University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor's thesis

Extension of neural network architecture

Plzeň 2020 Roman Kalivoda

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Roman KALIVODA**
Osobní číslo:	**A16B0049P**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informatika**
Téma práce:	**Rozšíření architektury neuronové sítě**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s aktuálním stavem v oblasti modelování neuronů a neuronových sítí z pohledu jejich schopnosti napodobovat biologické neuronové sítě.
2. Seznamte se s dostupnými nástroji – simulátory různých modelů neuronů a neuronových sítí.
3. Na základě bodů 1 a 2 navrhněte a implementujte rozšíření architektury neuronové sítě tak, aby toto rozšíření reflektovalo vybrané biologické chování.
4. Porovnejte chování neuronové sítě z bodu 3 s chováním neuronových sítí podobných architektur.
5. Zhodnoťte dosažené výsledky.

Rozsah bakalářské práce:	**doporuč. 30 s. původního textu**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování bakalářské práce:	**tištěná**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce:	**Ing. Roman Mouček, Ph.D.** Katedra informatiky a výpočetní techniky
Datum zadání bakalářské práce:	**7. října 2019**
Termín odevzdání bakalářské práce:	**7. května 2020**

Radová

Doc. Dr. Ing. Vlasta Radová
děkanka

Západočeská univerzita v Plzni
L.S.
Fakulta aplikovaných

[illegible]

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 15. října 2019

Declaration

I hereby declare that this bachelor's thesis is completely my own work and that I used only the cited sources.

Plzeň, May 6, 2020

Roman Kalivoda

Abstract

Spiking neural networks (SNNs) are artificial neural networks designed to mimic sparse and asynchronous nature of information processing observed in biology. In recent years, deep spiking networks emerged with efforts to draw on experiences with classic deep networks. There also appeared attempts to reuse the available state-of-the-art analogue neural networks (ANNs) completely, and replace the neurons for inference. This thesis contributes with analysis of the dominant methods used in the development of SNNs, and comparison of the major SNN simulation platforms. An application of the SNNs was demonstrated on detection of event-related potentials in EEG data.

Abstrakt

Impulsní neuronové sítě jsou variantou umělých neuronových sítí, které jsou navrženy, aby simulovaly přirozenou rozptýlenost a asynchronii pozorovanou u biologických neuronových sítí. Pokrok v nedávné době umožnil vytváření vícevrstvých impulsních sítí. S tím se objevila i snaha dosáhnout podobných úspěchů jako s klasickými vícevrstvými sítěmi. Objevily se také snahy o opětovné využití již existujících a populárních architektur klasických neuronových sítí. Ty mohou být například použity během procesu učení a nahrazeny impulsními sítěmi až v provozu. Tato práce analyzuje hlavní metody využité při vytváření impulsních sítí a porovnává nástroje pro jejich simulaci. Také bylo ukázáno použití impulsní neuronové sítě na detekci kognitivních evokovaných potenciálů v EEG datech.

Contents

1 Introduction

A neural network is a set or population of specialised cells (neurons) that are interconnected by synapses. In biology, the neural network forms the structure of a neural system in animals. The interconnection pattern, size and spatial organisation of the population determines the architecture and specialisation of the network. The network specialises in carrying out a specific function when it is activated. These specialised networks then tie to one another to develop larger systems (e.g. animal brain). These natural phenomena inspired computer scientists to design a set of algorithms that models some aspects of the animal brain.

Artificial (analogue) neural networks (ANNs) were initially inspired by biological neural systems, but many concepts were simplified or modified to conform to their practical applications. On the other hand, the spiking neural networks (SNNs) were mostly meant to simulate their biological models as closely as possible and thus improve our understanding of how real biological networks achieve their cognitive abilities with incredibly low energy consumption. Although the differences between those approaches still last, some aspects of the two approaches are getting more interconnected as the research advances and one approach can exploit the other. Nowadays, deep ANNs achieve satisfying accuracy in many complex tasks, although with extensive requirements on computation power. Because of that, new goals for the improvement of their efficiency are being set. That is where deep SNNs appear to be an exciting topic of research because of better power efficiency [5, 40], although the applications generally do not accomplish the same accuracy yet. Apart from the applications in which ANNs are used, spiking networks open up new fields of utilisation.

There is a summary of principal differences between the two classes of neural networks in chapter 2. There is also a survey of the current state of the research in the area of spiking neural networks. Chapter 3 compares existing simulators of spiking neural networks. The remaining tools used in this work are described in chapter 4. Chapter 5 describes initial experiments with spiking networks and selected software. Also, there is proposed an approach on how to interconnect the two chosen SNN tools. Chapter 6 delineates a conversion and consequent simulation of a neural network used for binary classification of event-related potentials. Event-related potentials are brain responses to a specific stimulus. These responses are measurable as small voltage changes during electroencephalography (EEG).

2 Analogue neural networks and spiking neural networks

2.1 Artificial neural networks

Artificial neural networks are built of idealised units which compute output values from their weighted input values and continuous nonlinear activating functions [40]. The units interact with each other by propagating its continuous output. These units connect themselves into distinct layers. If there is more than one hidden layer (the hidden layer is the layer which takes input from the previous layer and outputs own results to another layer, that means it is neither input layer, nor output layer), the network is recognised as a deep network. In deep networks, each layer works with a set of features received from the output of the previous layer. This feature hierarchy enables deep networks to operate accurately on large data sets with plenty of parameters. Advances in the hardware computation power made it possible to use very deep networks for various artificial intelligence tasks. However, the (still quite) notable computation costs of these networks prevent them from applications in embedded and power-constrained environments.

The resulting neural network needs to be learned to become functional. Learning neural network means adjusting its connection weights and other parameters to optimise the accuracy of the results. There exist two major approaches to learning of a neural network, supervised learning and unsupervised learning.

2.1.1 Supervised learning in artificial neural networks

A prerequisite for a supervised learning method is a labelled data set. Supervised learning is useful for classification or regression tasks when we need to find a set of significant relationships or structure in the input data, that means we need to approximate a regression function. The functionality of the method is dependent on the complexity of the approximated function. Higher complexity is needed to learn the full structure of the data, but there is a threat of over-fitting the model, that means the model is too complicated and fails on general data. In the neural network, the supervised learning method takes advantage of the error backpropagation algorithm. The algorithm learns the network on the learning data set and finds the difference between the computed and desired output. Then the synaptic weights are adjusted to minimise the difference [36].

2.1.2 Unsupervised learning in artificial neural networks

Unsupervised learning means that the data set is missing labels, and the model attempts to find some structure in the underlying data set. Common use-cases for unsupervised learning are exploratory analysis and dimensional reduction. Exploratory analysis can get a fundamental inspection of the data in situations where it is difficult or impossible for people to find trends in the data. Typical applications for unsupervised learning methods are data clustering or visualisation. Another example of unsupervised learning is a concept of generative adversarial networks (GANs). GANs is an architecture that uses two neural networks to generate new synthetic data with a structure similar to real data [15].

2.2 Spiking neural networks

Although biological neural networks inspired the basic principles of artificial neural networks, the processes in the biological networks differ from the processes in ANNs. The spiking neural networks proceed from the knowledge retrieved from observations of the biological neural networks. The basic idea behind SNN is that the network consists of the spiking neurons which have a different characteristic than artificial neurons. Each spiking neuron has a membrane potential, a quantity related to the difference of electrical potentials at the membrane of biological neurons. When the membrane potential reaches a specified value (the threshold value of the neuron), a spike (a sudden increase of voltage) is generated. It can also be said that the neuron fired. Note that there is a difference between biological neurons and their mathematical models. The resting membrane potential (the value of the membrane potential when there is not received any activity from previous neurons) of the real biological neuron is typically negative. However, for mathematical simplification, it is convenient to assume in the spiking neuron models that the resting membrane potential is zero, and thus the current membrane potential is the sum of postsynaptic potentials [26]. The postsynaptic potential is a result of an action potential (spike) generated by previous neurons which connect to the observed neuron. The postsynaptic potential can be either excitatory or inhibitory. The excitatory postsynaptic potential (EPSP) increments (depolarises) the membrane potential and the inhibitory postsynaptic potential decrements (hyper-polarises) the membrane potential. Depending on the level of detail, various models of spiking neuron exist. Popular spiking neuron models are the leaky integrate-and-fire model (LIF), the Izhikevich neuron model or spike response model [40].

Besides the biological perspective on spiking neural networks, there also exists a more technical point of view used in neuromorphic engineering. In this perspective, the spikes are more often called events. This term originates from the address event representation protocol [30, 2], which is used to connect event-based neuromorphic peripherals. With this point of view, the emphasis on the biological plausibility of the model and level of detail of the neurons steps back to allow more pragmatic ways to use the networks in such neuromorphic applications. The combination of the spiking neural networks with event-based sensors is especially useful because it enables both parts to utilise its full potential in terms of power efficiency [32].

2.3 Deep spiking neural networks

Similarly to ANNs, if the spiking neural network consists of multiple layers, and at least one of them is hidden, the network is called deep. Deep neural networks, both spiking and non-spiking, are more capable. The interconnected neurons transfer information between themselves by firing trains of spikes (spike trains). Strictly speaking, the information is coded in the number of spikes and their frequencies, rather than amplitude characteristic of a single spike. That means the communication in the network is discrete in time in contrast to ANN, where the communication is going on continuous activation functions. This characteristic makes the spiking neural networks interesting to research because SNNs can be used to create more efficient low-energy consuming neural networks. Also, SNNs can be implemented on specialised dedicated hardware, which even more improves energy efficiency. Another advantage of the deep spiking networks is that the approximation of the final layer output is retrievable at the time of recording the first input spikes and the approximation improves over time [32]. Deep spiking neural networks have theoretically the same representational power as deep ANNs with lower energy requirements. Nevertheless, there is a problem with achieving the same results as with ANNs because optimal solutions for supervised learning of the SNNs do not exist yet. The gradient-based optimisation methods used in artificial neural networks require the activation function to be differentiable [40]. The spike trains generated in spiking neurons can be formally represented by sums of delta functions which do not have derivatives. Multiple approaches proposed how to overcome this problem, but the research of the learning methods for the SNNs is still at its beginning [40].

2.4 State of the art

Recently there appeared many novel theories and experiments on the topic of deep spiking neural networks and new approaches to learning such networks. As stated above, traditional methods for deep learning can be hardly used because of the distinct characteristics of the two models. One of the approaches to overcome this issue is using some sort of approximate derivatives or other substitutes, although this may reduce bio-plausibility of the model. [24] proposed a spiking network backpropagation rule with low-pass filtering to handle discontinuity at the time of the spike. This method demonstrates state-of-the-art results for deep SNNs.

A more biologically plausible approach is using local learning rules such as spike-timing-dependent plasticity (STDP). STDP adjusts the synaptic weights between presynaptic and postsynaptic neuron according to the relative difference of their spike times. If presynaptic neuron fires relatively briefly before the postsynaptic neuron, its synaptic weight increments. The weight decrements in the opposite case, that is in the case when presynaptic neuron spikes briefly after the postsynaptic neuron. Local learning rules can be used with unsupervised learning [40], but with supervised learning, there is once again missing a plausible backpropagation method. This problem can be covered, for example, by introducing recurrent connections to propagate the error signal. The use of local learning rules is also interesting for efficient implementations on dedicated hardware, such as SpiNNaker[14] or BrainScales. At present, the efficiency of the networks which use this method falls behind in terms of precision.

Another approach is to create a conversion between ANN and SNN. For the conversion, a conventional deep network is trained using backpropagation and other methods available in ANNs. Then the already trained network is transformed into its spiking variant. The conversion is done by mapping the analogue neurons onto spiking ones (the mapping can be both one-to-one and one-to-many) and adjusting the weights and parameters of the spiking neurons according to the trained analogue network. Pérez-Carrasco first proposed the conversion approach in [31]. This approach can enhance the performance of hardware implementations of deep neural networks. An advantage of this approach is that the best, state-of-the-art deep networks can be transformed into SNNs without previous modifications and the efficiency of the final SNN is approaching the efficiency of the original network. However not every ANN can be converted, because features or methods which are common in analogue networks might not have its spiking equivalents. [35] introduced additional techniques to convert more general class of ANNs by implementing features such as soft-max, batch normalisation or max-pooling (using a maximum filter for spatial reduction of the input feature set), which were previously not available in the spiking networks. The resulting accuracy of experiments which used the conversion approach is very promising [40, 32]. One of the disadvantages of the conversion is that the most often used technique uses rate codes which are quite inefficient. Rate-coding means that the activation values of the original analogue deep network are translated into firing rates of the spiking neurons, so multiple spikes are necessary to stand for a single activation value. Alternative spike codes need to be developed to overcome this. [44] proposes one such coding, which maintains the accuracy of the modelled artificial network while reducing the total count of spikes needed in comparison to other conversion methods.

There exists a modified hybrid approach known as *constrain-then-train*. Instead of starting with conventional learning of the ANN, additional constraints needed for spiking networks simulation (or neuromorphic hardware) are applied on the first network before its training. Then, the learned parameters of such constrained ANN can be used directly during the mapping onto SNN without further re-scaling [32]. The retrieved parameters are particular for just single settings of the spiking neuron model, so the network needs retraining if these parameters should change. The final network has an advantage that it adapts better to the target environment than generally converted networks.

Above, four approaches to supervised learning of the spiking neural networks are briefly described. The spike-based learning and local learning methods have a common need for improvement of the accuracy to match the results retrieved by the other two methods. However, they offer other notable features such as potentially better energy efficiency and the ability to exploit additional features of the neuromorphic hardware (e.g. precise timing). Also, inferior performance efficiency is being improved by recent works. On the other hand, the spiking networks created by the conversion techniques are comparable to its original artificial network models, but due to the used coding, its energy efficiency deteriorates.

2.4.1 Binary deep neural networks

Binary deep neural networks (BNNs) are an alternative to spiking neural networks in terms of energy efficiency. BNNs are deep neural networks that use binary activation values and binary-valued weights [37]. The state-of-the-art binary neural networks achieve slightly degraded accuracy in comparison with non-binary full precision networks, but with reduced memory requirements and less power consumption [10]. This advantage is more evident if both activation values and weights are binary. Then, more complicated arithmetic operations can be replaced with simpler bitwise operations [20]. With binary activations, BNNs can also directly use spikes from event-based hardware [25]. Compared to spiking networks, the information is still propagated synchronously and does not allow the fast propagation of most salient features [32].

2.4.2 Comparison

Standard classification benchmarks are usually used to compare the individual works, although [32] points out that these benchmarks might not be appropriate for evaluation of spiking networks and should be taken as a proof of concept.

The most used benchmark for demonstrating the performance of the spiking network is a classification of the modified NIST (MNIST) database of handwritten digits [23]. It consists of 70000 examples of grey-level images. The handwritten digits have been normalized to fit in a 20×20 pixels frame and centred in a 28×28 image. This database is divided into a training set with 60000 examples and a test set with 10000 examples.

Table 2.1 compares some of the reported classification accuracies on the MNIST data set. There are shown ANNs and SNNs with the best currently achieved accuracies. It can be seen that the SNNs are slightly worse than ANNs. The best accuracy for SNN was reported with a converted convolutional network by [35] in 2017. The other converted networks shown in the table (rows 2 and 3) are mentioned because they used different spike coding schemes, pulsed Sigma-Delta [44] and time-to-first-spike [43]. The SNN in row 4 represents a non-converted SNN trained with the backpropagation algorithm.

description	type	accuracy (%)
Rate-based conversion of 7-layer CNN with max-pooling [35]	SNN	99.44
Converted SNN with pulsed Sigma-Delta coding scheme [44]	SNN	99.14
Converted Lenet-5 with TTFS temporal coding [34]	SNN	98.57
Spiking CNN with low-pass backpropagation rule [24]	SNN	99.31
Branching & Merging CNN with Homogenous filter capsules [19]	ANN	99.79
Multi-column deep neural network [9]	ANN	99.77

Table 2.1: Comparison of selected deep neural networks.

3 Simulation platforms

Currently, there exist numerous platforms for simulation of biological neural networks. The first objective is to analyse those platforms from the perspective of capabilities and choose the most suitable of them. The main criterion is the current state of the community, and user base around the platform, their ability to provide support to the users and state of the project documentation. Because many of the projects are using GitHub to host its repositories, the metrics like code frequency, contributors count, number of issues and similar indicators displayed by GitHub can help to determine this. The next criterion is whether the platform provides means to interact with both spiking neural networks and artificial neural networks. Other criteria are user-friendliness of the platform and license conditions.

The Neural Simulation Tool (NEST) [17], Brian [39], Neuron [6] and Nengo [1] are platforms, which were analysed more precisely. Multiscale Object-Oriented Simulation Environment (MOOSE), STochastic Engine of Pathway Simulation (STEPS), Topographica, Neocortical Simulator (NCS), The Parallel Circuit SIMulator (PCSIM) and the GEneral NEural SImulation System (GENESIS) were other considered simulators. However, they were not included, out of consideration for their small community, range of supported models, or because they are no longer actively developed.

3.1 The Neural Simulation Tool (NEST)

NEST is a simulator for spiking neural network models. NEST provides approximately 50 neuron models and over ten synapse models. Its primary focus is on the dynamics, size and structure of the network rather than on the exact morphology of individual neurons. NEST is implemented in C++ and contains a scripting interface for Python and its native simulation language interpreter (SLI). Recently, the NEST initiative, which coordinates the development of NEST, developed a new markup language NESTML to make the creation of new neuron models easier [33]. NEST can be used on many computer architectures from low-end laptops to supercomputers. The project's page explicitly mentions Linux, MacOS X and IBM BlueGene as supported platforms. A windows operating system can utilise a prepared virtual machine image. The scaling of the existing model on a high-performance cluster or multi-core computer is convenient and minimal, or no changes are required, which is an advantage over other simulation platforms [41]. NEST is applied in projects like the Human Brain Project or BrainScaleS. NEST appears in over 290 papers as the simulator which was used [18].

3.2 Brian

Brian is a spiking neural network simulator written in Python. It is an equation oriented simulator, i.e. neural models are defined in its mathematical form with differential equations rather than written in programming language function. This feature makes the simulator very flexible. To increase the speed of simulation, Brian can run in several runtime code generation modes, for example, NumPy, Cython or generate a standalone C++ code (although some limitations apply). A downside of the Brian software is limited support for parallel computations and total lack of support for cluster computations [41]. The project's page does not state any numbers of its users, but there exists a Google group [3] with support topics, which has over 300 users. Maintainers of the project do not track any information about publications, which use Brian.

3.3 NEURON

NEURON is a simulation environment for modelling single neurons or neural networks. It specialises in empirically-based simulations. NEURON is more suitable for simulating single neuron models with more complex structure. NEURON contains a graphical user interface (GUI), which can be used to manipulate an existing model or create a new one. The GUI can be used to create both single neuron model and a network model. Neural networks and neuron models can be described with NEURON's own interpreted language called "hoc" or with Python script. New modules for neuron dynamics can be developed in its native simplified language NMODL [41]. NEURON utilises several methods to optimise the efficiency of its code. For example, it provides routines for tabulating values of often used equations to avoid their computation every time. It also supports optimisations for high-performance computing, such as embarrassingly parallel computation without the need to use other software. NEURON is the only simulator which can distribute computations for a single complex neuron over a cluster [41]. According to its web page, its user base is extensive with more than 1500 registered users at the forum and more than 1900 of scientific articles that reported the use of NEURON. Software installers are available for MS Windows, macOS and Linux.

3.4 Nengo

Nengo is a tool for simulating large neural models. Nengo exploits the theoretical modelling approach described by the Neural Engineering Framework. The Neural Engineering Framework proposes three principles (representation, transformation, and dynamics) for the construction of large-scale neural models which are simulator independent. Nengo decouples model creation and simulation so that multiple simulators can run created models. NengoDL is one such simulator built on top of the deep learning framework TensorFlow. Nengo also provides a backend for neuromorphic hardware Intel Loihi. From version 2 onwards, Nengo is written in the Python programming language. It provides packages for all major operating systems. Figure 3.1 shows a relation between the Nengo components. Nengo incorporates a proprietary license which allows free manipulation with Nengo and its source code for any non-commercial purpose as long as the copyright notice is included. According to the project's webpage, more than 100 publications mentioned Nengo. Nengo forum has over 200 registered users of which about 40 were active last month.

3.4.1 Neural engineering framework

The neural engineering framework (NEF) is a general method for building neural models. It offers a method to solve connection weights between network components by specifying properties of the used neuron models, the values which they represent and the functions which should be computed [38]. The NEF defines three principles for building large-scale neural models. These principles are called representation, transformation and dynamics.

The representation principle states that a population of neurons represents information. Neural populations represent time-varying signals through their spiking responses, and a signal is regarded as a vector of real numbers of arbitrary length. This way, neural computations can be defined by manipulating the information using conventional mathematics [1]. During the encoding process, a specific amount of current is released into a single neuron. The tuning curve of the neuron determines how likely it is that the neuron spike, as a function of the input signal. The spike frequency of the population represents the encoded information. The information can be decoded by observing these spike trains. First, the spike frequencies are filtered with exponentially decaying filter. The decoding weights multiply the filtered spike trains, and then the spike trains sum together. The accuracy of the decoded information increases with the increasing number of neurons

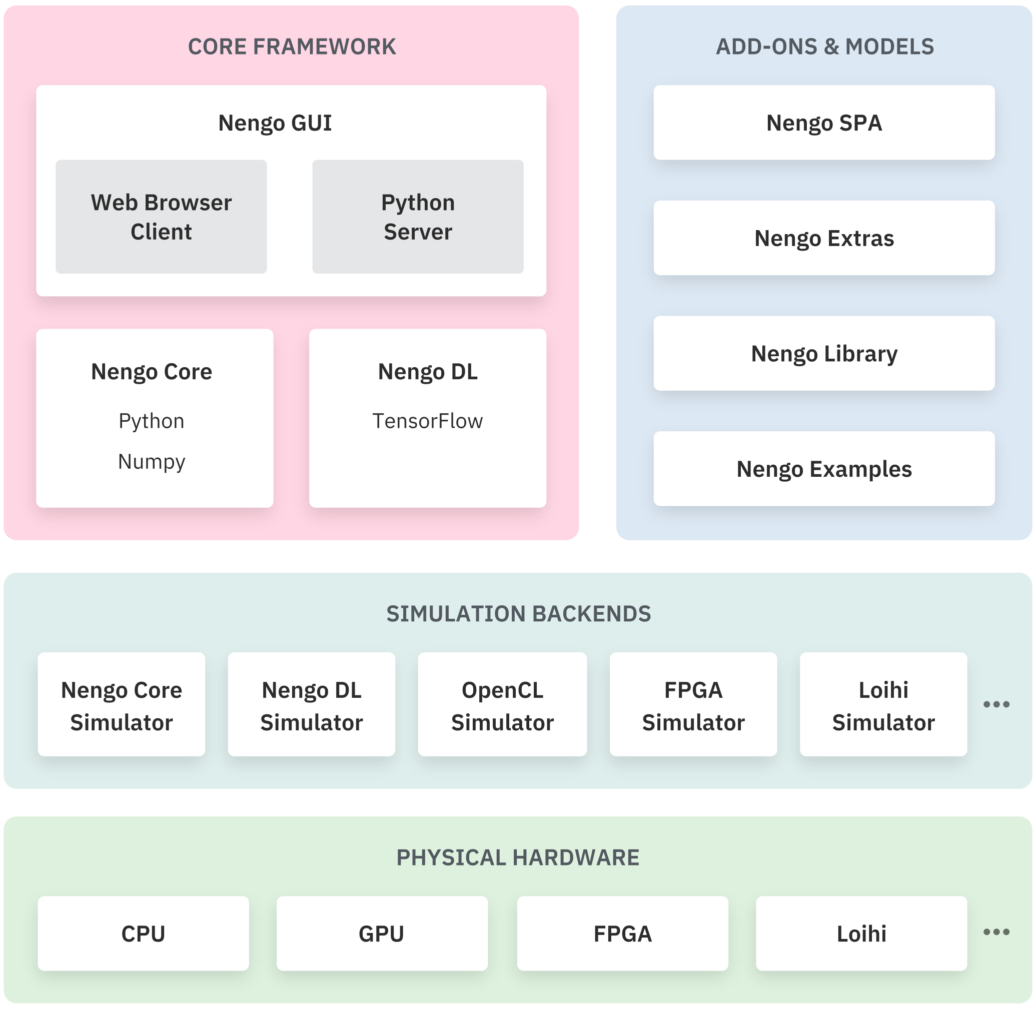

Figure 3.1: The Nengo ecosystem. Source: `https://www.nengo.ai/documentation/`

in the population.

The transformation principle describes how to decode spike trains to compute transformations of signals. It provides a way to compute the connection weights between populations to represent an arbitrary function.

When the neurons are connected recurrently, the vectors represented by neural populations can be interpreted as state variables of the dynamical system and methods of control theory can be applied [1].

3.4.2 Semantic pointer architecture

The semantic pointer architecture provides an approach to build cognitive models with large-scale spiking neural networks. As stated in [13], the semantic pointers are neural representations that carry partial semantic content and are composable into the representational structures to support complex cognition. The Semantic Pointer Architecture (SPA) uses vectors to represent concepts. The Nengo-SPA package contains the implementation of the SPA for the Nengo simulator.

Nengo used with SPA is an underlying foundation for the proof-of-concept Spaun 2.0 [8] model, which is according to the author, the largest functional brain model. The Spaun 2.0 model consists of several neural networks, each specialised for a different cognitive task. Spaun 2.0 deals with digit recognition, list memory, question answering and other tasks.

3.5 Summary of platform comparison

Out of available neural simulation platforms, four were chosen as the best candidates for further analysis. As a starting point for the comparison of Neuron, Brian and NEST, a previous comparative study [41] was used. Each of the platforms has some sort of forum or mailing list for communication of its users. Statistics retrieved from these pages were used to compare the number of users of the platform. Other statistics were retrieved from the GitHub sites of the projects. They can be used as an indicator of the activity of the projects in the last years. The collected statistics are shown in table 3.1 below. The Neuron simulator has by far the largest community on its user forum and is cited by other researchers most often. It is probably because the Neuron simulator is developed for the longest time.

On the other hand, The NEST simulator has almost three times more contributors to its source code than any other platform. Nengo is the only simulator which offers utilisation of the artificial neural networks to improve simulations of the spiking neural networks. A table in table 3.2 contains information about the state of the documentation of the four compared projects. Table 3.3 contains a table with an overview of some additional characteristics of the simulators.

Name	Contributors	Mailing list members	GitHub issues	GitHub watchers	License
NEST	77	N/A	630	36	GPL-2.0
Neuron	19	1516	95	13	BSD-3-Clause
Brian	27	324	675	42	CeCILL-2.1
Nengo	24	234	730	69	Proprietary

Table 3.1: Metrics of the community support for the selected simulation platforms

	NEST	Neuron	Brian	Nengo
A book describing the simulator	Computational Systems Neurobiology [22]	The NEURON Book [6]		How to build a brain: A neural architecture for biological cognition [13]
State of the documentation	minor imperfections in the Python interface documentation	complete API reference, poorly arranged	minor lacks in API documentation	detailed API reference and tutorials
Is the source code for the tutorials available?	Yes	Yes	Yes	Yes

Table 3.2: Overview of the state of the documentation of the simulators.

	NEST	Neuron	Brian	Nengo
GPU computation support	No	Yes (coreNEURON library)	Yes (Brian2GeNN package)	Yes (OpenCL)
parallel computation support	Yes	Yes (special version)	In development (OpenMP)	Yes
support for distributed computations	Yes	Yes	No	Yes
supported platforms	Linux, macOS, IBM BlueGene	Windows, Linux, macOS, IBM BlueGene, Cray XT3	Windows, Linux, macOS	Windows, macOS, Linux, Intel Loihi

Table 3.3: Parallelization, high-performance computation support and supported platforms.

4 Other libraries

This chapter contains a short description of the rest of the software, libraries and frameworks encountered during research. PyNN [11], a simulator-independent language for building neuronal network models, is mentioned at the end of this chapter. PyNN works as a frontend for Neuron, NEST and Brian simulators mentioned in the previous chapter.

4.1 SNN toolbox

SNN toolbox (SNN-TB) [35] is a conversion tool which automates the process of conversion of trained ANN to an SNN. The toolbox uses a deep learning model written in one of the supported ANN frameworks (Currently Keras / TensorFlow [7], Lasagne [12], Caffe [16] and PyTorch [29]). The provided input model is parsed, and a general abstract model in Keras is derived. This internal model is used to achieve the actual conversion to the spiking network. That is achieved by replacing the analogue neurons to spiking integrate-and-fire neurons and altering the connection weights correspondingly. The resulting spiking model can be exported to several spiking simulators or deployed on dedicated neuromorphic chip-sets. Currently, the supported simulation platforms are PyNN, Brian2 and built-in MegaSim and INIsim simulators. Intel Loihi and SpiNNaker project are the supported hardware platforms at the moment. Figure 4.1 illustrates the workflow of the SNN toolbox.

4.2 Keras / TensorFlow

Keras is a high-level deep learning API which can be used with multiple neural network or machine learning toolkits, mainly with TensorFlow. TensorFlow is an open-source machine learning platform. It consists of an interface for formulating machine learning algorithms and system-specific implementations to perform such algorithms.

4.3 PyNN

PyNN is a language for building models of neuronal networks independently of the used simulator. It provides a common API in the Python language

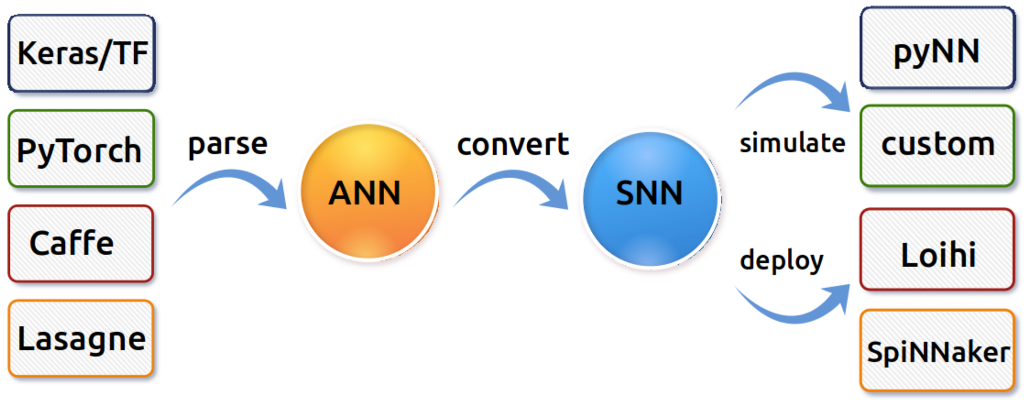

Figure 4.1: The illustration of the SNN toolbox workflow. An arbitrary supported deep learning model is transformed into an abstract Keras model, which is subsequently converted to a spiking network. Source: `https://snntoolbox.readthedocs.io/en/latest/guide/intro.html`

to write the code and run it on multiple backends. PyNN API is designed to support neural networks models at a high-level of abstraction. Currently, Neuron, Brian (version 1) and NEST are supported simulators. PyNN also supports the SpiNNaker and BrainScaleS neuromorphic hardware and is partially compatible with NeuroML model description language. Figure in fig. 4.2 shows the architecture of the PyNN interface [4].

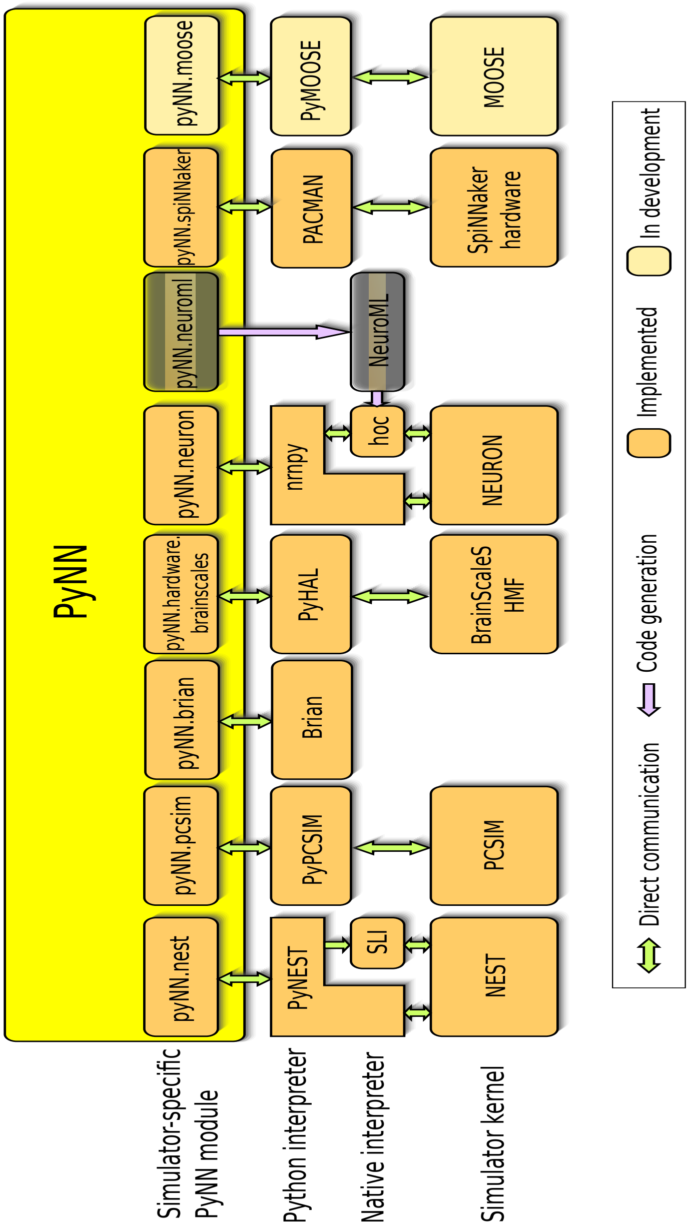

Figure 4.2: PyNN architecture

5 Exploratory use cases

This chapter describes the initial experiments with the Nengo simulator and the SNN-Toolbox, which were selected as the most appropriate tools for subsequent use in this work. The decision was based on the pieces of information stated about the individual tools in the previous chapters. Both tools make it possible to use analogue neural networks to train the model and then convert it to the corresponding spiking neural network for inference. This approach is more utilisable for practical needs at the moment because other approaches of using SNNs suffer from issues described in chapter 2 above. Thus, it is complicated to use them for any practical application such as image recognition. However, they are still necessary for research in the area of spiking networks and scientific experiments. Furthermore, the other simulators proved to be quite hard to work with, for example, because of unreliable interfaces of their dependencies or other issues.

5.1 Nengo

In the beginning, the Nengo simulator was inspected more thoroughly. It helps new users with its simple graphical interface, which contains several tutorials and examples. The tutorials focus mainly on an explanation of the Neural Engineering Framework and Semantic Pointer Architecture methods introduced by authors of the Nengo simulator, rather than on SNNs generally. The GUI is divided into a script editor and visualisation area, where the objects, created in the script editor, are visualised in current time. These visualisations help understand the whole course of the simulation of the spiking neural network. An additional package `nengo-gui` must be installed to access the graphical environment. The environment is an HTML5-based application, which is accessed through an internet browser. The interactive editor supports Nengo and NengoDL backends for running the simulation.

5.2 Spaun 2.0

The next goal was to execute the Spaun 2.0 model to inspect the capabilities of the Nengo even further. The objective was to simulate the handwritten digit recognition, which is one of the cognitive tasks that Spaun 2.0 supports. Although this use case should be quite straightforward according to the documentation of the Nengo, a few issues appeared. Firstly, there does not exist any documentation for the model. It means that the only way how to get information about its execution is to correspond with its authors about every detail, for example, to discover that the model is not compatible with recent versions of the Nengo package. The other problem is probably caused by the fact that the Spaun 2.0 model can require more than 26 gigabytes of memory to execute. Even though the model was executed on a cloud platform, which provides such resources, the program still finished the simulation with memory allocation errors.

5.3 Using the same pre-trained model for the Nengo and SNN-Toolbox

The next objective was to assess conversion abilities of both Nengo and SNN conversion toolbox on a similar use case. Conversion of a simple neural network model represented by Keras interface was selected as a suitable use case. The model was assembled and trained as an ordinary analogue neural network on a MNIST dataset. The model architecture must have been constrained to allow comparison of the two converters because Nengo currently does not support batch normalisation. Although, the NengoDL converter should be able to handle unsupported layers (by falling back to ANN, as explained below), it did not work if batch normalisation layers were in the model.

There was an issue with using the same network model directly because there exist two versions of Keras framework at this time. SNN-TB uses a standalone multi-backend version of Keras, which supports Theano and CNTK toolkits alongside with the TensorFlow backend. However, this version of Keras will not be further developed in the future. The other version of Keras is integrated directly into TensorFlow as its high-level API for neural network development. This version of Keras interface is used in Nengo platform. However, both versions work similarly, and the model exported from the multi-backend version can be migrated into the other without further complications.

5.3.1 Model architecture

The final selected model was a convolutional network, which consisted of multiple consecutive convolutional layers, and contained an average pooling layer. A visualisation of the network architecture is shown in fig. 5.1. The first convolution layer has consisted of thirty-two filters. The kernel size was 3×3. The second convolutional layer had sixty-four 3×3 filters with a stride length of 2×2. All layers have rectified linear unit (ReLU) as activation function, and the weight matrices were initialised with He uniform initialiser. The only exception is the last dense layer, which uses softmax activation. The presented model was configured to use the Adam [21] optimizer and categorical cross-entropy as a loss function. 25% of the training set was held out to serve as validation data. The ANN was trained on the remaining 75% of the MNIST training set for 2 epochs. The batch size was set to 1024. The trained Keras model was exported to the file system to be accessible to the conversion tools.

5.3.2 Conversion with SNN Toolbox

Conversion of the network with SNN-TB is quite straightforward. There must be created a configuration file, which contains all necessary parameters that must be passed to the `snntoolbox.bin.run.main()` function of the toolbox. The configuration file can be written with a few simple utility functions of the SNN-TB. The configuration file contains paths to the model and dataset as well as a specification of the used simulator and its parameters, parameters of the simulated spiking neuron cells or plots and variable logs, which should be generated by the toolbox. The SNN-TB then automatically performs the conversion of the network and evaluates it on the given test samples. There is an option to evaluate the ANN input model so the results can be compared instantly.

The configuration file specified following values for the execution of SNN-TB: The toolbox was set to evaluate the input model and parse it to SNN-TB internal representation. The activation value in the 99.9 percentile was used to normalise the parameters of each layer. Then, integrate-and-fire neurons replaced analogue neurons and by that was the ANN converted into SNN. The SNN was then evaluated on the whole test set of the MNIST dataset. The default simulator of the SNN-TB (INI simulator) was used because it supports most of the potential functionality of the conversion toolbox, so it was not necessary to introduce other limitations to the selected model. Simulation of each sample was configured to run for 120 ms time interval with a time resolution of 0.1 ms. Because the built-in INI simulator supports parallel simulations, the test samples were processed in batches of 500 samples. The simulated spiking neurons had a refractory period (a recovery phase of the neuron during which it can not spike again) and a delay parameters equal to the time resolution of the simulator ($dt = 0.1\ ms$).

There is an example of the configuration in listing 5.1.

Listing 5.1: Configuration of the SNN-TB

```
# import SNN-Toolbox functions
from snntoolbox.utils.utils import import_configparser
# initial parameters
dt = 0.1  # Time resolution of simulator.
snn_sim = 'INI' # Name of the used simulator.
timesteps = 30 # Time of simulation of one sample in
   ms.
sim_batch = 500 # Number of samples simulated in
   parallel.

# Create a config file with an experimental setup for
   SNN Toolbox.
configparser = import_configparser()
config = configparser.ConfigParser()

config['paths'] = {
     'path_wd': path_wd,  # Path to model.
     'dataset_path': path_wd,  # Path to dataset.
     'filename_ann': model_name  # Name of input model.
}
config['tools'] = {
     'evaluate_ann': True,  # Test ANN on dataset
        before conversion.
     'normalize': True,  # Normalize weights
}
config['simulation'] = {
     'simulator': snn_sim,
     'duration': timesteps,
     'num_to_test': n_samples,
     'batch_size': sim_batch,
     'dt': dt
}
config['cell'] = {
     'tau_refrac': dt,  # Refractory period
     'delay': dt
}
```

5.3.3 Conversion with NengoDL Converter

Several adjustments were needed to execute the selected model of the spiking network in the Nengo simulator. The input samples had to be reformatted to conform with the format which is required by the Nengo simulator. A new dimension was added to the data array to spread the data over the time of the simulation. The input image was flattened into a single dimension. The `nengo_dl.Converter` class served for the conversion of the ANN.

The converter can be configured to allow a fallback option for features which are not supported by the core Nengo simulator. This option causes that the layers which do not have spiking equivalents are wrapped into NengoDL's `TensorNode` class. This option makes it possible to use NengoDL to run even more complex network models. However, networks which contain `TensorNodes` can be run only with NengoDL simulator and cannot be ported to other Nengo environments such as Nengo-core simulator or Nengo backends for hardware platforms.

The last layer of the used model was excluded to avoid this silent drop back to a hybrid analogue network before it was used with Nengo. That was necessary because Nengo does not define a conversion of the softmax activation function. Then, NengoDL converter parsed the Keras model into the Nengo model. However, this model still consisted of non-spiking neurons. NengoDL's `RectifiedLinear` neurons were used because they provided an approximation for the spiking neurons and would be replaced with the `nengo.SpikingRectifiedLinear` neurons later. This model was used for training during which parameter optimizations were performed. A Nengo `Probe` object was needed to obtain the output of the last network layer. That output was needed to train the network. The converted model was compiled and trained with the same or equivalent arguments as the original Keras model, that means that Adam optimizer and categorical cross-entropy were specified during the compilation of the model. There is demonstrated a part of the code to convert and train network in listing 5.2. The `nengo_dl` module implements a simple interface for compilation, training and evaluation of the model in addition to the standard Nengo syntax. The implemented interface is designed to be highly similar to the Keras API as can be seen from the `compile` and `fit` methods in listing 5.2.

Listing 5.2: Conversion with NengoDL Converter class.

```
# convert keras model to nengo network
converter = nengo_dl.Converter(model)

# add a probe to the output layer
with converter.net:
    output_p = converter.outputs[converter.model.
       output]

# Compile the model for NengoDL
with nengo_dl.Simulator(converter.net, minibatch_size=
   sim_batch) as sim:
    sim.compile(
        optimizer=tf.optimizers.Adam(),
        loss={
            output_p: tf.losses.
               CategoricalCrossentropy(from_logits=True
               )
        },
        loss_weights={output_p: 1}
    )
    # run training
    sim.fit(
      {converter.inputs[converter.model.input]:
         x_train},
      {output_p: y_train},
      epochs=epochs,
      validation_split=0.25
    )

    # save the parameters to file
    sim.save_params(model_params)
```

The model fitted on 75% of the training set, and the remaining 25% was used for validation. The training was performed in two epochs, as was with the original Keras model. The retrieved parameters were saved on the file system for the testing phase.

For the evaluation, the original Keras model was converted with the NengoDL Converter again, but this time, the non-spiking analogue neurons were swapped for spiking leaky integrate-and-fire neurons. A low-pass synaptic smoothing with parameter 0.01 was applied to all the neurons. This operation causes that the spikes in the network are averaged over a short time window. This method helped to remove excess noise from the network output. The input data needed to be repeated once for each timestep of the simulation.

Finally, NengoDL simulator ran the prediction of the MNIST test set. Each sample was simulated for 30 timesteps. The parameters exported from the previously created NengoDL model were applied against the spiking network.

5.3.4 Comparison of the tools and their results

Both tools were tested on a subset of 8000 images of the MNIST test set, which has 10000 images in total. The set must have been reduced because the Nengo demanded too much memory. The simulation required more than 25 GB when it was executed on the 10000 samples. Although, the SNN-TB did not have such problems with the whole testing set, the number of tested samples was made equal to compare the results. The original Keras model was also tested on the 8000 samples subset for the same reason. The base Keras ANN achieved an accuracy of 96.55%. The SNN converted with SNN-TB performed slightly worse, and that is affirmative to results reported by previous works about spiking networks. The NengoDL simulator demonstrated the best accuracy, which is comparable with results of similar network architectures built with Nengo platform. The results can be seen in table 5.1. It should be stated that these results were not validated in any way. The primary purpose of this part of the work was to find out if the tools can be used for conversion of the same base model and how to achieve that.

Model	Test accuracy (%)
Keras base model	96.55
SNN in SNN-TB	94.67
SNN in NengoDL	98.45

Table 5.1: Test accuracies achieved on 8000 samples of the MNIST test set.

Several findings about the used tools emerged during this experiment. From the software engineering point of view, it seems that the Nengo ecosystem is better designed and might be extended easily. However, it takes time to fully understand it and use its full potential, which is far beyond the illustrated bits. It would probably require a more profound knowledge of the associated methods from NEF and SPA. The SNN conversion toolbox is, on the other hand, well suited for quick trials or familiarization with spiking networks. It offers an uncomplicated way to convert state-of-the-art ANNs to spiking alternatives, which works out of the box. However, it is also probably its sole purpose because it does not offer any interfaces for modifications of its functioning. All potential extensions have to be integrated directly into its source code.

Further investigation of the tools should be carried out to evaluate their time and memory requirements because there might be a significant difference as became apparent because of the reason described earlier. That could be achieved by averaging memory consumption and run-time across multiple repetitions of the simulations or by looking into the source code for possible design dissimilarities.

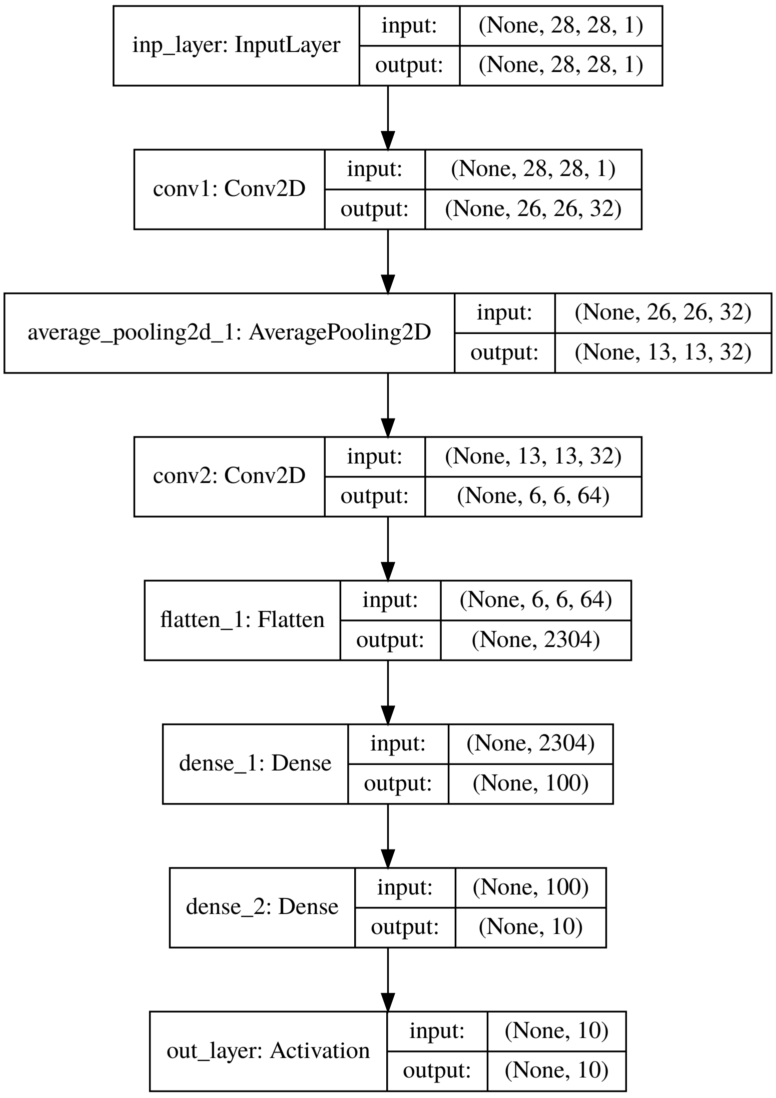

Figure 5.1: The Keras model of a convolutional network used for classification of the MNIST dataset.

6 Spiking classification on large multi-subject P300 dataset

A convolutional neural network described in [42] was converted with the SNN conversion toolbox. SNN-TB was selected for the conversion over Nengo simulation platform because the CNN contained batch normalisation layer and softmax activation function, which prevented Nengo to convert the network successfully.

The original CNN was used for binary classification of event-related potential (ERP) signals retrieved from large multi-subject P300 dataset, which was presented in [27]. The dataset was retrieved from EEG experiments, where subjects were told to concentrate on a single-digit number (from 0 to 9). Experimenters were trying to guess the number, which the subject selected, from recorded EEG data while the subject was stimulated with random sequences of all digits. It is described in [42] how were extracted short intervals of signal around all target stimuli from this dataset. An equal amount of non-target samples was acquired randomly from the remaining data. This newly-emerged dataset consisted of two distinct equal-sized classes, the target epochs (i.e. short time intervals of EEG data around the stimulus, which was the subject's thought number) and the non-target epochs. Each epoch contained three EEG channels and was 1200 ms long. The sampling frequency was 1 kHz. That means that each sample of the network input was a 3×1200 matrix.

The input data were preprocessed in a similar fashion as in [42]. A modified subset [28] of the original dataset was used for this work. This dataset was created during the classification with the original CNN. The input file contained one matrix of feature vectors for all target data and one matrix of feature vectors for all non-target data. Those matrices were concatenated to create the input vectors. The corresponding ground truth labels were produced by one-hot-encoding of the input data. Then, severely damaged epochs were discarded from the input the same way as in the original article [42]. It was done by discarding all epochs, where amplitudes were higher than 100 μV in any of the three channels. The steps of preprocessing is visualised in fig. 6.1.

These preprocessed data were divided into training and test subsets by

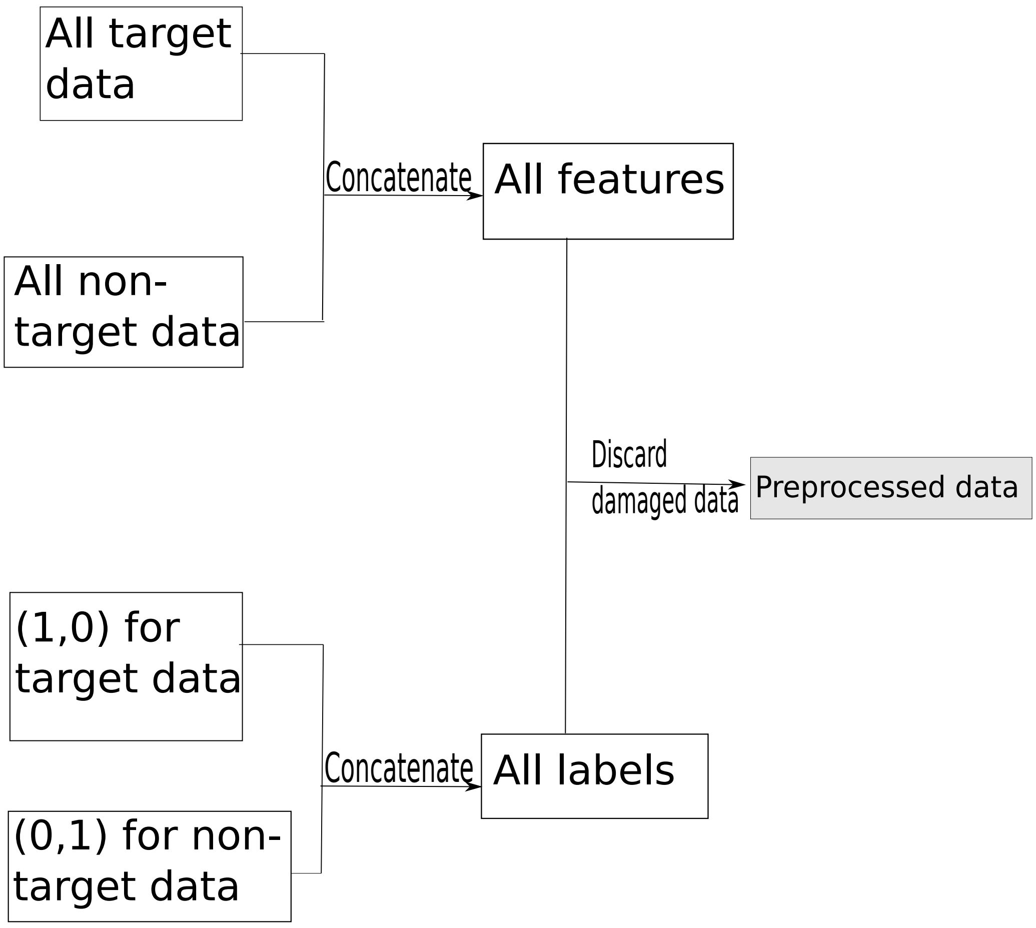

Figure 6.1: Flow of data preprocessing.

randomly picking out 25% of the dataset samples into the test set. The test set was saved to the file system so it could be passed to SNN-TB during conversion later.

The network architecture of the CNN model was replicated entirely from the original work. This model was cross-validated the same way as in the original article. That means 30 iterations of the Monte-Carlo cross-validation were performed, and each iteration held out 25% of the training set to create validation subset. In each iteration, a new instance of the network was created and trained. The training was done in 30 epochs while early stopping with patience value of 5 was used. After that, the trained ANN was evaluated on the test set. 10% of the dataset samples, which were used during the training, were saved for normalisation of the layer weights. SNN-TB did this normalisation before the conversion process. All the divisions of the dataset are shown in fig. 6.2.

The conversion toolbox parsed the evaluated ANN into its internal Keras

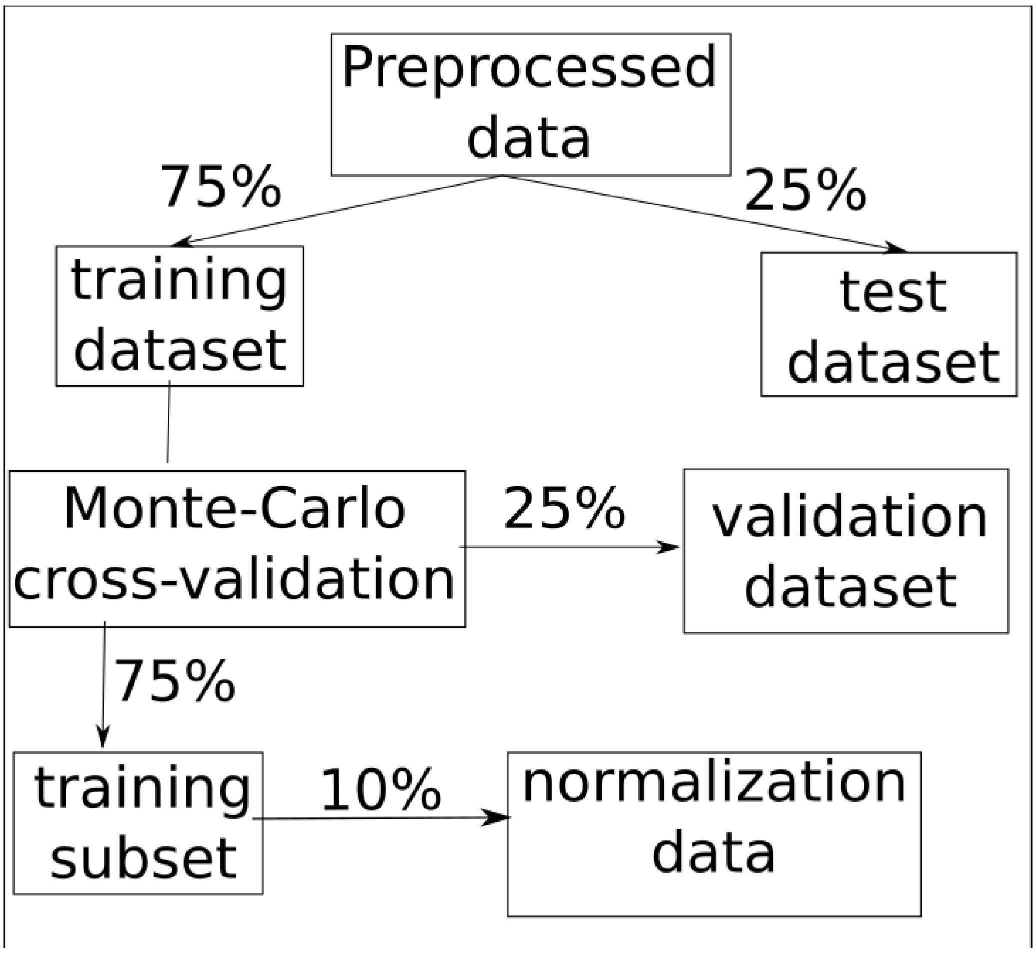

Figure 6.2: Data splitting. Proportions of the split are marked by percentages along the edges.

representation, weights of the parsed model were normalised with previously saved part of the training data. The normalised model was converted into a spiking model, and the spiking model was evaluated on the test set each iteration. The evaluation of each input sample of the spiking network was simulated for 50 timesteps. The simulator evaluated the whole test set gradually in batches of 49 samples.

Accuracies from the 30 iterations of cross-validation were recorded for both networks. The other metrics, which are shown in the article about the original neural network (AUC, precision, recall), could not be computed, because SNN-TB does not provide a simple way to define arbitrary metrics at this time [1].

6.1 Results

The average accuracy and sample standard deviation from the 30 cross-validation iterations of the experiment are shown in table 6.1. Figure 6.3 shows a comparison of the accuracies from the individual iterations. The decrease of accuracy on the SNN was expected as previous works from the field reports similar drops of performance. Also, this experiment validated only a single combination of network parameters, which is stated in the previous section. There are plenty of parameters (for the used neuron type, network or the whole simulator), so there surely is a space for fine-tuning and optimization. Whole other area of experimentation would appear if the SNN-TB was used with any other supported backend. That would possibly allow using other neuron types.

Model	Average test accuracy	Sample standard deviation
Original model	0.6370	0.0066
Converted model	0.5720	0.0156

Table 6.1: Average accuracy and sample standard deviation of the classification results received during cross-validation.

[1]This issue was discussed with a contributor to the SNN conversion toolbox on the project's GitHub page. The discussion did not lead to a successful implementation of the custom metrics before this work was completed. A link to the website `https://github.com/NeuromorphicProcessorProject/snn_toolbox/issues/56` is provided with a hope that the solution might appear there in future.

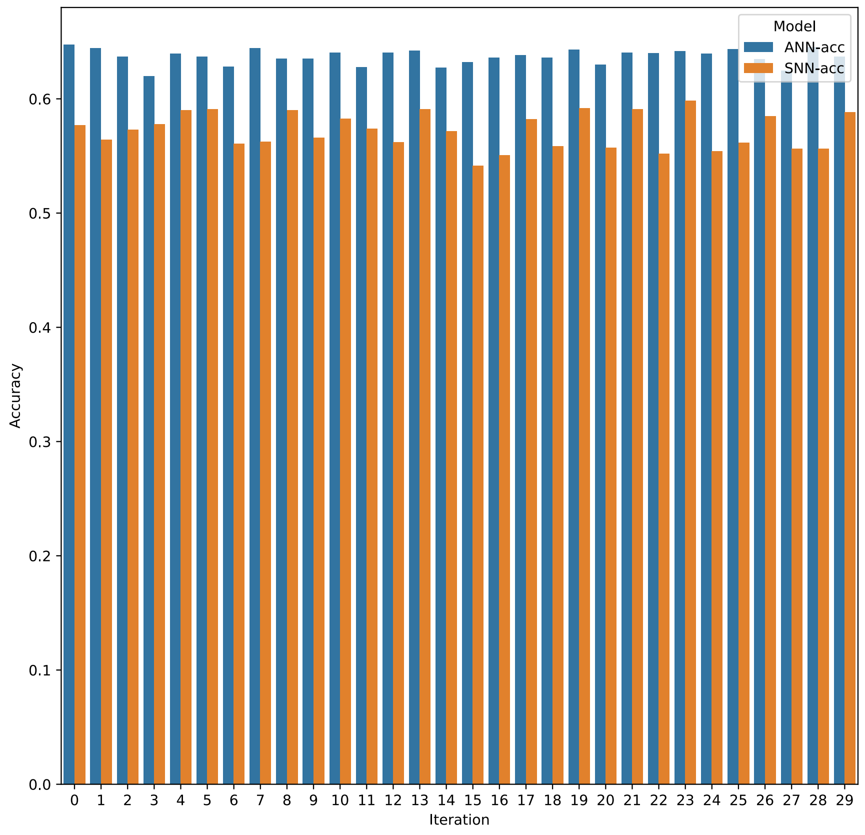

Figure 6.3: The accuracies across all iterations of the experiment.

7 Conclusion

This thesis clarifies the main distinctions between analogue neural networks and spiking neural networks, depict the current state of research in the area of the spiking networks and demonstrate the findings on a selected machine learning problem with the help of selected instruments. It was described how spiking networks could be beneficial in research for their biological similarities with biological neural systems, and their fitness for utilisation in embedded applications. The work later focuses on the collation of the most advanced simulators of spiking networks. Nengo ecosystem and SNN-Toolbox were selected for the subsequent experiments. It was outlined how these tools work on a use case where a base analogue network was converted with both tools and evaluated on the MNIST image classification dataset. It was shown that both instruments were able to convert an original non-spiking network to a spiking alternative with some modifications. There were also described personal experiences and impressions from using the tools. The results of both converted networks were comparable with the results of the base model.

The main contribution of this work is a conversion of an earlier convolutional network for event-related potentials classification. The achieved average accuracy (57.2%) does not surpass the accuracy of the original network. Values of other metrics could not be compared with original work because the used conversion toolbox does not make it possible to calculate those metrics. The results of this experiment could be improved with a more profound research of the effects of individual simulation parameters on the performance of the spiking networks and using algorithms for optimization of those parameters. Another improvement in the classification of event-related potentials could be achieved if a new model, which would exploit specific qualities of the spiking networks, was created.

This work focused on computer simulations of the spiking networks and overlooked available neuromorphic platforms such as Intel Loihi or SpiNNaker chips. It would be an impressive continuation of this work to experiment with these novel hardware architectures.

All source codes of the used and created applications were made public[1]. Even though the achieved results of any of the applications are not groundbreaking, the work and the personal experiences with the mentioned tools might be helpful to subsequent works in this field.

[1]All materials related to this thesis were published on `https://github.com/RKCZ/Extension-of-neural-network-architecture`.

Bibliography

[1] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C. Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. 2014. Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7, 48. ISSN: 1662-5196. DOI: 10.3389/fninf.2013.00048. Retrieved 03/11/2020 from https://www.frontiersin.org/articles/10.3389/fninf.2013.00048/full.

[2] K.A. Boahen. 2000. Point-to-point connectivity between neuromorphic chips using address events. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47, 5, (May 2000), 416–434. ISSN: 1558-125X. DOI: 10.1109/82.842110.

[3] [n. d.] Brian. Retrieved 03/11/2020 from https://groups.google.com/forum/#!aboutgroup/briansupport.

[4] Daniel Brüderle. 2009. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System*. Dissertation. University of Heidelberg. DOI: 10.11588/heidok.00009656. Retrieved 03/11/2020 from http://archiv.ub.uni-heidelberg.de/volltextserver/9656/.

[5] Yongqiang Cao, Yang Chen, and Deepak Khosla. 2015. Spiking Deep Convolutional Neural Networks for Energy-Efficient Object Recognition. *International Journal of Computer Vision*, 113, 1, (May 1, 2015), 54–66. ISSN: 1573-1405. DOI: 10.1007/s11263-014-0788-3. Retrieved 03/11/2020 from https://doi.org/10.1007/s11263-014-0788-3.

[6] Nicholas T. Carnevale and Michael L. Hines. 2006. *The NEURON Book*. Cambridge University Press, (January 12, 2006). 399 pages. ISBN: 978-1-139-44783-6.

[7] François Chollet. 2015. Keras. (2015). Retrieved 03/30/2020 from https://keras.io.

[8] Feng-Xuan Choo. 2018. *Spaun 2.0: Extending the World's Largest Functional Brain Model*. University of Waterloo, (May 17, 2018). https://uwspace.uwaterloo.ca/handle/10012/13308.

[9] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition.* 2012 IEEE Conference on Computer Vision and Pattern Recognition. (June 2012), 3642–3649. DOI: `10.1109/CVPR.2012.6248110`.

[10] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1, (March 17, 2016). arXiv: `1602.02830 [cs]`. Retrieved 03/25/2020 from `http://arxiv.org/abs/1602.02830`.

[11] Andrew P. Davison, Daniel Brüderle, Jochen M. Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. 2009. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2, 11. ISSN: 1662-5196. DOI: `10.3389/neuro.11.011.2008`. Retrieved 03/11/2020 from `https://www.frontiersin.org/articles/10.3389/neuro.11.011.2008/full`.

[12] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, diogo149, Brian McFee, Hendrik Weideman, takacsg84, peterderivaz, Jon, instagibbs, Dr. Kashif Rasul, CongLiu, Britefury, and Jonas Degrave. 2015. Lasagne: First release. Zenodo, (August 13, 2015). DOI: `10.5281/zenodo.27878`. Retrieved 03/30/2020 from `https://zenodo.org/record/27878#.XoJJW4gzaMo`.

[13] Chris Eliasmith. 2013. *How to Build a Brain: A Neural Architecture for Biological Cognition.* Oxford University Press, (June 27, 2013). 476 pages. ISBN: 978-0-19-979454-6.

[14] Steve B. Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. 2014. The SpiNNaker Project. *Proceedings of the IEEE*, 102, 5, (May 2014), 652–665. ISSN: 1558-2256. DOI: `10.1109/JPROC.2014.2304638`.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27.* Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors. Curran Associates, Inc., 2672–2680. Retrieved 03/11/2020 from `http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`.

[16] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (MM '14). Association for Computing Machinery, Orlando, Florida, USA, (November 3, 2014), 675–678. ISBN: 978-1-4503-3063-3. DOI: `10.1145/2647868.2654889`. Retrieved 03/30/2020 from `https://doi.org/10.1145/2647868.2654889`.

[17] Jakob Jordan, Håkon Mørk, Stine Brekke Vennemo, Dennis Terhorst, Alexander Peyser, Tammo Ippen, Rajalekshmi Deepu, Jochen Martin Eppler, Alexander van Meegen, Susanne Kunkel, Ankur Sinha, Tanguy Fardet, Sandra Diaz, Abigail Morrison, Wolfram Schenck, David Dahmen, Jari Pronold, Jonas Stapmanns, Guido Trensch, Sebastian Spreizer, Jessica Mitchell, Steffen Graber, Johanna Senk, Charl Linssen, Jan Hahne, Alexey Serenko, Daniel Naoumenko, Eric Thomson, Itaru Kitayama, Sebastian Berns, and Hans Ekkehard Plesser. 2019. NEST 2.18.0. Version 2.18.0. (June 27, 2019). `https://doi.org/10.5281/zenodo.2605422`.

[18] Jülich Aachen Research Alliance. 2015. NEST: The Neural Simulation Tool. (2015). Retrieved 03/11/2020 from `https://www.nest-simulator.org/wp-content/uploads/2015/04/JARA_NEST_final.pdf`.

[19] T. Kalganova, A. Byerly, and I. Dear. 2020. A Branching and Merging Convolutional Network with Homogeneous Filter Capsules.

[20] Minje Kim and Paris Smaragdis. 2016. Bitwise Neural Networks, (January 22, 2016). arXiv: `1601.06071 [cs]`. Retrieved 03/25/2020 from `http://arxiv.org/abs/1601.06071`.

[21] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization, (January 29, 2017). arXiv: `1412.6980 [cs]`. Retrieved 04/13/2020 from `http://arxiv.org/abs/1412.6980`.

[22] Nicolas Le Novère. 2012. *Computational Systems Neurobiology*. Springer Science & Business Media.

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86, 11, (November 1998), 2278–2324. ISSN: 1558-2256. DOI: `10.1109/5.726791`.

[24] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. 2016. Training Deep Spiking Neural Networks Using Backpropagation. *Frontiers in Neuroscience*, 10, 508. ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00508. Retrieved 03/11/2020 from https://www.frontiersin.org/articles/10.3389/fnins.2016.00508/full.

[25] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards Accurate Binary Convolutional Neural Network. In *Advances in Neural Information Processing Systems 30*. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors. Curran Associates, Inc., 345–353. Retrieved 03/25/2020 from http://papers.nips.cc/paper/6638-towards-accurate-binary-convolutional-neural-network.pdf.

[26] Wolfgang Maass. 1997. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10, 9, (December 1, 1997), 1659–1671. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(97)00011-7. Retrieved 03/09/2020 from http://www.sciencedirect.com/science/article/pii/S0893608097000117.

[27] R. Mouček, L. Vařeka, T. Prokop, J. Štěbeták, and P. Brůha. 2017. Event-related potential data from a guess the number brain-computer interface experiment on school children. *Scientific Data*, 4, 1, (March 28, 2017), 1–11, 1, (March 28, 2017). ISSN: 2052-4463. DOI: 10.1038/sdata.2016.121. Retrieved 04/01/2020 from https://www.nature.com/articles/sdata2016121.

[28] Roman Mouček, Lukáš Vařeka, Tomáš Prokop, Jan Štěbeták, and Petr Brůha. 2019. Replication Data for: Evaluation of convolutional neural networks using a large multi-subject P300 dataset. (July 30, 2019). DOI: 10.7910/DVN/G9RRLN. Retrieved 04/28/2020 from https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/G9RRLN.

[29] Adam Paszke, 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*. H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors. Curran Associates, Inc., 8026–8037. Retrieved

03/30/2020 from http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[30] R. Paz, F. Gomez-Rodriguez, M. A. Rodriguez, A. Linares-Barranco, G. Jimenez, and A. Civit. 2005. Test Infrastructure for Address-Event-Representation Communications. In *Computational Intelligence and Bioinspired Systems* (Lecture Notes in Computer Science). Joan Cabestany, Alberto Prieto, and Francisco Sandoval, editors. Springer, Berlin, Heidelberg, 518–526. ISBN: 978-3-540-32106-4. DOI: 10.1007/11494669_64.

[31] José Antonio Pérez-Carrasco, Bo Zhao, Carmen Serrano, Begoña Acha, Teresa Serrano-Gotarredona, Shouchun Chen, and Bernabé Linares-Barranco. 2013. Mapping from Frame-Driven to Frame-Free Event-Driven Vision Systems by Low-Rate Rate Coding and Coincidence Processing–Application to Feedforward ConvNets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35, 11, (November 2013), 2706–2719. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2013.71.

[32] Michael Pfeiffer and Thomas Pfeil. 2018. Deep Learning With Spiking Neurons: Opportunities and Challenges. *Frontiers in Neuroscience*, 12. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00774. Retrieved 03/11/2020 from https://www.frontiersin.org/articles/10.3389/fnins.2018.00774/full.

[33] Dimitri Plotnikov, Bernhard Rumpe, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, and Abgail Morrison. 2016. NESTML: a modeling language for spiking neurons, (June 9, 2016). arXiv: 1606.02882. Retrieved 03/11/2020 from http://arxiv.org/abs/1606.02882.

[34] Bodo Rueckauer and Shih-Chii Liu. 2018. Conversion of analog to spiking neural networks using sparse temporal coding. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018 IEEE International Symposium on Circuits and Systems (ISCAS). (May 2018), 1–5. DOI: 10.1109/ISCAS.2018.8351295.

[35] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. 2017. Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification. *Frontiers in Neuroscience*, 11. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682. Retrieved 03/09/2020 from https://www.frontiersin.org/articles/10.3389/fnins.2017.00682/full.

[36] R. Sathya and Annamma Abraham. 2013. Comparison of supervised and unsupervised learning algorithms for pattern classification. *International Journal of Advanced Research in Artificial Intelligence*, 2, 2, 34–38.

[37] Taylor Simons and Dah-Jye Lee. 2019. A Review of Binarized Neural Networks. *Electronics*, 8, 6, (June 2019), 661, 6, (June 2019). DOI: `10.3390/electronics8060661`. Retrieved 03/25/2020 from `https://www.mdpi.com/2079-9292/8/6/661`.

[38] Terrence C Stewart. 2012. A Technical Overview of the Neural Engineering Framework. Centre for Theoretical Neuroscience, University of Waterloo.

[39] Marcel Stimberg, Romain Brette, and Dan FM Goodman. 2019. Brian 2: an intuitive and efficient neural simulator. *eLife*, 8. ISSN: 2050-084X. DOI: `10.7554/eLife.47314`. pmid: `31429824`. Retrieved 03/11/2020 from `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6786860/`.

[40] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. 2019. Deep learning in spiking neural networks. *Neural Networks*, 111, (March 1, 2019), 47–63. ISSN: 0893-6080. DOI: `10.1016/j.neunet.2018.12.002`. Retrieved 03/11/2020 from `http://www.sciencedirect.com/science/article/pii/S0893608018303332`.

[41] Ruben A. Tikidji-Hamburyan, Vikram Narayana, Zeki Bozkus, and Tarek A. El-Ghazawi. 2017. Software for Brain Network Simulations: A Comparative Study. *Frontiers in Neuroinformatics*, 11, 46. ISSN: 1662-5196. DOI: `10.3389/fninf.2017.00046`. Retrieved 03/11/2020 from `https://www.frontiersin.org/articles/10.3389/fninf.2017.00046/full`.

[42] Lukáš Vařeka. 2020. Evaluation of convolutional neural networks using a large multi-subject P300 dataset. *Biomedical Signal Processing and Control*, 58, (April 1, 2020), 101837. ISSN: 1746-8094. DOI: `10.1016/j.bspc.2019.101837`. Retrieved 04/08/2020 from `http://www.sciencedirect.com/science/article/pii/S1746809419304185`.

[43] Julius von Kügelgen. 2017. *On Artificial Spiking Neural Networks: Principles, Limitations and Potential.* (June 18, 2017).

[44] Davide Zambrano and Sander M. Bohte. 2016. Fast and Efficient Asynchronous Neural Computation with Adapting Spiking Neural Networks, (September 7, 2016). arXiv: `1609.02053 [cs]`. Retrieved 03/11/2020 from `http://arxiv.org/abs/1609.02053`.