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

Bachelor's thesis

Use of spiking neural networks

Plzeň 2021 Václav Honzík

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta aplikovaných věd

Akademický rok: 2020/2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Václav HONZÍK**
Osobní číslo:	**A19B0674P**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informatika**
Téma práce:	**Využití impulzních neuronových sítí**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s aktuálním stavem poznání v oblasti impulzních neuronových sítí.
2. Vymezte zásadní charakteristiky impulzních neuronových sítí a rozdíly vůči klasickým neuronovým sítím.
3. Seznamte se s dostupnými nástroji pro vytváření a simulaci impulzních neuronových sítí.
4. Na základě bodů 1, 2 a 3 navrhněte a implementujte vhodné příklady demonstrující využití impulzních neuronových sítí.
5. Ověřte řešení z bodu 4 na netriviálních případech a 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: **5. října 2020**
Termín odevzdání bakalářské práce: **6. května 2021**

L.S.

Doc. Dr. Ing. Vlasta Radová
děkanka

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

V Plzni dne 26. října 2020

Declaration

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

Plzeň, May 5, 2021

Václav Honzík

Abstract

In the last decade, artificial (analog) neural networks have become the new norm of solving many tasks from the fields of machine learning and native language processing. Despite their success, however, analog networks fail to accurately capture the behaviour of biological neural networks as the model of an analog neuron depends on continuous activations rather than discrete sequences of action potentials. Spiking neural networks, on the other hand, present a new approach to model such biological nets much closely while also attaining performance close to artificial networks. This thesis studies current knowledge of spiking networks and compares them to analog ones. Subsequently, state-of-the-art tools for simulation in the spiking setting are overviewed and their subset is applied to selected brain-computer interface experiments and image datasets.

Abstrakt

Umělé (analogové) neuronové sítě se staly v posledním desetiletí novou normou pro řešení mnoha úloh z oblasti strojového učení a zpracování přirozeného jazyka. Nicméně i přes jejich úspěch, analogové sítě nejsou schopné přesně zachytit chování biologických neuronových sítí, protože model analogového neuronu závisí na spojitých aktivacích místo diskrétních posloupností akčních potenciálů. Impulzní neuronové sítě, na druhou stranu, představují nový přístup jak modelovat biologické sítě mnohem přesněji a zároveň dosahují výkonu velmi blízko analogovým sítím. Tato bakalářská práce studuje současné znalosti z impulzních sítí a porovnává je s analogovými. Následně práce shrnuje nejmodernější nástroje pro simulaci v impulzním prostředí a část z nich je aplikována na vybraných experimentech pro rozhraní mozek-počítač a obrazových datasetech.

Contents

1 Introduction

Artificial neural networks (ANNs) are systems inspired by biological neural networks found in the brains of humans and animals. Compared to their biological counterparts, ANNs were altered to fit a more pragmatic approach in the field of computer science. Similarly to human (or animal) neural network, an analog neural network consists of units called neurons. These neurons are connected with synapses which allow transferring signals (information) between them. The strength of these synapses can be adjusted which allows the system to learn. Such trained networks attain high accuracy in various sophisticated tasks such as image classification, audio and object recognition, trend prediction, and many more.

While ANNs perform very well in many fields, they are still limited in their utilization. To run the majority of high accuracy ANNs, high-end graphics processing units (GPUs) are required for sufficient performance and a low computation time. This makes such networks impossible to implement efficiently in power restricted hardware such as embedded devices. Additionally, these networks are not suitable for biological simulations due to their oversimplified representation of biological neural nets. Spiking neural networks (SNNs) are often thought of as the next generation of computer-simulated neural networks. This type of nets is specifically modelled with both energy consumption and biological plausibility in mind. Thanks to their architecture, SNNs can be utilized in various fields of science, ranging from the energy-saving replacements of conventional ANNs to efficient processors for inputs from event-based sensors [30].

However, in their current state, SNNs are still in early development and require a lot more research to be applicable. This work aims to contribute to the research of spiking neural networks and explore their potential applications. Chapter 2 overviews basic principles of ANNs and SNNs, compares both architectures, and analyses the state-of-the-art in the field. Chapter 3 examines several tools and frameworks that can be used to model and simulate spiking neural networks. A subset of these tools is used in Chapter 4 where several models of spiking networks are applied on image and brain-computer-interface datasets.

2 Non-spiking and spiking neural networks

2.1 Analog neural networks

An analog neural network[1] (ANN) is a system of processing units that are connected together to simulate desired behaviour. Such units are typically referred to as neurons and their connections as synapses due to the resemblance to biological neural networks.

A neuron in an analog network is a cell that transforms a specific input signal to a corresponding output signal. These transformations are achieved using the neuron's activation function. The function receives signals (information) from synapses (typically real numbers), sums them (or they are summed beforehand) and produces an output (a visualization can be seen in Fig. 2.1). Synapses are not only used to transfer the signal but also to control its strength. If the summed up input is strong enough, the activation "fires", i.e., it produces a strong output. Not every function can, however, be used as activation and specific properties are required. Typical requirements involve non-linearity and differentiability. Non-linearity allows the network to perform complex operations, whereas differentiability is usually required by state-of-the-art training methods since they use some form of a gradient descent.

Neurons group into layers where neurons in each layer use the same type of activation. The information is fed to the input layer (the first layer in the network) and then gradually processed by the following layers (hidden layers) until the result is produced in the output layer. According to the number of connections in the layer, it can be either dense, i.e., each neuron connects to all neurons from the previous layer, or sparse - with a lower number of synapses.

Depending on the way the information is transferred, neural nets can be either feedforward or recurrent (see Fig. 2.2). Feedforward networks allow the information to only move forward, i.e., the output of a specific neuron cannot be fed to it again. Recurrent networks (RNNs), on the other hand, permit feedback loops and the signal can be even transferred to the previous

[1] In this context, the term analog neural network and artificial neural network are equivalent. Even though spiking network is considered artificial as well, it is commonly not referred to it this way.

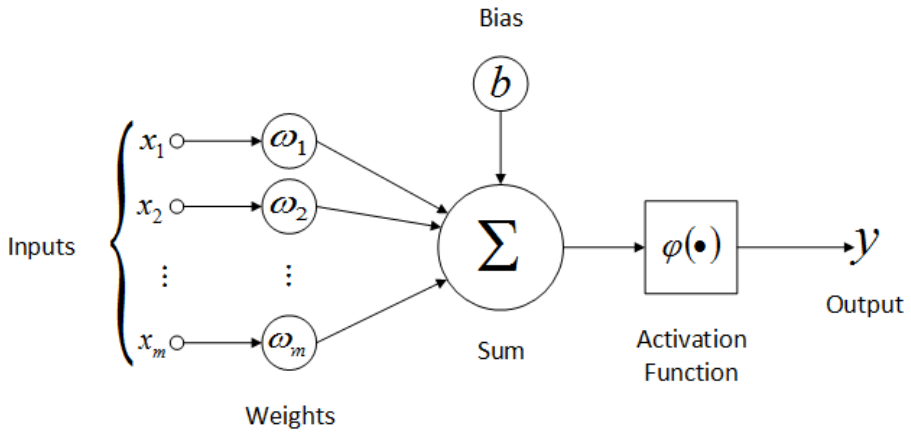

Figure 2.1: Visualization of an analog neuron. The input is a series of real numbers multiplied by the weights (synapses), summed up, and fed to the activation function, producing an output. An additional bias can be added as well, which is typically a real number. Source: [28]

layers.

If the network has two or more hidden layers, it is called a deep neural network (DNN). These networks are more capable in real-world applications than shallow ANNs (with one or zero hidden layers) as they contain more trainable parameters and transform the data in a non-linear fashion. With the current technology, DNNs can comprise hundreds of hidden layers and achieve high performance in various complex AI tasks, sometimes even surpassing humans.

2.1.1 Applications

One of the main applications of neural networks is supervised learning. Supervised learning tasks involve learning a general pattern across a given data. This pattern can then be used to predict on previously unseen data, ideally with high accuracy. Each sample from the learning dataset comprises two parts - input data (features) and a label. The model has to learn the pattern by adjusting its parameters so the predicted label is the same as the target (original) label.

The two most common applications of supervised learning are classification and regression tasks. Classification tasks concern with selecting a correct label (class) for specific input. Depending on the requirements, the model can choose one or multiple classes. There exist many real-world applications such as optical character recognition, spam detection, image

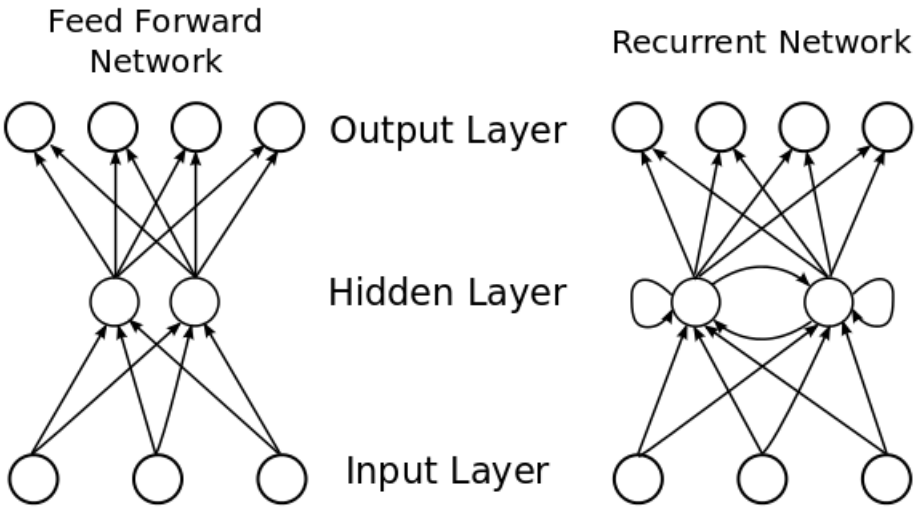

Figure 2.2: Visualization of feedforward and recurrent neural networks. Source: [17]

segmentation, and countless others. Regression also involves determining a relationship between the given set of features and labels. Unlike classification, however, regression models are used to predict continuous values (not a distinct class). The most common applications of regression are risk-factor analysis, market predictions, sports analysis, and so forth.

In the case of neural networks, backpropagation algorithms are used to train the ANN for a specific supervised learning task. These methods calculate the gradient of the loss function (a function that computes the difference between the network's output and the label) and use it to adjust the parameters of the network appropriately. Typically, the training of a neural network requires more samples than less complex algorithms such as naive Bayes, support vector machines or linear regression, which usually makes ANNs unpractical for small datasets. In some cases, the learning of neural network may not converge to optimal parameters and the network becomes either overfit or underfit. Overfit networks tend to perform well on training data and poorly on testing data, while underfit networks often perform poorly on both. There are many approaches to solving fitting issues, e.g., by modifying the training dataset - expanding it, improving its quality or using entirely different samples. The overall performance can also be improved by applying some sort of regularization to the network - dropout layers [40], batch normalization layers, and L1 or L2 regularization.

Unsupervised learning is another field neural nets perform exceptionally well in. The main difference is that in unsupervised learning, no labels are available. The goal of such tasks is to learn without any external error cor-

rection, i.e., it is not possible to adjust the model's parameters according to the labels. The most common application of unsupervised learning is cluster analysis (i.e., grouping data with similar properties to the same groups) and dimension reduction.

As of right now, there exists many specializations of analog networks for both unsupervised and supervised learning. The following three sections serve as a very brief overview of the most popular architectures of ANNs.

2.1.2 Generative adversarial networks

An example of an architecture used in unsupervised learning (though not exclusively) are generative adversarial networks (GANs) [15]. GANs are used for generative modelling - a task that is concerned with creating new (artificial) samples from an existing set of data. In recent years, these networks have found especially huge success in problems regarding image generation. One such example [23] used GANs to generate additional samples to improve (deliberately) imbalanced versions of popular datasets such as MNIST [21] and CIFAR-10 [20] which resulted in improved accuracy on the datasets.

2.1.3 Convolutional networks

Another state-of-the-art architecture used in ANNs are convolutional neural networks (CNNs). CNNs expand on the concept of feedforward networks by introducing convolutional and pooling layers. Both types of layers are typically placed before the set of dense layers and their task is to perform feature extraction (see Fig. 2.3).

Convolutional layers take in a vector/tensor (pixel grid, signal data, etc.) and apply a linear transformation - a convolutional filter. The filter extracts specific features such as shapes, lines, or other domain-specific objects that are subsequently going to be processed by the rest of the network. With more convolutions applied, the extracted features become more complex which, in theory, should increase the probability of correct classification.

Each convolutional layer is often followed by a pooling layer. A pooling layer downsamples the output to introduce a translation invariance to small shifts and distortions [46]. In practice, this means that the network should be able to recognize the same objects in slightly different positions and rotations. The result of the last pooling layer is then fed to a sequence of dense layers which serve the same purpose as in a conventional classifier. CNNs perform especially well on image-based datasets such as MNIST, Fashion

MNIST [45], CIFAR-10, and others. Other applications involve native language processing, signal processing or classification of medical data.

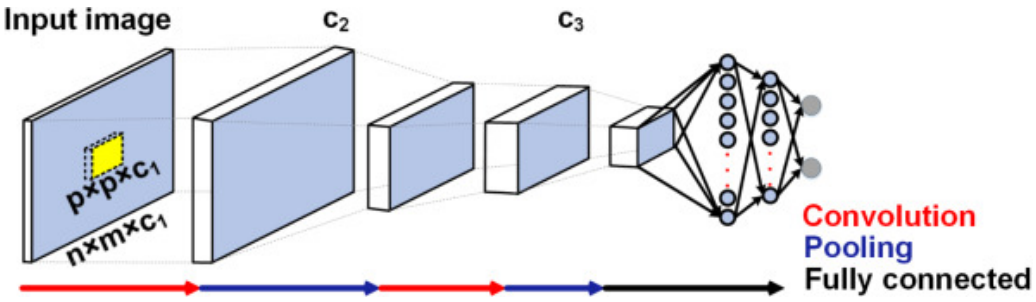

Figure 2.3: Visualization of the CNN architecture. Source: [42]

2.1.4 LSTM networks

Long short-term memory networks (LSTMs) are RNNs that present state-of-the-art machines for processing time-series data. Such networks contain specialized LSTM cells that help the network to overcome long-term dependencies and vanishing gradient problem introduced by recurrent loops.

The concept of LSTM cells is not unified, and there exist various implementations of LSTM neurons. The most basic model is a recurrent neuron with a forget gate. A forget gate is typically represented by a vector which is used to determine whether the information in the cell should be removed or not. Other more advanced models involve LSTMs using peephole connections, gated recurrent units or minimal gated units [47].

Apart from the analysis of time-series data, the application of LSTMs overlaps with CNNs as both are used in areas such as signal processing. There even exist hybrid networks that use both LSTM and CNN layers. One example is shown in this work [49] that has used such "C-LSTM" network for text classification.

2.2 Spiking neural networks

Spiking neural networks (SNNs) represent a new generation of computer-simulated neural networks. SNNs are designed to model biological neural networks more precisely while also aiming to improve several shortcomings of analog nets such as slow response times, high power consumption or lack of asynchronous computing.

Due to the expectations set, the structure of spiking networks needed to be changed. A spiking network consists of spiking neurons (see Fig. 2.4),

which are fundamentally different from the neurons found in ANNs. While spiking neurons also share information via synaptic connections, they do not have an activation function. Instead, every spiking neuron has a membrane potential. Neurons receive and share information via sequences of action potentials, also known as spike trains [42], which alter the membrane potential. Whenever a certain threshold voltage is exceeded, the neuron produces a spike (a stimulation) and the membrane potential is reset towards some defined baseline. This phenomenon is also commonly referred to as a firing of the neuron.

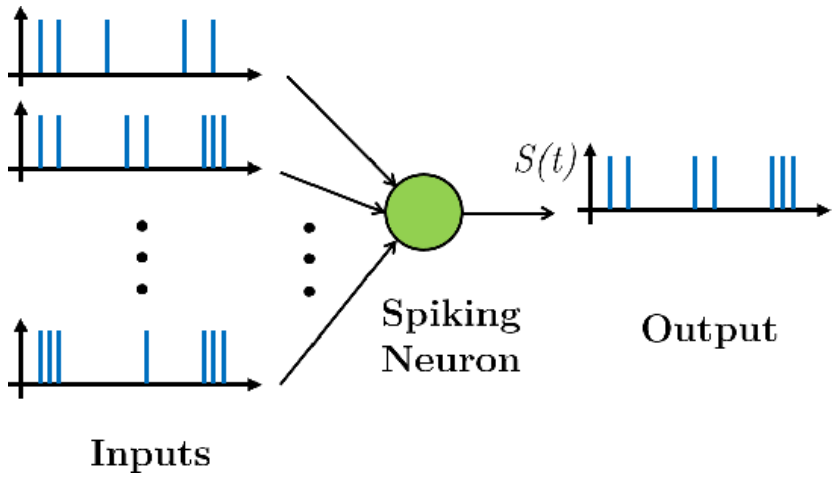

Figure 2.4: Visualization of a spiking neuron. Inputs are sequences of spikes (blue) which influence neuron's membrane potential. The output is also spikes that are generated upon exceeding the membrane potential. Source: [1]

Usually, all spikes are assumed to be stereotypical events which reduces the production of spikes to two main factors [30]. The first factor is the timing of the incoming spikes such as firing rates (how many spikes were fired in a given time frame) and firing patterns. Together, both properties are the closest representation of an activation function in an analog network. The second factor involves the types of synapses used to transfer the spike sequences. Synapses in the spiking setting can be either excitatory or inhibitory. The excitatory synapse increases the membrane potential, while the inhibitory decreases it. As is the case with analog networks, synapses in spiking nets are also represented by real-valued weights. Positive weights are typically used for excitatory synapses while negative weights are used for inhibitory ones.

The presence of action potentials fundamentally changes the way spiking networks process information. The signal in SNNs is not continuous

or differentiable due to the discrete nature of spikes. Even though this reduces the overall usability of backpropagation methods, this property can be utilized to make the information processing asynchronous. In an ideal case, each neuron behaves as an independent processing unit that is not synchronized with the rest of the network. This means that if there are no spike trains present, the neuron can be fully turned off or can at least consume less energy. If implemented well, and there is an overall low number of spikes to encode inputs, this can lead to a very power-efficient implementation. The reduction of power consumption is crucial because it makes it feasible to use SNNs in embedded devices, which cannot be achieved by the state-of-the-art ANNs. Spiking nets can be improved even further if paired with event-based sensors. This addition makes it possible to detect the first approximate output of the network after registering the first input spikes (even for deep networks) [30]. Such phenomenon can be exploited to achieve lower response times than conventional analog networks have, while also maintaining high accuracy [27].

Currently, there exist several models of spiking neurons. These vary in biological plausibility, level of detail, and computational cost. Arguably, the most popular model is the LIF (leaky integrate-and-fire) neuron [10]. However, there are other viable candidates like the Izhikevich model [19] or the spike response model. Even though these neuron models are more complex than the standard neurons found in ANNs, they still represent a relatively simplified version of the biological models. In a typical spiking neuron, only threshold dynamics is assumed while there are other important factors such as refractoriness, hysteresis, resonance dynamics, etc. [42]. Another common simplification is setting the resting membrane potential to 0 Volts which is, however, not the case in the biological neurons where it is usually negative.

While there are crucial differences between the working principles of spiking and non-spiking neurons, the architecture of an SNN remains very similar to the one of an ANN. Spiking neurons are contained in layers that use synapses to transfer information throughout the network. Again, the most popular are deep spiking neural networks (i.e., networks with at least two hidden layers) since they contain more trainable parameters and perform better than simpler shallow SNNs. Just as ANNs, spiking networks can also process information in a feedforward manner or even adopt a recurrent architecture. With some changes, they can even implement concepts from state-of-the-art analog networks, such as the use of convolutional layers, pooling layers, dropout layers, etc.

2.2.1 Neuromorphic hardware

Unfortunately, there are not many advantages of using spiking networks implemented in the conventional von Neumann computers. To work effectively, SNNs need hardware with independent processing units that are not synchronized by a system clock. One such promising hardware for spiking networks are neuromorphic devices.

Neuromorphic hardware aims to model neuro-biological structures found in the nervous systems of humans and animals. Such devices share the locality of data to reduce on-chip traffic which is mostly reflected by using trains of spikes for communication between the components [30]. Neuromorphic hardware can be based on various implementations. Some neuromorphic devices comprise analog electronic circuits, while others use digital electronics. The key distinction is in the mapping between the hardware and the neurons. Analog based devices are typically mapped one-to-one, while digital neuromorphic machines emulate hundreds or thousands of neurons per core.

Even though the efficiency is greatly increased (compared to the von Neumann architecture), the structure of neuromorphic systems is very restrictive, and spiking networks have to be modified to fit the specific hardware. Many platforms also demand the SNN to be trained before it is put in the chip, though some such as SpiNNaker, BrainScales [37], and Loihi [9] implement spike-timing-dependent plasticity (STDP, see Section 2.3.3) for on-chip learning. There is also an issue of implementing deep spiking networks and only a few successful examples exist. Possibly the best result was achieved using the TrueNorth [24] platform in article [13] that managed to construct a deep SNN with 99.41% accuracy on the MNIST dataset.

2.3 State of the art

Since the discontinuity of spikes makes it extremely difficult to use back-propagation algorithms, which are de facto a standard in the state-of-the-art ANNs, a large part of the current research is focused on finding alternative methods of training that would close the performance gap between the spiking and analog architecture. There are also additional challenges, such as finding the optimal coding for the information recognized by the network as it needs to be represented by spikes rather than a series of floating-point numbers (which is the case in analog networks). This section overviews several state-of-the-art approaches of training spiking networks and evaluates their results in the comparison section.

2.3.1 ANN to SNN conversion

The first solution to the lack of optimal learning methods is to avoid them in the spiking environment entirely. Instead, a conventional ANN is trained via backpropagation and then converted to its spiking equivalent. The conversion step consists of transforming the non-spiking neurons to their spiking variants as well as adjusting their specific parameters (refactor times, leak rates, etc.) and weights of their synapses.

A great advantage of this method is that state-of-the-art analog networks can be utilized. Therefore, the converted SNN achieves similar accuracy to the original ANN since the conversion introduces minimal performance decrease. There are already some well-performing SNNs created with this method such as [11, 38] which achieve only marginally worse results on the standard ANN datasets such as MNIST.

Even though these conversions are a rather successful and viable solution to the supervised training of SNNs, there is still vast room for improvement. Typically, most of the converted networks use rate coding, i.e., the information is encoded in an average number of spikes over a given interval. This is, however, suboptimal since multiple spikes are required to represent a single activation of analog network [30]. Ideally, temporal codes with a sparse number of spikes (i.e., timing the spikes precisely) should be used to fully utilize the potential of a spiking network. They, however, in almost all cases lower the performance. Several papers are trying to improve rate codes, for example, [48] manages to decrease the overall number of spikes while maintaining similar accuracy to other rate-based codes. Histogram of averaged time surfaces (HATS) is another alternative that introduces a spatio-temporal coding for event-based hardware [39], defining a new type of memory time surface to compactly store spatio-temporal information.

An additional set of issues includes conversion of specific ANN components to the spiking environment. Activations with negative outputs prove difficult to convert since the firing rates of spiking neurons can only be positive. This affects frequently used activations such as softmax. There is a partial solution to this problem [29] which uses one neuron to encode positive and zero values, while the other one is used to encode negative values. A different circumvention is to simply avoid these activation functions entirely and substitute them with similar ones that output only positive values - e.g., a sigmoid in the case of softmax activation. The conversion of specific CNN layers such as max-pool causes problems as well. This is due to the maximum operation being nonlinear and unable to be computed on a spike-by-spike basis [30]. The inability to properly emulate max-pool in

the spiking setting made some state-of-the-art networks, such as [11], use average pooling instead. That, however, results in a slight performance decrease. There is also a workaround solution, which proposes that output units use a gating function that discards all spikes except the ones from the maximally firing neuron [35] which, in practice, results in functionality similar to the one found in a max-pooling layer.

2.3.2 Constrain-then-train

Constrain-then-train methods are similar to the conversion methods, except that they define additional constraints before the analog model is trained. These constraints are present because of the nature of the spiking neurons (or the hardware they are implemented in). Traditional training methods based on backpropagation are used to train the ANN which can then be converted to its spiking equivalent.

Unlike the ANN conversion methods, however, constrain-then-train uses constraints for a single configuration of spiking parameters, i.e, if the SNN parameters need to be changed, the entire ANN model has to be retrained as well (which is not the case in the conversion methods). Such an approach usually provides SNNs with better accuracy than the ones from the conversion method. The technique is for example applied in [18], where a neural network with leaky integrate-and-fire neurons was used. The resulting SNN achieved accuracy comparable to other state-of-the-art spiking networks.

2.3.3 Local learning rules, STDP

Local learning rules are one of the more biologically plausible methods for training neural networks. The most notable is the so-called spike-timing-dependent plasticity (STDP). STDP is a biology-inspired method that modifies the weights of synapses between neurons. If a presynaptic neuron fires shortly before the postsynaptic neuron, the weight of the synapse connecting them is strengthened [42]. Unlike the two previously mentioned methods, STDP allows training SNN directly. While this approach may seem like the most attractive one, there is, however, no proper way of performing backpropagation on feedforward networks (the adjustments can only be made locally). Therefore, STDP is mostly used for unsupervised learning. To allow STDP to be utilized for supervised learning, a feedforward network would have to be converted to a recurrent one. Even though several experiments are combining recurrent networks and STDP for supervised learning tasks, they attain worse results than the previously mentioned conversion

and constrain-then-train methods.

For unsupervised learning, there are already existing neuromorphic hardware implementations such as SpiNNaker, BrainScaleS or Loihi. These can additionally be exploited to accelerate biological simulations that would otherwise take a significantly longer time to simulate [30].

2.3.4 Approximation methods

Another approach is to approximate the discontinuity of the spike signals by a function that is continuous and differentiable so that backpropagation algorithms can be used. These methods generally show more potential than the previously mentioned conversion and constrain-then-train approaches as they allow training the spiking network directly. One of the successful implementations [22] used low-pass filtering with backpropagation to train a high accuracy SNN. Another method [26] defines a modified version of stochastic gradient descent which approximates the gradient itself (instead of changing the network).

2.3.5 Binary neural networks

Even though binary networks are not SNNs and instead constitute a special type of ANNs that contain only neurons with binary activations, they can be seen as a viable alternative to spiking networks. Thanks to binarization, these networks are much more energy-efficient than generic ANNs and can also be implemented in event-based hardware. To provide an even better performance, weights in the network can be binarized as well [8] which makes it possible to replace costly multiplication operations with computationally cheaper bitwise operations such as XNOR and bit counting [30].

Although it might seem practical to convert an ANN into its binarized form after it was trained, such a conversion usually results in a significant accuracy degradation. Therefore, it is preferred for the network to be binarized before the training is performed. State-of-the-art training methods, however, take longer than gradient descent methods (which are not usable due to binarization), and the resulting networks offer worse performance than conventional ANNs. Additionally, binary networks cannot substitute spiking networks in use cases where asynchronous data processing is required since they are still based on analog networks.

2.3.6 Comparison of state-of-the-art SNNs and ANNs

This section overviews several state-of-the-art analog and spiking networks on the MNIST dataset. Even though there are other, more suitable datasets for spiking networks such as N-MNIST (Neuromorphic MNIST), they are not used frequently enough for the comparison to be meaningful. Additionally, the MNIST dataset is also used by analog networks which makes it possible to measure performance differences between the state-of-the-art analog and spiking nets.

The MNIST dataset is commonly used for performance evaluation of machine learning methods since it is not difficult to achieve very high accuracy on it. The dataset consists of 28×28 grey-scale images of handwritten digits (from a range of 0 to 9) with 60000 training samples and 10000 testing samples.

It is worth noting, however, that unlike in ANNs, comparing only accuracy does not show the full potential of spiking networks. SNNs allow additional optimizations for lower response times and efficient power consumption, which is typically not a point of concern of analog networks. In many cases, achieving a highly performant SNN directly decreases the response time and the power efficiency since high firing rates and longer duration for spike integration are necessary [30]. Therefore, evaluating the performance of spiking networks by only using accuracy-related metrics may result in misleading interpretations.

Table 2.1 shows performance of the state-of-the-art analog and spiking networks. The table consists of three ANNs and four SNNs. The two best best performing networks are analog networks [5, 6] which achieve over 99.7% accuracy. The third ANN is a recurrent LSTM network that scored only 99% accuracy. This, however, is still remarkable since LSTM networks usually do not perform as well as CNNs in image classification.

The best spiking network is surprisingly implemented in neuromorphic hardware. The work [13] is an example of the approximation approach. It treated spikes and synapses as continuous probabilities which made it possible to use standard backpropagation techniques for training. The SNN managed to attain 99.42% accuracy which is very close to the best ANN on the list. The second best SNN is also an example of approximation methods and is very close to the first one. The network is a CNN trained using backpropagation and low-pass filtering [22] which allowed it to achieve 99.31% accuracy. The spiking CNN trained with weight and threshold balancing [11] is a case of the conversion methods and its accuracy is still relatively close to the first two SNNs. The worst performing network [34] in the table was

converted from an ANN as well but uses temporal coding instead of a rate based one. The temporal coding is expected to be more energy efficient but, as seen, comes with an accuracy decrease.

Neural network	**Network type**	**MNIST accuracy** [%]
Branching and Merging CNN with HFC [5]	ANN	**99.79**
Multi-column DNN for image classification [6]	ANN	99.77
SNN implemented in TrueNorth and trained with backpropagation [13]	SNN	99.42
Spiking CNN trained with BP and low-pass filtering [22]	SNN	99.31
Spiking ConvNet trained with weight and threshold balancing [11]	SNN	99.10
Batch normalized LSTM [7]	ANN	99.00
Spiking Lenet-5 with sparse temporal coding [34]	SNN	98.57

Table 2.1: Comparison of spiking and non-spiking state-of-the-art networks on the MNIST dataset (sorted from best to worst).

3 Tools for simulation of spiking networks

The next part of the thesis was to explore tools that were going to be used to model and simulate spiking networks. As of right now, there already exist several robust frameworks, typically each with a different specialization. Some frameworks are more theoretically oriented and allow very detailed simulation of individual neurons, while others are very pragmatically focused. Most of these simulators are designed to be used with the Python programming language since it offers simple syntax to work with complex algorithms while also being compatible with lower-level languages such as C and C++.

There were several factors involved to determine whether a specific platform was going to be examined. The main ones were the overall support from the user base and developers, the number of features it offers, and the quality of documentation. Frameworks with a large community were preferred since they typically have more educative materials and tend to be more user-friendly. The ability to convert an analog network to a spiking one was also welcome since it allows to directly compare how well does the SNN perform. Other factors were GPU support, compatibility with other machine learning frameworks (e.g., TensorFlow and PyTorch) or simple installation and setup.

3.1 NEURON

NEURON is arguably the most used platform among researchers as it appears in more than 2000 scientific papers. The platform targets mostly users with a neuroscience background since its intention is to simulate biologically plausible neurons and networks.

Most of the simulation can be done either via NEURON's GUI, in "hoc" (a programming language) or in Python. Alternatively, the framework also offers its own domain specific language called NMODL which the users can declare models in. NMODL makes the construction of the models very straightforward and can also be compiled to highly optimized code for both CPUs and GPUs. Clusters such as IBM Blue Gene or Beowulf are supported as well. NEURON additionally offers integrator-independent model defini-

tion. Users can select from different numerical integration methods such as Euler method, Crack-Nicholson method or adaptive integration methods which are more accurate.

3.2 BindsNET

BindsNET [16] is a Python simulation library that is built on top of PyTorch. The main goal of the framework is to provide biologically acceptable simulation while also maintaining straightforward development by using PyTorch objects to construct the network.

BindsNET offers tools for both unsupervised and supervised learning as well as reinforcement learning. The framework contains several models of spiking neurons that can be used to construct an SNN such as the LIF model, the Izhikevich model, etc. There are also premade models of networks available, however, those are typically simple two-layer SNNs.

Training of the networks can be done by application of learning rules (this is typically preferred over backpropagation due to biological plausibility). Rules can be either two-factor or three-factor. Two-factor learning rules change parameters of the network according to pre-synaptic and post-synaptic activity. These can be for example Hebbian Learning (neurons that fire at the same time have a strong synaptic connections to one another) and STDP (see Section 2.3.3). Three-factor learning rules allow additional changes to the network on a global level.

A great advantage of this framework is that it uses functionality from PyTorch. This means that costly operations during training (such as matrix computations) can be done on a GPU rather than on a CPU and the training can be faster. BindsNET should also, in theory, be convertible to various hardware platforms such as FPGA, ASIC or ARM devices [16] to execute simulations. However, no tools for such conversion have yet been created.

3.3 NEST

NEST [12] is another platform for simulation of spiking neural networks. Its main advantage is its wide range of features for creating such models. The framework contains over 50 types of spiking neurons and 10 types of synapses. A NEST network can also comprise a combination of these components, i.e., multiple variants of neurons and synapses can be used within a single network. The properties of the SNN can be tracked and even changed during its run. NEST is also very popular in the scientific community, ap-

pearing in nearly 500 scientific papers and projects such as BrainScaleS and the Human Brain Project.

The framework itself is written in C++, however, it can also be used in Python (via PyNEST library), in a built-in simulation language interpreter (SLI), or in NESTML - a domain-specific language for modelling NEST neurons [31]. Additionally, it is compatible with PyNN which is a high-level framework for simulator independent model definition (see Section 3.5). NEST is primarily designed to run on UNIX systems - Linux, macOS, etc. Even though there is no direct support of Windows operating systems, the page mentions compatibility via a virtual machine (or Windows Subsystem for Linux). Another advantage of NEST is its scalability. The authors mention compatibility with a wide range of systems ranging from MacBooks to computer clusters such as IBM BlueGene. Unfortunately, the framework does not currently support computation on GPUs, which makes it in certain cases slower than other similarly oriented frameworks [43].

3.4 Brian

Brian is a mathematically based simulator programmed in Python. It offers an equation-oriented definition of SNNs rather than implementing them programmatically. Neurons, synapses, and other parts of the network can be described by differential equations which makes Brian very flexible and relatively straightforward for scientific purposes. On the other hand, such equations also require expert knowledge in computational neuroscience and might not be suitable for everyone.

Models defined in Brian can be converted to highly optimized Python, Cython, and even native C++ code. Such compiled C++ code can also be used as an input to the GeNN library [41] which makes it possible to further optimize the entire simulation by running part of it on Nvidia GPUs.

3.5 PyNN

PyNN is a framework that can be used to define abstract models of neural networks. Such models can then be run on various supported backends such as the previously mentioned NEURON, NEST, and Brian. There is also support for running PyNN networks on neuromorphic platforms such as SpiNNaker or BrainScaleS. The library contains a set of neurons and synapses which are compatible across all frameworks. If the user does not need simulation on multiple backends, PyNN can also use platform-specific

features and serves as a high-level abstraction on top of the specific simulator.

3.6 Nengo

Nengo [2] is a Python framework that supports simulation of both non-spiking and spiking networks. A great advantage of this platform is that it is very flexible while also offering a large number of features. Nengo uses a similar approach as PyNN which allows defining an abstract model that can run on various simulation backends. These involve the conventional x86/x64 platform (Linux, Windows, macOS) as well as various neuromorphic devices.

3.6.1 Nengo Core

Nengo core contains essential functionality which is reused in other parts of the library. It consists of several objects that are used to model a neural network. A network itself is represented by the *Network* object which comprises ensembles (groups of neurons), nodes (neurons), connections (synapses) or even other networks. Models can be trained using learning rules - similar to other biologically inspired frameworks. Nengo also allows to define custom learning rules and node types.

Since the simulations of biological models are typically very complex, and there might not be enough memory to monitor all components of the network during the simulation, Nengo uses a *Probe* object, which collects data from a particular object in the model, e.g., a neuron. Probes can record statistics such as the number of spikes, membrane potential, and others. Models created in Nengo can also be visualized using NengoGUI, which is a web application. This application shows all parts of the network and can even be used to control the simulation.

3.6.2 NengoDL

NengoDL contains a deep learning simulator for Nengo models. Even though some learning and simulation could be done with components from the core, this library contains more features and functionality dedicated to deep learning tasks.

NengoDL uses TensorFlow and Keras as the backend for the simulation. User can use pure Nengo models, define models that comprise parts from TensorFlow and Keras, or even use networks that are made in Keras entirely. Before the simulation, any non-native network must be converted. This involves replacing TensorFlow objects (layers, activations ...) with native

Nengo objects (if such converter exists) or creating a *TensorNode* which provides compatibility between the two frameworks.

To run the network, NengoDL introduces a *Simulator* object which acts as a wrapper around Keras API. This is very useful since most of the TensorFlow (Keras) functionality can be used alongside Nengo, including optimizers, callbacks, and losses. Training and evaluation can be done in the same way as in Keras as well, i.e., a network is created, compiled, fit, and then evaluated. It is also possible to change certain parts (such as activations) of the trained network. This, in practice, means that the user can train a state-of-the-art analog network and then swap its activations with spiking ones, resulting in a highly performant SNN.

3.6.3 KerasSpiking

KerasSpiking contains functionality to construct and train spiking neural networks in the Keras framework. Unlike NengoDL, it does not use objects from Nengo and is intended to be a lightweight library used alongside Keras rather than a feature-rich framework.

The main feature of this library is its ability to convert any Keras / TensorFlow activation into a spiking one. Models with spiking activations can be run and trained in the same way as analog ones. This is achieved by using a feature called "spiking aware training" that swaps the spiking activations with analog ones whenever the network parameters are being adjusted (i.e., during training). There is also a PyTorch variant of this framework called PyTorchSpiking which offers the same set of features. The main disadvantage of both of these frameworks is that the created networks cannot be converted to native Nengo models, and therefore are tied to their specific backend.

3.6.4 The rest of the ecosystem

As previously mentioned, Nengo also contains modules that make it possible to implement spiking networks in hardware. Various hardware platforms can be used such as FPGAs, Intel Loihi, OpenCL-based devices (i.e., set of CPU, GPU and other computing devices that can be programmed with the OpenCL framework), and SpiNNaker. There is also a library for semantic pointer architecture implemented in Nengo SPA. This allows to define a (spiking) neural network by semantic pointers which represent higher-level cognitive functions. The visualization of the entire ecosystem can be seen in Figure 3.1.

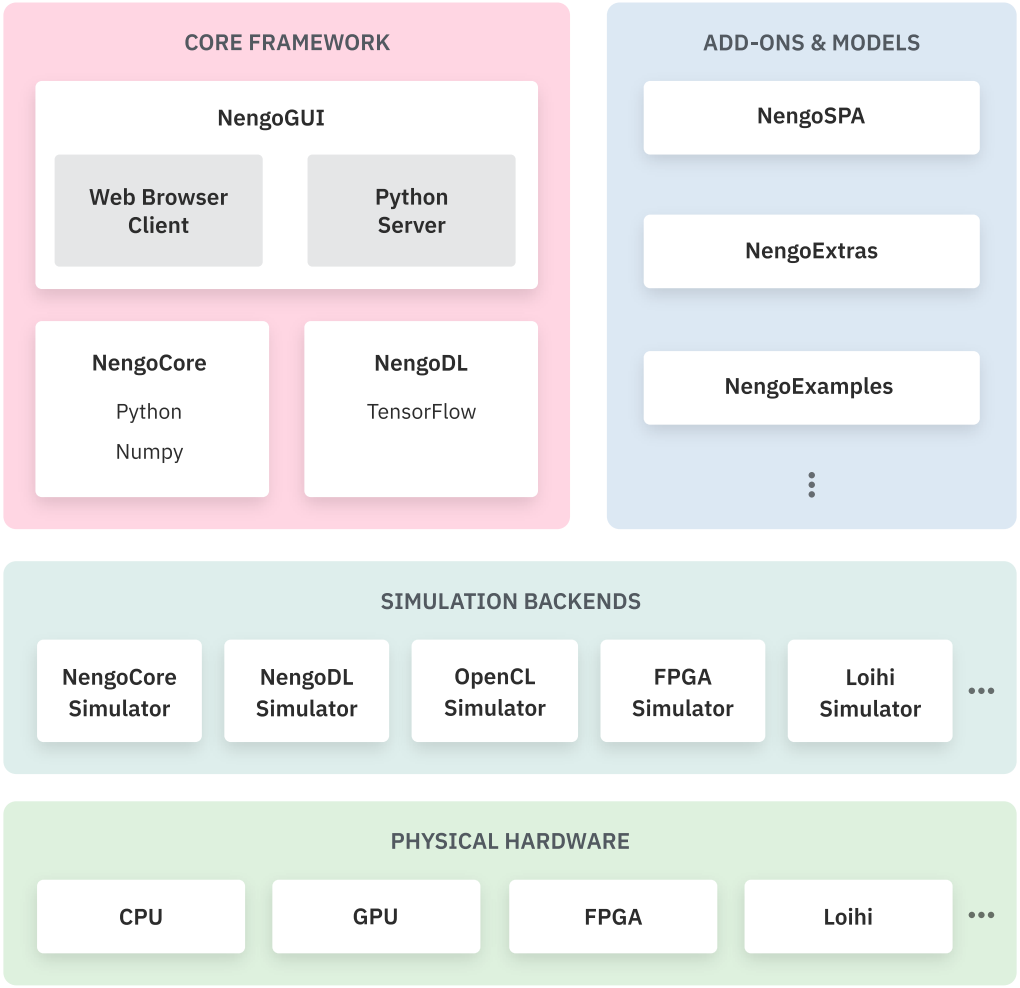

Figure 3.1: Visualization of the Nengo ecosystem. Source: [3]

3.7 SNN-Toolbox

SNN-Toolbox is a Python library that specializes in converting artificial networks into spiking ones. The framework functions as a mediator between analog and spiking platforms. It can convert ANN models defined in non-spiking frameworks such as Keras, PyTorch, Caffe or Lasagne and make them usable in spiking simulators such as PyNN and Brian. User can also run converted SNNs in INIsim, which is a native simulator inside SNN-Toolbox.

Additionally, SNN-toolbox supports two popular hardware platforms - Loihi and SpiNNaker. Due to the differences of each platform, though, not all layers in analog network can be converted (see Fig. 3.2). The most compatible is INIsim (for obvious reasons).

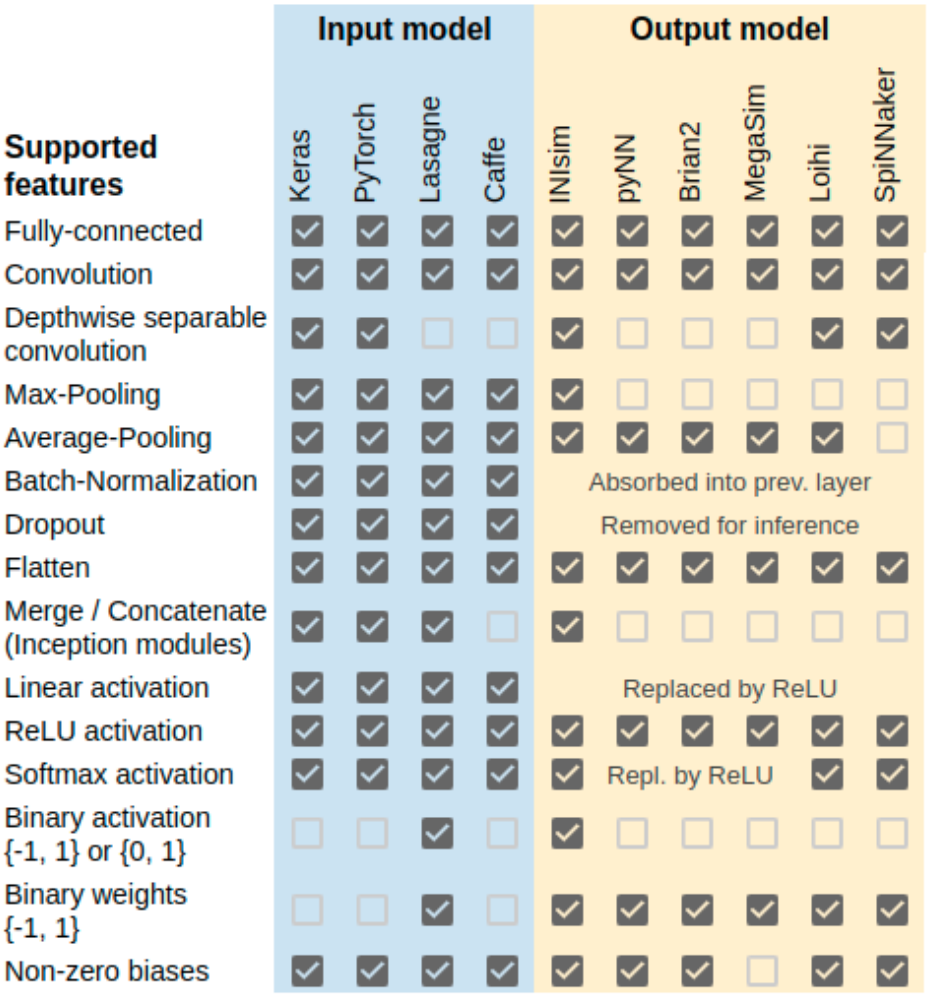

Figure 3.2: Supported features in SNN-Toolbox. Source: [36]

3.8 ANN simulation platforms

This section briefly overviews two main frameworks (TensorFlow and PyTorch) for the simulation of analog networks. While neither of these allows the creation and run of spiking neural networks directly, they are often used as a backend in many spiking simulators.

3.8.1 TensorFlow and Keras

TensorFlow is one of the most popular ANN platforms. The platform is very beginner-friendly and also contains a large set of features and customizability for advanced users. Tensorflow can be controlled via Keras API which

contains high-level components to simplify the creation of neural networks. The platform is very scalable - TensorFlow networks can be run in Python, server-side using TensorFlowJS, or even in mobile devices using the TensorFlow Lite platform. There is also support for GPU computing using the Nvidia CUDA toolkit.

3.8.2 PyTorch

PyTorch is another mature machine learning platform that targets the same user base as TensorFlow. It is designed to be used in Python, offering similar features to TensorFlow - support for mobile devices, enormous scalability, and GPU support (again via the CUDA platform). Its main advantage over TensorFlow is that it allows to define computational graph (a graph that describes relations between parts of the network) dynamically, i.e., a model can be edited while the simulation is running. There is also a low-level C++ API called LibTorch if the user needs to optimize their code further.

3.9 Summary

The most feature-rich and popular simulators for spiking networks were covered. While there are many other spiking frameworks available, they are either in the very early stage of development, or most their features are already implemented in one of the mentioned frameworks.

This section compares features of the six previously mentioned simulators - NEURON, BindsNET, NEST, Brian, Nengo, and SNN-Toolbox. PyNN was excluded since it is more of a high-level API to simplify work with other simulators. Regarding Nengo, only Nengo core and NengoDL were considered since they are the most used parts of the framework while others such as PyTorchSpiking or KerasSpiking are currently in more experimental stages. Table 3.1 shows data gathered from the GitHub repository of each project as GitHub has become arguably the largest platform for distributing software in the last few years. The site provides statistics that can be used to compare popularity, size of the community, and general support from the developers. Unfortunately, GitHub does not publicly disclose the number of unique or recent downloads per project, and thus, only metrics such as the number of stars, the number of releases, fork count, and the "used by" number were considered. Such collected data might, however, not be entirely accurate since some simulation platforms offer downloads directly from their websites (which in certain cases could be preferred by their user base).

If only data from GitHub is examined, the two closest metrics to the popularity are the "used by" number (i.e., how many repositories use the project) and the number of stars (user can add a star to a project to show that they like it). Sadly, not every repository tracks the "used by" statistics, and thus only the number of stars were considered. Using this metric, it can be seen that the three most popular frameworks are BindsNET, Nengo, and Brian.

Platform	**Releases**	**Forks**	**Stars**	**Contributors**	**Used by**
NEURON	18	63	139	35	N/A
BindsNET	17	218	834	25	N/A
NEST	20	271	341	92	N/A
Brian	35	162	531	39	99
Nengo	20	161	615	32	109
SNN-Toolbox	5	75	187	11	3

Table 3.1: GitHub statistics of each project.

Another important metric can also be the number of publications each framework appears in. This is shown in Table 3.2. Unfortunately, Brian, BindsNET, and SNN-Toolbox do not provide any information about the number of publications that mention them. Nengo appears in over 100 publications, however, the authors also mention that the list is incomplete. Judging by this metric, the NEURON framework is most likely the most used platform in the scientific community.

Platform	**Number of publications**
NEURON	2304
BindsNET	N/A
NEST	495
Brian	N/A
Nengo	over 100
SNN-Toolbox	N/A

Table 3.2: Number of publications for each platform.

Lastly, Table 3.3 shows various features of each framework such as their scalability, GPU support, and others. It could be argued that Nengo and BindsNET are probably the best platforms for "general purpose" simulations since they offer an easy setup of the environment (both are compatible with either TensorFlow or PyTorch) and have a much flatter learning curve

Platform	**GPU supp.**	**Cluster supp.**	**Language**
NEURON	Yes	Yes	GUI, Python / hoc, NMODL
BindsNET	Yes	N/A	Python
NEST	No	Yes	Python, C++, NESTML
Brian	Yes (via Brian2GeNN)	No	Python
Nengo	Yes (in NengoDL)	Yes	Python
SNN-Toolbox	N/A	No	Python

Table 3.3: Various features of each platform - GPU support, cluster computation support, and the language the user controls the simulation in.

(though this is very subjective). SNN-Toolbox also fits this category. However, its modelling features are very restrictive, and most of them are already present in Nengo or BindsNet. On the other side of the spectrum, NEURON, Brian, and NEST are more theoretically oriented and offer extensive features for modelling and implementing custom spiking neurons.

4 Applications of spiking neural networks

After evaluating all possible tools for the simulation of SNNs, the next task was to use these tools to construct a spiking network and apply it in a meaningful way. To find a fitting use case, only classification problems were considered since the results can be easily evaluated. Classification tasks are also one of the most common applications of neural networks so there exist many datasets from various fields such as medicine, computer vision, statistics, and many others.

One of the main criteria of choosing the dataset was whether it would be suitable for the spiking network. Since SNNs process information via trains of spikes, they are very suitable for spatio-temporal data. Image datasets are another great candidate since they can be classified by CNNs which are also realizable in the spiking setting (though with more constraints). In total, 4 datasets were used - two with spatio-temporal characteristics (the P300 and BNCI Horizon datasets) while the other two (MNIST and Fashion MNIST) were image datasets:

- A large multi-subject P300 dataset [25]
- BNCI Horizon spatial attention shifts to colored items dataset [32]
- MNIST handwritten digits image dataset [21]
- Fashion MNIST image dataset [45]

The datasets were used across three experiments. The networks applied in these experiments were run using Nengo, NengoDL, and PyTorch (with additional implementation of a spiking network and a learning algorithm). Nengo and NengoDL were selected since they offer a very straightforward simulation of a spiking network. The SNN is constructed from an already trained ANN which also makes it possible to compare the performance between both architectures. Other frameworks were not chosen as they did not prove to be very practical and/or did not offer any advantage over Nengo and PyTorch.

All experiments were run in Python 3.8 using Jupyter notebooks. Jupyter notebooks were used because they allow splitting the entire Python script into smaller code blocks that can be executed separately. In practice, this

allows to develop part of the program, run it, and see the results immediately (e.g., the user can print some text in the console or display a graph), while also being able to use the variables declared in previous code blocks. Each experiment was run on a personal computer with Intel Core i7 9700f CPU, 16 GB of RAM, and Nvidia GTX 1060 6 GB GPU.

4.1 Large multi-subject P300 dataset spiking conversion

The first experiment used a multi-subject P300 dataset from [25]. The dataset contains electroencephalographic (EEG) samples from the guess the number (GTN) brain-computer interface (BCI) experiment. In this BCI experiment, the examined subject chooses an arbitrary number (ranging from 1 to 9) to which they refer to as the "target number". Subsequently, the participant is shown a random sequence of digits (again from the same range) while the experimenters record the EEG signal and try to guess the target number from event-related potential waveforms [25]. Overall, the GTN dataset contains EEG data from 250 children between ages 7 - 17. The samples can be divided into epochs, i.e., a short time interval during which a stimulus (in this case a random number) occurs.

The experiment performed here aims to create a spiking neural network that can classify whether data from a specific epoch correspond to a target or a non-target number. Since one paper [44] already used this dataset to create a CNN which achieved relatively high accuracy ($62 - 64\%$), a very similar model was used here as well. There is also a thesis [33] which utilizes this CNN with SNN-Toolbox to achieve accuracy around 63.7% in analog and 57.2% in spiking setting.

Before performing the experiment, preprocessing of the original P300 dataset was applied (in the same manner as in the paper [44]). The entire EEG signal from each participant was split into epochs. For each one, the data from 200 ms before the stimulus and 1000 ms after the stimulus were extracted, forming a 1200 ms long sample. To remove damaged epochs, those with amplitude higher than 100 μV were taken out. In total, such processing produced 8036 samples of 3×1200 tensors (3 EEG channels recorded for 1200 ms with 1000 Hz sampling rate) that could be fed to the model. The information whether a given epoch represents a target or a non-target number was used as labels, which were hot-encoded - a $(0, 1)$ vector for a non-target and a $(1, 0)$ vector for a target number.

4.1.1 Model architecture

To simulate a spiking network, NengoDL and TensorFlow (Keras) frameworks were used. The CNN from the original paper [44] was created in Keras and then converted to a Nengo model. This operation required small changes due to compatibility issues between the frameworks (this involved replacing ELU activations with ReLUs, changing padding to "same" in the average pooling layer, and removing the first batch normalization layer). Unfortunately, not all parts of the network could be replaced by native Nengo objects and were instead replaced by TensorNodes. This did not cause any issues during this experiment but may potentially create problems if the model needs to be simulated on a different backend. The visualization of the model is shown in Fig. 4.1.

4.1.2 Network training and evaluation

Apart from the minor changes in the model, both features and labels needed to be reshaped since Nengo required a specific shape of data for the simulation. The features were flattened from 3×1200 tensor to a 3600-element vector. Additionally, a dimension representing time was necessary for both features and labels (so the simulator could use it for the time steps during simulation - i.e., for how long is the data fed to the network).

The target and non-target features with their respective labels were concatenated to form the dataset. The training approach was very similar to the article with the original CNN [44]. The dataset was shuffled and 25% of the data was held out as testing data. The other part (75%) was used in cross-validation (CV), i.e., a technique where the model is iteratively reset and retrained with different training subset to find the best performing parameters. The CV method applied here is called Monte Carlo. The algorithm randomly picks a part of the dataset as training samples, while the other part is used to validate the model (this is done every iteration).

The CV was performed in 30 iterations (i.e., creating 30 different configurations of the model). In each one, 25% of the non-testing data were used as the validation data while the remainder was fitted to the neural network. Each configuration of the analog model was trained for 30 epochs with an early stopping callback of five epochs (the training would stop, restoring the best weights, if the loss on validation data did not improve in the last 5 epochs) and a batch size of 64. The Adam optimizer and binary cross-entropy loss were used to optimize the network.

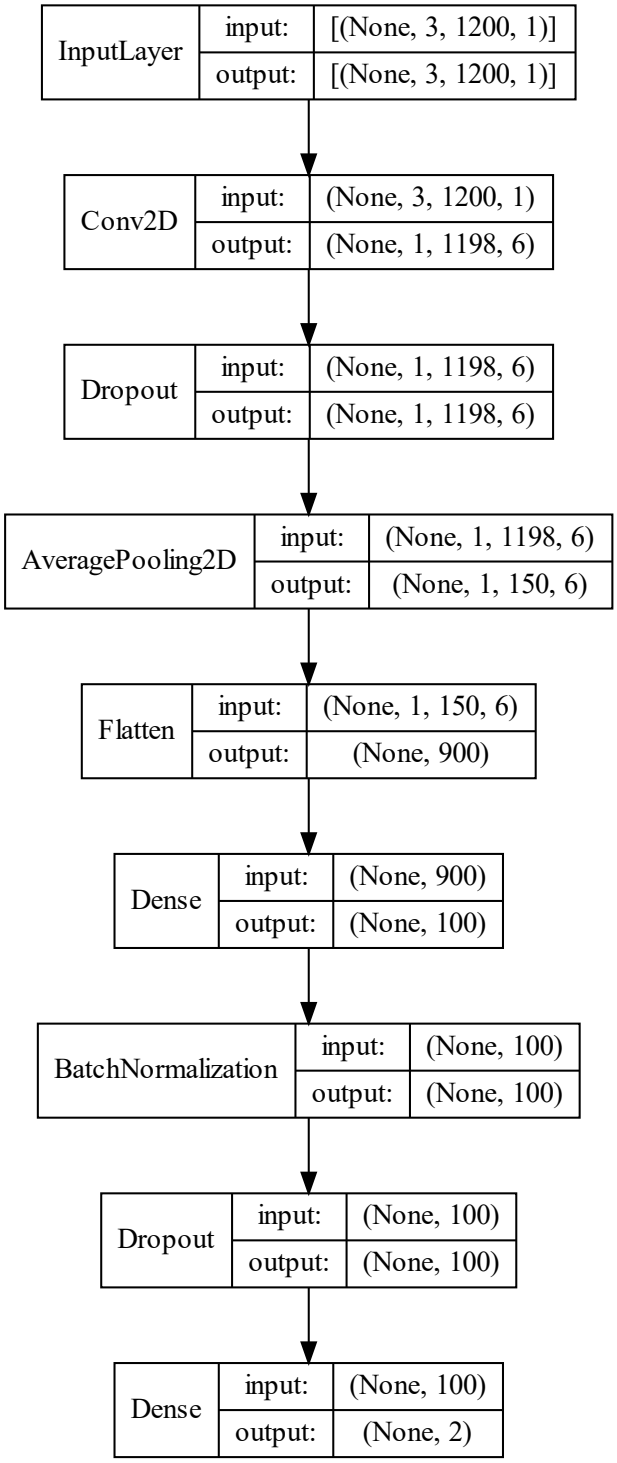

Figure 4.1: TensorFlow model of the convolutional neural network tested on the P300 dataset.

4.1.3 Conversion to spiking network

The trained ANN was evaluated on the previously unseen testing data and its parameters were saved so it could be converted to a spiking network. The conversion in Nengo was relatively straightforward. It consisted of loading the parameters from a file and replacing non-spiking activations with spiking ones. In this case, all ReLUs were swapped with Nengo's spiking rectified linear activations.

Apart from changing the activations, the Nengo converter has an additional set of parameters that can further improve the performance of the spiking network. These are spike firing rate scaling and synaptic smoothing. The first mentioned increases the number of spikes by applying a linear scaling to the input of the neurons (subsequently the output needs to be divided by the same scale as well) [4]. With high enough firing rates, the performance of the SNN should converge to the performance of the ANN, though it also implies less energy efficiency due to the number of fired spikes. Synaptic smoothing (synapse parameter) applies smoothing to the spikes using a low-pass filter which should also improve performance [4]. Without synaptic smoothing, there is always a chance that the spike will not occur in the last time step, whereas with it the output is averaged over more time steps. The disadvantage of this feature is that it will increase the network's latency.

The converted spiking network was evaluated with both features turned on and off - i.e, 4 variants of the spiking network were tested (see Table 4.1) which resulted in a total of 120 runs of SNNs across the entire CV. The spike firing rate scaling was set to 1000 and the synaptic smoothing was set to 0.01. These values were found mostly experimentally as increasing spike firing rate scaling further, or changing the synaptic smoothing to different values did not improve the overall accuracy in any noticeable way. The testing data was fed to the network in 50 time steps and the output of the last time step was used to compute the performance of the model. The example code of running a spiking network is in Fig. 4.2.

SNN variant	**Time steps**	**Spike firing rates scaling**	**Synaptic smoothing**
1	50	1000	0.01
2	50	1000	Off
3	50	Off	0.01
4	50	Off	Off

Table 4.1: Combinations of SNN parameters used in Nengo simulator.

```
# Convert TensorFlow network to a spiking Nengo network
# model - the TensorFlow model
# swap_activations - replacement of ReLU activations with Spiking ReLUs
# scale_firing_rates - firing rates scaling parameter
# synapse - synaptic smoothing
converter = nengo_dl.Converter(
        model=model,
        swap_activations={ tf.nn.relu: nengo.SpikingRectifiedLinear() },
        scale_firing_rates=scale_firing_rates,
        synapse=synapse
    )

# Input layer of the network
input_layer = converter.inputs[model.get_layer('input_layer')]

# Output layer of the network
output_layer = converter.outputs[model.get_layer('output_layer')]

# Tile test features (x_test) to desired timesteps for the simulator
x_test_tiled = np.tile(x_test, (1, timesteps, 1))

# Run NengoDL simulator to test the spiking network
with nengo_dl.Simulator(converter.net, minibatch_size=64) as simulator:
        # Load weights of the trained analog network
        simulator.load_params(trained_network_params)

        # Get predictions for the test data
        predictions = simulator.predict({ input_layer: x_test_tiled })

        # Extract output from the last layer and get the last timestep
        predictions = predictions[output_layer][:, -1, :]

        # Apply argmax to get the label for each testing sample
        predictions = np.argmax(predictions, axis=-1)

        # Compute accuracy of the spiking network
        snn_accuracy = (predictions == y_test).mean()
print(snn_accuracy)
```

Figure 4.2: Conversion of a TensorFlow model to a spiking Nengo model and getting its accuracy on testing data.

4.1.4 Results

The original ANN model was expected to perform the best since a slight decrease in performance is usually present by conversion to a spiking network. This was the case for variants 3 and 4, which both had no spike firing rate scaling enabled. However, variants 1 and 2 achieved marginally higher average and maximum accuracy than the ANN (most likely due to the high scaling rate applied). The best accuracy, 64.96%, was attained by variant 2 (the one without synaptic smoothing enabled). The worst performing model was the third variant which had synaptic smoothing enabled and scaling disabled. This is surprising since smoothing is expected to improve performance which is not the case here. Variant 4, which had both features off, performed slightly better than variant 3 but was still very close to a coin flip.

Performance metrics can be seen in Table 4.2, which contains information about accuracy of each model and Table 4.3 that provides information about precision, recall, and F1 score. Surprisingly enough, in terms of accuracy and recall, both the analog model and the spiking variants 1 and 2 performed slightly better than the CNN in the original article [44] (which attained an average accuracy of 62.18%, an average precision of 62.76%, and an average recall of 61.34%).

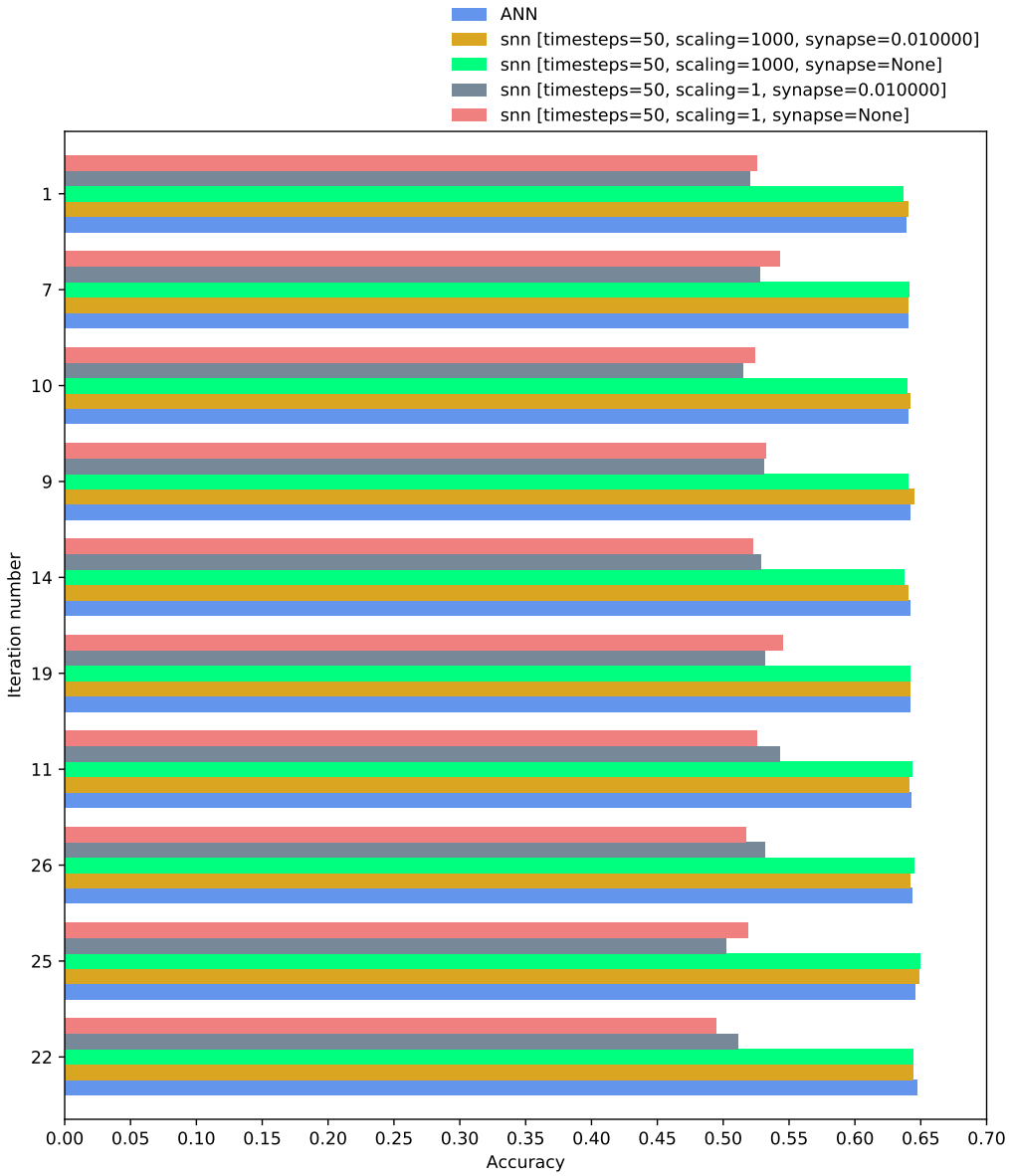

Figure 4.3: Top 10 ANN accuracies in comparison with SNN. From best (iteration 1) to worst (iteration 22) - sorted by the best ANN accuracy.

Model	Average accuracy	Maximum accuracy	Accuracy SD
ANN	0.6334	0.6472	0.0107
SNN (variant 1)	**0.6343**	0.6486	0.0106
SNN (variant 2)	0.6336	**0.6496**	0.0107
SNN (variant 3)	0.5197	0.5431	0.0137
SNN (variant 4)	0.5219	0.5456	0.0126

Table 4.2: Accuracy statistics of the analog CNN model and its spiking variants (from Table 4.1) on the P300 dataset.

Model	Average precision	Maximum precision	Average recall	Maximum recall	Average F1	Maximum F1
ANN	0.6433	0.6720	0.6004	0.6700	0.6204	0.6462
SNN (variant 1)	0.6447	0.6732	0.6008	0.6683	0.6212	0.6446
SNN (variant 2)	0.6441	0.6740	0.5996	0.6673	0.6202	0.6449
SNN (variant 3)	0.5263	0.5567	0.3918	0.4801	0.4480	0.5092
SNN (variant 4)	0.5288	0.5608	0.3905	0.4671	0.4484	0.4955

Table 4.3: Precision, recall, and F1 score of the analog CNN model and its spiking variants (from Table 4.1).

4.2 Training deep spiking networks using surrogate gradient

The next performed experiment was based on training a spiking network using a so-called surrogate gradient (SG) method [26]. This method (unlike the conversion in Nengo) allows training a spiking network directly, i.e., no analog network is required to produce a trained spiking model.

As previously mentioned, the issue of training in the spiking environment is that the spikes are discrete events, and the signal is thus discontinuous and non-linear. These properties make it impossible to perform gradient descent to optimize the loss function. To circumvent this issue, some form

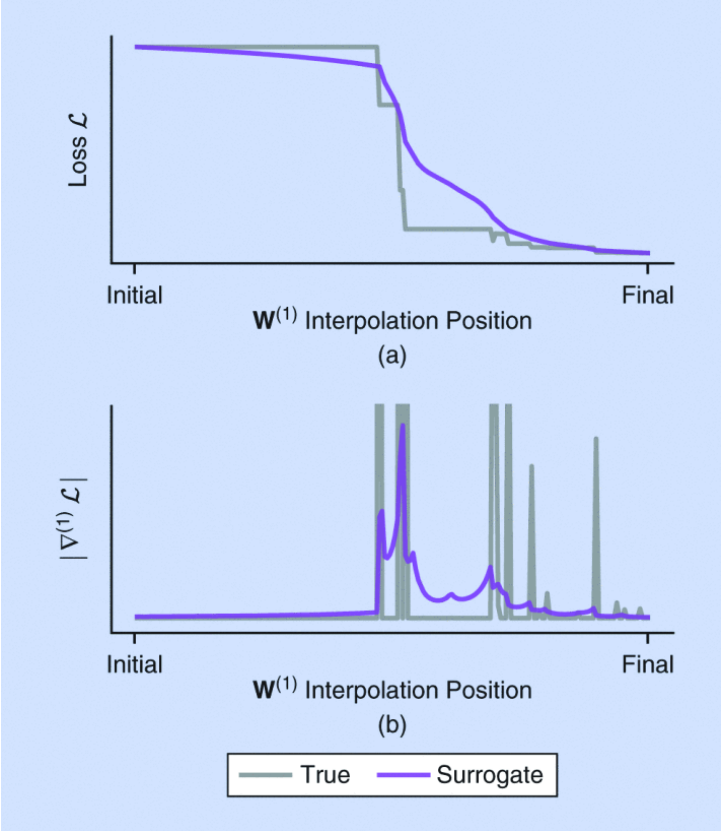

Figure 4.4: A comparison of the "true" gradient and the surrogate gradient in an SNN. The top graph shows an example of a loss function (grey) and its surrogate form (violet). The bottom graph shows the norm of the gradient of the loss function in the first graph (grey) and its surrogate form (violet). Source: [26]

of approximation must be applied in order to use any backpropagation-based algorithm. While many training methods make a direct change to the model, the surrogate gradient proposes a different approach. No modifications of the spiking network are necessary and instead, the regular gradient of the loss function is replaced by a surrogate gradient during the training. The SG has desired properties that make it possible to use it with backpropagation, i.e., it is continuous and non-zero (to avoid the vanishing gradient problem which would make training ineffective). The surrogate gradient can also be interpreted as a gradient of virtual surrogate loss function which in itself is some sort of an approximation [26]. The visualisation of surrogate and "true" gradient can be seen in Fig. 4.4.

To apply this training method, a part of the implementation of the SG from the SpyTorch GitHub repository [14] was used (since it was also ref-

erenced in the original article). The repository contains several examples of training spiking neural networks with the SG. These examples involve simple one-hidden-layer spiking nets using LIF neurons which are simulated in PyTorch.

In this experiment, several deep spiking networks were constructed and tested on MNIST (described in Section 2.3.6) and Fashion MNIST datasets. Fashion MNIST [45] is very similar to the regular MNIST. It is an image dataset that comprises 60000 training and 10000 testing examples of 28×28 grey-scale images. Each image shows a piece of clothing with a label associated with one of 10 classes - t-shirt, coat, shirt, sneaker, etc. This dataset aims to provide a more difficult classification than the original MNIST while retaining its format (i.e., it should be interchangeable with the regular MNIST without any required modifications to the classifying network).

While it would be ideal to use a state-of-the-art architecture such as CNN, it would also be rather difficult to make the SG work with specialized layers such as convolutional or pooling ones. Therefore, only fully connected deep spiking networks were considered. In total, four SNNs were tested on both datasets - two recurrent and two feedforward. Each network comprised an input layer of 784 units (since the images were 28×28) and an output layer of 10 units (one per class). The first feedforward network composed of two hidden layers with 256 and 128 units, while the other feedforward network used an additional 64-unit hidden layer (which was the last hidden layer in the network). The recurrent networks used the architecture from the fully connected SNNs and also contained recurrent weights in each hidden layer. The architecture of all four models can be seen in Table 4.4.

Because neither the MNIST nor the Fashion MNIST are intended for use with a spiking neural network, an additional conversion to spikes was required. To do so, time-to-first-spike coding (TTFS) was applied, which is a spatio-temporal coding that was also used in the GitHub examples. Apart from the structure of hidden layers, all models used the same model of leaky integrate-and-fire spiking neuron and were trained in the same manner. TTFS encoded data were fed to the model in 100 time steps where each time step took 1 ms. Every model was trained for 30 epochs on both datasets with the Adam optimizer and the negative log-likelihood loss.

Model	Network Type	Hidden layers	Units in hidden layers
1	Feedforward	2	[256, 128]
2	Feedforward	3	[256, 128, 64]
3	Recurrent	2	[256, 128]
4	Recurrent	3	[256, 128, 64]

Table 4.4: Architecture of the used SNNs.

4.2.1 Results

The best performing model was model 1 (see Table 4.4) which was a two-hidden-layer feedforward SNN. The model achieved around 97.09% accuracy on MNIST and 85.52% accuracy on Fashion MNIST. Models 2 and 3 performed similarly to the first one on the MNIST dataset, scoring 96.63% and 96.33% accuracy respectively, but attaining lower performance on the Fashion MNIST dataset. Overall, the feedforward networks (models 1 and 2) performed better than recurrent ones (models 3 and 4).

Note that the accuracy of each tested network could be vastly improved. The tested SNNs comprised only fully connected layers with a very basic structure and did not use any regularization. Implementing SG learning in state-of-the-art architectures (such as convolutional networks) should introduce additional performance increase. Compared to the CNN with homogenous filter capsules [5] which achieves state-of-the-art results on both datasets, model 1 had around 2.67% worse accuracy on MNIST and 8.10% worse accuracy on Fashion MNIST.

Dataset	Model	Accuracy	Precision	Recall	F1
MNIST	1	**0.9709**	0.9705	0.9708	0.9706
	2	0.9663	0.9660	0.9659	0.9660
	3	0.9633	0.9629	0.9628	0.9628
	4	0.9262	0.9259	0.9261	0.9253
Fashion MNIST	1	**0.8552**	0.8556	0.8552	0.8541
	2	0.8379	0.8391	0.8380	0.8347
	3	0.8223	0.8294	0.8224	0.8178
	4	0.7464	0.7500	0.7465	0.7387

Table 4.5: Accuracy, precision, recall, and F1 score of the SNN models from Table 4.4 on the MNIST and Fashion MNIST datasets.

4.3 Spatial attention shifts to colored items dataset classification

The second researched spatio-temporal dataset was gathered from the BNCI Horizon website (data from Spatial attention shifts to colored items experiment) [32]. Similarly to the P300 dataset in Section 4.1, this dataset also contains EEG signals from a BCI experiment. In this experiment, each participant was asked several questions to which they could respond with either "yes" or "no" answer. For every question asked, the participant made a response by paying attention to either green "+"-cross (to answer "yes") or to red "x"-cross (to answer "no"). The experiment consisted of 7 runs per participant, where each run contained 24 trials (i.e., 168 samples per participant). In the first two runs, the participants were not asked any questions and instead focused on the red/green cross. Runs 3-6 composed of only objective questions (e.g., 'Is Berlin a city?'), while the last run comprised purely subjective ones (e.g., 'Are you a vegetarian?'). The aim of the spiking network here is again very similar to the P300 experiment. The classifier is trained on the part of the EEG data to be able to decide whether a specific sample corresponds to a "yes" or "no" answer from the participant, while the rest is used to test its performance.

4.3.1 Data preprocessing

Firstly, the dataset was preprocessed in the same manner as in the original article [32]. From the original 29 channels in the EEG signal, only 14 parieto-occipital channels were used and data from the rest of the channels were discarded. The pre-stimulus part of the EEG signal was trimmed and only the remainder was used. Subsequently, the data were transformed using a zero-phase IIR Butterworth bandpass filter and resampled from 250 Hz to 50 Hz. The signal was split into epochs, where each epoch took 750 ms after stimulus onset. Such preprocessing resulted in a $14 \times 36 \times 10$ tensor for every trial (i.e., a sequence of 10 stimuli in 36 sampling points per channel). In total, 2976 trials (samples) were extracted and could be used by a neural network.

4.3.2 Model architecture

The next step was to construct an analog network that could recognize the data and be converted to an SNN. Similarly to the experiment with the GTN dataset in Section 4.1, Nengo was used here as well to perform the conversion.

Three different models of neural networks were tested on the dataset. The first one was a CNN with two convolutional layers (one with 32 filters using 5×5 kernel and the other with 64 filters and a kernel size of 3×3), dropout, and an average pooling layer after each convolution, followed by a series of fully connected layers to perform the classification. The convolutional and dense layers used ReLU activations (except the output dense layer which used softmax). The architecture of the network can be seen in Fig. 4.5. The other two tested models were the CNN from the previous experiment with the P300 dataset (as shown previously in Fig 4.1) and an LSTM network. The LSTM network was composed of two LSTM layers, each followed by a dropout layer to introduce regularization. The visualization of the model is in Fig. 4.6. Unfortunately, the LSTM network could not be run using Nengo, and therefore it was simulated in TensorFlow and was only used for the comparison with the other two models (no SNN model was made).

4.3.3 Training on the entire dataset

Firstly, each of the three models was trained on the EEG data from the entire dataset. The dataset was split and 25% of it was used as testing data. The non-testing data (i.e., the remaining 75%) was used for K-fold cross-validation of 10 iterations ($k = 10$). The CV method split the data into 10 folds (groups) and in each iteration, one fold was selected to serve as validation data, while the rest was used to train the network. The model was trained for 30 epochs with an early stopping callback to avoid overfitting (patience of 8 epochs was used, since it performed the best during the initial tests; the best weights were restored as in the P300 experiment). The Adam optimizer and binary cross-entropy were used to train the network.

After fitting the model, it was evaluated on the 25% of the previously held out data and converted to a spiking network (except for the LSTM). The SNN was constructed in the same way as in the previous P300 experiment in Chapter 4.1 (i.e., swapping ReLU activations with spiking rectified linear activations). The SNN ran with spike scaling set to 1000 and synaptic smoothing set to 0.01 to closely match the performance of the original analog model. Subsequently, the SNN was fed the training data in 50 time steps and the last time step was used to compute classification metrics.

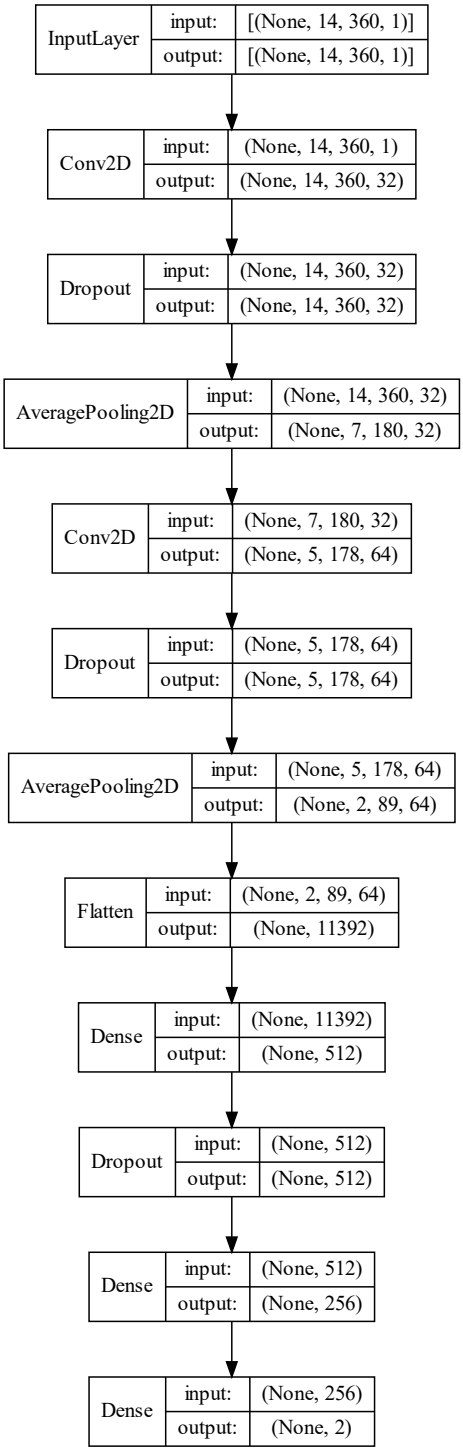

Figure 4.5: Tensorflow model of the first convolutional neural network used for the binary classification of the BNCI Horizon dataset.

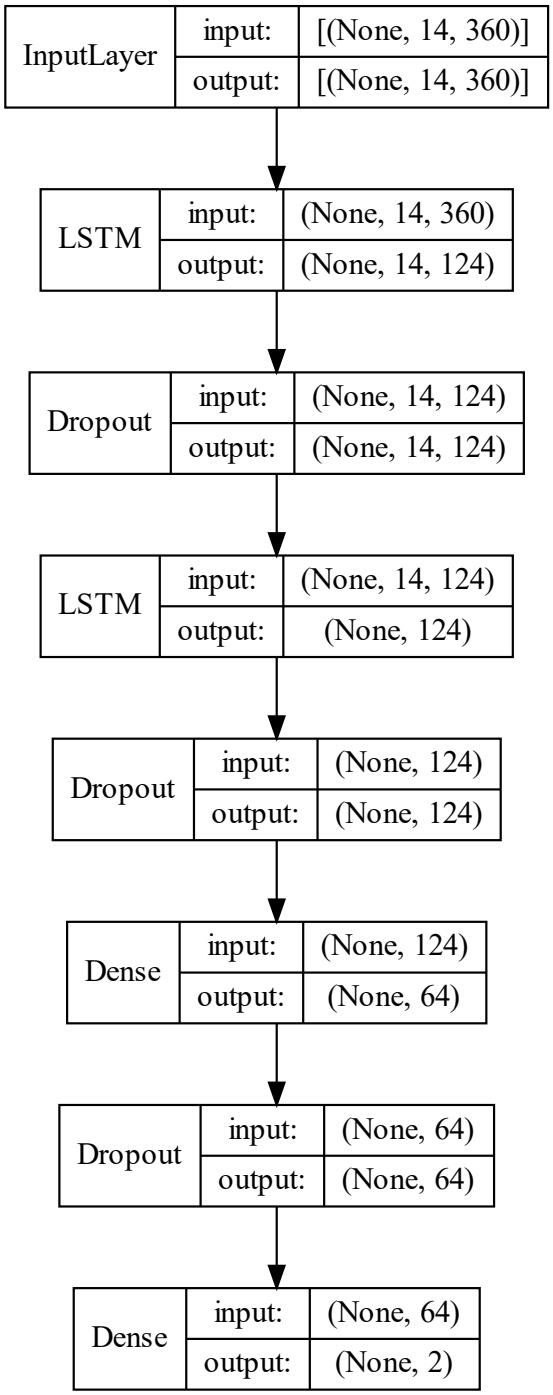

Figure 4.6: Tensorflow model of the LSTM network used for the binary classification of the BNCI Horizon dataset.

4.3.4 Training on samples from male and female subjects

Since the dataset also includes information about the gender of each participant, two smaller datasets were formed - one containing only samples from female subjects (1680), while other contained only samples from male subjects (1296). Both datasets were tested in the exact same manner as in the case of all samples (i.e., 10-fold CV was run, training the network for 30 epochs and then converting both CNN models to a spiking network).

4.3.5 Training on samples from each participant separately

The third part of the experiment involved training each of the three models on data from a single participant (separately). This part of was done to find how well can both spiking and non-spiking networks perform in comparison with a method from article [32] which was used with this dataset as well. This method did not use a neural network and instead applied canonical correlation analysis (CCA). The analysis technique computed correlation vectors with which it was possible to classify a given EEG signal. The method, however, did not use the entire dataset (of all 18 participants) but rather data from a single subject.

All three models were trained similarly to the CCA-based model. The process of training and evaluation was split into 18 iterations, each containing samples from one participant. The model was iteratively reset (so it could not remember the data from the previous subject), trained with 75% of the samples and tested with the remaining 25%. Due to the low number of samples per participant, no cross-validation was applied as it would negatively impact the training process. The individual training of the model was again split into 30 epochs and an early stopping callback was applied (since there was no validation data available, the loss calculated from the training data was used instead). The conversion of the analog model was also the same as in the case of the entire dataset and data from male/female subjects. ReLU activations were swapped with spiking ones, firing rate scaling was configured to 1000, synapse parameter was set to 0.01, and the input was fed to the network in 50 time steps.

4.3.6 Results

Unfortunately, not a single tested model performed well on this dataset. The average accuracy in all cases stayed around 50% which implies that the model cannot generalize and performs similarly to picking the answer randomly. Since the spiking models were converted from a "trained" analog model, their performance did not improve either as they essentially used the same weights. Additionally, there were also issues with how both CNN models were able to classify the data. In some cases the model completely ignored one label and classified only the other. This was not the case with the LSTM network which did classify both classes equally, however, it did not achieve better results.

The statistics regarding accuracy can be seen in Table 4.6. Table 4.7 shows precision, recall, and F1 score metrics. Note that since the CNNs sometimes did not classify one class, this heavily influenced maximum recall. Thus, maximum recall and maximum F1 score were excluded from the table and only averages are shown. The best-recorded accuracy was achieved by the spiking CNN model from the GTN experiment on data from individuals - 61.90%. Though this might be misleading since there was relatively a small number of samples per individual. Using the data from the entire dataset, the best performing network was the analog LSTM which attained 52.96% maximum accuracy. Use of data only from one specific gender slightly improved the maximum accuracy as the spiking variant of the first CNN model (Fig. 4.5) achieved around 56.25% on the samples from male subjects and 54.09% on data from female subjects (for both spiking and non-spiking variant).

The low performance might be caused by various reasons such as a low number of samples to successfully train any neural network on, the architecture of the tested models, or the preprocessing applied. Overall, the CCA-based classifier tested in the original article of the dataset [32] seems superior to any neural network tested here, though the method is less complex and most likely more suitable for such a small dataset.

Part of the dataset	**Model**	**Type**	**Average accuracy**	**Max. accuracy**	**Accuracy SD**
All subjects	CNN	ANN	0.5060	0.5285	0.0230
	CNN	SNN	0.5039	0.5231	0.0195
	CNN (p300)	ANN	0.4905	0.5217	0.0221
	CNN (p300)	SNN	0.4853	0.5204	0.0204
	LSTM	ANN	0.5067	0.5296	0.0204
Male subjects	CNN	ANN	0.4806	0.5406	0.0433
	CNN	SNN	0.4844	0.5625	0.0366
	CNN (p300)	ANN	0.5141	0.5438	0.0270
	CNN (p300)	SNN	**0.5178**	0.5438	0.0274
	LSTM	ANN	0.5009	0.5247	0.0190
Female subjects	CNN	ANN	0.4892	0.5409	0.0242
	CNN	SNN	0.4962	0.5409	0.0248
	CNN (p300)	ANN	0.4925	0.5216	0.0125
	CNN (p300)	SNN	0.4933	0.5240	0.0139
	LSTM	ANN	0.5010	0.5238	0.0231
Individuals	CNN	ANN	0.5013	0.5476	0.0277
	CNN	SNN	0.5093	0.5714	0.0366
	CNN (p300)	ANN	0.4601	0.5714	0.0592
	CNN (p300)	SNN	0.4733	**0.6190**	0.0742
	LSTM	ANN	0.5090	0.5952	0.0636

Table 4.6: Average, maximum accuracy, and its standard sample deviation computed from the testing data for each model. **All subjects** - data from all 2976 samples in the dataset, **male subjects** - samples corresponding to only male subjects, **female subjects** - samples corresponding to only female subjects, **individuals** - fitting each model on samples from a single individual. **CNN** - the first CNN model, shown in Fig. 4.5. **CNN (p300)** - the CNN used in the previous GTN P300 experiment (Section 4.1), shown in Fig. 4.1. **LSTM** - the recurrent model, shown in Fig. 4.6.

Part of the dataset	Model	Type	Average precision	Max. precision	Average recall	Average F1 score
All subjects	CNN	ANN	0.4980	0.5263	0.7281	0.5564
	CNN	SNN	0.5184	0.5417	0.6398	0.5335
	CNN (p300)	ANN	0.5186	0.6061	0.4729	0.4536
	CNN (p300)	SNN	0.5140	0.6111	0.4651	0.4484
	LSTM	ANN	0.5289	0.5714	0.5577	0.4904
Male subjects	CNN	ANN	0.4996	0.5412	0.5584	0.4991
	CNN	SNN	0.5075	0.5609	0.5035	0.4736
	CNN (p300)	ANN	0.5326	0.5429	0.7578	0.6044
	CNN (p300)	SNN	0.5354	0.5522	0.7624	0.6096
	LSTM	ANN	0.5380	0.5714	0.5098	0.5133
Female subjects	CNN	ANN	0.4941	0.5404	0.6662	0.5589
	CNN	SNN	0.5013	0.5401	0.7290	0.5869
	CNN (p300)	ANN	0.4948	0.5447	0.5143	0.4442
	CNN (p300)	SNN	0.4870	0.5484	0.5119	0.4422
	LSTM	ANN	0.5053	0.5301	0.4887	0.4892
Individuals	CNN	ANN	0.3646	0.5882	0.6920	0.4733
	CNN	SNN	0.3709	0.5882	0.6286	0.4582
	CNN (p300)	ANN	0.4556	0.6000	0.3901	0.4130
	CNN (p300)	SNN	0.4646	0.6500	0.3958	0.4214
	LSTM	ANN	0.5064	0.6667	0.4854	0.4915

Table 4.7: Precision, recall, and F1 score computed from the testing data for each model (average/maximum). **All subjects** - data from all 2976 samples in the dataset, **male subjects** - samples corresponding to only male subjects, **female subjects** - samples corresponding to only female subjects, **individuals** - fitting each model on samples from a single individual. **CNN** - the first CNN model, shown in Fig. 4.5. **CNN (p300)** - the CNN used in the previous GTN P300 experiment (Section 4.1), shown in Fig. 4.1. **LSTM** - the recurrent model, shown in Fig. 4.6.

5 Conclusion

To sum up, this thesis overviewed the current knowledge of spiking networks and their comparison to analog networks. It described the relation of spiking networks to neuromorphic hardware as well as the state-of-the-art in the field. A wide range of tools to model and simulate spiking neural networks were investigated and their core features were overviewed. Out of all tested simulators and tools, Nengo, TensorFlow, and PyTorch were used in the experiments in Section 4 where several applications of spiking networks were demonstrated.

In total, three different experiments were conducted, two involving BCI EEG data and one using MNIST and Fashion MNIST datasets. Arguably, two out of the three experiments resulted in a success and their outcomes might be useful for further research in the field. The first experiment (Section 4.1) trained on a P300 dataset containing samples from guess the number experiment and used a modified CNN from article [44]. Surprisingly enough, the resulting spiking network had a better average (63.43%) and maximum accuracy (64.96%) than the original CNN (which scored around 62 - 64% maximum and 62.18% average accuracy). The performance could likely be even improved, either by increasing the number of samples or by using a different model. The second experiment (Section 4.2) trained a spiking net directly using the surrogate gradient on the MNIST and Fashion MNIST datasets. The best model for both datasets had an accuracy of 97.01% on MNIST and 85.52% on Fashion MNIST. The experiment did not utilize any regularization or special layers (i.e., convolutional or pooling), and therefore, applying these alongside SG-learning could further increase the overall performance. The only unsuccessful experiment was the last one where the BNCI Horizon dataset was used (Section 4.3). None of the tested models scored significantly better than a coin flip, and different preprocessing or more samples would most likely be necessary to improve their performance.

This thesis mainly focused on simulating spiking nets on an x64 system that follows classic von Neumann architecture, however, the work could be further researched from the perspective of different hardware platforms, e.g., neuromorphic devices. It would also be interesting to apply different types of spike encoding as here mostly rate-based coding was used.

The source code for this thesis also is available on a GitHub repository[1].

[1]The repository can be found here: `https://github.com/honzikv/use-of-snn`.

List of Abbreviations

ANN Analog (artificial) neural network.

ARM Advanced Risc Machine.

ASIC Application-specific integrated circuit.

BCI Brain-computer interface.

CCA Canonical correlation analysis.

CNN Convolutional neural network.

CPU Central processing unit.

CV Cross-validation.

DNN Deep neural network.

EEG Electroencephalography.

ELU Exponential linear unit.

FPGA Field programmable gate array.

GAN Generative adversarial network.

GPU Graphical processing unit.

GTN Guess the number (experiment).

GUI Graphical user interface.

HATS Histogram of averaged time surfaces.

LIF Leaky integrate-and-fire.

LSTM Long short-term memory (unit, network, etc.).

N-MNIST Neuromorphic MNIST.

RAM Random access memory.

ReLU Rectified linear unit.

RNN Recurrent neural network.

SG Surrogate gradient.

SNN Spiking neural network.

SPA Semantic pointer architecture.

STDP Spike-timing-dependent plasticity.

TTFS Time-to-first-spike.

Bibliography

[1] Navin Anwani and B. Rajendran. 2020. Training multilayer spiking neural networks using normad based spatio-temporal error backpropagation. *ArXiv*, abs/1811.10678. https://arxiv.org/pdf/1811.10678.pdf.

[2] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence 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, 1–13. ISSN: 1662-5196. DOI: 10.3389/fninf.2013.00048. http://compneuro.uwaterloo.ca/files/publications/bekolay.2014.pdf.

[3] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. 2021. Nengo: a Python tool for building large-scale functional brain models. (2021). https://www.nengo.ai/.

[4] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. 2021. Nengo: a Python tool for building large-scale functional brain models. (2021). https://www.nengo.ai/nengo-dl/examples/tensorflow-models.html.

[5] Adam Byerly, Tatiana Kalganova, and Ian Dear. 2020. A branching and merging convolutional network with homogeneous filter capsules. (2020). arXiv: 2001.09136 [cs.CV].

[6] D. Ciregan, U. Meier, and J. Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 3642–3649. DOI: 10.1109/CVPR.2012.6248110.

[7] Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. 2017. Recurrent batch normalization. (2017). arXiv: 1603.09025 [cs.LG].

[8] Matthieu Courbariaux and Yoshua Bengio. 2016. Binarynet: training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830. arXiv: 1602.02830. http://arxiv.org/abs/1602.02830.

[9] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. 2018. Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38, 1, 82–99. DOI: 10.1109/MM.2018.112130359.

[10] Arnaud Delorme, Jacques Gautrais, Rufin van Rullen, and Simon Thorpe. 1999. Spikenet: a simulator for modeling large networks of integrate and fire neurons. *Neurocomputing*, 26-27, 989–996. ISSN: 0925-2312. DOI: https://doi.org/10.1016/S0925-2312(99)00095-8. https://www.sciencedirect.com/science/article/pii/S0925231299000958.

[11] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer. 2015. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International Joint Conference on Neural Networks (IJCNN)*, 1–8. DOI: 10.1109/IJCNN.2015.7280696.

[12] Jochen Martin Eppler, Robin Pauli, Alexander Peyser, Tammo Ippen, Abigail Morrison, Johanna Senk, Wolfram Schenck, Hannah Bos, Moritz Helias, Maximilian Schmidt, and et al. 2015. Nest 2.8.0, (September 2015). DOI: 10.5281/zenodo.32969.

[13] Steve K Esser, 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*. C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors. Volume 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2015/file/10a5ab2db37feedfdeaab192ead4ac0e-Paper.pdf.

[14] Friedemann Zenke and Manu Halvagal. 2019. Spytorch. (2019). https://github.com/fzenke/spytorch.

[15] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial networks. (2014). arXiv: 1406.2661 [stat.ML].

[16] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. 2018. Bindsnet: a machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics*, 12, 89. ISSN: 1662-5196. DOI:

10.3389/fninf.2018.00089. https://www.frontiersin.org/article/10.3389/fninf.2018.00089.

[17] Dana Hughes and Nikolaus Correll. 2016. Distributed machine learning in materials that couple sensing, actuation, computation and communication. (2016). arXiv: 1606.03508 [cs.LG].

[18] Eric Hunsberger and Chris Eliasmith. 2015. Spiking deep networks with lif neurons. (2015). arXiv: 1510.08829 [cs.LG].

[19] E. M. Izhikevich. 2003. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14, 6, 1569–1572. DOI: 10.1109/TNN.2003.820440.

[20] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n. d.] Cifar-10 (canadian institute for advanced research). http://www.cs.toronto.edu/~kriz/cifar.html.

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

[22] 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. https://www.frontiersin.org/article/10.3389/fnins.2016.00508.

[23] Giovanni Mariani, Florian Scheidegger, Roxana Istrate, Costas Bekas, and Cristiano Malossi. 2018. Bagan: data augmentation with balancing gan. (2018). arXiv: 1803.09655 [cs.CV].

[24] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. 2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345, 6197, 668–673. ISSN: 0036-8075. DOI: 10.1126/science.1254642. eprint: https://science.sciencemag.org/content/345/6197/668.full.pdf. https://science.sciencemag.org/content/345/6197/668.

[25] 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 2017), 160121. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.121. https://doi.org/10.1038/sdata.2016.121.

[26] E. O. Neftci, H. Mostafa, and F. Zenke. 2019. Surrogate gradient learning in spiking neural networks: bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36, 6, 51–63. DOI: 10.1109/MSP.2019.2931595.

[27] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. 2016. Phased lstm: accelerating recurrent network training for long or event-based sequences. (2016). arXiv: 1610.09513 [cs.LG].

[28] Rodrigo M. S. de Oliveira, Ramon C. F. Araújo, Fabrício J. B. Barros, Adriano Paranhos Segundo, Ronaldo F. Zampolo, Wellington Fonseca, Victor Dmitriev, and Fernando S. Brasil. 2017. A system based on artificial neural networks for automatic classification of hydro-generator stator windings partial discharges. *Journal of Microwaves, Optoelectronics and Electromagnetic Applications*, 16, 628–645. ISSN: 2179-1074. http://www.scielo.br/scielo.php?script=sci_arttext&pid=S2179-10742017000300628.

[29] J. A. Pérez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. 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, 2706–2719. DOI: 10.1109/TPAMI.2013.71.

[30] Michael Pfeiffer and Thomas Pfeil. 2018. Deep learning with spiking neurons: opportunities and challenges. *Frontiers in Neuroscience*, 12, 774. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00774. https://www.frontiersin.org/article/10.3389/fnins.2018.00774.

[31] Dimitri Plotnikov, Bernhard Rumpe, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, and Abgail Morrison. 2016. Nestml: a modeling language for spiking neurons. (2016). arXiv: 1606.02882 [cs.SE].

[32] Christoph Reichert, Igor Fabian Tellez Ceja, Catherine M. Sweeney-Reed, Hans-Jochen Heinze, Hermann Hinrichs, and Stefan Dürschmid. 2020. Impact of stimulus features on the performance of a gaze-independent brain-computer interface based on covert spatial attention shifts. *Frontiers in Neuroscience*, 14, 1250. ISSN: 1662-453X. DOI:

10.3389/fnins.2020.591777. https://www.frontiersin.org/article/10.3389/fnins.2020.591777.

[33] Kalivoda Roman. 2020. *Extension of neural network architecture.* Bachelor's Thesis. University of West Bohemia. http://hdl.handle.net/11025/41790.

[34] B. Rueckauer and S. Liu. 2018. Conversion of analog to spiking neural networks using sparse temporal coding. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 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, 682. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682. https://www.frontiersin.org/article/10.3389/fnins.2017.00682.

[36] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. 2021. Snn-toolbox. https://snntoolbox.readthedocs.io/en/latest/_images/features.png.

[37] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. 2010. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1947–1950. DOI: 10.1109/ISCAS.2010.5536970.

[38] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. 2019. Going deeper in spiking neural networks: vgg and residual architectures. *Frontiers in Neuroscience*, 13, 95. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00095. https://www.frontiersin.org/article/10.3389/fnins.2019.00095.

[39] Amos Sironi, Manuele Brambilla, Nicolas Bourdis, Xavier Lagorce, and Ryad Benosman. 2018. Hats: histograms of averaged time surfaces for robust event-based object classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* (June 2018).

[40] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15, 1, (January 2014), 1929–1958. ISSN: 1532-4435.

[41] Marcel Stimberg, Dan F. M. Goodman, and Thomas Nowotny. 2020. Brian2genn: accelerating spiking neural network simulations with graphics hardware. *Scientific Reports*, 10, 1, (January 2020), 410. ISSN: 2045-2322. DOI: 10.1038/s41598-019-54957-7. https://doi.org/10.1038/s41598-019-54957-7.

[42] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. 2019. Deep learning in spiking neural networks. *Neural Networks*, 111, 47–63. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2018.12.002. http://www.sciencedirect.com/science/article/pii/S0893608018303332.

[43] 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. https://www.frontiersin.org/article/10.3389/fninf.2017.00046.

[44] Lukáš Vařeka. 2020. Evaluation of convolutional neural networks using a large multi-subject p300 dataset. *Biomedical Signal Processing and Control*, 58, 101837. ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2019.101837. http://www.sciencedirect.com/science/article/pii/S1746809419304185.

[45] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747. arXiv: 1708.07747. http://arxiv.org/abs/1708.07747.

[46] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. 2018. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9, 4, (August 2018), 611–629. ISSN: 1869-4101. DOI: 10.1007/s13244-018-0639-9. https://doi.org/10.1007/s13244-018-0639-9.

[47] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. 2019. A review of recurrent neural networks: lstm cells and network architectures. *Neural Computation*, 31, 7, 1235–1270. PMID: 31113301. DOI: 10.1162/neco_a_01199. eprint: https://doi.org/10.1162/neco_a_01199. https://doi.org/10.1162/neco_a_01199.

[48] Y. Zhang, P. Li, Y. Jin, and Y. Choe. 2015. A digital liquid state machine with biologically inspired learning and its application to speech recognition. *IEEE Transactions on Neural Networks and Learning Systems*, 26, 11, 2635–2649. DOI: 10.1109/TNNLS.2015.2388544.

[49] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis C. M. Lau. 2015. A c-lstm neural network for text classification. (2015). arXiv: 1511.08630 [cs.CL].

A User guide to run the experiments

A.1 Setting up the tools

This attachment describes how to set up and run all three experiments conducted in the thesis. The experiments are written in Python in the form of Jupyter Notebook and were tested on a desktop computer with Windows 10[1].

Download the necessary files (i.e., the entire repository) from the GitHub repository: `https://github.com/honzikv/use-of-snn`, either via Git, or directly by clicking on **Code** → **Download Zip**. Extract the zip (if necessary) in a folder that will be used to launch the experiments.

A.1.1 Setting up Python

The next step is to set up Python. While there are many installers and distributions to choose from, the recommended one here is Anaconda as it allows a straightforward installation of the dependencies. Alternatively, the official installer (from `https://www.python.org/`) can be used as well, however, it might be more difficult to set up PyTorch.

Download Anaconda from: `https://www.anaconda.com/` and install it. Make sure that Anaconda is saved in the Windows PATH variable as it is going to be used from the console (using the conda command). Open the command line (press the win key, type "cmd", and click the icon). Now, create a new environment using the command:

```
conda create -n name_of_your_environment python=3.8.5
```

This will create an empty environment that can be used to install dependencies.

A.1.2 Installing Python dependencies

Installing dependencies is very straightforward and can be done via the command line as well. In the root directory of the repository, there is a "require-

[1]Linux and macOS operating systems should be compatible as well, however, they were not tested.

ments.txt" file that contains most of the necessary dependencies (PyTorch needs to be installed separately). Firstly, install this file using the following commands:

1. Activate the created conda environment using:

```
conda activate name_of_your_environment
```

2. Now this environment can be referenced via pip, which will install all dependencies from the requirements.txt file. Execute the following command:

```
pip install -r requirements.txt
```

After installing all dependencies from the requirements.txt files, PyTorch can be installed. Use either of the two commands, depending on whether the machine is equipped with an Nvidia GPU with CUDA support:

1. If the computer has CUDA GPU:

```
conda install pytorch torchvision torchaudio
   cudatoolkit=10.2 -c pytorch
```

2. If the computer does not have CUDA GPU:

```
conda install pytorch torchvision torchaudio cpuonly -
   c pytorch
```

Note that in some cases it might happen that PyTorch updates NumPy to a newer version, which may not be compatible with TensorFlow. To fix this, run:

```
pip install numpy==1.9.5
```

After these steps, the environment should be ready to run any of the three experiments.

A.2 Running the experiments

The experiments can be run using one or multiple Jupyter notebooks. Firstly, a Jupyter server needs to be launched, this can be either done in a command-line or in an IDE such as PyCharm Professional or Visual Studio Code. This example uses the command line approach as it is the easiest

one to set up. To run the server, simply open the command line in the root directory of the repository and execute the following command[2]:

```
jupyter notebook
```

This should prompt to open a web browser with the page that will be used to control the experiments. Alternatively, the URL will be displayed in the console. In the web application, navigate to the folder "experiments", where all three experiments are located. Each folder represents one experiment as described in Sections A.2.1 - A.2.3.

The experiments comprise (Jupyter) notebooks (files with .ipynb extension) that can be interactivelly run. Each notebook contains multiple "cells" which are blocks of executable code or text. Typically, the cells are meant to be executed sequentially, from top to bottom. Cells can be executed in many ways:

- To execute a single cell click the **Run** button or press **ctrl + enter**. (This also works for any text cells)
- It is also possible to run all cells sequentially by clicking on **Cell > Run All**

To restart the entire program (notebook), click on **Kernel > Restart**.

A.2.1 Spiking CNN on the guess the number Multi-subject P300 dataset

The first performed experiment was the experiment with P300 dataset. The experiment is located in the folder "experiments/p300_experiment". It contains two notebook files:

- **p300_dataset_exp_convnet.ipynb**
- **p300_stats_visualization.ipynb**[3]

Before running anything, make sure to download the EEG data first as they are not present in the repository. The download link is available here:

[2]Make sure to enable the Anaconda environment beforehand, otherwise, the command might not work correctly.

[3]This file is not necessary to run the experiment and was only used for visualization. To vizualize the model an installation of graphviz is necessary: `https://graphviz.gitlab.io/download/`.

https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/G9RRLN.

Place the downloaded file in the "dataset" folder in the root of the experiment and make sure it is named "VarekaGTNEpochs.mat". The first notebook from the previous list contains code to run the experiment and save statistics from it. It creates a CNN and performs a 30-epoch CV while also converting it to a spiking equivalent after every training. The second notebook uses the computed data and visualizes them (it is not required to run for the experiment).

A.2.2 Surrogate gradient training on MNIST and Fashion MNIST datasets with deep spiking networks

The second experiment tests four different models on the MNIST and Fashion MNIST datasets using the surrogate gradient. The experiment is located in the folder "experiments/surrogate_gradient_experiment". To perform it, simply run the Jupyter notebook file. No additional download of the datasets is necessary as they get directly downloaded via PyTorch. Note that it is recommended to run this experiment on a GPU since the computations are relatively long (in the span of hours).

A.2.3 BNCI Horizon Spatial attention shifts to colored items dataset experiment

The last experiment uses data from BNCI Horizon and is located in the folder "experiments/bnci_horizon_experiment".

To perform the experiment, the data from each participant firstly need to be downloaded from here: http://bnci-horizon-2020.eu/database/data-sets. On the website, navigate to the item with the number 28 named "Spatial attention shifts to colored items - an EEG-based BCI (002-2020)". Download all 18 participant files (P01 - P18) and place them in the "dataset" folder. Make sure that no other files (except ___init___.py) are present and the files are not renamed - i.e, the resulting folder will have the init.py file and the files P01 - P18.mat. The experiment contains five Jupyter notebook files and two Python files:

- **data_preprocessing.ipynb** and **data_preprocessing.py** - either of these files must be used before any other notebook as they preprocess the data and form the dataset

- **bnci_horizon_convnet_all_samples.ipynb** - is used to run the entire dataset, the samples from the female subjects, and the samples from the male subjects on the two used CNN models
- **bnci_horizon_convnet_individuals.ipynb** - is used to run both CNN models on data from a single subject
- **bnci_horizon_lstm_all_samples.ipynb** - runs the LSTM model on the entire dataset, the samples from the female subjects, and the samples from the male subjects
- **bnci_horizon_lstm_individuals.ipynb** - runs the LSTM model on data from a single subject
- **bnci_utils.py** - this file contains part of the functionality that is used across the notebooks and should not be edited.

Before running anything else, it is necessary to preprocess the dataset. To do so, either run the Jupyter notebook "data_preprocessing.ipynb" or execute the script "data_preprocessing.py" like so:

```
python data_preprocessing.py
```

After sucessfully running the preprocessing, the output (all preprocessed files) will be located in "dataset_result" folder. Subsequently, each of the four notebooks (with prefix bnci_horizon) can be run in any order. Files with all samples for both the CNNs and the LSTM can additionally be configured to run only with samples from female / male subjects (there is an explanation how to do it in the notebooks).