardioGram
[27]Heidelberg AnaloG Evolvable neural Network
[28]Multiply-ACcumulate operations
[29]Federal Ministry of Education and Research (*Bundesministerium für Bildung und Forschung*)
[30]The pass-fail performance criterion was to detect AF with at least 90 % recall accuracy, while classifying at most 20 % of sinus rhythm wrongly as AF.
[31]BrainScaleS-2 Mobile Analog Neuromorphic Hardware System, [Stradmann et al., 2021]
[32]Advanced RISC Machines

spent more productive.

Final evaluation is then performed via a standalone executable, essentially running in the ASIC[33] only. Unfortunately, at the time of writing, no "official" results obtained by external referees were published. With our own equipment, we measure (93.7 ± 0.7) % recall accuracy at a false positive rate of (14.0 ± 1.0) %. Inference of the validation dataset took 138 ms at 5.6 W total system power consumption, of which a mere 0.7 W were consumed by the BrainScaleS-2 ASIC, less than 15 %. This translates to 0.78 J required to classify the full validation set, of which 96 mJ are consumed by the BrainScaleS-2 ASIC. All results are summarized in Table 13.1.

Prior to final evaluation by external referees, we verify temperature stability by performing continuous inference via `quiggeldy` while varying room temperature between 17–32 °C, corresponding to chip temperatures between 36–51 °C. Overall, the setup managed to stay within competition criteria over the full duration of almost 4 hours.

The next steps for fast analog inference are twofold: We need to improve our understanding of the chip's effective transfer-function in order to efficiently scale to large problems. While the mock mode (cf. Section 13.2.3) was proven to be useful in the development of models prior to the arrival of prototype chips, it clearly does not capture all qualitative aspects of the hardware's non-linearity (cf. Figure 13.11). Hence, the assumed simple linear gradient differs from the actual one. The difference is not catastrophic since we are able to meet the competition's stability criteria. However, a better understanding of hardware characteristics leads to *enhanced calibration*, thereby improving learning significantly, especially for larger and harder tasks. This includes higher order effects such as cross-talk between adjacent synaptic lines and locality effects depending on where in the synaptic array a calculation is performed. Understanding and calibrating for these kinds of fixed pattern noise must be part of this treatment, as it has been for all other applications to analog substrates. Especially for rolled out convolutions matrices (cf. Section 13.2.2) it is important that all instances perform the same logical operation. Once models become deeper, these differences in successive layers add up. During the competition, we investigated preliminary methods to train and compensate for these differences by attuning each weight in the convolutional matrix while learning identities. However, due to time constraints they were not included in the final model that already met all criteria. The long term goal is for calibration to achieve a true separation of model function and used analog substrate, akin to their digital counterparts. New insights could then influence the design of future chip revisions.

Typical model sizes for real world tasks such as video analysis or speech translation contain about $10^7 - 10^9$ parameters [Aharoni et al., 2019]. These are 2–4 orders of magnitude larger than a single HICANN-X chip, requiring partitioning and distributed computation across several chips. The presented software stack (cf. Chapter 6), in particular `hxtorch`[34] and `grenade` already has working support for partitioned execution [Spilger, 2021], but true multi-chip execution has not yet been demonstrated on a practical example. Here, `quiggeldy` is an essential building block to create a *cloud-like infrastructure* where models are scaled across multiple chips. Working in close tandem with a `grenade`-based

[33]Application-Specific Integrated Circuit

[34]`PyTorch` for BrainScaleS-2, [Spilger et al., 2020]

executor, it could relay information about its current experiment load and queue status, allowing for most efficient resource allocation in true `*aaS`[35] style.

Another interesting avenue of application is *edge computing* [Shi et al., 2016; Park et al., 2018; Dongarra et al., 2019; Chen et al., 2019]. Here, data processing as well as inference happen in direct proximity to data storage. This is relevant for I/O-bound tasks in data centers, where energy cost of data transfer are expected to exceed the energy cost for computation in the future. The same is true about embedded IoT[36] devices that could additionally face privacy concerns if data is not allowed to be transmitted off-site (e.g., detecting dangerous situations in a security camera feed). Power- and cost-efficiency are key to allow for data-local computation and massive parallel deployment. We have demonstrated the feasibility of our approach in this regard. The next step is a comparative benchmark on TinyML applications [Lin et al., 2020; Banbury et al., 2021], an emerging field of machine learning that focusses on resource-efficient and low-powered learning on embedded devices. Possible applications include a ultra-low-powered pre-trigger to identify relevant sensor data that is then forwarded up the processing chain (cf. Section 13.3.6).

Finally, we can combine both modes of operation presented in this thesis: spiking and non-spiking. In hybrid execution mode, part of a network evolves in accelerated realtime, exchanging spikes, while another part performs non-spiking MAC-like operations. Besides BrainScaleS-2, we know of only one other hardware platform supporting this mode of execution: Tianjic [Pei et al., 2019]. The challenge will be to combine the explicit passage of time with purely logical mathematical operations. For example, TTFS-trained early layers could act as fast indexing, quickly deciding which forms of more involved MAC-based computation need to be performed in deeper layers. Here, many creative new avenues are still left to be explored.

In summary, despite its early prototyping stage we have demonstrated the feasibility of analog inference on HICANN-X.

Closing remark

In this thesis we have shown that we can successfully conquer the Lernaean Hydra, that is development of a neuromorphic hardware platform alongside its control software stack, all the while supporting modelling from both internal and external researchers. By doing things “properly” – containerization, review, CI, etc. – we allow each person to tackle their respective hydra head, ensuring it not to grow back unexpectedly.

It is the author’s personal belief to have showcased that proper development, software or otherwise, does not happen out of thin air but through proper tooling where necessary. This is especially true when undertaking projects exceeding the involvement of any one single person. Ultimately, it is this tooling that increases scientific output because all methods described here lead to fewer work hours wasted on re-inventions of the wheel. In a sense, we facilitate learning by tooling.

[35]* as a Service

[36]Internet-of-Things

IV
Appendix

Contributions A

Developing a neuromorphic compute platform is a group effort too daunting for one person to accomplish alone. Hence, we briefly detail the contributions by the author of this thesis. An overview of all software contributions can be found in Appendix A.4.

A.1 In Thesis

Besides explicitly labelled *Contributions*-paragraphs, the author predominantly contributed to the topics described in this thesis in the following:

Chapter 7	Maintenance and partial setup of the development pipeline.
Section 8.1	Main architect and developer for extending Spack's functionality to fit into the visionary container workflow. Core maintainer of visionary Spack packages described in Section 8.1.4, alongside Dr. Eric Müller and Dr. Andreas Baumbach.
Section 8.2	Main architect and developer of visionary containers and their deployment structure.
Section 8.3	Main architect and developer of `yashchiki` and the visionary container build process controlled via Gerrit comments.
Chapter 9	Main architect, deployment and maintenance of the new containerized cluster architecture, except for `nmpm_custom_resource`.
Chapter 10	Conception, sole developer and responsible for deployment of `quiggeldy` and `hagen-daas`, as well as all prerequisite changes to the software stack to implement connection handles (cf., Section 10.5.2).
Chapter 12	Supervision of Julian Göltz during his Master's thesis on TTFS. Initial development of the software framework used for TTFS. Extensive software support to increase performance of TTFS framework, including a CUDA[1] implementation of Lambert W function for software simulations.
Chapter 13	Supervision of Arne Emmel during his Master's thesis on developing a model for ECG classification with CNNs on BrainScaleS-2. Deployment of `quiggeldy` to BrainScaleS-2 Mobile. Training, verification and tweaking of

[1]Compute Unified Device Architecture, [Nickolls et al., 2008]

final model implementation on BrainScaleS-2 Mobile via `quiggeldy` in cooperation with Arne Emmel. Performing temperature validation in cooperation with Joscha Ilmberger, detailed in Section 13.3.5.

A.2 Supervision

Several students were supervised over the course of this thesis. Not all work could be included in this manuscript. Nevertheless, they provide valuable contributions, both to the development of the BrainScaleS compute platform as well as science in general.

Neural Sampling with Linear Feedback Shift Registers [Großkinsky, 2016] investigates how LFSRs[2] and modified Gold-code generators can be used to generate stochastic background noise in future generations of neuromorphic hardware. These are of special importance in the context of Neural Sampling [Petrovici et al., 2016]. Marcel Großkinsky was supervised by the author.

Struktur schafft Robustheit: Eine Untersuchung hierarchischer neuronaler Netzwerke mit unpräzisen Komponenten [Schroeder, 2016] systematically investigates the influence of parameter variations (especially synaptic delays and refractory periods) on Neural Sampling [Petrovici et al., 2016], partially implementing networks on the Spikey[3] chip. Anna Schröder was partially supervised by the author, mainly providing software support by implementing neuron models with configurable stochastic parameters in NEST.[4]

Accelerated Classification in Hierarchical Neural Networks on Neuromorphic Hardware [Fischer, 2016] continues [Schroeder, 2016] by implementing a two-layer Boltzmann-machine on the Spikey chip in order to classify MNIST. Carola Fischer was supervised by the author.

Simulated Tempering in Spiking Neural Networks [Korcsak-Gorzo, 2017] investigates the mixing properties of Restricted Boltzmann machines when background noise is varied in frequencies, suggesting a functional role of macroscopic neural oscillations observed in cortex. This work is continued in [Korcsak-Gorzo et al., 2021]. Agnes Korcsak-Gorzo was partially supervised by the author, mainly providing software support by adapting configurable Poisson sources originally detailed in [Breitwieser, 2015] for inclusion in `sbs`,[5] a framework to easily perform Neural Sampling in.

Towards Spike–based Expectation Maximization in a Closed–Loop Setup on an Accelerated Neuromorphic Substrate [Schneider, 2018] implements the prerequisites for NSEM[6] by controlling the activity of neurons from host-computer via closed-loop

[2]Linear-Feedback Shift Registers
[3]Spikey chip, [Pfeil et al., 2013]
[4]NEural Simulation Tool, [Diesmann et al., 2002]
[5]Spike-Based Sampling – a library for fast Neural Sampling, [Breitwieser et al., 2020; Breitwieser, 2015]
[6]Neuromorphic Spike-Based Expectation Maximization, [Breitwieser, 2015]

homeostasis on BrainScaleS-1. Felix Schneider was jointly supervised with Christian Mauch.

Spike-based Expectation Maximization on the HICANN-DLSv2 Neuromorphic Chip [Spilger, 2018] implements NSEM on HICANN-DLS[7] while adding support for random number generation to `libnux`.[8] Philipp Spilger was supervised by the author.

Training Deep Networks with Time-to-First-Spike Coding on the BrainScaleS Wafer-Scale System [Göltz, 2019] derives update rules applicable to backpropagation on single spike-times of LIF neuron. It represents an early version of the work detailed in Chapter 12. Julian Göltz was supervised by the author.

Towards an Automated Platform to Implement Artificial Neural Network Topologies on Neuromorphic Hardware [Raman, 2019] investigates possibilities to automatically translate ANNs[9] to equivalent rate-coded SNNs[10] via software pipelines, allowing for a more automated approach to investigate the capabilities of neuromorphic hardware. Aruna Raman was jointly supervised with Dr. Eric Müller.

Inference with Convolutional Neural Networks on Analog Neuromorphic Hardware [Emmel, 2020] investigates classifying ECG traces via fast analog inference. It resembles part of the work detailed in Chapter 13. Arne Emmel was supervised by the author.

A.3 | Publications

The author contributed to several publications, not all of which could be included in this manuscript.

A.3.1 | Peer-reviewed

Mihai A. Petrovici, Anna Schroeder, **Oliver Breitwieser**, Andreas Grübl, Johannes Schemmel, Karlheinz Meier **Robustness from structure: Inference with hierarchical spiking networks on analog neuromorphic hardware**. *Proceedings of the 2017 IEEE International Joint Conference on Neural Networks*, 2017, `http://dx.doi.org/10.1109/IJCNN.2017.7966123`

Contribution: Software architecture and support for custom software models.

This study not discussed in this thesis.

[7]HICANN Dreieck Ludwighafen Süd: successor to HICANN chip and based on the technology test chip route65 which inspired the reference to BAB65, [Aamir et al., 2018; Friedmann et al., 2017]
[8]Library to Interface with PPU Codenamed Nux
[9]Artificial Neural Networks
[10]Spiking Neural Networks

Mihai A. Petrovici, Sebastian Schmitt, Johann Klähn, David Stöckel, Anna Schroeder, Guillaume Bellec, Johannes Bill, **Oliver Breitwieser**, Ilja Bytschok, Andreas Grübl, Maurice Güttler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Sebastian Jeltsch, Vitali Karasenko, Mitja Kleider, Christoph Koke, Alexander Kononov, Christian Mauch, Eric Müller, Paul Müller, Johannes Partzsch, Thomas Pfeil, Stefan Schiefer, Stefan Scholze, Anand Subramoney, Vasilis Thanasoulis, Bernhard Vogginger, Robert Legenstein, Wolfgang Maass, René Schüffny, Christian Mayr, Johannes Schemmel, Karlheinz Meier, **Pattern representation and recognition with accelerated analog neuromorphic systems**, *Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, `https://doi.org/10.1109/ISCAS.2017.8050530`

Contribution: Software architecture and support for custom software models.

This study not discussed in this thesis.

Luziwei Leng, Roman Martel, **Oliver Breitwieser**, Ilja Bytschok, Walter Senn, Johannes Schemmel, Karlheinz Meier, Mihai A. Petrovici, **Spiking neurons with short-term synaptic plasticity form superior generative networks**, *Scientific Reports 8, 10651 (2018)*, 2018, `https://doi.org/10.1038/s41598-018-28999-2`

Contribution: Software & modeling support via custom NEST models wrapped in `sbs`.

This study not discussed in this thesis.

Akos F. Kungl, Sebastian Schmitt, Johann Klähn, Paul Müller, Andreas Baumbach, Dominik Dold, Alexander Kugele, Eric Müller, Christoph Koke, Mitja Kleider, Christian Mauch, **Oliver Breitwieser**, Luziwei Leng, Nico Gürtler, Maurice Güttler, Dan Husmann, Kai Husmann, Andreas Hartel, Vitali Karasenko, Andreas Grübl, Johannes Schemmel, Karlheinz Meier and Mihai A. Petrovici, **Accelerated Physical Emulation of Bayesian Inference in Spiking Neural Networks**. *Frontiers in Neuroscience — Neuromorphic Engineering, 14 November 2019 Volume 13 pages 1201*, 2019, `https://doi.org/10.3389/fnins.2019.01201`

Contribution: Software architecture and experiment realization.

This study not discussed in this thesis.

Jakob Jordan, Mihai A. Petrovici, **Oliver Breitwieser**, Johannes Schemmel, Karlheinz Meier, Markus Diesmann, Tom Tetzlaff, **Deterministic networks for probabilistic computing**, *Scientific Reports 9, 18303 (2019)*, 2019,
`https://doi.org/10.1038/s41598-019-54137-7`

Contribution: Extending `sbs` to support the proposed framework and simulations, software simulation, data analysis.

This study not discussed in this thesis.

Dominik Dold, Ilja Bytschok, Akos F. Kungl, Andreas Baumbach, **Oliver Breitwieser**, Walter Senn, Johannes Schemmel, Karlheinz Meier and Mihai A. Petrovici, **Stochasticity from**

function Why the Bayesian brain may need no noise. *Neural Networks; November 2019, Volume 119, Pages 200-213*, 2019, `https://doi.org/10.1016/j.neunet.2019.08.002`

Contribution: Software architecture and support, predominantly via `sbs`.

This study not discussed in this thesis.

Sebastian Billaudelle, Yannik Stradmann, Korbinian Schreiber, Benjamin Cramer, Andreas Baumbach, Dominik Dold, Julian Göltz, Akos F. Kungl, Timo C. Wunderlich, Andreas Hartel, Eric Müller, **Oliver Breitwieser**, Christian Mauch, Mitja Kleider, Andreas Grübl, David Stöckel, Christian Pehle, Arthur Heimbrecht, Philipp Spilger, Gerd Kiene, Vitali Karasenko, Walter Senn, Mihai A. Petrovici, Johannes Schemmel, Karlheinz Meier, **Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate**, *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, `https://doi.org/10.1109/ISCAS45731.2020.9180741`

Contribution: BrainScaleS-2 software architecture and TTFS experiment discussed in Chapter 12.

Johannes Weis, Philipp Spilger, Sebastian Billaudelle, Yannik Stradmann, Arne Emmel, Eric Müller, **Oliver Breitwieser**, Andreas Grübl, Joscha Ilmberger, Vitali Karasenko, Mitja Kleider, Christian Mauch, Korbinian Schreiber, Johannes Schemmel, **Inference with Artificial Neural Networks on Analog Neuromorphic Hardware**. *Gama J. et al. (eds) IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning. ITEM 2020, IoT Streams 2020. Communications in Computer and Information Science, vol 1325. Springer, Cham.*, `https://doi.org/10.1007/978-3-030-66770-2_15`

Contribution: Software architecture and support.

This study not discussed in this thesis. It presents concepts of calibration of `calix`,[11] discussed in Chapter 6.

Philipp Spilger, Eric Müller, Arne Emmel, Aron Leibfried, Christian Mauch, Christian Pehle, Johannes Weis, **Oliver Breitwieser**, Sebastian Billaudelle, Sebastian Schmitt, Timo C. Wunderlich, Yannik Stradmann, Johannes Schemmel **hxtorch: PyTorch for BrainScaleS-2**. *Gama J. et al. (eds) IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning. ITEM 2020, IoT Streams 2020. Communications in Computer and Information Science, vol 1325. Springer, Cham.*, 2020, `https://doi.org/10.1007/978-3-030-66770-2_14`

Contribution: Software architecture and support.

This study are partially discussed in Chapters 6 and 10 as top level of the software stack.

[11]CALIbration Framework for HICANN-X, [Weis, 2020]

A.3.2 Preprints / Submitted for Review

Julian Göltz, Andreas Baumbach, Sebastian Billaudelle, **Oliver Breitwieser**, Dominik Dold, Laura Kriener, Akos F. Kungl, Walter Senn, Johannes Schemmel, Karlheinz Meier, Mihai A. Petrovici, **Fast and deep neuromorphic learning with time-to-first-spike coding**, *arXiv preprint*, 2019 (date of first preprint [Göltz et al., 2019]), `https://arxiv.org/abs/1912.11443`

Contribution: Extensive software support.

This study is discussed in detail in Chapter 12.

Eric Müller, Christian Mauch, Philipp Spilger, **Oliver Julien Breitwieser**, Johann Klähn, David Stöckel, Timo Wunderlich, Johannes Schemmel **Extending BrainScaleS OS for BrainScaleS-2**. 2020, `https://arxiv.org/abs/2003.13750`

Contribution: Software architecture (`quiggeldy`), example NSEM-experiment

The study is discussed in Chapters 6 and 10.

Yannik Stradmann, Sebastian Billaudelle, **Oliver Breitwieser**, Falk Leonard Ebert, Arne Emmel, Dan Husmann, Joscha Ilmberger, Eric Müller, Philipp Spilger, Johannes Weis, Johannes Schemmel, **Demonstrating Analog Inference on the BrainScaleS-2 Mobile System**. *arXiv preprint*, 2021, `https://arxiv.org/abs/2103.15960`

Contribution: Extensive software support, modelling and verification.

This study is discussed in detail in Chapter 13.

Agnes Korcsak-Gorzo, Michael G. Müller, Andreas Baumbach, Luziwei Leng, **Oliver Julien Breitwieser**, Sacha J. van Albada, Walter Senn, Karlheinz Meier, Robert Legenstein, Mihai A. Petrovici **Cortical oscillations implement a backbone for sampling-based computation in spiking neural networks** *arXiv preprint*, 2021, `https://arxiv.org/abs/2006.11099`

Contribution: Extensive software support.

This study not discussed in this thesis.

A.4 Software

Repository	# of Changes	# of Patchsets per Change		Insertions + Deletions per Change	
		Mean	Median	Mean	Median
`yashchiki`	273	8.5	4	30.5	7
`spack`	189	5.5	4	67.9	4
`config-slurm`	108	2.5	2	16.4	4
`lib-rcf`	96	11.3	5	9574.5	37
`visions-slurm`	95	4.3	2	393.6	10
`model-nmsampling-sbs`	68	8.0	3	165.2	34
`hxcomm`	57	19.3	7	140.6	26
`waf`	43	5.3	3	64.6	17
`haldls`	26	12.2	9	90.4	32
`model-hw-hdbioai`	24	12.0	13	51.8	22
`model-visionary-nest`	23	6.2	3	325.1	65
`service-ldap`	17	8.8	2	190.9	18
`hate`	11	3.0	2	60.8	66
`sctrltp`	10	4.1	1	47.3	50
`fisch`	9	15.9	5	279.9	38
`hxtorch`	6	6.3	6	22.2	22
`doc-visionshome`	6	2.3	2	17.5	6
`vision-bibtex`	5	4.6	4	77.4	7
`genpybind`	5	2.8	1	76.0	13
`calix`	5	6.6	4	33.2	30
`visionary-simlibs`	4	1.0	1	3.2	2
`meta-nmpm-software`	4	4.0	4	6.5	6
`hwdb`	3	1.3	1	6.7	1
`code-format`	3	4.0	3	8.3	2
`tools-slurm`	2	2.0	2	50.0	50
`nest`	2	5.0	5	1653.5	1653
`marocco`	2	4.0	4	12.0	12
`logger`	2	2.5	2	96.0	96
`jenlib`	2	10.5	10	89.0	89
`halco`	2	3.5	3	24.5	24
`euter`	2	1.5	1	8.0	8
`libnux`	1	16.0	16	38.0	38
`hmf-fpga`	1	1.0	1	28.0	28
(Omitted administrative commits to 18 *other repositories.)*					

The table above shows statistics of author's direct software contributions over the course of the thesis. Since listing all changes here would take up far too much space, we merely give Gerrit statistics, including number of changes per repository, number of patchsets (iterations) per change and number of insertions/deletions. Please note that these metrics are by no means useful to gauge code quality. Furthermore, these do *not* contain code review performed on other contributors' changes or deployment work.

Supplementary information B

B.1 Full List of quiggeldy CLI Arguments

`-a / --architecture <arch>`
Can be used to differentiate between different target architectures. Currently, there is only `vx` supported.

`-i / --connect-ip <ip>`
Can be used to explicitly specify which hardware resource this instance of `quiggeldy` should sentinel. If it is not specified it will be inferred from the environment as long as information is unique.

`--connect-port <port>`
In case of a co-simulation connection, we also need to provide the port to connect to.

`--connection-{arq,axi,sim}`
Select which type of connection the quiggeldy daemon should proxy. As shown in Section 10.5.3, due to its templated implementation, `quiggeldy` trivially support all connection types in `hxcomm`.[1] Currently there is support enabled for:

- connecting via Ethernet directly to the controlling FPGA that communicates with the HICANN-X via ARQ.[2]
- on the BrainScaleS-2 Mobile used for the BMBF Pilotinnovationswettbewerb "Energieeffizientes KI-System" (see Section 13.3), i.e., a Zynq Ultrascale+,[3] connecting from the Arm-based processor to the HICANN-X via AXI.[4]
- connecting via `flange`[5] to a SystemVeriolog[6]-based co-simulation.

`--delay-after-connect-ms <ms> / --max-num-connection-attempts <num>`
Adjust delay between and maximum number of connection attempts when establishing a connection to the hardware backend.

`--listen-ip`
Limit listening to a specific interface/address. By default, `quiggeldy` listens on all

[1]Low-Level Communication With HICANN-X via Hostarq
[2]Automatic Repeat reQuest protocol, [Philipp, 2008; Karasenko, 2020]
[3]`https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html` (visited on 2021-03-22)
[4]Advanced eXtensible Interface
[5]Linking C++ Software Stacks with SystemVeriolog using DPI
[6]SystemVerilog Programming Language, [IEEE, 2018]

available interfaces for incoming connections.

`-p/--listen-port <port>`
: Specify which port to listen on. This is especially important if several `quiggeldy` daemons run on the same host.

`-l/--logevel <num>`
: `quiggeldy`, like the rest of the BrainScaleS-2 software stack, supports several loglevels. They include in decreasing order of verbosity

0 ⇔ `TRACE`	display all messages
1 ⇔ `DEBUG`	display most important information useful for debugging or more severe
2 ⇔ `INFO`	display general runtime information or more severe
3 ⇔ `WARN`	only display warnings that might indicate failure or more severe
4 ⇔ `ERROR`	only display errors that cause `quiggeldy` to abort operation

`--logfile <file>`
: Write log to specified logfile. The logfile can then be parsed for usage information that is needed for reporting.

`--ignore-parent-death`
: By default, `quiggeldy` will terminate itself if its parent process dies. Setting `--ignore-parent-death` disables this behaviour.

`-c / --max-connections <num>`
: Set the maximum number of allowed connections. RCF[7] uses one file descriptor per established connection. In case a lot of simultaneous connections are open (for example when submitting many asynchronous requests), this can lead to a situation where `quiggeldy` is unable to accept new connections. To combat this, `quiggeldy` tries to increase its own `ulimit` for open file descriptors as long as its permitted and keeps track of how many connections are open. In case more than 95 % of available connections are in use, `quiggeldy` will start printing warnings to indicate a possible source of error. If possible, the limit on open file descriptors should be increased in this case.

`--mock-mode`
: `quiggeldy` features a mock-mode.[8] When enabled, `quiggeldy` accepts connections as usual, but does not connect to the "real" hardware backend. Instead, empty responses are returned. This is useful to troubleshoot connectivity issues without blocking any physical hardware resources. It is used in `hxcomm` software tests to ensure principle functionality.

`--no-allocate-license`
: In its default mode of operation, `quiggeldy` acquires a Slurm license prior to con-

[7]Remote Call Framework – a cross-platform interprocess communication framework for C++, [Delta V Software, 2020]

[8]Not to be confused with the mock-mode in `hxtorch` that emulates hardware behavior.

necting to the backend. With this switch specified, quiggeldy assumes the Slurm license has already been allocated and will not attempt to acquire or release it.

`-n / --num-threads-input <num>`
: The number of threads used in an RCF `ThreadPool` (see Figure 10.2) handles incoming experiment submission as well as on demand upload of reinit data.

`-m / --num-threads-output <num>`
: Likewise, the number of threads used by `OutputQueue` (see Figure 10.2) to deliver experiment results back to users can be set. Typically, it does not need be set as high as the number of input threads, but having more than one thread delivering output decreases chances of large result data – or a bad connection to a users node – prohibiting throughput.

`-r / --release <seconds>`
: Sets the number of seconds between seconds between releases of Slurm allocations. This allows other jobs that require exclusive hardware access to be scheduled. A value of zero causes `quiggeldy` to immediately release the Slurm allocation as soon as there no experiments pending execution.

`--slurm-license <license>`
: In case a Slurm is to be acquired, it needs to be specified.

`--slurm-partition <name>`
: The Slurm partition in which to allocate the given license can be customized.

`-t / --timeout <s>`
: Especially when combined with `hagen-daas`[9] (which is introduced in Section 10.9), `quiggeldy` daemons should not idly waste compute resources that are not needed. Hence, the number of seconds after which `quiggeldy` shuts itself down after being idle can be specified. If set to zero, `quiggeldy` will *not* terminate itself when idling. The timeout should be chosen larger than any conceivable update period for a user job. In case of shutdown, `hagen-daas` will reactivate `quiggeldy` if a new user job for the same hardware resource is scheduled. See Section 10.9 for details.

`-u / --user-period-ms <ms>`
: `quiggeldy` supports the concept of user periods. It is the minimal amount of milliseconds that a given user has access to the hardware before `quiggeldy` switches to another user, if there are any. By default, this feature is disabled, but it might be useful in future scenarios. One such scenario involves several users performing parameter sweeps for an experiment with a reinit that takes a relatively long time compared to executed experiment-steps. If experiment-steps are submitted asynchronously, `quiggeldy` can execute several experiment-steps for one user in rapid succession for one reinit prior to switching to another user. In this case, switching after executing a single experiment would lead to a decrease in performance. However, in the de-

[9]Howto Avoid Grabbing Emulators Nightlong – Dls As A Service

fault case we expect relatively short reinit durations compared to actual runtime of experiment-steps.

`-v / --version`
Due do to the "living at HEAD" paradigm employed at Electronic Vision(s), semantic versioning appears rather pointless. Instead, `quiggeldy` notes down the state of `hxcomm` and all dependency repositories at the time of compilation, i.e., which `git`[10]-commit `HEAD` points to and if the working area is "dirty" (i.e., modified). This information, as well as the compilation date, can be requested via this switch. As explained in Section 10.5.5, versioning information can also be obtained from remote running `quiggeldy` daemons via `viggeldy`.

B.2 | Parameter & Software States

B.2.1 | Time-To-First-Spike

B.2.1.1 | Parameters

Table B.1 contains neuron, network and training parameters used for results in Section 12.3.1.

Table B.2 contains network and training parameters for training on BrainScaleS-2 used for results in Section 12.3.2.

B.2.1.2 | Code

Table B.3 details the code state for simulations performed in Chapter 12.

Most of the repositories, described in Chapter 6, are available publicly at

`https://github.com/electronicvisions`

or upon request. Development states are tracked via Gerrit (cf. Section 7.2) at

`https://gerrit.bioai.eu`

that is only accessible for group members and selected collaborators.

Bitfile
`/ley/data/bitfiles/hxfpga/cube_ethernet/HXv2/stable/2021-03-08_1/bitfile.bit`

[10]Git – a distributed version-control system for tracking changes in source code during software development, see Section 4.4.1, [Torvalds et al., 2005]

Parameter name	**Yin-Yang**	**MNIST**
Neuron parameters		
g_ℓ	1.0	1.0
E_ℓ	0.0	0.0
ϑ	1.0	1.0
τ_m	1.0	1.0
τ_s	1.0	1.0
Network parameters		
size input	5	784
size hidden layer	120	350
size output layer	3	10
bias time[1]	$[0.9\tau_\mathrm{s}, 0.9\tau_\mathrm{s}]$	no bias
weight init mean[1]	$[1.5, 0.5]$	$[0.05, 0.15]$
weight init stdev[1]	$[0.8, 0.8]$	$[0.8, 0.8]$
t_early	0.15	0.15
t_late	2.0	2.0
Training parameters		
training epochs	300	150
batch size	150	80
optimizer	Adam	Adam
Adam parameter β	$(0.9, 0.999)$	$(0.9, 0.999)$
Adam parameter ϵ	10^{-8}	10^{-8}
learning rate	0.005	0.005
lr-scheduler	StepLR	StepLR
lr-scheduler step size	20	15
lr-scheduler γ	0.95	0.9
input noise σ	no noise	0.3
max ratio missing spikes[1]	$[0.3, 0.0]$	$[0.15, 0.05]$
max allowed Δw	0.2	0.2
weight bump value	0.0005	0.005
α	0.005	0.005
ξ [2]	0.2	0.2

[1] Parameter given layer wise [hidden layer, output layer].

[2] ξ implemented differently in code-base, i.e., as its own inverse.

Table B.1: Neuron, network and training parameters used to produce the results in Section 12.3.1. Adapted from: [Göltz et al., 2021, Table A].

Parameter name	**Yin-Yang**	**16×16 MNIST**
Network parameters		
size input	25	256
size hidden layer	120	246
size output layer	3	10
bias time[1]	$[0.9\tau_\mathrm{s}, \text{no bias}]$	no bias
weight init mean[1]	$[0.1, 0.075]$	$[0.01, 0.006]$
weight init stdev[1]	$[0.12, 0.15]$	$[0.03, 0.1]$
t_early	0.15	0.15
t_late	2.0	2.0[3]
Training parameters		
training epochs	400	50
batch size	40	50
optimizer	Adam	Adam
Adam parameter β	$(0.9, 0.999)$	$(0.9, 0.999)$
Adam parameter ϵ	10^{-8}	10^{-8}
learning rate	0.002	0.003
lr-scheduler	StepLR	StepLR
lr-scheduler step size	20	10
lr-scheduler γ	0.95	0.9
input noise σ	no noise	0.3
max ratio missing spikes[1]	$[0.3, 0.05]$	$[0.5, 0.5]$
max allowed Δw	0.2	0.2
weight bump value	0.0005	0.005
α	0.005	0.005
ξ [2]	0.2	0.2

[1] Parameter given layer wise [hidden layer, output layer].

[2] ξ implemented differently in code-base developed by the authors.

[3] After translating pixel values to spike-times, inputs spikes with $t_\mathrm{input} = t_\mathrm{late}$ were not sent into the network.

Table B.2: Network and training parameters for training on BrainScaleS-2 used to produce the results in Section 12.3.2. In contrast to Table B.1, the neuron parameters are not given here, as they are determined by the used chip. Adapted from: [Göltz et al., 2021, Table B].

Repository	Commit-Hash	Commit Message
calix	f7e9b72aae47929a5de27bd1897a042c14d10ad0	Add loglevel to default calib generator script
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	Change include order for C/C++ in clang-format config
fisch	2147c148af5f681453fac8e81649bc087739afcc	Use get_connection_from_env() for test execution
flange	fcde2aafe69805487789ca0b1a8a245caf5fb8ed	Support builds w/o generating Python bindings
halco	00ddc0f868b37be07bbf577f97168c4cef1205e6	Add SendingRepeaterOnWafer
haldls	9f34f9b1ea0e34dd1141198c61d2a0b25bfc8e92	Use get_connection_from_env() for test execution
hate	c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1	Enable ccache in CI
hmf-fpga	4c64b3f40501cb1eb2b01728a1b42a828b255102	Fix Jenkinsfile
hwdb	04c88357dab609aef7a8d36af58d10285d5cae51	Provide hwdb4cpp::database::get_yaml_entries
hxcomm	a01ba278fb4994463a9e539aaeeadb950f05256e	visit_connection: Add support for const variants
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	Update content description in Readme
lib-rcf	5b16326ae30ee08a322a6569887ca8bd2684c252	Fixes for log4cxx@0.11.0
libnux	32597fe35fa93b331de41da2adee127eb40e46e1	Add vector generator base address
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	Add log4cxx_level_v2
model-hx-strobe	6fdc7b05a3334ad1c2497e563a3c4c1c41feeceb	uncomment overwrite of clocks, and stuff (dirty)
pyhid	82c64b5569928d2ec8344b8dffbd930da23d0004	Fix cppcheck warnings
pywrap	83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb	Support builds w/o generating Python bindings
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	Update to gtest 2.0.0
sctrltp	b5f825007b842f44f3e6401f00cf93387e5e3f3c	Support builds w/o generating Python bindings
tools-kintex7	08395cc8429031281fd024e5cbf071d521f03259	Add option to bring FPGA in flashable state
ttfs_pytorch	7c0ab6ee073b35ebdb9c1f9ed37a705654ed9138	transition complete. YY good results
visions-slurm	5e7ea560235b068fc12f26e3f0d002d415f76cf9	Add hwdb yaml environment export
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	Add .gitreview

Table B.3: Repository state for TTFS simulations performed in Chapter 12.

Container Image
`/containers/stable/2021-02-19_1.img`
(available from `https://container.bioai.eu`)

App `visionary-dls`

In regard to required compute performance, all simulations were run on `Intel(R) Core(TM) i7-4771 CPU @ 3.50GHz` or comparable hardware. Furthermore, some software simulations for [Göltz et al., 2021] were performed using an `Nvidia Tesla P100`. By scaling batchsizes, executing experiments is possible on all current commercial hardware. Switching to an iterative approach – computing spike-times by solving differential equations rather than using Equations (12.31), (12.32), (12.42) and (12.43) to compute spike-times for all possible sets of input spikes in parallel – can alleviate memory problems as well, at the cost of runtime.

B.2.2 | Fast Analog Inference on BrainScaleS-2

B.2.2.1 | Parameter

Parameters for experiments in Chapter 13 are shown in Table B.4.

B.2.2.2 | Code

Most of the repositories, described in Chapter 6, are available publicly at

`https://github.com/electronicvisions`

or upon request. Development states are tracked via Gerrit (cf. Section 7.2) at

`https://gerrit.bioai.eu`

that is only accessible for group members and selected collaborators.

Container Image
`/containers/stable/2020-12-15_2.img`
(available from `https://container.bioai.eu`)

App `visionary-dls`

The state of all code-related repositories can be found in Tables B.5 and B.6. Additionally, a special Gerrit sandbox is provided in each repository at `refs/heads/sandbox/breitwieser/wettbewerb_final_state`. This allows checking out the *exact* commit-tree via:

Parameter Name	Accurate Variant	Efficient Variant
Layer Parameter		
input data	$102\,\mathrm{s} @ 32\,\mathrm{Hz} = 3278$	$13.5\,\mathrm{s} @ 32\,\mathrm{Hz} = 432$
conv1d kernel size	43	91
conv1d stride	5	11
conv1d out channels	14	8
MaxPool width	36	—
linear (fully-connected) # 1	252×127	256×123
linear (fully-connected) # 2	127×2	123×10
Neuron Parameter		
neuron capacitance	30 LSB	24 LSB $\longrightarrow$ 16 LSB
refractory counter	24 LSB	19 LSB $\longrightarrow$ 12 LSB
Training Parameter		
dropout after linear #2	—	50 %
ξ	63	
ω_{AF}	$^3/_5$	
$\omega_{\text{sinus rhythm}}$	$^2/_5$	
batch size	1000	
optimizer	Adam	
Adam parameter β	$(0.9, 0.999)$	
Adam parameter ϵ	10^{-8}	
learning rate	0.025 (and manually adjusted lower)	

Table B.4: Network and training parameters for training on BrainScaleS-2 used to produce the results in Section 13.3. Neuron capacitance and refractory counter were reduced mid-experiment as detailed in Section 13.3.3. Mock parameters are determined automatically upon experiment start on the given hardware. All other neuron parameters are set by calibration (which can be obtained by the given `calix` commit in Table B.5).

```
$ cd <cloned-repository>
$ git fetch origin refs/heads/sandbox/breitwieser/wettbewerb_final_state
$ git checkout FETCH_HEAD
```

The software state of the `x86-64`-based client stack is given in Table B.5, whereas the state of the ARM64-based `quiggeldy` implementation is given in Table B.6. For the ARM64-based deployment we used a cross-compiled variant of the `visionary-dls-core` app, tracked in `spack`.[11] In regard to required compute performance, all simulations were run on `Intel(R) Core(TM) i7-4771 CPU @ 3.50GHz` or comparable hardware. Executing non-hardware runs with `hxtorch` is not very memory intensive. By scaling batchsizes, executing experiments is possible on all current commercial hardware.

[11]Commit-hash `99c4c1d208d44e7b6468f3930620b5a0f3a92d2e` with commit message "`util-linux: add python variant`" in Spack repository.

Repository	Commit-Hash	Commit Message
calix	a2b7791baa1d8b1bb3cf3dc10e3728ffd59c99d8	WIP dirty calix changes during wettbewerb
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	Change include order for C/C++ in clang-format config
fisch	724039ca52c516754bf965fcbc84c510553bf4e5	[FRICKEL] Add test target for axi connection
flange	fcde2aafe69805487789ca0b1a8a245caf5fb8ed	Support builds w/o generating Python bindings
grenade	d03fa205e00973304dfc7a4adc63a585c962a6fd	Perform baseline read directly before actual operation
halco	e816a79b4b6e852d6b928a1bf2fdc812621914bd	Adjust FPGA DRAM address range to competition bitfile
haldls	738e7715cdbdb1ee9f062bc71e9fd6b553d19e7f	[Zynq frickel] Disable correlation voltages
hate	c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1	Enable ccache in CI
hwdb	2eec72d253cc5b360b0508b3d5a8fa98c02074a1	Support builds w/o generating Python bindings
hxcomm	46c00c7735d8ef9e408e496dcb83fef1e6e541c3	[Hack] quiggeldy: Ensure new remote connection
hxtorch	ebcec51fc62c59cddfe39a88fdad77d01af2e3cb	learn_fixed_pattern: Add dynamic stop capabilities
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	Update content description in Readme
lib-rcf	edeca3b8a2f037546bffe91967e41f328db7980d	WorkerThread: Trace that work is performed
libnux	334d87b70febd0cd4568c9913bc08e0e59bdd287	GlobalAddress: Fix local SRAM access
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	Add log4cxx_level_v2
model-hw-hdbioai	ae58751ca64a14d6c67725b0ffca8101fb52fa61	Adjusted temperature plots for thesis
pyublas	fb538e8c313a3f04d1a5b77200d192fece3ea901	Add .gitreview
pywrap	83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb	Support builds w/o generating Python bindings
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	Update to gtest 2.0.0
sctrltp	b5f825007b842f44f3e6401f00cf93387e5e3f3c	Support builds w/o generating Python bindings
visions-slurm	3777a9dc36a7067be3657ce06253efec32db260e	Log licenses in elastic search plugin
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	Add .gitreview

Table B.5: Repository state for simulations performed in Chapter 13 (client-side). Since it contains the exact state used for competition simulations, their commit contents represent a work-in-progress state. Additionally, a special Gerrit sandbox is provided in each repository at refs/heads/sandbox/breitwieser/wettbewerb_final_state.

Repository	Commit-Hash	Commit Message
hxcomm	46c00c7735d8ef9e408e496dcb83fef1e6e541c3	[Hack] quiggeldy: Ensure new remote connection
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	Change include order for C/C++ in clang-format config
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	Add log4cxx_level_v2
halco	e816a79b4b6e852d6b928a1bf2fdc812621914bd	Adjust FPGA DRAM address range to competition bitfile
hate	c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1	Enable ccache in CI
sctrltp	59a991f6d85ceaf81dbcf8958969a724958aba3c	Support non-x86_64 atomics
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	Update to gtest 2.0.0
hwdb	2eec72d253cc5b360b0508b3d5a8fa98c02074a1	Support builds w/o generating Python bindings
visions-slurm	3777a9dc36a7067be3657ce06253efec32db260e	Log licenses in elastic search plugin
flange	fcde2aafe69805487789ca0b1a8a245caf5fb8ed	Support builds w/o generating Python bindings
lib-rcf	dc05a1558799e02cb589012879073a9c0c466dc9	WorkerThread: Trace that work is performed
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	Add .gitreview
pywrap	83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb	Support builds w/o generating Python bindings
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	Update content description in Readme

Table B.6: Repository state for simulations performed in Chapter 13 (client-side). Since it contains the exact state used for competition simulations, their commit contents represent a work-in-progress state. Additionally, a special Gerrit sandbox is provided in each repository at `refs/heads/sandbox/breitwieser/wettbewerb_final_state`.

Acronyms and Technical Terms

C

(Citations are found at first usage, where applicable.)

***aaS**	* as a Service 241
BrainScaleS-2 Mobile	BrainScaleS-2 Mobile Analog Neuromorphic Hardware System27, 32, 140, 160 f., 171, 211, 225, 227 ff., 241, 247, 255
ABI	Application Binary Interface 91
Adam	ADAptive Moment estimation 11, 193, 221
ADC	Analog-to-Digital Converter 29, 69
AdEx	Adaptive Exponential 22, 29
AF	atrial fibrillation . . . . . 211–215, 217, 219, 221, 227 f., 234, 241
AI	Artificial Intelligence 12
Anaconda	Anaconda Software Distribution 46, 106
ANC	Analog Network Core 22
ANN	Artificial Neural Network . . . . . 12, 14 f., 17, 20, 192, 195, 249
API	Application Programming Interface 25, 49, 51 f., 54, 60, 63, 69, 71, 74, 79, 85, 89, 142, 146, 169
APT	Advanced Packaging Tool 45
ARM64	Advanced RISC Machines . . . . . 160 f., 225, 232, 241, 262
ARQ	Automatic Repeat reQuest protocol 255
ASIC	Application-Specific Integrated Circuit . . 20, 32 f., 100, 106, 121, 211, 225, 229, 232, 234 f., 241
AWK	AWK Programming Language 132
AXI	Advanced eXtensible Interface 255
bash	The Bourne-Again SHell 79, 97, 112, 115, 128, 132, 172
binutils	GNU Binutils 67
BMBF	Federal Ministry of Education and Research (*Bundesministerium für Bildung und Forschung*) 33, 160, 211, 214, 219, 225–228, 241, 255, 312
BPTT	BackPropagation Through Time 17
BrainScaleS	BrainScaleS Mixed-Signal Accelerated Neuromorphic Systems . 2 ff., 20 ff., 27, 42, 60, 171, 199, 239, 248
BrainScaleS-1	BrainScaleS-1 Wafer-Scale Mixed-Signal Accelerated Neuromorphic System21 f., 25, 28 f., 63, 65, 67, 69, 86, 93, 103, 105, 128, 130, 168, 180, 193, 203, 205, 239, 248
BrainScaleS-2	BrainScaleS-2 Analog Neuromorphic Hardware System . 27, 29, 32 f., 59 ff., 63, 65–69, 78, 86, 92, 104, 113, 131, 141, 155, 161, 174, 180, 193, 199, 201 ff., 211, 215, 217, 221, 224, 226 f., 229, 232, 234 f., 239 ff., 243, 247, 251, 256, 258
BT	Bluetooth 235

C	C Programming Language . . . . 28, 51, 53, 68 f., 78, 91, 97, 147, 167
C++	C++ Programming Language . . 28, 46, 66–69, 78, 91, 93, 96, 145–148, 153, 161, 193, 225 f.
`cabal`	Haskell Cabal 44
CADC	Correlation ADC 29, 31, 224, 226, 231
`calix`	CALIbration Framework for HICANN-X59, 63, 65, 103, 141, 229, 251
CapMem	Capacitive Memory . . . . 28 f., 66, 234
`cargo`	`cargo`: A Package Manager for Rust 44, 78
`cereal`	a C++11 Header-only Library for Serialization . . . . 65 f., 155, 166
`chroot`	change root directory syscall . . . . 50, 52 f.
CI	Continuous Integration . . . . 4, 41, 71 f., 74, 76, 78, 80, 82, 237, 243
CLI	Command-Line Interface . . 98, 104, 107, 130, 132 f., 141 f., 157, 168 f., 173, 238, 255, 257
CMOS	Complementary Metal-Oxide-Semiconductor . . . . 2, 20 f., 27
CNCF	Cloud Native Computing Foundation . . . . 52 f.
CNN	Convolutional Neural Network 14 f., 211, 215, 217, 219, 221, 223, 225, 241, 247
CoBa	Conductance-Based . . . . 22, 193 f., 239
CoCo	Coordinate/Container-Pair . . . . 66, 141 f.
Colab	Colaboratory 164, 174
`conda`	conda: A Cross-Platform, Python-Agnostic Binary Package Manager 46
CPU	Central Processing Unit 19, 73, 111, 120, 142, 173, 225, 232
CuBa	Current-Based . . . . 29, 180, 193 f., 239
CUDA	Compute Unified Device Architecture 60, 247
DAC	Digital-to-Analog Converter 24, 66, 145, 199
DAG	Directed Acyclic Graph 54, 87, 90, 171
DBT	Dynamic Binary Translation 48
DDOS	Distributed Denial-of-Service Attack 95
DenMem	Dendritic Membrane 22, 24
DFKI	German Research Centre for Artificial Intelligence Kaiserslautern226
DMA	Direct Memory Access . . . . 33, 225 f.
Docker	Docker . . . . 52 ff., 98, 100, 110, 119
`dotkit`	`dotkit`, Simple Module Files via Shell Scripts 46, 92
DPI	Direct Programming Interface 69
DRAM	Dynamic Random-Access Memory 28, 32, 226, 234, 240
DSL	Domain Specific Language 25, 47, 87, 140
EasyBuild	EasyBuild: Building Software with Ease. 47, 86
ECG	ElectroCardioGram211 ff., 215, 217, 219, 221, 223, 225–228, 234 f., 241, 247, 249
Electronic Vision(s)	Electronic Vision(s) Group at the Kirchhoff-Institute for Physics in Heidelberg . . . 4, 21, 38 f., 44, 47, 54, 59 ff., 73–76, 78 f., 85 f., 92 ff., 96, 98, 100 f., 103, 105 f., 108, 110, 125 f., 129, 132, 137, 140 f., 158, 160 f., 167 ff., 171 f., 225, 237, 239, 258
`ext3`	third EXTended filesystem . . . . 99 f.
FAIR	Findable, Accessible, Interoperable, and Reusable 40
FG	Floating Gates 22, 25, 28, 203

FIR	Finite Impulse Response 215, 234
fisch	FPGA Instruction Set Compiler for HICANN . . 59, 66 ff., 141 f., 146 f., 153
flange	Linking C++ Software Stacks with SystemVeriolog using DPI66, 68 f., 140, 146, 255
fMRI	functional Magnetic Resonance Imaging 36
Fortran	Fortran Programming Language 78, 91
FPGA	Field-Programmable Gate Array . 20, 27 ff., 32 f., 65–69, 130, 141, 143, 145 ff., 156, 158 f., 166, 168, 173, 202, 215, 225 ff., 231, 234, 240 f., 255
FUSE	Filesystem in Userspace 54
gcc	GNU Compiler Collection 67
genpybind	Autogeneration of Python Bindings from Manually Annotated C++ Headers . . . . 66 ff., 311
Gerrit	Gerrit Code Review . . 74, 76 f., 79, 112, 115, 118, 121 ff., 161, 237, 247, 253, 258, 262
GID	Group IDentifier number 50, 99
GIL	Global Interpreter Lock 116
git	Git – a distributed version-control system for tracking changes in source code during software development, see Section 4.4.1 42 f., 45, 74, 76, 78, 87, 91, 115, 128, 158, 258
GNU modules	GNU Environment Modules 46, 92, 97, 113, 125, 134
go	go Programming Language 44, 97, 172
GPGPU	General Purpose Graphical Processing Unit 19, 21, 60
GPU	Graphics Processing Unit 14, 19, 53, 97, 100, 107, 193
grenade	GRaph-based Experiment Notation And Data-flow Execution 63, 65, 140, 154, 159 f., 167, 226, 238, 241
gres	Generic RESource 145
HAGEN	Heidelberg AnaloG Evolvable neural Network . 29, 31, 159, 164, 211, 221, 241
hagen-daas	Howto Avoid Grabbing Emulators Nightlong – Dls As A Service141, 145, 157, 167 ff., 257
halco	Hardware Abstraction Layer providing COordinates for BrainScaleS-1-based and BrainScaleS-2-based neuromorphic systems . . . . . . 66 ff.
haldls	Hardware Abstraction Layer for HICANN-DLS . . 59, 63, 66 ff., 140 f., 145, 153
HashDist	HashDist 47, 86, 91
Haskell	Haskell Programming Language 44
HBP	Human Brain Project 21, 27, 174
HICANN	High Input Count Analog Neural Network . . . . 22, 24 f., 205
HICANN-DLS	HICANN Dreieck Ludwighafen Süd: successor to HICANN chip and based on the technology test chip route65 which inspired the reference to BAB65 . . . . 27, 29, 103, 145, 195, 203, 248
HICANN-DLS-SR-HX	HICANN Dreieck LudwighafenSüd: Spikey Replacement with HAGEN eXtensions 27, 29
HICANN-X	Short Form of HICANN-DLS-SR-HX . 27, 29, 31, 145 f., 164, 195, 202, 221, 239 ff., 243, 255
HPC	High-Performance Computing 46 f., 52 f., 73, 86, 89, 97, 106, 110, 125, 137, 171, 237

`hwdb`	Electronic Vision(s) Hardware Database . . . . 69, 129 f., 169
`hxcomm`	Low-Level Communication With HICANN-X via Hostarq . . . . 67 ff., 140 ff., 145 ff., 149, 151, 153 f., 156 ff., 161, 225, 255 f., 258
`hxtorch`	`PyTorch` for BrainScaleS-2 . 59, 65, 141 f., 154, 157, 159, 164, 167, 221, 226, 241, 256, 262
I^2C	Inter-Integrated Circuit 226
I/O	Input/Output . . . . 2 f., 67, 235, 238, 241
IC	Integrated Circuit 33, 231
IDL	Interface Definition Language 69
IF	Integrate-and-Fire 180, 182, 198
IoT	Internet-of-Things 241
`iptables`	Administration Tool for IPv4 Packet Filtering and NAT 132
ISA	Instruction Set Architecture . . . . 48, 160 f.
ISO	International Organization for Standardization 35
Java	Java programming language . . . . 71, 78, 89, 93 f.
Javascript	ECMAScript 2020 93
Jenkins	Jenkins . . . . . 71, 73–77, 79, 81, 92 f., 97, 103 ff., 115 f., 118 ff., 122, 172
`jenlib`	Jenlib – Shared Library for Visionary Jenkins Pipelines . . . . 73 f.
JIT	Just-In-Time 226
JSC	Jülich Supercomputing Centre 54
JSON	JavaScript Object Notation 53, 79
JTAG	Joint Test Action Group industry standard for verifying designs and testing printed circuit boards after manufacture 67
Jupyter	Project Jupyter – Julia Python and R 163, 174
KVM	Kernel-based Virtual Machine 48
L1	Layer-1 . . . . 24 f.
L2	Layer-2 24
LFSR	Linear-Feedback Shift Register 25, 248
`lib-rcf`	Library Wrapping and Extending RCF 69, 140, 142, 155
`libnux`	Library to Interface with PPU Codenamed Nux 67, 93, 248
LIF	Leaky-Integrate-and-Fire . . . . 22, 29, 180, 182, 188, 193 f., 239, 249
Lmod	Lua-based Module System . . . . 46 f., 92
`lola`	LOgical LAyer . . . . 66 ff.
LRZ	Leibniz Supercomputing Center 86
LUT	Look-Up Table 240
LXC	LinuX Containers 52, 54
MAC	Multiply-ACcumulate operation14, 21, 29, 31, 159, 164, 211, 217, 219, 221 f., 224, 228 f., 231, 241, 243
MADC	Membrane ADC 29
mamba	The Fast Cross-Platform Package Manager 46
`man`	Terminal-accessible Manual Page 92
Mattermost	Mattermost 74, 118
MIT DB	The Massachusetts Institute of Technology-Beth Israel Hospital Arrhythmia Database (48 records, 30 minutes each) 217
MixDown	MixDown: Meta-build tool for managing collections of third-party libraries 44
MNIST	MNIST Database 9, 17, 181, 191, 193, 195, 198, 202, 207, 239, 248

MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor	1
MPI	Message Passing Interface	53, 89
MUNGE	MUNGE Uid ‘N’ Gid Emporium	126, 129, 132, 151, 154, 156
NCC	Neural Computability Condition	16
NEST	NEural Simulation Tool	87, 248, 250
NFS	Network File System	73, 97, 101, 107, 128 f., 172
NICE 2021	Neuro-Inspired Computational Elements Conference 2021	35, 81, 140, 156, 161, 167, 174
Nix	Nix Package Manager	47, 86, 89, 91
`npm`	Node Package Manager	44
`nscd`	Name Service Cache Daemon	132
NSEM	Neuromorphic Spike-Based Expectation Maximization	103, 248, 252
OCI	Open Container Initiative	52 f., 98
OS	Operating System	45, 48 f., 52, 106
OverlayFS	OverlayFS	99, 105
`pacman`	Package Manager	45
`pbmem`	PlayBack MEMory program	63, 66 f., 141–145, 147, 152 f., 159, 164, 234
PCB	Printed Circuit Board	32 f.
`PID`	Process ID	50
pip	Package Installer for Python	44
POD	Plain Old Data type	147, 166
POSIX	Portable Operating System Interface	50
PPU	Plasticity Processing Unit	28 f., 33, 67, 143, 202, 217, 219, 225 f., 231, 234, 240
PSP	Post-Synaptic-Potential	16, 186, 199
PXE	Preboot Execution Environment	128
`pybind11`	Seamless operability between C++11 and Python	68
PyNN	A Python package for simulator-independent specification of Neuronal Network models	25, 63, 65, 141, 159 f.
`pynn.brainscales2`	PyNN-backend for BrainScaleS-2	59, 63, 65
Python	Python Programming Language	44, 47, 60, 63, 69, 78 f., 87, 89, 91, 94, 96 f., 100 f., 104, 106, 111, 116, 121, 141, 145–148, 153, 161, 163, 173 f., 193, 225, 232
`python-setuptools`	Python Setuptools	44, 78, 94
`PyTorch`	Python-Implementation of Lua-library `torch`	60, 65, 193, 199, 221, 224, 227
QEMU	Quick EMUlator	48
RAII	Resource Allocation Is Initialization	142, 148, 153
RAM	Random-Access Memory	116
RCF	Remote Call Framework – a cross-platform interprocess communication framework for C++	68 f., 140, 149, 151 f., 154 f., 256 f.
ReLU	Rectified Linear Unit	13, 211, 217, 219, 224 ff., 228, 231
REPL	Read-Eval-Print-Loop	137
REST	REpresentational State Transfer	54, 71, 74
Rietveld	Rietveld Code Review Tool	74

`RPATH`	Run-Time Search Path	91, 128, 134
RPC	Remote Procedure Calls	125 f.
RPM	RPM Package Manager	45
RTL	Register-Transfer Level	71
`runc`	runC	52 f.
Rust	Rust Programming Language	44, 54, 78, 153
`sbs`	Spike-Based Sampling – a library for fast Neural Sampling	16, 93, 105, 111, 248, 250 f.
SCIF	SCIentific Filesystem	104
`sctrltp`	Slow ConTRoL Transport Protocol	50, 59, 68 f., 141 f., 173
SF	Serialization Framework of RCF	155, 166
SGA3	Human Brain Project Specific Grant Agreement 3	174
SGD	Stochastic Gradient Descent	11, 17
SIF	Singularity Image Format	100, 108, 120
SIFT	Scale-Invariant Feature Transform	14
SIMD	Single Instruction stream – Multiple Data streams	19, 27 f.
Singularity	Singularity Container	54, 73, 92, 97–100, 104, 106–110, 112 f., 117, 119 f., 129, 132 ff., 171 f.
sinus rhythm	Sinus Rhythm	211, 214, 217, 219, 221, 227 f., 241
SISD	Single Instruction stream – Single Data stream	19
Slurm	Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management	53, 69, 73, 93, 107, 125–135, 137 f., 141, 145, 151, 157, 163, 167 ff., 174, 238, 256 f.
`smithy`	Smithy	47, 86
SNN	Spiking Neural Network	3, 15, 17, 20 f., 179, 249
SO-DIMM	Small Outline Dual In-line Memory Module	32
SoC	System on a Chip	27, 32, 160 f., 225 f.
Spack	Supercomputing PACKage manager	47, 54, 76, 86 f., 89–92, 94–97, 104 f., 112 f., 115–122, 128, 171 ff., 225, 237, 247, 262
spec	Specification of Package Configurations as used by Spack	87, 89 ff., 115, 117, 121, 225
SPI	Serial Peripheral Interface	145
Spikey	Spikey chip	93, 105, 171, 248
SQL	Structured Query Language	125
SquashFS	SquashFS	100, 108, 120
SRAM	Static Random-Access Memory	22, 28 f., 31, 240
SSD	Solid State Disk	172
`ssh`	Secure SHell	71, 79, 98, 117
`stack`	Haskell Tool Stack	44
`stadls`	STAteful encapsulation for HICANN-DLS	59, 63, 65–68, 141 f., 147, 167
STDP	Spike-Timing Dependent Plasticity	28
STP	Short Term Plasticity	24, 29, 31
`StreamRC`	`Stream`-interace with for `R`emote exe`C`ution	151
`sudo`	su “do”	110, 132
SURF	Speeded-Up Robust Features	14
`svn`	Apache Subversion	74

symlink	Symbolic Link	101
`symwaf2ic`	Electronic Vision(s)-specific fork of `waf`	44, 78 f., 237
syscall	System Call	49 f., 52, 267
`systemd`	`systemd` System and Service Manager	52, 128 f.
SystemVeriolog	SystemVerilog Programming Language	69, 146, 255
TCP	Transmission Control Protocol	52, 111, 152
TDD	Test-Driven Development	60
TensorFlow	TensorFlow	60, 65
Theano	Theano: A Python framework for fast computation of mathematical expressions	60
TM	Tsodyks-Markram	24
TOC/TUO	Time-Of-Check to Time-Of-Use	110, 171
TPU	Tensor Processing Unit	21, 60
TTFS	Time-To-First-Spike	17, 103, 179 f., 192 f., 195 f., 203, 207, 209, 239 f., 243, 247, 251
TUI	Text-based User Interface	97
UID	User IDentifier number	50, 99 f.
UNICORE	Uniform Interface to Computing Resources	93
URL	Uniform Resource Locator	79
USB	Universal Serial Bus	145, 171, 226
UT	Universal Translator	67 ff., 141–144, 146, 153
UUID	Universally Unique IDentifier	151
VLSI	Very Large Scale Integration	19
VM	Virtual Machine	48, 126, 172
`waf`	Waf: the meta build system	42, 73, 77 ff., 81, 171
WSF	weight-scale factor	194, 203, 205
YAML	YAML Ain't Markup Language	54, 69
`yashchiki`	from Russian, ящики, meaning boxes or "Schachtel" in German	106, 112 f., 115–118, 120 f., 134, 171 ff., 237, 247
YUM	Yellowdog Updater, Modified	45
`zsh`	Z SHell	115

List of Figures D

2.1 Reducing overfitting by using more data. 10
2.2 Schematic of a densely connected feed-forwar ANN with multiple layers. 12
2.3 Example of discrete convolution. 14

3.1 Photographs of BrainScaleS-1. 22
3.2 Architecture of the BrainScaleS-1 wafer-scale hardware system. 23
3.3 Conceptual overview of the BrainScaleS-1 synapse driver. 24
3.4 Overview of the BrainScaleS-1 software stack. 26
3.5 BrainScaleS-2 single-chip “cube” setup. 27
3.6 Internal structure and layout of BrainScaleS-2 ASIC. 30
3.7 Operation principle of BrainScaleS-2. 30
3.8 Overview of BrainScaleS-2 Mobile 32
3.9 Block diagram of the major functional units of the FPGA of BrainScaleS-2 Mobile. 32

4.1 Overview over different types of virtualization. 48

6.1 Overview of BrainScaleS-2 software stack. 64

7.1 Overview of the general software development workflow at Electronic Vision(s). 72
7.2 Gerrit workflow. 74
7.3 Examples for inter-repository dependencies mapped via `Depends-On`. 77
7.4 Examples of dependency relations that lead to errors, but are supported now. . . 80

8.1 General concretization workflow in Spack. 90
8.2 Example of full Spack dependency graph. 90
8.3 Singularity usage workflow. 99
8.4 Overview of deployed container’s NFS folder structure 102
8.5 Example for nested container workflows. 109
8.6 Overview of visionary container file structure. 114

9.1 Overview of deployment organization on `slurmviz`. 127

10.1 Core principle of `quiggeldy` operation. 139
10.2 Overview of `quiggeldy` implementation. 150
10.3 Simplified workflow schematic of on-demand upload. 153
10.4 Overview of `grenade`’s JIT graph execution. 160
10.5 Two examples of interactive user demos, both spiking and non-spiking. 162
10.6 Rate of executed experiment-steps via `quiggeldy` during the BrainScaleS-2 hands-on tutorial at NICE 2021. 163

10.7 Overhead evaluation for `quiggeldy` in non-spiking mode with different types of serialization. 165
10.8 Evaluating hardware utilization via `quiggeldy`. 166
10.9 Overview of `hagen-daas` workflow. 168

12.1 The problem with analog readout of state in analog neuromorphic hardware. . . 179
12.2 100 example patterns from MNIST test dataset [LeCun et al., 1998]. 182
12.3 Overview: TTFS coding and learning. 183
12.4 TTFS: Choice of branch for case with $\tau_\mathrm{m} = \tau_\mathrm{s}$. 186
12.5 Classification of the Yin-Yang dataset (figure). 197
12.5 Classification of the Yin-Yang dataset (caption). 198
12.6 Classification of the MNIST dataset. 199
12.7 Classification of Yin-Yang on BrainScaleS-2. 200
12.8 Classification of the MNIST dataset on BrainScaleS-2. 201
12.9 Voltage traces on BrainScaleS-1. 203
12.10 Analysis of spike-time variation for BrainScaleS-1 205
12.11 Visualization of the mapping of the network in Section 12.3.3. 206
12.12 Training a spiking network on BrainScaleS-1. 206
12.13 Investigating hardware “imperfections”. 208
12.14 Robustness to variations after training. 209

13.1 Schematic representation of human heart including conduction system. 212
13.2 The individual components of the electrocardiogram of a simplified sinus rhythm signal. 212
13.3 Some examples of ECG signals from MIT DB with annotated beat positions. . . 214
13.4 Lorenz plot of BMBF dataset. 215
13.5 Dependence of average loss on starting position of time-slice. 216
13.6 Preprocessing steps. 216
13.7 Possible variants of implementing 1D-convolutions on BrainScaleS-2. 218
13.8 Accurate model structure BrainScaleS-2. 219
13.9 Efficient model structure of model used in BMBF competition. 220
13.10 Comparison of noise BrainScaleS-2 and mock-mode. 222
13.11 Comparison mock-mode/BrainScaleS-2. 223
13.12 Evaluation setup for the BrainScaleS-2 mobile system. 225
13.13 Confusion matrices for the accurate model variant presented in Figure 13.8. . . . 227
13.14 Training progress of final model on BrainScaleS-2 Mobile. 230
13.15 Temperature validation system #3. 232
13.16 Temperature stability of BrainScaleS-2 Mobile. 233

List of Tables

E

4.1 Comparison between different container implementations. 55

7.1 Gerrit statistics. 75

12.1 Classification results when learning with IF-neurons. 182
12.2 Summary of TTFS results. 196
12.3 Literature review for pattern recognition models on neuromorphic back-ends. . 204

13.1 Measured results for the classification of 500 randomly selected ECG traces. . . 228

B.1 Neuron, network and training parameters used to produce the results in Section 12.3.1. Adapted from: [Göltz et al., 2021, Table A]. 259
B.2 Network and training parameters for training on BrainScaleS-2 used to produce the results in Section 12.3.2. 260
B.3 Repository state for TTFS simulations performed in Chapter 12. 261
B.4 Network and training parameters for training on BrainScaleS-2 used to produce the results in Section 12.3.2. 263
B.5 Repository state for simulations performed in Chapter 13 (client-side). 265
B.6 Repository state for simulations performed in Chapter 13 (client-side). 266

V

Bibliography

Aamir, S. A., Y. Stradmann, P. Müller, C. Pehle, A. Hartel, A. Grübl, J. Schemmel, and K. Meier (Dec. 2018). "An Accelerated LIF Neuronal Network Array for a Large-Scale Mixed-Signal Neuromorphic Architecture". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.12, pp. 4299–4312. ISSN: 1549-8328. DOI: 10.1109/TCSI.2018.2840718.

Aamir, Syed Ahmed, Paul Müller, Andreas Hartel, Johannes Schemmel, and Karlheinz Meier (2016). "A highly tunable 65-nm CMOS LIF neuron for a large-scale neuromorphic system". In: *Proceedings of IEEE European Solid-State Circuits Conference (ESSCIRC).*

Aamir, Syed Ahmed, Paul Müller, Laura Kriener, Gerd Kiene, Johannes Schemmel, and Karlheinz Meier (2017). "From LIF to AdEx neuron models: Accelerated analog 65 nm CMOS implementation". In: *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 1–4.

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.* URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

Aharoni, Roee, Melvin Johnson, and Orhan Firat (June 2019). "Massively Multilingual Neural Machine Translation". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* Minneapolis, Minnesota: Association for Computational Linguistics, pp. 3874–3884. DOI: 10.18653/v1/N19-1388. URL: https://www.aclweb.org/anthology/N19-1388.

Ahmadia, Aron, Volker Braun, Ondrej Certik, Chris Kees, Fernando Perez, Dag Sverre Seljebotn, and Andy Terrel (2012). *HashDist.* URL: http://github.com/hashdist/hashdist (visited on 12/06/2020).

Aho, Alfred V, Brian W Kernighan, and Peter J Weinberger (1987). *The AWK programming language.* Addison-Wesley Longman Publishing Co., Inc.

Akkerhuis, G. A. J. M. Jagers op and C. Damgaard (1999). "Using Resource Dominance to Explain and Predict Evolutionary Success". In: *Oikos* 87.3, pp. 609–614. ISSN: 00301299, 16000706. URL: http://www.jstor.org/stable/3546828.

Akopyan, Filipp, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. (2015). "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10, pp. 1537–1557.

Albada, Sacha J. van, Andrew G. Rowley, Johanna Senk, Michael Hopkins, Maximilian Schmidt, Alan B. Stokes, David R. Lester, Markus Diesmann, and Steve B. Furber (2018). "Performance Comparison of the Digital Neuromorphic Hardware SpiNNaker and the Neural Network Simulation Software NEST for a Full-Scale Cortical Microcircuit Model". In: *Frontiers in Neuroscience* 12, p. 291. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00291. URL: https://www.frontiersin.org/article/10.3389/fnins.2018.00291.

Altini, Marco, Salvatore Polito, Julien Penders, Hyejung Kim, Nick Van Helleputte, Sunyoung Kim, and Firat Yazicioglu (2011). “An ECG Patch Combining a Customized Ultra-Low-Power ECG SoC with Bluetooth Low Energy for Long Term Ambulatory Monitoring”. In: *Proceedings of the 2nd Conference on Wireless Health*. WH ’11. San Diego, California: Association for Computing Machinery. ISBN: 9781450309820. DOI: 10.1145/2077546.2077564. URL: https://doi.org/10.1145/2077546.2077564.

Andrae, Anders SG and Tomas Edler (2015). “On global electricity usage of communication technology: trends to 2030”. In: *Challenges* 6.1, pp. 117–157.

Anzt, Hartwig, Felix Bach, Stephan Druskat, Frank Löffler, Axel Loewe, Bernhard Y. Renard, Gunnar Seemann, Alexander Struck, Elke Achhammer, Piush Aggarwal, and et al. (Apr. 2020). “An environment for sustainable research software in Germany and beyond: current state, open challenges, and call for action”. In: *F1000Research* 9, p. 295. ISSN: 2046-1402. DOI: 10.12688/f1000research.23224.1. URL: http://dx.doi.org/10.12688/f1000research.23224.1.

Armenise, Valentina (2015). “Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery”. In: *Proceedings of the Third International Workshop on Release Engineering*. RELENG ’15. Florence, Italy: IEEE Press, pp. 24–27.

Authority, Python Packaging (2006). *Python setuptools*. URL: https://setuptools.readthedocs.io/en/latest/history.html#credits.

– (2008). *pip: the package installer for Python*. URL: https://pip.pypa.io/en/stable/.

AVNET (2020). *Ultra96-V2*. URL: http://zedboard.org/sites/default/files/product_briefs/5365-pb-ultra96-v2-v10b.pdf.

Baker, Monya (2016). “1,500 scientists lift the lid on reproducibility”. In: *Nature News* 533.7604, p. 452.

Banbury, Colby R., Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, Danilo Pau, Jae-sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav (2021). *Benchmarking TinyML Systems: Challenges and Direction*. arXiv: 2003.04821 [cs.PF].

Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield (Oct. 2003). “Xen and the Art of Virtualization”. In: *SIGOPS Oper. Syst. Rev.* 37.5, pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: https://doi.org/10.1145/1165389.945462.

Bartheld, Christopher S von, Jami Bahney, and Suzana Herculano-Houzel (2016). “The search for true numbers of neurons and glial cells in the human brain: A review of 150 years of cell counting”. In: *Journal of Comparative Neurology* 524.18, pp. 3865–3895.

Baumbach, Andreas (Nov. 2021). “From microscopic dynamics to ensemble behavior in spiking neural networks”. PhD thesis. Universität Heidelberg.

Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool (July 2006). “SURF: Speeded up robust features”. In: vol. 3951, pp. 404–417. ISBN: 978-3-540-33832-1. DOI: 10.1007/11744023_32.

Beck, Kent (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Belguidoum, Meriem and Fabien Dagnat (2007). “Dependency Management in Software Component Deployment”. In: *Electronic Notes in Theoretical Computer Science* 182. Proceedings of the Third International Workshop on Formal Aspects of Component Software

(FACS 2006), pp. 17–32. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2006.09.029`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066107003830`.

Belkin, Maxim, Roland Haas, Galen Wesley Arnold, Hon Wai Leong, Eliu A. Huerta, David Lesny, and Mark Neubauer (2018). "Container Solutions for HPC Systems: A Case Study of Using Shifter on Blue Waters". In: *Proceedings of the Practice and Experience on Advanced Research Computing*. PEARC '18. Pittsburgh, PA, USA: Association for Computing Machinery. ISBN: 9781450364461. DOI: `10.1145/3219104.3219145`. URL: `https://doi.org/10.1145/3219104.3219145`.

Bellard, Fabrice (2005). "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA, p. 46.

Bellec, Guillaume, Franz Scherr, Elias Hajek, Darjan Salaj, Robert A. Legenstein, and Wolfgang Maass (2019). "Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets". In: *arXiv preprint arXiv:1901.09049*.

Bellec, Guillaume, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass (July 2020). "A solution to the learning dilemma for recurrent networks of spiking neurons". In: *Nature Communications* 11.1, p. 3625. ISSN: 2041-1723. DOI: `10.1038/s41467-020-17236-y`. URL: `https://doi.org/10.1038/s41467-020-17236-y`.

Benedicic, Lucas, Felipe A. Cruz, Alberto Madonna, and Kean Mariotti (2019). "Sarus: Highly Scalable Docker Containers for HPC Systems". In: *High Performance Computing*. Ed. by Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode. Cham: Springer International Publishing, pp. 46–60. ISBN: 978-3-030-34356-9.

Benedyczak, K., B. Schuller, M. Petrova-El Sayed, J. Rybicki, and R. Grunzke (2016). "UNICORE 7 — Middleware services for distributed and federated computing". In: *2016 International Conference on High Performance Computing Simulation (HPCS)*, pp. 613–620. DOI: `10.1109/HPCSim.2016.7568392`.

Bengio, Yoshua, Aaron C Courville, and Pascal Vincent (2012). "Unsupervised feature learning and deep learning: A review and new perspectives". In: *CoRR, abs/1206.5538* 1, p. 2012.

Benjamin, Ben Varkey, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen (2014). "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations". In: *Proceedings of the IEEE* 102.5, pp. 699–716.

Bernstein, D. (2014). "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3, pp. 81–84. DOI: `10.1109/MCC.2014.51`.

Bhatt, Asti, Valentic, Todd, Reimer, Ashton, Lamarche, Leslie, Reyes, Pablo, and Cosgrove, Russell (2020). "Reproducible Software Environment: a tool enabling computational reproducibility in geospace sciences and facilitating collaboration". In: *J. Space Weather Space Clim.* 10, p. 12. DOI: `10.1051/swsc/2020011`. URL: `https://doi.org/10.1051/swsc/2020011`.

Bill, Johannes, Lars Buesing, Stefan Habenschuss, Bernhard Nessler, Wolfgang Maass, and Robert Legenstein (Aug. 2015). "Distributed Bayesian Computation and Self-Organized Learning in Sheets of Spiking Neurons with Local Lateral Inhibition". In: *PLOS ONE*

10.8, pp. 1–51. DOI: 10.1371/journal.pone.0134356. URL: https://doi.org/10.1371/journal.pone.0134356.

Bill, Johannes and Robert Legenstein (2014). “A compound memristive synapse model for statistical learning through STDP in spiking neural networks”. In: *Frontiers in Neuroscience* 8.412. ISSN: 1662-453X. DOI: 10.3389/fnins.2014.00412. URL: http://www.frontiersin.org/neuromorphic_engineering/10.3389/fnins.2014.00412/abstract.

Billaudelle, S., Y. Stradmann, K. Schreiber, B. Cramer, A. Baumbach, D. Dold, J. Göltz, A. F. Kungl, T. C. Wunderlich, A. Hartel, E. Müller, O. Breitwieser, C. Mauch, M. Kleider, A. Grübl, D. Stöckel, C. Pehle, A. Heimbrecht, P. Spilger, G. Kiene, V. Karasenko, W. Senn, M. A. Petrovici, J. Schemmel, and K. Meier (Oct. 2020). “Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. DOI: 10.1109/iscas45731.2020.9180741.

Billaudelle, Sebastian, Benjamin Cramer, Petrovici Mihai A, Korbinian Schreiber, David Kappel, Johannes Schemmel, and Karlheinz Meier (2019). “Structural plasticity on an accelerated analog neuromorphic hardware system”. In: *arXiv preprint; arXiv:1912.12047*.

Bishop, Christopher M and Nasser M Nasrabadi (2006). *Pattern recognition and machine learning*. Vol. 1. springer New York.

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer. ISBN: 978-0387-31073-2. URL: http://research.microsoft.com/en-us/um/people/cmbishop/prml/.

Boell, Richard (2018). “Visualization of Mapping and Routing of the BrainScaleS System”. Bachelorarbeit. Universität Heidelberg.

Bohte, Sander M., Joost N. Kok, and Han La Poutré (2000). “SpikeProp: backpropagation for networks of spiking neurons”. In: *ESANN 2000, 8th European Symposium on Artificial Neural Networks, Brugres, Belgium, April*.

Bolte, Matthias, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann (2010). “Non-intrusive virtualization management using libvirt”. In: *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 574–579. DOI: 10.1109/DATE.2010.5457142.

Booch, Grady (1990). *Object Oriented Design with Applications*. USA: Benjamin-Cummings Publishing Co., Inc. ISBN: 0805300910.

BrainScaleS (2011). *BrainScales - Brain-inspired multiscale computation in neuromorphic hybrid systems*. Accessed: 2019-07-30. URL: http://brainscales.kip.uni-heidelberg.de/.

Bray, T. (Dec. 2017). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. URL: https://www.rfc-editor.org/rfc/rfc8259.html (visited on 11/27/2020).

Breitwieser, Oliver (2015). “Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization”. Master thesis. Ruprecht-Karls-Universität Heidelberg.

– (Feb. 2020). *obreitwi/py-veer: Seamless execution of Python code in different environments, be it simple sub-processes or singularity containers*. Version v0.1.0. DOI: 10.5281/zenodo.3675308. URL: https://doi.org/10.5281/zenodo.3675308.

Breitwieser, Oliver, Andreas Baumbach, Agnes Korcsak-Gorzo, Johann Klähn, Max Brixner, and Mihai Petrovici (Feb. 2020). *sbs: Spike-based Sampling (v1.8.2)*. Version v1.8.2. This open source software code was developed in part in the Human Brain Project, funded from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 720270 (HBP SGA1) and 785907 (HBP SGA2). DOI: `10.5281/zenodo.3686011`. URL: `https://doi.org/10.5281/zenodo.3686011`.

Brette, R. and W. Gerstner (2005). "Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity". In: *J. Neurophysiol.* 94, pp. 3637–3642. DOI: `10.1152/jn.00686.2005`.

Brooks, Frederick P. (1978). *The Mythical Man-Month: Essays on Softw.* 1st. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201006502.

– (1987). "No Silver Bullet: Essence and Accidents of Software Engineering". In: *IEEE Computer* 20, pp. 10–19.

Brooks, Rodney, Demis Hassabis, Dennis Bray, and Amnon Shashua (2012). "Is the brain a good model for machine intelligence?" In: *Nature* 482.7386, p. 462.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei (2020). "Language Models are Few-Shot Learners". In: arXiv: `2005.14165 [cs.CL]`.

Brunel, Nicolas and Mark CW Van Rossum (2007). "Lapicque's 1907 paper: from frogs to integrate-and-fire". In: *Biological cybernetics* 97.5-6, pp. 337–339.

Buesing, L., J. Bill, B. Nessler, and W. Maass (2011). "Neural dynamics as sampling: A model for stochastic computation in recurrent networks of spiking neurons". In: *PLoS Computational Biology* 7.11, e1002211.

Bundesministerium für Bildung und Forschung (BMBF) (2019). *Bekanntmachung: Richtlinie zur Förderung des Pilotinnovationswettbewerbs "Energieeffizientes KI-System"*. German. URL: `https://www.bmbf.de/foerderungen/bekanntmachung-2371.html` (visited on 09/07/2020).

Cai, Fuxi, Justin M Correll, Seung Hwan Lee, Yong Lim, Vishishtha Bothra, Zhengya Zhang, Michael P Flynn, and Wei D Lu (2019). "A fully integrated reprogrammable memristor–CMOS system for efficient multiply–accumulate operations". In: *Nature Electronics* 2.7, pp. 290–299.

Canon, Richard Shane and Doug Jacobsen (2016). "Shifter: containers for HPC". In: *Proceedings of the Cray User Group.*

Cao, Yongqiang, Yang Chen, and Deepak Khosla (2015). "Spiking deep convolutional neural networks for energy-efficient object recognition". In: *International Journal of Computer Vision* 113.1, pp. 54–66.

Chen, Gregory K, Raghavan Kumar, H Ekin Sumbul, Phil C Knag, and Ram K Krishnamurthy (2018). "A 4096-neuron 1M-synapse 3.8-pJ/SOP spiking neural network with on-chip STDP learning and sparse weights in 10-nm FinFET CMOS". In: *IEEE Journal of Solid-State Circuits* 54.4, pp. 992–1002.

Chen, Jim X., Jeffrey Carver, Steven Gottlieb, Douglass E. Post, and Barry I. Schneider, eds. (2019). *Computing in Science & Engineering* 21.1: *Race to Exascale.*

Chen, W., H. Lu, L. Shen, Z. Wang, N. Xiao, and D. Chen (2008). “A Novel Hardware Assisted Full Virtualization Technique”. In: *2008 The 9th International Conference for Young Computer Scientists*, pp. 1292–1297. DOI: 10.1109/ICYCS.2008.218.

Chrissis, Mary Beth, Michael Konrad, and Sandra Shrum (2011). *CMMI for Development: Guidelines for Process Integration and Product Improvement.* Third. Addison-Wesley Professional.

Chua, Leon O and Sung Mo Kang (1976). “Memristive devices and systems”. In: *Proceedings of the IEEE* 64.2, pp. 209–223.

Chung, Mina K, Lee L Eckhardt, Lin Y Chen, Haitham M Ahmed, Rakesh Gopinathannair, José A Joglar, Peter A Noseworthy, Quinn R Pack, Prashanthan Sanders, and Kevin M Trulock (2020). “Lifestyle and Risk Factor Modification for Reduction of Atrial Fibrillation: A Scientific Statement From the American Heart Association”. In: *Circulation (New York, N.Y.)* 141.16, e750–e772. DOI: 10.1161/CIR.0000000000000748.

Cireşan, Dan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber (2012). “Multi-column deep neural network for traffic sign classification”. In: *Neural Networks* 32, pp. 333–338.

Clifford, Gari D, Chengyu Liu, Benjamin Moody, Li-Wei H Lehman, Ikaro Silva, Qiao Li, A E Johnson, and Roger G Mark (2017). “AF Classification from a Short Single Lead ECG Recording: the PhysioNet/Computing in Cardiology Challenge 2017”. In: *Computing in Cardiology* 44. ISSN: 2325-8861 and 2325-887X.

CollabNet, Inc. and Apache Software Foundation (2000). *Apache Subversion Homepage.* URL: https://subversion.apache.org/ (visited on 11/23/2020).

Combe, T., A. Martin, and R. Di Pietro (2016). “To Docker or Not to Docker: A Security Perspective”. In: *IEEE Cloud Computing* 3.5, pp. 54–62. DOI: 10.1109/MCC.2016.100.

Comsa, Iulia M, Thomas Fischbacher, Krzysztof Potempa, Andrea Gesmundo, Luca Versari, and Jyrki Alakuijala (2020). “Temporal coding in spiking neural networks with alpha synaptic function”. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8529–8533.

Cook, Stephen A (1971). “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing.* ACM, pp. 151–158.

Coral (Aug. 2020). *Edge TPU performance benchmarks.* URL: https://coral.ai/docs/edgetpu/benchmarks/.

Cota, E. G., P. Bonzini, A. Bennée, and L. P. Carloni (2017). “Cross-ISA machine emulation for multicores”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 210–220. DOI: 10.1109/CGO.2017.7863741.

Cox, Russ (Aug. 2019). “Surviving Software Dependencies”. In: *Commun. ACM* 62.9, pp. 36–43. ISSN: 0001-0782. DOI: 10.1145/3347446. URL: https://doi.org/10.1145/3347446.

Cramer, Benjamin, Sebastian Billaudelle, Simeon Kanya, Aron Leibfried, Andreas Grübl, Vitali Karasenko, Christian Pehle, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Johannes Schemmel, and Friedemann Zenke (2020). “Training spiking multi-layer networks with surrogate gradients on an analog neuromorphic substrate”. In: *arXiv preprint.* arXiv: 2006.07239 [cs.NE]. URL: https://arxiv.org/abs/2006.07239.

Czierlinski, Milena (2020). “PyNN for BrainScaleS-2”. Bachelorarbeit. Universität Heidelberg.

Czischek, Stefanie, Andreas Baumbach, Sebastian Billaudelle, Benjamin Cramer, Lukas Kades, Jan M. Pawlowski, Markus K. Oberthaler, Johannes Schemmel, Mihai A. Petrovici, Thomas Gasenzer, and Martin Gärttner (2020). *Spiking neuromorphic chip learns entangled quantum states.* arXiv: `2008.01039 [cs.ET]`.

D. Noveck, Ed. and C. Lever (Aug. 2020). *Network File System (NFS) Version 4 Minor Version 1 Protocol.* RFC 8881. URL: `https://www.rfc-editor.org/rfc/rfc8881.html` (visited on 11/23/2020).

Dauphin, Yann, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio (2014). *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization.* arXiv: `1406.2572 [cs.LG]`.

Davies, Mike (2019). "Benchmarks for progress in neuromorphic computing". In: *Nature Machine Intelligence* 1.9, pp. 386–388.

Davies, Mike, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. (2018). "Loihi: A neuromorphic manycore processor with on-chip learning". In: *IEEE Micro* 38.1, pp. 82–99.

Davies, Mike, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A. Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R. Risbud (2021). "Advancing Neuromorphic Computing With Loihi: A Survey of Results and Outlook". In: *Proceedings of the IEEE*, pp. 1–24. DOI: `10.1109/JPROC.2021.3067593`.

Davis, A., J. Parikh, and W. E. Weihl (2004). "Edgecomputing: Extending Enterprise Applications to the Edge of the Internet". In: *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters.* WWW Alt. '04. New York, NY, USA: Association for Computing Machinery, pp. 180–187. ISBN: 1581139128. DOI: `10.1145/1013367.1013397`. URL: `https://doi.org/10.1145/1013367.1013397`.

Davison, A. P., D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger (2009). "PyNN: a common interface for neuronal network simulators". In: *Front. Neuroinform.* 2.11. DOI: `3389/neuro.11.011.2008`.

Dayan, Peter and L. F. Abbott (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems.* Cambride, Massachusetts: The MIT press. ISBN: 0-262-04199-5.

Deane, Phyllis M and Phyllis M Deane (1979). *The first industrial revolution.* Cambridge University Press.

Delta V Software (2020). *Remote Call Framework 3.2.* URL: `https://www.deltavsoft.com/doc/index.html` (visited on 11/27/2020).

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei (2009). "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee, pp. 248–255.

Diehl, Peter U, Guido Zarrella, Andrew Cassidy, Bruno U Pedroni, and Emre Neftci (2016). "Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware". In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8.

Diesmann, Markus and Marc-Oliver Gewaltig (2002). "NEST: An Environment for Neural Systems Simulations". In: *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001.* Ed. by Theo Plesser and Volker Macho. Vol. 58. GWDG-Bericht. Göttingen: Ges. für Wiss. Datenverarbeitung, pp. 43–70.

Doherty, Rina A and Paul Sorenson (2015). “Keeping users in the flow: mapping system responsiveness with user experience”. In: *Procedia Manufacturing* 3, pp. 4384–4391.

Dold, Dominik (2020). “Harnessing function from form: towards bio-inspired artificial intelligence in neuronal substrates”. PhD thesis. Universität Heidelberg.

Dold, Dominik, Ilja Bytschok, Akos F Kungl, Andreas Baumbach, Oliver Breitwieser, Walter Senn, Johannes Schemmel, Karlheinz Meier, and Mihai A Petrovici (2019). “Stochasticity from function—Why the Bayesian brain may need no noise”. In: *Neural Networks* 119, pp. 200–213.

Dolstra, Eelco, Merijn De Jonge, Eelco Visser, et al. (2004). “Nix: A Safe and Policy-Free System for Software Deployment.” In: *LISA*. Vol. 4, pp. 79–92.

Dongarra, Jack, Steven Gottlieb, and William T. C. Kramer (2019). “Race to Exascale”. In: *Computing in Science & Engineering* 21.1, pp. 4–5. DOI: `10.1109/MCSE.2018.2882574`.

Duan, Charles (2010). *Understanding Git Conceptually*. URL: `https://www.sbf5.com/~cduan/technical/git/` (visited on 04/28/2021).

Dubois, P. F., T. Epperly, and G. Kumfert (2003). “Why Johnny can’t build [portable scientific software]”. In: *Computing in Science Engineering* 5.5, pp. 83–88. DOI: `10.1109/MCISE.2003.1225867`.

Eklund, Anders, Thomas E. Nichols, and Hans Knutsson (2016). “Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates”. In: *Proceedings of the National Academy of Sciences* 113.28, pp. 7900–7905. ISSN: 0027-8424. DOI: `10.1073/pnas.1602413113`. eprint: `https://www.pnas.org/content/113/28/7900.full.pdf`. URL: `https://www.pnas.org/content/113/28/7900`.

Emmel, Arne (Nov. 2020). “Inference with Convolutional Neural Networks on Analog Neuromorphic Hardware”. Master’s Thesis. Universität Heidelberg.

Epperly, Tom, Chris White, Lorin Hochstein, and Prakashkumar Thiagarajan (2010). *Mix-Down: Meta-build tool for managing collections of third-party libraries*. URL: `https://github.com/tepperly/MixDown` (visited on 12/06/2020).

Esmaeilzadeh, Hadi, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger (2011). “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, pp. 365–376.

Esser, Steve K, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha (2015). “Backpropagation for energy-efficient neuromorphic computing”. In: *Advances in Neural Information Processing Systems*, pp. 1117–1125.

Esser, Steven K, Paul A Merolla, John V Arthur, Andrew S Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J Berg, Jeffrey L McKinstry, Timothy Melano, Davis R Barch, et al. (2016). “From the Cover: Convolutional networks for fast, energy-efficient neuromorphic computing”. In: *Proceedings of the National Academy of Sciences of the United States of America* 113.41, p. 11441.

Fagan, M. E. (1976). “Design and code inspections to reduce errors in program development”. In: *IBM Systems Journal* 15.3, pp. 182–211. DOI: `10.1147/sj.153.0182`.

Fei-Fei, Li, Rob Fergus, and Pietro Perona (2006). “One-shot learning of object categories”. In: *IEEE transactions on pattern analysis and machine intelligence* 28.4, pp. 594–611.

Felter, W., A. Ferreira, R. Rajamony, and J. Rubio (2015). “An updated performance comparison of virtual machines and Linux containers”. In: *2015 IEEE International Sym-*

posium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172. doi: 10.1109/ISPASS.2015.7095802.

Fischbach, Gerald D. (1992). “Mind and Brain”. In: *Scientific American* 267.3, pp. 48–59. issn: 00368733, 19467087. url: http://www.jstor.org/stable/24939212.

Fischer, Carola (2016). “Accelerated Classification in Hierarchical Neural Networks on Neuromorphic Hardware”. Bachelorarbeit. Universität Heidelberg.

Fogel, Itzhak and Dov Sagi (1989). “Gabor filters as texture discriminator”. In: *Biological cybernetics* 61.2, pp. 103–113.

Fox, Brian (Aug. 1988). *bash: Bourne-Again Shell (tarball).* url: http://ftp.gnu.org/gnu/bash/bash-1.14.7.tar.gz (visited on 12/01/2020).

FP Complete (June 2015). *stack 0.1 released.* url: https://www.fpcomplete.com/blog/2015/06/stack-0-1-release/ (visited on 12/01/2020).

Free Software Foundation (Nov. 2020). *Binutils website.* Version 2.35.1. url: http://www.gnu.org/software/binutils/ (visited on 11/16/2020).

Frenkel, Charlotte, Jean-Didier Legat, and David Bol (2020). “A 28-nm Convolutional Neuromorphic Processor Enabling Online Learning with Spike-Based Retinas”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5.

Friedmann, S., J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier (2017). “Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System”. In: *IEEE Transactions on Biomedical Circuits and Systems* 11.1, pp. 128–142. issn: 1932-4545. doi: 10.1109/TBCAS.2016.2579164.

Friedmann, Simon (2013). “A New Approach to Learning in Neuromorphic Hardware”. PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Friedmann, Simon, Nicolas Frémaux, Johannes Schemmel, Wulfram Gerstner, and Karlheinz Meier (2013). “Reward-based learning under hardware constraints — using a RISC processor embedded in a neuromorphic substrate”. In: *Frontiers in Neuroscience* 7, p. 160. issn: 1662-453X. doi: 10.3389/fnins.2013.00160. url: http://journal.frontiersin.org/article/10.3389/fnins.2013.00160.

Furber, Steve (2016). “Large-scale neuromorphic computing systems”. In: *Journal of neural engineering* 13.5, p. 051001.

Furber, Steve B, Francesco Galluppi, Steve Temple, and Luis A Plana (2014). “The spinnaker project”. In: *Proceedings of the IEEE* 102.5, pp. 652–665.

Furlani, John L (1991). “Modules: Providing a flexible user environment”. In: *Proceedings of the fifth large installation systems administration conference (LISA V)*, pp. 141–152.

Fuster, Valentin, Lars E. Rydén, David S. Cannom, Harry J. Crijns, Anne B. Curtis, Kenneth A. Ellenbogen, Jonathan L. Halperin, Jean-Yves Le Heuzey, G. Neal Kay, James E. Lowe, S. Bertil Olsson, Eric N. Prystowsky, Juan Luis Tamargo, and Samuel Wann (2006). “ACC/AHA/ESC 2006 Guidelines for the Management of Patients With Atrial Fibrillation—Executive Summary: A Report of the American College of Cardiology/American Heart Association Task Force on Practice Guidelines and the European Society of Cardiology Committee for Practice Guidelines (Writing Committee to Revise the 2001 Guidelines for the Management of Patients With Atrial Fibrillation) Developed in Collaboration With the European Heart Rhythm Association and the Heart Rhythm Society”. In: *Journal of the American College of Cardiology* 48.4, pp. 854–906. doi: 10.1016/j.jacc.2006.07.009.

Gamblin, Todd, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral (2015). "The Spack Package Manager: Bringing Order to HPC Software Chaos". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 40:1–40:12. ISBN: 978-1-4503-3723-6. DOI: `10.1145/2807591.2807623`.

Geimer, M., K. Hoste, and R. McLay (2014). "Modern Scientific Software Management Using EasyBuild and Lmod". In: *2014 First International Workshop on HPC User Support Tools*, pp. 41–51. DOI: `10.1109/HUST.2014.8`.

Georgiou, Yiannis and Matthieu Hautreux (2013). "Evaluating Scalability and Efficiency of the Resource and Job Management System on Large HPC Clusters". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 134–156. ISBN: 978-3-642-35867-8.

Gerstner, Wulfram (1998). "Spiking neurons". In: *MIT Press.*

– (2001). "What is different with spiking neurons?" In: *Plausible neural networks for biological modelling*, pp. 23–48.

Gerstner, Wulfram and Werner Kistler (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press.

Gerstner, Wulfram and Richard Naud (2009). "How good are neuron models?" In: *Science* 326.5951, pp. 379–380.

Gilb, Tom, Dorothy Graham, and Susannah Finzi (1993). *Software Inspection*. 5th. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201631814.

Godlove, David (2019). "Singularity: Simple, Secure Containers for Compute-Driven Workloads". In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*. PEARC '19. Chicago, IL, USA: Association for Computing Machinery. ISBN: 9781450372275. DOI: `10.1145/3332186.3332192`. URL: `https://doi.org/10.1145/3332186.3332192`.

Göltz, Julian (2019). "Training Deep Networks with Time-to-First-Spike Coding on the BrainScaleS Wafer-Scale System". Masterarbeit. Universität Heidelberg.

Göltz, Julian, Andreas Baumbach, Sebastian Billaudelle, Oliver Breitwieser, Dominik Dold, Laura Kriener, Akos Ferenc Kungl, Walter Senn, Johannes Schemmel, Karlheinz Meier, and Mihai Alexandru Petrovici (2019). *Fast and deep neuromorphic learning with time-to-first-spike coding*. arXiv: `1912.11443 [cs.NE]`.

Göltz, Julian, Laura Kriener, Andreas Baumbach, Sebastian Billaudelle, Oliver Breitwieser, Benjamin Cramer, Dominik Dold, Akos Ferenc Kungl, Walter Senn, Johannes Schemmel, Karlheinz Meier, and Mihai Alexandru Petrovici (2021). *Fast and energy-efficient neuromorphic deep learning with first-spike times*. arXiv: `1912.11443 [cs.NE]`.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT press.

Goodstein, David (2010). *On fact and fraud: Cautionary tales from the front lines of science*. Princeton University Press.

Goswami, Pronnoy, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao (2020). "Investigating The Reproducibility of NPM Packages". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 677–681. DOI: `10.1109/ICSME46990.2020.00071`.

Graham, Richard L., Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine (Sept. 2006). “Open MPI: A High-Performance, Heterogeneous MPI”. In: *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*. Barcelona, Spain.

Grant, W. Shane and Randolph Voorhies (2017). *cereal - A C++11 library for serialization*. URL: `http://uscilab.github.io/cereal/` (visited on 11/16/2020).

Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton (2013). “Speech recognition with deep recurrent neural networks”. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649. DOI: `10.1109/ICASSP.2013.6638947`.

Großkinsky, Marcel (2016). “Neural Sampling with Linear Feedback Shift Registers as a Source of Noise”. Bachelorarbeit. Universität Heidelberg.

Grübl, Andreas, Sebastian Billaudelle, Benjamin Cramer, Vitali Karasenko, and Johannes Schemmel (2020). “Verification and Design Methods for the BrainScaleS Neuromorphic Hardware System”. In: *arXiv preprint*. URL: `http://arxiv.org/abs/2003.11455`.

Guo, Shangqi, Zhaofei Yu, Fei Deng, Xiaolin Hu, and Feng Chen (2017). “Hierarchical bayesian inference and learning in spiking neural networks”. In: *IEEE transactions on cybernetics* 49.1, pp. 133–145.

Gütig, Robert and Haim Sompolinsky (Mar. 2006). “The tempotron: a neuron that learns spike timing-based decisions”. In: *Nat Neurosci* 9.3, pp. 420–428. ISSN: 1097-6256. URL: `http://dx.doi.org/10.1038/nn1643`.

Guyton, A.C. and J.E. Hall (2006). *Textbook of Medical Physiology*. Elsevier Saunders. ISBN: 9780721602400.

Harris, Trey (2020). *Gerrit Code Review*. URL: `https://www.gerritcodereview.com/about.html` (visited on 11/23/2020).

Hashimoto, Mitchell (2013). *Vagrant: up and running: create and manage virtualized development environments*. " O’Reilly Media, Inc."

Hassabis, Demis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick (2017). “Neuroscience-inspired artificial intelligence”. In: *Neuron* 95.2, pp. 245–258.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034.

Heimbrecht, Arthur (Mar. 2017). “Compiler Support for the BrainScaleS Plasticity Processor”. Bachelorarbeit. Universität Heidelberg.

Hinton, G., Li Deng, Dong Yu, G.E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T.N. Sainath, and B. Kingsbury (Nov. 2012a). “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups”. In: *Signal Processing Magazine, IEEE* 29.6, pp. 82–97. ISSN: 1053-5888. DOI: `10.1109/MSP.2012.2205597`.

Hinton, Geoffrey E, Simon Osindero, and Yee-Whye Teh (2006). “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7, pp. 1527–1554.

Hinton, Geoffrey E, Terrence J Sejnowski, and David H Ackley (1984). “Boltzmann machines: Constraint satisfaction networks that learn”. In: *Carnegie-Mellon University, Department of Computer Science Pittsburgh*.

Hinton, Geoffrey E., Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov (2012b). *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv: `1207.0580 [cs.NE]`.

Hock, M., A. Hartel, J. Schemmel, and K. Meier (Sept. 2013). "An analog dynamic memory array for neuromorphic hardware". In: *Circuit Theory and Design (ECCTD), 2013 European Conference on*, pp. 1–4. DOI: `10.1109/ECCTD.2013.6662229`.

Hock, Matthias (2014). "Modern Semiconductor Technologies for Neuromorphic Hardware". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Hoste, Kenneth, Jens Timmerman, Andy Georges, and Stijn De Weirdt (2012). "EasyBuild: Building Software with Ease". In: *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. SCC '12. USA: IEEE Computer Society, pp. 572–582. ISBN: 9780769549569. DOI: `10.1109/SC.Companion.2012.81`. URL: `https://doi.org/10.1109/SC.Companion.2012.81`.

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio (2017). "Quantized neural networks: Training neural networks with low precision weights and activations". In: *The Journal of Machine Learning Research* 18.1, pp. 6869–6898.

Huh, Dongsung and Terrence J Sejnowski (2018). "Gradient Descent for Spiking Neural Networks". In: *Advances in Neural Information Processing Systems 31*, pp. 1433–1443.

Husmann, Kai-Hajo (2012). "Handling Spike Data in an Accelerated Neuromorphic System". BSc Thesis. Ruprecht-Karls-Universität Heidelberg.

IEEE (2001). "IEEE Standard Test Access Port and Boundary-Scan Architecture". In: *IEEE Std 1149.1-2001*, pp. i–200. DOI: `10.1109/IEEESTD.2001.92950`.

"IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language" (2018). In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315. DOI: `10.1109/IEEESTD.2018.8299595`.

Inc, Mattermost (2015). *Mattermost Changelog*. URL: `https://docs.mattermost.com/administration/changelog.html` (visited on 12/15/2020).

Inc., Continuum Analytics (Nov. 2016). *Anaconda Software Distribution*. Version Vers. 2-2.4.0. URL: `https://docs.anaconda.com/` (visited on 12/02/2020).

– (2017). *Conda: A Cross-Platform, Python- Agnostic Binary Package Manager*. URL: `http://conda.pydata.org` (visited on 12/02/2020).

Indiveri, Giacomo and Timothy K Horiuchi (2011a). "Frontiers in neuromorphic engineering". In: *Frontiers in neuroscience* 5, p. 118.

Indiveri, Giacomo, Bernabe Linares-Barranco, Tara Julia Hamilton, André van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, Johannes Schemmel, Gert Cauwenberghs, John Arthur, Kai Hynna, Fopefolu Folowosele, Sylvain Saighi, Teresa Serrano-Gotarredona, Jayawan Wijekoon, Yingxue Wang, and Kwabena Boahen (2011b). "Neuromorphic silicon neuron circuits". In: *Frontiers in Neuroscience* 5.0. ISSN: 1662-453X. DOI: `10.3389/fnins.2011.00073`. URL: `http://www.frontiersin.org/Journal/Abstract.aspx?s=755&name=neuromorphic%20engineering&ART_DOI=10.3389/fnins.2011.00073`.

International, ECMA (2020). *ECMAScript® 2020 Language Specification*. URL: `https://www.ecma-international.org/ecma-262/11.0/` (visited on 12/14/2020).

ISO (2005). *ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE).* Geneva, CH.

– (Dec. 2017). *ISO/IEC 14882:2017 Information technology — Programming languages — C++.* Fifth. Geneva, Switzerland: International Organization for Standardization, p. 1605. URL: `https://www.iso.org/standard/68564.html` (visited on 12/01/2020).

ISO/IEC (Nov. 2018). *ISO/IEC 1539-1:2018 Information technology — Programming languages — Fortran — Part 1: Base language.* Geneva, CH. URL: `https://www.iso.org/standard/72320.html` (visited on 11/26/2020).

ISO (June 2018). *Information technology — Programming languages — C.* Standard. Geneva, CH. URL: `https://www.iso.org/standard/74528.html` (visited on 12/01/2020).

Ivakhnenko, Alexey Grigorevich (1971). "Polynomial theory of complex systems". In: *IEEE transactions on Systems, Man, and Cybernetics* 4, pp. 364–378.

Izhikevich, Eugene M (2004). "Which model to use for cortical spiking neurons?" In: *IEEE Transactions on Neural Networks* 15.5, pp. 1063–1070.

Jaeger, Herbert (2021). "Toward a generalized theory comprising digital, neuromorphic, and unconventional computing". In: *Neuromorphic Computing and Engineering.* URL: `http://iopscience.iop.org/article/10.1088/2634-4386/abf151`.

Jakob, Wenzel, Jason Rhinelander, and Dean Moldovan (2019). *pybind11 – Seamless operability between C++11 and Python.* https://github.com/pybind/pybind11.

Jeltsch, Sebastian (2014). "A Scalable Workflow for a Configurable Neuromorphic Platform". PhD thesis. Universität Heidelberg.

Jo, Sung Hyun, Ting Chang, Idongesit Ebong, Bhavitavya B Bhadviya, Pinaki Mazumder, and Wei Lu (2010). "Nanoscale memristor device as synapse in neuromorphic systems". In: *Nano letters* 10.4, pp. 1297–1301.

Johnston, Phillip and Rozi Harris (2019). "The Boeing 737 MAX saga: lessons for software organizations". In: *Software Quality Professional* 21.3, pp. 4–12.

Jon Postel, Ed. (Sept. 1981). *Transmission Control Protocol.* RFC 793. URL: `https://www.rfc-editor.org/rfc/rfc793.html` (visited on 02/10/2021).

Jones, C. (1994). "Assessment and control of software risks". In: *Yourdon Press Computing Series.*

Jones, Isaac, Simon Peyton Jones, Simon Marlow, Malcolm Wallace, and Ross Patterson (2005). *The Haskell Cabal – A Common Architecture for Building Applications and Tools.* URL: `https://www.haskell.org/cabal/proposal/index.html` (visited on 12/01/2020).

Jones, Nick and Mark R Fahey (2008). "Design, Implementation, and Experiences of Third-Party Software Administration at the ORNL NCCS". In: *Proceedings of the 50th Cray User Group (CUG08).*

Jones, Steven L, Andrew J Sullivan, Naveen Cheekoti, Michael D Anderson, and Dilip Malave (2004). "Traffic simulation software comparison study". In: *UTCA report* 2217.

Jordan, Jakob, Mihai A Petrovici, Oliver Breitwieser, Johannes Schemmel, Karlheinz Meier, Markus Diesmann, and Tom Tetzlaff (2019). "Deterministic networks for probabilistic computing". In: *Scientific Reports* 9.1, pp. 1–17.

Jouppi, Norman P, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. (2017). "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12.

Justesen, Niels and Sebastian Risi (2017). "Learning macromanagement in starcraft from replays using deep learning". In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 162–169. DOI: `10.1109/CIG.2017.8080430`.

Kacher, Ilyes, Maxime Portaz, Hicham Randrianarivo, and Sylvain Peyronnet (Feb. 2020). "Graphcore c2 card performance for image-based deep learning application: A report". In: *arXiv preprint*. arXiv: `2002.11670 [cs.CV]`.

Kaiser, G. E., D. E. Perry, and W. M. Schell (1989). "Infuse: fusing integration test management with change management". In: *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*, pp. 552–558. DOI: `10.1109/CMPSAC.1989.65147`.

Kaiser, Jakob (2020). "Implementation of Large Scale Neural Networks on the Neuromorphic BrainScaleS-1 System". Masterarbeit. Universität Heidelberg.

Karasenko, Vitali (2020). "Von Neumann bottlenecks in non-von Neumann computing architectures". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Karpathy, Andrej and Li Fei-Fei (2015). "Deep visual-semantic alignments for generating image descriptions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3128–3137.

Katz, Yehuda, Carl Lerche, and Alex Crichton (Mar. 2014). *Cargo, a package manager for Rust*. URL: `https://github.com/rust-lang/cargo`.

Kawaguchi, Kohsuke and CDF (2011). *Jenkins website*. URL: `https://jenkins.io` (visited on 11/18/2020).

Kerrisk, Michael (2013). "Namespaces in operation, part 5: User namespaces". In: *Linux Weekly News*. URL: `https://lwn.net/Articles/532593/` (visited on 05/03/2021).

Kharraz, Amin, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda (2015). "Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Magnus Almgren, Vincenzo Gulisano, and Federico Maggi. Cham: Springer International Publishing, pp. 3–24. ISBN: 978-3-319-20550-2.

Kheradpisheh, Saeed Reza and Timothée Masquelier (2019). "S4NN: temporal backpropagation for spiking neural networks with one spike per neuron". In: *arXiv preprint arXiv:1910.09495*.

Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. arXiv: `1412.6980 [cs.LG]`.

Kivity, Avi, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori (2007). "kvm: the Linux virtual machine monitor". In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada, pp. 225–230.

Klähn, Johann (2017). "Training Functional Networks on Large-Scale Neuromorphic Hardware". Master. Universität Heidelberg.

Klähn, Johann and Electronic Vision(s) (Mar. 2020). *genpybind 0.2.1*. Version v0.2.1. DOI: `10.5281/zenodo.3726274`. URL: `https://doi.org/10.5281/zenodo.3726274`.

Kleider, Mitja (2017). "Neuron Circuit Characterization in a Neuromorphic System". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Koke, Christoph (2017). “Device variability in synapses of neuromorphic circuits”. PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Kononov, Alex (2011). “Testing of an Analog Neuromorphic Network Chip”. HD-KIP-11-83. Diploma thesis. Ruprecht-Karls-Universität Heidelberg.

Korcsak-Gorzo, Agnes (2017). “Simulated Tempering in Spiking Neural Networks”. In: *Masterarbeit. Universität Heidelberg.*

Korcsak-Gorzo, Agnes, Michael G. Müller, Andreas Baumbach, Luziwei Leng, Oliver Julien Breitwieser, Sacha J. van Albada, Walter Senn, Karlheinz Meier, Robert Legenstein, and Mihai A. Petrovici (2021). *Cortical oscillations implement a backbone for sampling-based computation in spiking neural networks.* arXiv: `2006.11099 [q-bio.NC]`.

Krafczyk, M. S., A. Shi, A. Bhaskar, D. Marinov, and V. Stodden (2021). “Learning from reproducing computational results: introducing three principles and the “Reproduction Package””. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379.2197, p. 20200069. DOI: `10.1098/rsta.2020.0069`. URL: `https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2020.0069`.

Krafczyk, Matthew, August Shi, Adhithya Bhaskar, Darko Marinov, and Victoria Stodden (2019). “Scientific Tests and Continuous Integration Strategies to Enhance Reproducibility in the Scientific Software Context”. In: *Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems.* P-RECS ’19. Phoenix, AZ, USA: Association for Computing Machinery, pp. 23–28. ISBN: 9781450367561. DOI: `10.1145/3322790.3330595`. URL: `https://doi.org/10.1145/3322790.3330595`.

Kreiser, Raphaela, Timoleon Moraitis, Yulia Sandamirskaya, and Giacomo Indiveri (2017). “On-chip unsupervised learning in winner-take-all networks of spiking neurons”. In: *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS).* IEEE, pp. 1–4.

Kriener, Laura, Julian Göltz, and Mihai A. Petrovici (2021). “The Yin-Yang dataset”. In: *arXiv.* URL: `https://arxiv.org/abs/2102.08211`.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25.* Ed. by F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger. Curran Associates, Inc., pp. 1097–1105. URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

Kungl, Akos (2016). “Sampling with leaky integrate-and-fire neurons on the hicannv4 neuromorphic chip”. In: *Masterarbeit, Universität Heidelberg.*

Kungl, Akos F., Sebastian Schmitt, Johann Klähn, Paul Müller, Andreas Baumbach, Dominik Dold, Alexander Kugele, Eric Müller, Christoph Koke, Mitja Kleider, Christian Mauch, Oliver Breitwieser, Luziwei Leng, Nico Gürtler, Maurice Güttler, Dan Husmann, Kai Husmann, Andreas Hartel, Vitali Karasenko, Andreas Grübl, Johannes Schemmel, Karlheinz Meier, and Mihai A. Petrovici (2019). “Accelerated Physical Emulation of Bayesian Inference in Spiking Neural Networks”. In: *Frontiers in Neuroscience* 13, p. 1201. ISSN: 1662-453X. DOI: `10.3389/fnins.2019.01201`. URL: `https://www.frontiersin.org/article/10.3389/fnins.2019.01201`.

Kungl, Ákos Ferenc (2020). “Robust learning algorithms for spiking and rate-based neural networks”. PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Kurtzer, Gregory M., Vanessa Sochat, and Michael W. Bauer (May 2017). “Singularity: Scientific containers for mobility of compute”. In: *PLOS ONE* 12.5, pp. 1–20. DOI: 10.1371/journal.pone.0177459.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (May 2015). “Deep learning”. In: *Nature* 521.7553, pp. 436–444. ISSN: 0028-0836. DOI: http://dx.doi.org/10.1038/nature1453910.1038/nature14539.

LeCun, Yann and Corinna Cortes (1998). *The MNIST database of handwritten digits.*

Lee, Jong-Ho, Sung Yun Woo, Sung-Tae Lee, Suhwan Lim, Won-Mook Kang, Young-Tak Seo, Soochang Lee, Dongseok Kwon, Seongbin Oh, Yoohyun Noh, et al. (2019). “Review of candidate devices for neuromorphic applications”. In: *ESSDERC 2019-49th European Solid-State Device Research Conference (ESSDERC).* IEEE, pp. 22–27.

Lee, Sang Wan, John P. O’Doherty, and Shinsuke Shimojo (Apr. 2015). “Neural Computations Mediating One-Shot Learning in the Human Brain”. In: *PLOS Biology* 13.4, pp. 1–36. DOI: 10.1371/journal.pbio.1002137. URL: https://doi.org/10.1371/journal.pbio.1002137.

Leng, Luziwei, Roman Martel, Oliver Breitwieser, Ilja Bytschok, Walter Senn, Johannes Schemmel, Karlheinz Meier, and Mihai A Petrovici (2018). “Spiking neurons with short-term synaptic plasticity form superior generative networks”. In: *Scientific Reports* 8.1, pp. 1–11.

Leshno, Moshe, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken (1993). “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function”. In: *Neural networks* 6.6, pp. 861–867.

Leveson, N. G. and C. S. Turner (1993). “An investigation of the Therac-25 accidents”. In: *Computer* 26.7, pp. 18–41.

Li, Yibo, Zhongrui Wang, Rivu Midya, Qiangfei Xia, and J Joshua Yang (2018). “Review of memristor devices in neuromorphic computing: materials sciences and device challenges”. In: *Journal of Physics D: Applied Physics* 51.50, p. 503002.

Lin, Chit-Kwan, Andreas Wild, Gautham N Chinya, Yongqiang Cao, Mike Davies, Daniel M Lavery, and Hong Wang (2018). “Programming Spiking Neural Networks on Intel’s Loihi”. In: *Computer* 51.3, pp. 52–61.

Lin, Ji, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han (2020). *MCUNet: Tiny Deep Learning on IoT Devices.* arXiv: 2007.10319 [cs.CV].

Lin, Jian-Liang and Hong-Sen Yan (2016). *Decoding the mechanisms of antikythera astronomical device.* Springer. ISBN: 978-3-662-48447-0.

Linnainmaa, Seppo (1970). “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. In: *Master’s Thesis (in Finnish), Univ. Helsinki*, pp. 6–7.

Lovelace, Augusta Ada (Aug. 1843). “Notes [on translation of Luigi Federico Menebrae’s paper on Babbage’s Analytical Engine]”. In: *Taylor’s Scientific Memoirs.*

Lowe, D.G. (1999). “Object recognition from local scale-invariant features”. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision.* Vol. 2, 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410.

Luederitz, Berndt (1993). *Geschichte der Herzrhythmusstörungen — Von der antiken Pulslehre zum implantierbaren Defibrillator.* Springer-Verlag Berlin Heidelberg. ISBN: 978-3-642-77940-4. DOI: 10.1007/978-3-642-77940-4.

Maass, Wolfgang (2016). “Searching for principles of brain computation”. In: *Current Opinion in Behavioral Sciences* 11, pp. 81–92.

Markram, H. (2012). “The Human Brain Project”. In: *Scientific American* 306.6, pp. 50–55.

Markram, H., A. Gupta, A. Uziel, Y. Wang, and M. Tsodyks (1998). “Information processing with frequency-dependent synaptic connections.” In: *Neurobiol Learn Mem* 70.1-2, pp. 101–112.

Marlow, Simon et al. (2010). *Haskell 2010 Language Report.* URL: `https://www.haskell.org/onlinereport/haskell2010/` (visited on 12/01/2020).

Matsakis, Nicholas D. and Felix S. Klock (Oct. 2014). “The Rust Language”. In: *Ada Lett.* 34.3, pp. 103–104. ISSN: 1094-3641. DOI: `10.1145/2692956.2663188`. URL: `https://doi.org/10.1145/2692956.2663188`.

Mayr, Christian, Sebastian Hoeppner, and Steve Furber (2019). “SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning”. In: *arXiv preprint arXiv:1911.02385.*

McCulloch, Warren S. and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity”. In: *Bulletin of Mathematical Biophysics*, pp. 127–147.

McLay, Robert, Karl W. Schulz, William L. Barth, and Tommy Minyard (2011). “Best Practices for the Deployment and Management of Production HPC Clusters”. In: *State of the Practice Reports.* SC ’11. Seattle, Washington: Association for Computing Machinery. ISBN: 9781450311397. DOI: `10.1145/2063348.2063360`. URL: `https://doi.org/10.1145/2063348.2063360`.

Mead, C. A. (1989). *Analog VLSI and Neural Systems.* Reading, MA: Addison Wesley.

– (1990). “Neuromorphic Electronic Systems”. In: *Proceedings of the IEEE* 78, pp. 1629–1636.

Medina, Eitan and Eran Dagan (2020). “Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor”. In: *IEEE Micro* 40.2, pp. 17–24. DOI: `10.1109/MM.2020.2975185`.

Mehta, Pankaj, Marin Bukov, Ching-Hao Wang, Alexandre GR Day, Clint Richardson, Charles K Fisher, and David J Schwab (2019). “A high-bias, low-variance introduction to machine learning for physicists”. In: *Physics reports.*

Menabrea, Luigi F. (1842). “Notions sur la Machine Analytique de M. Charles Babbage”. In: *Bibliothèque Universelle de Genève* 41, pp. 352–376.

Merkel, Dirk (2014). “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239, p. 2.

Mesnard, Olivier and Lorena A. Barba (2016). *Reproducible and replicable CFD: it’s harder than you think.* arXiv: `1605.04339 [physics.comp-ph]`.

Millner, Sebastian (Nov. 2012). “Development of a Multi-Compartment Neuron Model Emulation”. PhD thesis. Ruprecht-Karls University Heidelberg. URL: `http://www.ub.uni-heidelberg.de/archiv/13979`.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller (2013). “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602.*

Moore, Gordon E et al. (1965). *Cramming more components onto integrated circuits.*

Moradi, S. and G. Indiveri (Feb. 2014). “An Event-Based Neural Network Architecture With an Asynchronous Programmable Synaptic Memory”. In: *IEEE Transactions on*

Biomedical Circuits and Systems 8.1, pp. 98–107. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2013.2255873.

Moradi, Saber, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri (2018). "A scalable multicore architecture with heterogeneous memory structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)". In: *IEEE Trans. Biomed. Circuits Syst.* 12.1, pp. 106–122.

Mostafa, H., B. U. Pedroni, S. Sheik, and G. Cauwenberghs (May 2017). "Fast classification using sparsely active spiking networks". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4. DOI: 10.1109/ISCAS.2017.8050527.

Mostafa, Hesham (2017). "Supervised learning based on temporal coding in spiking neural networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.7, pp. 3227–3235.

Müller, Eric, Christian Mauch, Philipp Spilger, Oliver Julien Breitwieser, Johann Klähn, David Stöckel, Timo Wunderlich, and Johannes Schemmel (Mar. 2020a). "Extending BrainScaleS OS for BrainScaleS-2". In: *arXiv preprint*. arXiv: 2003.13750 [cs.NE]. URL: http://arxiv.org/abs/2003.13750.

Müller, Eric, Sebastian Schmitt, Christian Mauch, Hartmut Schmidt, José Montes, Joscha Ilmberger, Johann Klähn, Felix Passenberg, Christoph Koke, Mitja Kleider, Sebastian Jeltsch, Maurice Güttler, Dan Husmann, Sebastian Billaudelle, Paul Müller, Andreas Grübl, Jakob Kaiser, Jonas Weidner, Bernhard Vogginger, Johannes Partzsch, Christian Mayr, and Johannes Schemmel (Mar. 2020b). "The Operating System of the Neuromorphic BrainScaleS-1 System". In: *arXiv preprint*. arXiv: 2003.13749 [cs.NE]. URL: http://arxiv.org/abs/2003.13749.

Müller, Eric, Philipp Spilger, Christian Mauch, Elias Arnold, Yannik Stradmann, Oliver Julien Breitwieser, Milena Czierlinski, Andreas Baumbach, Sebastian Billaudelle, Benjamin Cramer, Philipp Dauer, Falk Leonard Ebert, Arne Emmel, Joscha Ilmberger, Jakob Kaiser, Simeon Kanya, Vitali Karasenko, Aron Leibfried, Christian Pehle, Sebastian Schmitt, Korbinian Schreiber, Raphael Stock, Johannes Weis, and Johannes Schemmel (Oct. 2021). "A scalable approach to modeling on accelerated neuromorphic hardware". In: *Frontiers in Neuromorphic Engineering (planned)*.

Müller, Eric Christian (2014). "Novel Operation Modes of Accelerated Neuromorphic Hardware". HD-KIP 14-98. PhD thesis. Ruprecht-Karls-Universität Heidelberg. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=3112.

Müller, Johann Helfrich von and Ph. E. Klipstein (1786). *Beschreibung seiner neu-erfundenen Rechenmaschine, nach ihrer Gestalt, ihrem Gebrauch und Nutzen*. Frankfurt und Mainz : bey Varrentrapp Sohn und Wenner. DOI: https://doi.org/10.3931/e-rara-12979/.

Nagy, Thomas (2005). *waf: The meta build system*. URL: https://waf.io.

National Academies of Sciences, Engineering and Medicine (2019). *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press. ISBN: 978-0-309-48616-3. DOI: 10.17226/25303. URL: https://www.nap.edu/catalog/25303/reproducibility-and-replicability-in-science.

Navarro, Cristobal A, Nancy Hitschfeld-Kahler, and Luis Mateu (2014). "A survey on parallel computing and its applications in data-parallel problems using GPU architectures". In: *Communications in Computational Physics* 15.2, pp. 285–329.

Nawrocki, Robert A, Richard M Voyles, and Sean E Shaheen (2016). "A mini review of neuromorphic architectures and implementations". In: *IEEE Transactions on Electron Devices* 63.10, pp. 3819–3829.

Neckar, Alexander, Sam Fok, Ben V Benjamin, Terrence C Stewart, Nick N Oza, Aaron R Voelker, Chris Eliasmith, Rajit Manohar, and Kwabena Boahen (2018). "Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model". In: *Proceedings of the IEEE* 107.1, pp. 144–164.

Neftci, Emre, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs (2014). "Event-driven contrastive divergence for spiking neuromorphic systems". In: *Frontiers in Neuroscience* 7, p. 272.

Neftci, Emre O, Hesham Mostafa, and Friedemann Zenke (2019). "Surrogate gradient learning in spiking neural networks". In: *arXiv preprint arXiv:1901.09948*.

Neftci, Emre O, Bruno U Pedroni, Siddharth Joshi, Maruan Al-Shedivat, and Gert Cauwenberghs (2016). "Stochastic synapses enable efficient brain-inspired learning machines". In: *Frontiers in Neuroscience* 10, p. 241.

Neumann, J. von (1945). *First draft of a report on the EDVAC*. Tech. rep. Transscript in: M. D. Godfrey: Introduction to "The first draft report on the EDVAC" by John von Neumann. IEEE Annals of the History of Computing 15(4), 27–75 (1993). Moore School of Electrical Engeneering Library, University of Pennsylvania.

Ng, Andrew (2016). "What artificial intelligence can and can't do right now". In: *Harvard Business Review* 9.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron (2008). "Scalable parallel programming with CUDA". In: *Queue* 6.2, pp. 40–53.

Nieh, Jason and Ozgur Can Leonard (2000). "Examining vmware". In: *Dr. Dobb's Journal* 25.8, p. 70.

O'Regan, Gerard (2012). *A Brief History of Computing*. 2nd ed. 2012. London: Springer London. URL: `http://dx.doi.org/10.1007/978-1-4471-2359-0`.

Oliphant, Travis E. (2007). "Python for Scientific Computing". In: *IEEE Computing in Science and Engineering* 9.3, pp. 10–20.

Olshausen, Bruno A and David J Field (1996). "Emergence of simple-cell receptive field properties by learning a sparse code for natural images". In: *Nature* 381.6583, pp. 607–609.

OpenAI, : Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang (2019). *Dota 2 with Large Scale Deep Reinforcement Learning*. arXiv: `1912.06680 [cs.LG]`.

Owens, John D, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell (2007). "A survey of general-purpose computation on graphics hardware". In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library, pp. 80–113.

Pakkenberg, Bente, Dorte Pelvig, Lisbeth Marner, Mads J Bundgaard, Hans Jørgen G Gundersen, Jens R Nyengaard, and Lisbeth Regeur (2003). "Aging and the human neocortex". In: *Experimental gerontology* 38.1, pp. 95–99.

Park, Jongsoo, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy (2018). *Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications.* arXiv: `1811.09886 [cs.LG]`.

Passenberg, Felix Constantin (2019). "Improving the BrainScaleS-1 place and route software towards real world waferscale experiments". Masterarbeit. Universität Heidelberg.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32.* Curran Associates, Inc., pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Payeur, Alexandre, Jordan Guerguiev, Friedemann Zenke, Blake A. Richards, and Richard Naud (2020). "Burst-dependent synaptic plasticity can coordinate learning in hierarchical circuits". In: *bioRxiv 10.1101/2020.03.30.015511.* DOI: `10.1101/2020.03.30.015511`.

Peachey, Kevin (Apr. 2021). *Post Office scandal: What the Horizon saga is all about.* URL: `https://archive.is/LzkFQ` (visited on 04/24/2021).

Pei, Jing, Lei Deng, Sen Song, Mingguo Zhao, Youhui Zhang, Shuang Wu, Guanrui Wang, Zhe Zou, Zhenzhi Wu, Wei He, Feng Chen, Ning Deng, Si Wu, Yu Wang, Yujie Wu, Zheyu Yang, Cheng Ma, Guoqi Li, Wentao Han, Huanglong Li, Huaqiang Wu, Rong Zhao, Yuan Xie, and Luping Shi (Aug. 2019). "Towards artificial general intelligence with hybrid Tianjic chip architecture". en. In: *Nature* 572.7767, pp. 106–111.

Petrovici, Mihai A, Sebastian Schmitt, Johann Klähn, David Stöckel, Anna Schroeder, Guillaume Bellec, Johannes Bill, Oliver Breitwieser, Ilja Bytschok, Andreas Grübl, et al. (2017a). "Pattern representation and recognition with accelerated analog neuromorphic systems". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4.

Petrovici, Mihai A, Anna Schroeder, Oliver Breitwieser, Andreas Grübl, Johannes Schemmel, and Karlheinz Meier (2017b). "Robustness from structure: Inference with hierarchical spiking networks on analog neuromorphic hardware". In: *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2209–2216.

Petrovici, Mihai A, Bernhard Vogginger, Paul Müller, Oliver Breitwieser, Mikael Lundqvist, Lyle Muller, Matthias Ehrlich, Alain Destexhe, Anders Lansner, René Schüffny, Johannes Schemmel, and Karlheinz Meier (2014). "Characterization and Compensation of Network-Level Anomalies in Mixed-Signal Neuromorphic Modeling Platforms". In: *PLOS ONE* 9.10, e108590. DOI: `10.1371/journal.pone.0108590`.

Petrovici, Mihai A. (2016). *Form Versus Function: Theory and Models for Neuronal Substrates.* Springer. DOI: `10.1007/978-3-319-39552-4`.

Petrovici, Mihai A., Johannes Bill, Ilja Bytschok, Johannes Schemmel, and Karlheinz Meier (2013). "Stochastic inference with deterministic spiking neurons". In: *arXiv preprint arXiv:1311.3211.*

– (Oct. 2016). “Stochastic inference with spiking neurons in the high-conductance state”. In: *Physical Review E* 94.4. DOI: 10.1103/PhysRevE.94.042312. URL: http://journals.aps.org/pre/abstract/10.1103/PhysRevE.94.042312.

Pfeiffer, Michael and Thomas Pfeil (2018). “Deep learning with spiking neurons: opportunities and challenges”. In: *Frontiers in Neuroscience* 12.

Pfeil, Thomas, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A. Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier (2013). “Six networks on a universal neuromorphic computing substrate”. In: *Frontiers in Neuroscience* 7, p. 11. ISSN: 1662-453X. DOI: 10.3389/fnins.2013.00011. URL: http://www.frontiersin.org/neuromorphic_engineering/10.3389/fnins.2013.00011/abstract.

Philipp, Stefan (2008). “Design and Implementation of a Multi-Class Network Architecture for Hardware Neural Networks”. PhD thesis. Ruprecht-Karls Universität Heidelberg.

Pike, Rob (2009). “The go programming language”. In: *Talk given at Google's Tech Talks* 14.

Pike, Rob, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom (1995). “Plan 9 from bell labs”. In: *Computing systems* 8.3, pp. 221–254.

Posselt, Emanuel Anthony (1887). *The Jacquard machine analyzed and explained: with an appendix on the preparation of Jacquard cards.* Philadelphia, Pa., Published under the auspices of the school [Pennsylvania museum and school of industrial art]. URL: https://archive.org/details/jacquardmachinea00poss (visited on 10/29/2020).

Pouchard, Line, Sterling Baldwin, Todd Elsethagen, Shantenu Jha, Bibi Raju, Eric Stephan, Li Tang, and Kerstin Kleese Van Dam (2019). “Computational reproducibility of scientific workflows at extreme scales”. In: *The International Journal of High Performance Computing Applications* 33.5, pp. 763–776. DOI: 10.1177/1094342019839124. eprint: https://doi.org/10.1177/1094342019839124. URL: https://doi.org/10.1177/1094342019839124.

PowerISA (July 2010). *PowerISA Version 2.06 Revision B.* Specification. Power.org. URL: http://www.power.org/resources/reading/.

Pratt, Vaughan (1995). “Anatomy of the Pentium bug”. In: *TAPSOFT '95: Theory and Practice of Software Development.* Ed. by Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 97–107. ISBN: 978-3-540-49233-7.

Priedhorsky, Reid and Tim Randles (2017). “Charliecloud: Unprivileged containers for user-defined software stacks in hpc”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10.

Qiao, Ning, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawska, and Giacomo Indiveri (2015). “A Re-configurable On-line Learning Spiking Neuromorphic Processor comprising 256 neurons and 128K synapses”. In: *Frontiers in Neuroscience* 9.141. ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00141. URL: http://www.frontiersin.org/neuromorphic_engineering/10.3389/fnins.2015.00141/abstract.

Quinlan, Daniel, Paul Russell, and Christopher Yeoh (2004). *Filesystem hierarchy standard.* URL: https://www.pathname.com/fhs/pub/fhs-2.3.html (visited on 02/15/2021).

Raichle, Marcus E. and Debra A. Gusnard (2002). “Appraising the brain’s energy budget”. In: *Proceedings of the National Academy of Sciences* 99.16, pp. 10237–10239. ISSN: 0027-8424. DOI: 10.1073/pnas.172399499. eprint: https://www.pnas.org/content/99/16/10237.full.pdf. URL: https://www.pnas.org/content/99/16/10237.

Rajendran, Bipin, Abu Sebastian, Michael Schmuker, Narayan Srinivasa, and Evangelos Eleftheriou (2019). “Low-Power Neuromorphic Hardware for Signal Processing Applications: A review of architectural and system-level design approaches”. In: *IEEE Signal Processing Magazine* 36.6, pp. 97–110.

Ramachandran, Prajit, Barret Zoph, and Quoc V Le (2017). “Searching for activation functions”. In: *arXiv preprint arXiv:1710.05941*.

Raman, Aruna (Oct. 2019). “Towards an Automated Platform to Implement Artificial Neural Network Topologies on Neuromorphic Hardware”. Master’s thesis. Universität Heidelberg.

Rangayyan, Rangaraj M. (2002). *Biomedical Signal Analysis - A Case-Study Approach*. New York: Wiley. ISBN: 978-0-471-20811-2.

Rauch, Alexander, Giancarlo La Camera, Hans-Rudolf Luscher, Walter Senn, and Stefano Fusi (2003). “Neocortical pyramidal cells respond as integrate-and-fire neurons to in vivo–like input currents”. In: *Journal of Neurophysiology* 90.3, pp. 1598–1612.

Richards, Blake A, Timothy P Lillicrap, Philippe Beaudoin, Yoshua Bengio, Rafal Bogacz, Amelia Christensen, Claudia Clopath, Rui Ponte Costa, Archy de Berker, Surya Ganguli, et al. (2019). “A deep learning framework for neuroscience”. In: *Nature Neuroscience* 22.11, pp. 1761–1770.

Ritchie, D. M. and K. Thompson (1978). “The UNIX Time-Sharing System†”. In: *Bell System Technical Journal* 57.6, pp. 1905–1929. DOI: https://doi.org/10.1002/j.1538-7305.1978.tb02136.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1978.tb02136.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1978.tb02136.x.

Robbins, Herbert and Sutton Monro (1951). “A Stochastic Approximation Method”. In: *The Annals of Mathematical Statistics* 22.3, pp. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586.

Rosen, Rami (2013). “Resource management: Linux kernel namespaces and cgroups”. In: *Haifux, May* 186, p. 70.

Rosenblatt, Frank (1962). “A comparison of several perceptron models”. In: *Self-Organizing Systems*, pp. 463–484.

Rossum, Guido van (2008). *An Open Source App: Rietveld Code Review Tool*. URL: https://web.archive.org/web/20151017112923/https://cloud.google.com/appengine/articles/rietveld (visited on 11/23/2020).

Rossum, Guido Van (2000). *Python Reference Manual: February 19, 1999, Release 1.5.2*. Ed. by Fred L. Drake. iUniverse, Incorporated. ISBN: 1583483748.

Roy, Kaushik, Akhilesh Jaiswal, and Priyadarshini Panda (2019). “Towards spike-based machine intelligence with neuromorphic computing”. In: *Nature* 575.7784, pp. 607–617.

RTI, Strategic Planning (2002). “The economic impacts of inadequate infrastructure for software testing”. In: *National Institute of Standards and Technology*.

Ruder, Sebastian (2017). *An overview of gradient descent optimization algorithms*. arXiv: 1609.04747 [cs.LG].

Rueckauer, Bodo, Connor Bybee, Ralf Goettsche, Yashwardhan Singh, Joyesh Mishra, and Andreas Wild (2021). *NxTF: An API and Compiler for Deep Spiking Neural Networks on Intel Loihi.* arXiv: `2101.04261 [cs.ET]`.

Rumelhart, D. E., G. E. Hinton, and Williams R.J. (1986). "Learning representations by back-propagating errors". In: *Nature* 323, pp. 533–536.

S. Lehtinen C. Lonvick, Ed. (Jan. 2006). *The Secure Shell (SSH) Protocol Assigned Numbers.* RFC 4250. URL: `https://www.rfc-editor.org/rfc/rfc4250.html` (visited on 11/26/2020).

Sacramento, João, Rui Ponte Costa, Yoshua Bengio, and Walter Senn (Oct. 2018). "Dendritic cortical microcircuits approximate the backpropagation algorithm". In: arXiv: `1810.11393 [q-bio.NC]`.

Sampedro, Zebula, Aaron Holt, and Thomas Hauser (2018). "Continuous Integration and Delivery for HPC: Using Singularity and Jenkins". In: *Proceedings of the Practice and Experience on Advanced Research Computing.* PEARC '18. Pittsburgh, PA, USA: Association for Computing Machinery. ISBN: 9781450364461. DOI: `10.1145/3219104.3219147`. URL: `https://doi.org/10.1145/3219104.3219147`.

Samuel, A. L. (1959). "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3, pp. 210–229. DOI: `10.1147/rd.33.0210`.

Schemmel, J., S. Hohmann, K. Meier, and F. Schürmann (2004). "A Mixed-Mode Analog Neural Network using Current-Steering Synapses". In: *Analog Integrated Circuits and Signal Processing* 38.2-3, pp. 233–244.

Schemmel, Johannes, Sebastian Billaudelle, Philipp Dauer, and Johannes Weis (2020). "Accelerated Analog Neuromorphic Computing". In: *arXiv preprint.* arXiv: `2003.11996 [cs.NE]`. URL: `https://arxiv.org/abs/2003.11996`.

Schemmel, Johannes, Daniel Brüderle, Andreas Grübl, Matthias Hock, Karlheinz Meier, and Sebastian Millner (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling". In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 1947–1950.

Schemmel, Johannes, Johannes Fieres, and Karlheinz Meier (2008). "Wafer-scale integration of analog neural networks". In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence).* IEEE, pp. 431–438.

Schemmel, Johannes, Laura Kriener, Paul Müller, and Karlheinz Meier (2017). "An Accelerated Analog Neuromorphic Hardware System Emulating NMDA- and Calcium-Based Non-Linear Dendrites". In: *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2217–2226. DOI: `10.1109/IJCNN.2017.7966124`.

Schilling, Moritz (2010). "A Highly Efficient Transport Layer for the Connection of Neuromorphic Hardware Systems". HD-KIP-10-09. Diploma thesis. Ruprecht-Karls-Universität Heidelberg.

Schmitt, Sebastian, Johann Klähn, Guillaume Bellec, Andreas Grübl, Maurice Güttler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Sebastian Jeltsch, Vitali Karasenko, Mitja Kleider, Christoph Koke, Alexander Kononov, Christian Mauch, Eric Müller, Paul Müller, Johannes Partzsch, Mihai A. Petrovici, Bernhard Vogginger, Stefan Schiefer, Stefan Scholze, Vasilis Thanasoulis, Johannes Schemmel, Robert Legenstein, Wolfgang Maass, Christian Mayr, and Karlheinz Meier (2017). "Classification With Deep Neural Networks on an Accelerated Analog Neuromorphic System". In: *Proceedings of* *the 2017 IEEE International Joint Conference on Neural Networks*. DOI: 10.1109/IJCNN.2017.7966125. URL: http://ieeexplore.ieee.org/document/7966125/.

Schneider, Felix (2018). "Towards Spike–based Expectation Maximization in a Closed–Loop Setup on an Accelerated Neuromorphic Substrate". Bachelorarbeit. Universität Heidelberg.

Schreiber, K., T. C. Wunderlich, C. Pehle, M. A. Petrovici, J. Schemmel, and K. Meier (2020). "Closed-Loop Experiments on the BrainScaleS-2 Architecture". In: *Proceedings of the Neuro-Inspired Computational Elements Workshop*. NICE '20. Heidelberg, Germany: Association for Computing Machinery. ISBN: 9781450377188. DOI: 10.1145/3381755.3381776. URL: https://doi.org/10.1145/3381755.3381776.

Schreiber, Korbinian (Jan. 2021). "Accelerated neuromorphic cybernetics". PhD thesis. Universität Heidelberg.

Schroeder, Anna (2016). "Struktur schafft Robustheit: Eine Untersuchung hierarchischer neuronaler Netzwerke mit unpräzisen Komponenten". Bachelorarbeit. Universität Heidelberg.

Schuman, Catherine D, Thomas E Potok, Robert M Patton, J Douglas Birdwell, Mark E Dean, Garrett S Rose, and James S Plank (2017). "A survey of neuromorphic computing and neural networks in hardware". In: *arXiv preprint arXiv:1705.06963*.

Schwartz, Marc-Olivier (2013). "Reproducing Biologically Realistic Regimes on a Highly-Accelerated Neuromorphic Hardware System". PhD thesis. Universität Heidelberg.

Schwartz, Roy, Jesse Dodge, Noah A. Smith, and Oren Etzioni (Nov. 2020). "Green AI". In: *Commun. ACM* 63.12, pp. 54–63. ISSN: 0001-0782. DOI: 10.1145/3381831. URL: https://doi.org/10.1145/3381831.

Sejnowski, Terrence J (2018). "The deep learning revolution". In: *MIT Press*.

– (2020). "The unreasonable effectiveness of deep learning in artificial intelligence". In: *Proceedings of the National Academy of Sciences* 117.48, pp. 30033–30038. ISSN: 0027-8424. DOI: 10.1073/pnas.1907373117. eprint: https://www.pnas.org/content/117/48/30033.full.pdf. URL: https://www.pnas.org/content/117/48/30033.

Shen, Juncheng, De Ma, Zonghua Gu, Ming Zhang, Xiaolei Zhu, Xiaoqiang Xu, Qi Xu, Yangjing Shen, and Gang Pan (2016). "Darwin: a neuromorphic hardware co-processor based on spiking neural networks". In: *Science China Information Sciences* 59.2, pp. 1–5.

Shi, Weisong, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu (2016). "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5, pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

Shumaker, Robert W, Kristina R Walkup, and Benjamin B Beck (2011). *Animal tool behavior: the use and manufacture of tools by animals*. JHU Press.

Silva, Gustavo Noronha (2001). *APT HOWTO*. URL: https://www.debian.org/doc/manuals/apt-howto/ (visited on 12/01/2020).

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587, pp. 484–489.

Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. (2017). "Mastering the game of go without human knowledge". In: *Nature* 550.7676, p. 354.

Simes (May 2002). *How to break out of a chroot() jail.* URL: https://web.archive.org/web/20160209154009/http://www.bpfh.net/simes/computing/chroot-break.html (visited on 01/28/2021).

Slaughter, Sandra A., Donald E. Harter, and Mayuram S. Krishnan (Aug. 1998). "Evaluating the Cost of Software Quality". In: *Commun. ACM* 41.8, pp. 67–73. ISSN: 0001-0782. DOI: 10.1145/280324.280335. URL: https://doi.org/10.1145/280324.280335.

Smith, Jim and Ravi Nair (2005). *Virtual machines: versatile platforms for systems and processes.* Elsevier.

Snyder, Laura J (2011). *The philosophical breakfast club : four remarkable friends who transformed science and changed the world.* eng. 1st ed. New York: Broadway Books. ISBN: 9780767930482.

Sochat, Vanessa (Mar. 2018). "The Scientific Filesystem". In: *GigaScience* 7.5. giy023. ISSN: 2047-217X. DOI: 10.1093/gigascience/giy023. eprint: https://academic.oup.com/gigascience/article-pdf/7/5/giy023/24813254/giy023.pdf. URL: https://doi.org/10.1093/gigascience/giy023.

Soergel, DAW (2015). "Rampant software errors may undermine scientific results [version 2; peer review: 2 approved]". In: *F1000Research* 3.303. DOI: 10.12688/f1000research.5930.2.

Spilger, Philipp (Nov. 2018). "Spike-based Expectation Maximization on the HICANN-DLSv2 Neuromorphic Chip". Bachelorarbeit. Universität Heidelberg.

– (Feb. 2021). "From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach". Master's thesis. Universität Heidelberg.

Spilger, Philipp, Eric Müller, Arne Emmel, Aron Leibfried, Christian Mauch, Christian Pehle, Johannes Weis, Oliver Breitwieser, Sebastian Billaudelle, Sebastian Schmitt, Timo C. Wunderlich, Yannik Stradmann, and Johannes Schemmel (2020). "hxtorch: PyTorch for BrainScaleS-2 — Perceptrons on Analog Neuromorphic Hardware". In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Cham: Springer International Publishing, pp. 189–200. ISBN: 978-3-030-66770-2. DOI: 10.1007/978-3-030-66770-2_14.

Srowig, André, Jan-Peter Loock, Karlheinz Meier, Johannes Schemmel, Holger Eisenreich, Georg Ellguth, and René Schüffny (2007). "Analog Floating Gate Memory in a 0.18 μm Single-Poly CMOS Process". In: *FACETS internal documentation.*

Stall, Shelley, Lynn Rees Yarmey, Reid Boehm, Helena Cousijn, Patricia Cruse, Joel Cutcher-Gershenfeld, Robin Dasler, Anita de Waard, Ruth Duerr, Kirsten Elger, et al. (2018). "Advancing FAIR data in Earth, space, and environmental science". In: *Eos, Earth and Space Science News* 99.

Standards, National Institute of and Technology (Aug. 2015). "Secure hash standard (SHS)". In: DOI: 10.6028/NIST.FIPS.180-4.

Stewart, Graeme A., Benjamin Morgan, Javier Cervantes Villanueva, and Hobbs A. Willett (2020). "Modern Software Stack Building for HEP". In: *EPJ Web Conf.* 245, p. 05016. DOI: 10.1051/epjconf/202024505016. URL: https://doi.org/10.1051/epjconf/202024505016.

Stodden, Victoria, Matthew S. Krafczyk, and Adhithya Bhaskar (2018). "Enabling the Verification of Computational Results: An Empirical Evaluation of Computational Reproducibility". In: *Proceedings of the First International Workshop on Practical Repro-*

ducible Evaluation of Computer Systems. P-RECS'18. Tempe, AZ, USA: Association for Computing Machinery. ISBN: 9781450358613. DOI: `10.1145/3214239.3214242`. URL: `https://doi.org/10.1145/3214239.3214242`.

Strachan, James, Guillaume Laforge, Jochen Theodorou, Paul King, Cedric Champeau, and Apache Software Foundation (2020). *Groovy Language Documentation.* Version 3.0.6. URL: `https://groovy-lang.org/single-page-documentation.html` (visited on 11/18/2020).

Stradmann, Yannik (2019). "Verification and Commissioning of Mixed-Signal Neuromorphic Substrates". Master's Thesis. Ruprecht-Karls-Universität Heidelberg.

Stradmann, Yannik, Sebastian Billaudelle, Oliver Breitwieser, Falk Leonard Ebert, Arne Emmel, Dan Husmann, Joscha Ilmberger, Eric Müller, Philipp Spilger, Johannes Weis, and Johannes Schemmel (2021). *Demonstrating Analog Inference on the BrainScaleS-2 Mobile System.* arXiv: `2103.15960 [cs.AR]`.

Stromatias, Evangelos, Daniel Neil, Francesco Galluppi, Michael Pfeiffer, Shih-Chii Liu, and Steve Furber (2015). "Scalable energy-efficient, low-latency implementations of trained spiking deep belief networks on spinnaker". In: *2015 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

Tavanaei, Amirhossein, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothee Masquelier, and Anthony Maida (2019). "Deep learning in spiking neural networks". In: *Neural Networks* 111, pp. 47–63.

Teeter, Corinne, Ramakrishnan Iyer, Vilas Menon, Nathan Gouwens, David Feng, Jim Berg, Aaron Szafer, Nicholas Cain, Hongkui Zeng, Michael Hawrylycz, et al. (2018). "Generalized leaky integrate-and-fire models classify multiple neuron types". In: *Nature Communications* 9.1, p. 709.

Texas Instruments (2020). *INA219 Zerø-Drift, Bidirectional Current/Power Monitor With I2C Interface.* URL: `https://www.ti.com/lit/ds/symlink/ina219.pdf`.

Thakur, Chetan Singh, Jamal Lottier Molin, Gert Cauwenberghs, Giacomo Indiveri, Kundan Kumar, Ning Qiao, Johannes Schemmel, Runchun Wang, Elisabetta Chicca, Jennifer Olson Hasler, Jae-sun Seo, Shimeng Yu, Yu Cao, André van Schaik, and Ralph Etienne-Cummings (2018). "Large-Scale Neuromorphic Spiking Array Processors: A Quest to Mimic the Brain". In: *Frontiers in Neuroscience* 12, p. 891. ISSN: 1662-453X. DOI: `10.3389/fnins.2018.00891`. URL: `https://www.frontiersin.org/article/10.3389/fnins.2018.00891`.

Thalheim, Jörg, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci (2018). "Cntr: Lightweight OS Containers". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 199–212.

Theano Development Team (May 2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688. URL: `http://arxiv.org/abs/1605.02688`.

Theis, Thomas N. and H.-S. Philip Wong (2017). "The End of Moore's Law: A New Beginning for Information Technology". In: *Computing in Science Engineering* 19.2, pp. 41–50. DOI: `10.1109/MCSE.2017.29`.

Thorpe, Simon, Arnaud Delorme, and Rufin Van Rullen (2001). "Spike-based strategies for rapid processing". In: *Neural Networks* 14.6-7, pp. 715–725.

Thorpe, Simon, Denis Fize, and Catherine Marlot (1996). "Speed of processing in the human visual system". In: *Nature* 381.6582, p. 520.

Tokunaga, Kohei (Apr. 2020). *Startup Containers in Lightning Speed with Lazy Image Distribution on Containerd.* URL: `https://archive.is/e3Cs1` (visited on 02/12/2021).

Toole, Jameson (Nov. 2019). *Deep learning has a size problem: Shifting from state-of-the-art accuracy to state-of-the-art efficiency.* URL: `https://archive.is/fxMZg` (visited on 04/23/2021).

Torvalds, Linus, Junio Hamano, et al. (2005). *Git – distributed version-control system for tracking changes in source code during software development.* URL: `https://git-scm.com` (visited on 11/19/2020).

Tovar, B., N. Hazekamp, N. Kremer-Herman, and D. Thain (2018). "Automatic Dependency Management for Scientific Applications on Clusters". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 41–49. DOI: `10.1109/IC2E.2018.00026`.

Tran, Binh (2013). "Demonstrationsexperimente auf neuromorpher Hardware". In: *Bachelor thesis (german), Universität Heidelberg.*

Troan, Erik, Marc Ewing, and Red Hat (Nov. 1995). *rpm.org Timeline.* URL: `https://rpm.org/timeline.html`.

Tsodyks, Misha, Klaus Pawelzik, Henry Markram, and M. Tsodyks (1998). "Neural Networks with Dynamic Synapses". In: *Neural Computation* 10, pp. 821–835.

Tsodyks, Misha V and Henry Markram (1997). "The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability". In: *Proceedings of the National Academy of Sciences* 94.2, pp. 719–723.

Tweedie, Stephen (2000). "Ext3, journaling filesystem". In: *Ottawa Linux Symposium.* Vol. 20. URL: `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html` (visited on 02/01/2021).

Vanarse, Anup, Adam Osseiran, and Alexander Rassau (2016). "A review of current neuromorphic approaches for vision, auditory, and olfactory sensors". In: *Frontiers in neuroscience* 10, p. 115.

Vidal, Seth (2011). *yum: Yellowdog Update, Modified.* URL: `https://yum.baseurl.org` (visited on 12/01/2020).

Vinet, Judd and Pacman Development Team (2002). *pacman package manager.* URL: `https://www.archlinux.org/pacman/`.

Virtanen, Pauli, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. (2020). "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature methods* 17.3, pp. 261–272.

Vision(s), Electronic (May 2021). *yashchiki.* DOI: `10.5281/zenodo.4740256`. URL: `https://doi.org/10.5281/zenodo.4740256`.

Vollprecht, Wolf, Mario Buikhuizen, Marianne Corvellec, Johan Mabille, and David Brochart (June 2020). *Open Software Packaging for Science.* URL: `https://archive.is/5k096` (visited on 02/22/2021).

Wang, Runchun M, Chetan S Thakur, and André van Schaik (2018). "An FPGA-based massively parallel neuromorphic cortex simulator". In: *Frontiers in neuroscience* 12, p. 213.

Weidner, Jonas (2019). “Experiment Visualization and Simulations towards a Cortical Microcircuit on the BrainScaleS Neuromorphic Hardware”. Bachelorarbeit. Universität Heidelberg.

Weis, Johannes (Sept. 2020). “Inference with Artificial Neural Networks on Neuromorphic Hardware”. Master’s thesis. Universität Heidelberg.

Weis, Johannes, Philipp Spilger, Sebastian Billaudelle, Yannik Stradmann, Arne Emmel, Eric Müller, Oliver Breitwieser, Andreas Grübl, Joscha Ilmberger, Vitali Karasenko, Mitja Kleider, Christian Mauch, Korbinian Schreiber, and Johannes Schemmel (2020). “Inference with Artificial Neural Networks on Analog Neuromorphic Hardware”. In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Cham: Springer International Publishing, pp. 201–212. ISBN: 978-3-030-66770-2. DOI: 10.1007/978-3-030-66770-2_15.

Werbos, Paul J (1982). “Applications of advances in nonlinear sensitivity analysis”. In: *System modeling and optimization*, pp. 762–770.

Wilson, Greg, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson (Jan. 2014). “Best Practices for Scientific Computing”. In: *PLOS Biology* 12.1, pp. 1–7. DOI: 10.1371/journal.pbio.1001745. URL: https://doi.org/10.1371/journal.pbio.1001745.

Wolfram, Stephen (2015). *Untangling the Tale of Ada Lovelace*. URL: https://writings.stephenwolfram.com/2015/12/untangling-the-tale-of-ada-lovelace/ (visited on 10/29/2020).

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean (2016). *Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. arXiv: 1609.08144 [cs.CL].

Wunderlich, Timo, Akos F. Kungl, Eric Müller, Andreas Hartel, Yannik Stradmann, Syed Ahmed Aamir, Andreas Grübl, Arthur Heimbrecht, Korbinian Schreiber, David Stöckel, Christian Pehle, Sebastian Billaudelle, Gerd Kiene, Christian Mauch, Johannes Schemmel, Karlheinz Meier, and Mihai A. Petrovici (2019). “Demonstrating Advantages of Neuromorphic Computation: A Pilot Study”. In: *Frontiers in Neuroscience* 13, p. 260. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00260. URL: https://www.frontiersin.org/article/10.3389/fnins.2019.00260.

Wunderlich, Timo C. and Christian Pehle (2020). *EventProp: Backpropagation for Exact Gradients in Spiking Neural Networks*. arXiv: 2009.08378 [q-bio.NC].

Xilinx (2019). *Zync UltraScale+ MPSoC Data Sheet*. URL: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

Yamazaki, Tadashi, Jun Igarashi, and Hiroshi Yamaura (2021). “Human-scale Brain Simulation via Supercomputer: A Case Study on the Cerebellum”. In: *Neuroscience* 462. In Memoriam: Masao Ito—A Visionary Neuroscientist with a Passion for the Cerebellum, pp. 235–246. ISSN: 0306-4522. DOI: https://doi.org/10.1016/j.neuroscience.

2021.01.014. URL: https://www.sciencedirect.com/science/article/pii/S030645222100021X.

Yang, Chenghai, James H. Everitt, and Dale Murden (2011). “Evaluating high resolution SPOT 5 satellite imagery for crop identification”. In: *Computers and Electronics in Agriculture* 75.2, pp. 347–354. ISSN: 0168-1699. DOI: 10.1016/j.compag.2010.12.012. URL: https://www.sciencedirect.com/science/article/pii/S0168169910002632.

Ye, Peide, Thomas Ernst, and Mukesh V. Khare (2019). “The last silicon transistor: Nanosheet devices could be the final evolutionary step for Moore’s Law”. In: *IEEE Spectrum* 56.8, pp. 30–35. DOI: 10.1109/MSPEC.2019.8784120.

Yoo, Andy B, Morris A Jette, and Mark Grondona (2003). “Slurm: Simple linux utility for resource management”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 44–60.

Zenke, Friedemann and Surya Ganguli (2018). “Superspike: Supervised learning in multilayer spiking neural networks”. In: *Neural computation* 30.6, pp. 1514–1541.

Zhang, W., J. Lin, W. Xu, H. Fu, and G. Yang (2017). “SCStore: managing scientific computing packages for hybrid system with containers”. In: *Tsinghua Science and Technology* 22.6, pp. 675–681. DOI: 10.23919/TST.2017.8195349.

Zhu, Jiadi, Teng Zhang, Yuchao Yang, and Ru Huang (2020). “A comprehensive review on emerging artificial neuromorphic devices”. In: *Applied Physics Reviews* 7.1, p. 011312. DOI: 10.1063/1.5118217. eprint: https://doi.org/10.1063/1.5118217. URL: https://doi.org/10.1063/1.5118217 (visited on 05/01/2020).

Zhu, Xiaojin and Andrew B Goldberg (2009). “Introduction to semi-supervised learning”. In: *Synthesis lectures on artificial intelligence and machine learning* 3.1, pp. 1–130.

Zou, Hui (2006). “The adaptive lasso and its oracle properties”. In: *Journal of the American statistical association* 101.476, pp. 1418–1429.

Acknowledgments

I would like to extend my gratitude to the following people:

The late Prof. Dr. Karlheinz Meier for championing the quite unique Electronic Vision(s) group, promoting his vision of Neuromorphic Computing while shielding us from all things funding, so that we could focus on the things that mattered. Still, he managed to pierce any concept's hard-points after mere minutes of presentation.

Dr. habil. Johannes Schemmel for agreeing to take over supervision of my thesis and the Electronic Vision(s) after Prof. Meier's untimely demise.

Prof. Dr. Holger Fröning who graciously agreed to be my second supervisor.

Prof. Dr. Manfred Salmhofer and Prof. Dr. Rüdiger Klingeler for taking part in the final demonstration of how much more there is to know about Physics, i.e., my defense.

Dr. Mihai A. Petrovici, Bereichsleiter Rail, for sparking my original interest in both neuroscience and machine learning and defining the Vision(s) way of life. Ever since that first round of Quake with Eric and the old guard, right after our very first in-person meeting, I knew I was in the right place with the right people.

Dr. Eric Müller, coffee connoisseur, for showing me how to build proper software. If only his taste in music could rival his skills in all things software…

All proof readers of this manuscript, helping to keep the number of errors at a manageable, albeit still far too high, level: Andreas Baumbach, Julian Göltz, Christian Mauch, Eric Müller, Philipp Spilger.

All students I had the pleasure to supervise (fully, jointly or partially): Marcel Großkinsky, Anna Schröder, Carola Fischer, Agnes Korcsak-Gorzo, Felix Schneider, Philipp Spilger, Julian Göltz, Aruna Raman and Arne Emmel. In particular, I am very pleased to see Philipp Spilger, Julian Göltz and Arne Emmel infused with the proper tooling spirit, especially when it comes to producing state-of-the-art figures.

All collaborators I had the pleasure of working with, in particular Jakob Jordan.

My fellow TMAlers, for being my first home; my fellow S0fties, for slowly but steadily absorbing me.

Christian Mauch, Grillmeister, for sharing the pain of both cluster-administrative and win-rate-related burdens with me… also Greek wine.

A Turing machine in the flesh, Philipp Spilger.

Dr. Sebastian Schmitt for maintaining an off-site arXiv-backup in `~Journal Club`.

Yannik Stradmann for his driving skills to and from retreats.

Johann Klähn for continuing to support `genpybind`[1] in his free time well after leaving the group.

[1]Autogeneration of Python Bindings from Manually Annotated C++ Headers, [Klähn et al., 2020]

David Stöckel for initially setting up the digital containerkasse, Christian Mauch and Joscha Ilmberger for maintaining it tirelessly.

Dr. Björn Kindler for his custom HTTP-based authentication API that is easy-to-use from Rust-based LDAP-plugins.

Simeon Kanya for initiating the Hüttenwochende, our retreat-methadone, happening mere weeks prior to shutdown.

The Pandemic Legacy crisis committee – Christian Mauch, Joscha Ilmberger, Hartmut Schmidt and Philipp Spilger – that had to be postponed by the very outbreak it predicted; never forget the Colonel's last stand in Taipei!

The Visionary Brewers Joscha Ilmberger, Christian Mauch, Maurice Güttler, Dan Husmann and Eric Müller; Hopfen stopfen!

All the Visionaries who joined, left or still are in the group. Keep up the visionary group spirit that, over the years, has included last suppers, Quake, Neurovision, Laser-Tag, Doto (including one very special LAN), movie nights, legendary Christmas parties, grill0rn, Secret Hitler/Among Us, and of course: Dienstende.

Allen, mit denen ich abseits der Arbeit Zeit verbringe.

Meinen Eltern und meiner Familie.

Lara, mein Komplementär zum großen Ganzen, insbesondere für das gemeinsame Durchstehen der letzten Monate der Corona-beengten Schreibphase. Danke!

Funding Statement

This research has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nos. 720270, 785907 and 945539 (Human Brain Project, HBP).

This research has received funding from the BMBF[2] under grant number 16ES1127 as part of the *Pilotinnovationswettbewerb „Energieeffizientes KI-System"*.

The work carried out in this dissertation used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 720270, 785907 and 945539 (Human Brain Project, HBP).

Furthermore, we acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG (bwForCluster NEMO).

[2]Federal Ministry of Education and Research (*Bundesministerium für Bildung und Forschung*)