Philipp Josef Plank, BSc

Implementation of novel networks of spiking neurons on the Intel Loihi chip

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Biomedical Engineering

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl.-Ing. Dr.techn. Robert Legenstein

Institute of Theoretical Computer Science

Em.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass

Graz, December 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Contents

Abstract

In recent years, the development of neuronal networks and the improvement of applications based on artificial intelligence became a major focus of both industry and research. The advancement of computational resources enabled deep neural networks to become a very strong tool for image recognition and speech processing. However, artificial neural networks need an enormous amount of computing power which prevents the simulation of larger neural networks, that come close to the complexity of the human brain. Spiking neural networks would be more efficient in achieving this goal, in particular using specialized neuromorphic computing chips. The recently developed neuromorphic chip Loihi from Intel allows for efficient simulations of spiking neural networks. Loihi was examined in detail and a novel spiking neural network architecture, the long short-term memory spiking neural network, was implemented on the chip. Furthermore, the sequential MNIST classification task was performed on Loihi. Therefore, a TensorFlow model was adapted in order to train a network for Loihi with the supervised learning method backpropagation through time. It was shown that the Loihi implementation achieved results on the sequential MNIST task that were similar to software simulations of both artificial and spiking neural networks.

Zusammenfassung

Die Entwicklung von auf künstlicher Intelligenz basierenden Anwendungen, vor allem mittels Neuronalen Netzwerken, rückten in den letzten Jahren immer stärker in den Fokus von Industrie und Forschung. Die steigende Verfügbarkeit von Rechenleistung ermöglichte den Einsatz von tiefen Neuronalen Netzwerken, die vor allem in der Bilderkennung und Sprachverarbeitung ihren Einsatz finden. Allerdings benötigen diese Neuronalen Netzwerke eine solch enorme Menge an Rechenleistung, dass Simulationen von größeren Netzwerken, deren Komplexität der des menschlichen Gehirns nahe kommt, nicht machbar sind. Eine mögliche Lösung für dieses Problem ist die Verwendung von spikenden neuronalen Netzwerken, vor allem in Kombination mit neuromorpher Hardware. Intel entwickelte einen innovativen neuromorphen Chip namens Loihi, welcher für solche Aufgaben eingesetzt werden könnte. Im Zuge dieser Arbeit wurde untersucht ob neuartige Algorithmen für spikende neuronale Netzwerke auch auf dem Loihi Chip von Intel angewandt werden können. Dafür wurde das Long short-term memory spiking neural network Modell auf Loihi umgesetzt und ein Bilderkennungsexperiment, genannt sequentielles MNIST, durchgeführt. Außerdem wurde ein TensorFlow Modell weiterentwickelt, welches es ermöglicht, das an Loihi angepasste Modell zu simulieren und zu trainieren. Die auf Loihi erzielten Klassifikationsergebnisse waren ähnlich gut wie jene von in Software simulierten spikenden Netzwerken und von klassischen künstlichen Neuronalen Netzwerken.

Acknowledgment

I would like to thank my thesis advisers Professor Robert Legenstein and Professor Wolfgang Maass of the Institute for Theoretical Computer Science at Graz University of Technology for the opportunity to work on this project and the constant advice and help during my research and writing.

I would also like to thank Mike Davies and Andreas Wild from Intel for their active support of this research project and the opportunity to work with them. I am also grateful for the support and insight of Guillaume Bellec and Darjan Salaj, also from the Institute for Theoretical Computer Science.

Finally, I must express my very profound gratitude to my parents and to my friends for providing me with continuous encouragement throughout my years of study and during the writing of this thesis. This achievement would not have been possible without them.
Thanks to all of you!

Philipp Plank

1 Introduction

In recent years Artificial Neural Networks (ANNs) provided the basis for groundbreaking progress in many machine learning applications, from classic applications like object recognition and speech translation to locomotion of robots [1] and medical diagnosis [2]. Therefore ANNs got a lot of focus from industry and academia to improve the performance of these deep learning models on established hardware architectures and exploit it for extremely fast dense matrix multiplications [3]. The easiest and safest way to improve the performance of a deep neural network is to increase the number of neurons. This however this leads to two drawbacks: overfitting and dramatically increased use of computational resources [4]. Stacking up the computational resources might be a solution [5], but certainly not an efficient one. A more sophisticated solution which copes with both problems, would be to use sparsely connected ANNs [6, 4, 7]. This might provide a novel solution to two big problems in machine learning, but hardly anybody uses it, because the frameworks and infrastructures were optimized for fast dense matrix multiplication, to a point that it would not make sense to use a sparse neural network [4]. Thus, [4] aimed for an intermediate architecture that still could exploit current hardware features.

This raises the question if ignoring the established hardware optimizations and using a different sparse architecture would be beneficial after all. The brain, as the biggest and most complex working neural network known so far, heavily relies on sparsity. So sparsity seems to be a key attribute for an efficient usage of neural networks with higher numbers of neurons. Another key attribute of the human brain or brains in general would be the use of pulses of current or spikes to transfer information between neurons instead of continues values. Spiking Neural Network (SNN) architectures have been around quite some time and have in theory already been proven to be as powerful if not more powerful than present ANNs [8]. A lack of an effective supervised training method and the relative computationally-intensive simulation on conventional hardware prevented their practical usage so far.

A human brain consists of roughly 100 billion nerve cells which are interconnected by 100 trillion synapses. It would need a whole lot of resources to simulate it, even with simplifications and approximations for parts of this very complex machine. Simulating the human brain or parts of it for various purposes, e.g. new research methods for brain disorders, network designs and new perspectives on deep learning, have been a big challenge since the first neuron models were proposed. A simulation of the human brain with ANNs or SNNs on established hardware was never possible and probably would still need

more powerful hardware than currently available [9]. As contemporary chip architectures struggle with efficient calculation of these big, sparse and more complex models a different approach might provide some solutions - neuromorphic hardware.

In the last decade large-scale neuromorphic hardware has been developed, such as SpiNNaker [10], BrainScaleS [11], TrueNorth [12] and most recently Loihi [13] which should allow for efficient simulations of very large neural networks and practical real time applications of SNNs. Also effective supervised training algorithms have been recently proposed [14, 15] or even the conversion of ANNs to SNNs [16]. Thus, developing feasible SNN applications could finally progress to the same level as with ANNs.

The central question of this thesis was if a neuromorphic hardware implementation of a SNN could compete with software SNN simulations and ANN implementations on conventional hardware. Therefore a novel SNN architecture, the long short-term memory spiking neural network (LSNN) [17] was implemented on Intel's neuromorphic hardware chip Loihi. As application the sequential MNIST [18, 19] classification task was chosen which was also performed by [17]. It was shown that LSNNs can achieve similar performance to LSTM [20] networks on the sequential MNIST task [17]. In order to train the LSNN efficiently the LSNN TensorFlow [21] model from [17] was modified to represent and incorporate the dynamics and constraints of the Loihi chip.

Chapter 2 gives an overview on SNNs, the current status of supervised learning methods for SNNs and an introduction to neuromorphic hardware. A detailed description of the Loihi neuromorphic platform can be found in chapter 3 and the implementation of LSNNs on Loihi is described in chapter 4. The necessary TensorFlow adaptions for the software model are specified in chapter 5 and the comparison of the results on the sequential MNIST task chapter are stated in chapter 6. The thesis concludes with a discussion of the found results in chapter 7 and a concluding statement in chapter 8.

2 Background

2.1 Spiking Neural Networks

Spiking neural networks (SNNs) more closely resemble biological neural networks compared to other ANN models, which are based on highly simplified dynamics [8]. In comparison to ANNs, SNNs not only use the neuronal and synaptic state in their models, but also the concept of time. This means neurons do not propagate their value at a fixed clock rate, but dependent on their previous inputs and outputs. This event based behavior was derived from biological neuron models and experiments.

In 1952 Hodgkin and Huxley proposed their popular scientific model of a biological spiking neuron which incorporates ionic interactions that lead to action potentials [22]. Further on action potentials or spikes are not transmitted directly between the neurons in biology as it involves synapses [23]. So in order to model the biological behavior the type of the synapse, which can be electrical or chemical has to be chosen. If it would be a chemical synapse the exchange of neurotransmitters in the synaptic gap [24] would have to be modeled. It is also known that even more complex processes throughout the whole brain affect the synaptic behavior. These are not fully understand yet and topic of ongoing research. Therefore much simpler models, which require less equations to compute are widely used for spiking neural networks [25].

The leaky integrate-and-fire (LIF) model is one of these models and was used in the experiments of this thesis. It is a derivative of the integrate-and-fire model [26] in which an applied input current increases the membrane potential of a neuron over time until a certain threshold is reached. If the threshold is reached the neuron ejects a spike as output and the membrane potential resets to some resting potential. In the LIF model a leak term is additionally added which decreases the membrane potential over time. This reflects the diffusion behavior of ions through the membrane if a cell equilibrium is not reached [25]. The LIF model follows Eq. (2.1):

$$C_m \frac{dV_m(t)}{dt} = I(t) - \frac{V_m(t)}{R_m}, \tag{2.1}$$

where $I(t)$ represents the input current at time t, $V_m(t)$ is the membrane potential at time t, R is the membrane resistance and C_m is the membrane capacitance. This equation represents a resistor-capacitor circuit with the time constant $R_m C_m$. The integration of $I(t)$ is due to the capacitor, which is in parallel to the leakage causing resistor. The LIF

model does not explicitly model spiking events. If the membrane potential $V_m(t)$ reaches a certain threshold V_{th} a spike is generated and V_m is instantly reset to a lower value V_r. Subsequently the process of Eq. (2.1) starts again with $V_m = V_r$.

2.2 Backpropagation through time for SNNs

One of the big disadvantages of SNNs was the lack of suitable supervised learning algorithms. The application of error back-propagation [27] in deep artificial neural networks showed great successes in the past years, especially on recurrent neural networks (RNNs) [28]. Backpropagation through time (BPTT) is one popular method used to train recurrent neural networks. The recurrent network is unfolded in time and after that the backpropagation algorithm is applied. Backpropagation is used to calculate gradients in order to use them for the computation of the weight updates during training of a neural network. An error is computed at the output layer, which gets propagated backwards through the layers of the network. A detailed description of the algorithm, its variations as well as its applications in machine learning is given in [28]. Although, BPTT does not reflect biological motivated learning models and there is no neurophysiological evidence that BPTT or similar techniques are used within the brain [29], it would definitely be a powerful method to use for training SNNs. The major problem of BPTT in SNNs is that it requires the derivative of the loss function to be known. This is challenging as the derivative of the membrane potential, i.e. the neuronal state is non-existent at the time a spike is elicited. Recently, it was shown that an approximated derivative can be used to address the non-differentiability of spikes. This was achieved for binary activations in feed forward SNNs [14, 15] as well as for recurrent SNNs [17]. The used pseudo-derivative continuously increased from 0 to 1 and decayed back to 0, as seen in Eq. (2.2):

$$\frac{dz_j(t)}{dv_j(t)} = \max\left\{0, 1 - |v_j(t)|\right\}, \tag{2.2}$$

where $v_j(t)$ represents the normalized membrane potential and $z_j(t)$ denotes the spike train. The authors of [17] used an additional dampening factor $\gamma < 1$ on the amplitude of the pseudo-derivative in order to stabilize this approach for very deep unrolled recurrent SNNs:

$$\frac{dz_j(t)}{dv_j(t)} = \gamma \max\left\{0, 1 - |v_j(t)|\right\}. \tag{2.3}$$

This approach Eq. (2.3) with a $\gamma = 0.3$ was used to train the recurrent SNN models of the experiments described in this thesis.

2.3 Neuromorphic Hardware

Traditional CPUs are based on the von Neumann architecture which is over 60 years old and still dominates, beside minor adaptions, the current development of computing chips. Two major aspects of the von Neumann model are the distinction of the computing units and the working memory as well as the data processing on a clock rate basis. Many years ago the increase of the clock rate was the main improvement point for new CPUs, as a higher clock rate allowed for more calculations per time unit. Around ten years ago the increase of the clock rate reached a physical limit due to heat loss and chip sizes. After that, improvements were mainly achieved by optimizing memory streams and adding more computing units to the CPUs. These parallel optimization approaches have one big disadvantage - bigger CPUs with more computing units need more resources and power. At the same time the downsizing of transistors is slowly but surely coming to an end. This means that after 70 years there is a realistic chance that future general purpose CPUs are based on a different architecture. One possible replacement of the present widely used CPU architecture could be found in the neuromorphic approach.

This approach of creating digital or analog equivalents of neurons in silicon, which communicate in parallel using pulses of electric current without a clocked time could be the future of computing [30]. The neuro-units of such chips process incoming flows of electricity in a similar matter as our brain does and also with much higher efficiency compared to contemporary computing units. The neuromorphic TrueNorth chip from IBM for example consumes only 70 milliwatts of power while having five times more transistors than a standard Intel processor which uses at least 35 watts [12]. At the same time some approaches of neuromorphic chips use standard processors for simulating described behavior, e.g. SpiNNaker [10] or incorporate analog neurons e.g. BrainScaleS [11]. Most recently also Intel presented a silicon implementation of a neuromorphic chip named Loihi [13], which is also very energy efficient and scalable to brain size amounts of neurons and synapses. A more detailed description of the design and possibilities of Loihi is given in chapter 3, as Loihi was used to perform the experiments.

3 Loihi

Loihi is a neuromorphic manycore processor which is capable of using on-chip learning rules [13]. It is a 60-mm^2 chip and fabricated in Intel's 14-nm FinFET process. Loihi has a fully digital architecture, that approximates the usually continuous time dynamics with a fixed-size discrete time step model. As the distributed dynamics of the neurons have to evolve in a well-defined manner all neurons need to maintain a synchronized timestamp throughout the entire chip. Before the cores advance their time step all spikes need to reach their destination. After all spikes arrived at their destination a barrier synchronization message is sent and the cores advance their time step. Therefore the time steps have no direct relationship to the hardware execution time. The length of a time step depends on the amount of spikes and amount of updates of the learning engine occur during that time step [13].

3.1 Architecture

A Loihi chip is composed of 128 neuromorphic cores and three embedded x86 processor cores. A maximum of 1024 spiking neural units or compartments can be implemented per neuromorphic core. This compartments share ten architectural memories with their fan-in and fan-out connectivity, configuration and state variables [13]. This means that densely connected neuron models need more memory for their fan-in and fan-out information or synapses, than sparser models. As the memory is limited there is a trade-off between the maximum amount of compartments which can be configured and their number of synapses per neuromorphic core. The more compartments are configured in a neuromorphic core the less synapses can be configured and vice versa. The same concept applies for other state variables which may be configured. This means that the actual possible amount of neurons and synapses to configure depend on the complexity of the model.

Loihi uses a variation of the current-based (CUBA) leaky-integrate-and-fire model which implements the synaptic response current u and the membrane potential v as two internal state variables. The current u is the sum of filtered and weighted input spikes and an optional bias current. The membrane potential integrates the current u and generates a spike when it passes its firing threshold potential. Both state variables are leaky and have their own time constant describing their exponential decay. The exact equations for u and v for one compartment can be seen in Eq. (3.1) and Eq. (3.2):

Table 3.1 Precision of the parameters for the neuronal dynamics on Loihi. Variables with a preceding u/s are unsigned/signed. The sign is included in the number of bits specified.

Symbol	Precision	Description
wgt	u8 bit	Synaptic weight used for spike accumulation.
$wgtExp$	s4 bit	Shared weight exponent used to scale magnitude of weights.
u	s24 bit	Compartment current.
v	s24 bit	Compartment membrane potential.
$v^{threshold}$	u17 bit	Membrane threshold for a compartment.
τ_u	u12 bit	Shared decay constant for compartment current u.
τ_v	u12 bit	Shared decay constant for compartment membrane voltage v.

$$u(t) = \left\{ \left[u(t-1) \cdot (2^{12} - integer(\frac{1}{\tau_u} \cdot 2^{12})) \right] >> 12 \right\} + 2^{6+wgtExp} \cdot wgt \cdot s_{in}(t) \quad (3.1)$$

$$v(t) = \left\{ \left[v(t-1) \cdot (2^{12} - integer(\frac{1}{\tau_v} \cdot 2^{12})) \right] >> 12 \right\} + u(t) + b \quad (3.2)$$

$$s_{out}(t) = \begin{cases} 1, & \text{if } v(t) > v^{threshold} \cdot 2^6 \\ 0, & \text{otherwise} \end{cases} \quad (3.3)$$

where τ_u and τ_v are the respective time constants for the leak, $integer(..)$ refers to rounding the value to an integer, the operator $>>$ refers to a right-shift with LSB truncation, wgtExp and wgt refer to the synaptic weight value with the actual weight being $wgt \cdot 2^{6+wgtExp}$, s_{in} as binary input spike, s_{out} as binary output spike following Eq. (3.3) and b is an optional additive constant bias voltage. If there is an input spike, s_{in} is one and the weight value is added to the first decayed old current u. After that the membrane potential v is also decayed and integrates the new u as well as add an optional bias. If the membrane potential v reaches the threshold $v^{threshold} \cdot 2^6$ an outgoing spike is generated and the membrane potential is reset to zero. The maximum precision of the parameters for the neuronal dynamics on Loihi are shown in Table 3.1.

The biologically inspired internal structure of a neuromorphic core on Loihi is shown in Figure 3.1. The SYNAPSE unit processes the incoming spikes according to the synaptic weights and the DENDRITE unit updates the state variable u and v for all the neurons. Outgoing spikes are generated in the AXON unit for each neuron which reaches the threshold potential and the LEARNING unit updates the synaptic weights according to the implemented learning rule.

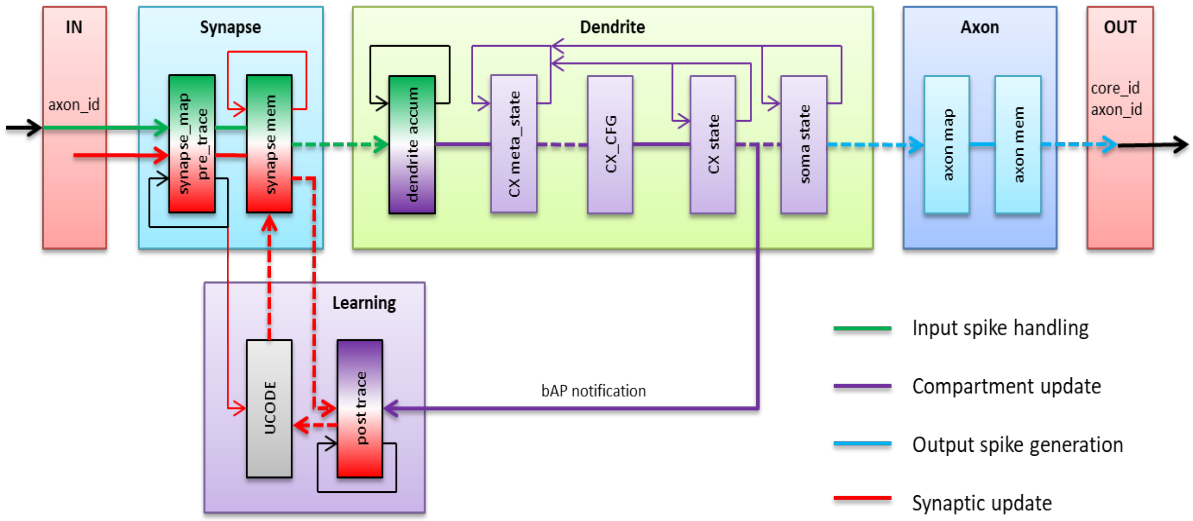


Figure 3.1 Top-Level Microarchitecture of a neuromorphic core.

3.2 Configuration

Loihi comes with a Python-based API to specify SNN topologies on the chip as well as a compiler and runtime library for building and executing SNNs on the Loihi chip [31]. This API is called NxAPI and allows to configure the network in the form of specifying nodes and their connections, similar to models in graph theory. Once the network, input spikes and output monitors are configured the compiler can build it and send it to the chip itself in order to run it. The monitored result data can then be inspected in Python again. The main state variables to monitor are the current u, the membrane potential v as well as the generated spikes s of the compartments.

There are three depth levels of configuring or programming the chip. The highest level option is using the NxNet API, which encapsulates the complex register based configurations of the neurons and synapses of the hardware in easy to use methods. Basically it provides a SNN API that does not require any knowledge about the underlying hardware. Therefore the NxNet API is similar to other SNN simulators or libraries like Brian [32] or PyNN [33]. While it is very convenient to use NxNet, it hides away some advanced configuration possibilities.

The NxCore API is still implemented in python but it requires knowledge of the hardware configuration and therefore it is needed to configure the registers explicitly. Thus, additional knowledge of the hardware is needed but it also means that registers can be configured in different ways than initially intended which makes the API more flexible. The third and lowest level to configure Loihi is through C. While it is not a full API, NxNet and NxCore offer the possibility to program symmetric neural interfacing processes (SNIPs).

This framework offers to include simple C programs which run on one of the x86 cores of Loihi. SNIPs offer an even deeper interaction level during the execution of the SNN on the chip but require also deeper knowledge of the hardware processes.

The experiments involving Loihi in this thesis were first configured solely with the NxCore API and were later ported to an implementation using NxNet and SNIPs. The SNIPs were used for encoding the input of the SNN directly on the chip rather than process it in python. To access a Loihi chip a remote connection via SSH to Intel Labs was used, as at that time the chips were only available there.

4 LSNN on Loihi

4.1 Long short-term memory in networks of spiking neurons

In ANNs long short-term memory (LSTM) cells made a big breakthrough in recent years. Due to availability of Big Data and enhanced computing power recurrent LSTM networks became a very popular architecture for many deep learning models and applications [34]. As deep in deep learning refers to vast amount of layers through which the data is transformed, an additional memory is needed to train them with error back-propagation methods. Otherwise the propagated error would diminish after being passed through a few layers and the cells in these layers would not change their parameters at all i.e. not learn anything. Therefore the cells got an explicit memory and information gating which helps to overcome this problem due to more options of propagating/processing the error signal over a longer time period. LSTM networks are used today in major application e.g. speech recognition [35] of smart phones and smart devices [36], chat bots [37], speech translation [38] or gaming bots which beat human players in various games [39].

It is clear that this long short-term memory is beneficial for RNNs, so the general idea was also applied to SNNs. [17] developed such a long short-term memory architecture for spiking neurons, the LSNN. However it is not quite the same as LSTMs. While an LSTM cell has a non-volatile memory which can be only changed through explicit gates an LSNN is somewhat simpler. An LSNN consists of a population of regular LIF neurons and a population of adaptive LIF neurons. As a spiking neuron has a threshold, which the membrane potential needs to reach in order to elicit a spike, adapting this threshold on a time scale of seconds becomes the memory i.e. holds the additional information.

The adapting LIF neurons differ from a regular LIF neuron in the way, that the adaption changes the firing threshold of a neuron in response to its own firing, which reduces its excitability. This means that an adapting neuron, which has just fired a spike due to receiving an input spike train, will be less likely to fire again when receiving the same input spike train for a certain time period. The adapted threshold decays back to a baseline with some time constant. This time constant determines then the volatility of the memory. [17] used a simple model for this adapting neurons:

$$I_j(t) = \sum_i W_{ji}^{in} x_i(t - d_{ji}^{in}) + \sum_i W_{ji}^{rec} z_i(t - d_{ji}^{rec}), \tag{4.1}$$

$$V_j(t+\delta t) = \exp(-\frac{\delta t}{\tau_m})V_j(t) + (1-\exp(-\frac{\delta t}{\tau_m}))R_m I_j(t) - B_j(t)z_j(t)\delta t, \tag{4.2}$$

$$B_j(t) = b_j^0 + \beta b_j(t), \tag{4.3}$$

$$b_j(t+\delta t) = \exp(-\frac{\delta t}{\tau_{a,j}})b_j(t) + (1-\exp(-\frac{\delta t}{\tau_{a,j}}))z_j(t). \tag{4.4}$$

$I_j(t)$ denotes the input current and is the weighted sum of incoming spikes from external inputs as well as inputs from other neurons in the network. W_{ji}^{in} and W_{ji}^{rec} represent the input and the recurrent synaptic weights and d_{ji}^{in} as well as d_{ji}^{rec} denote the corresponding synaptic delays. Neuron j fires at time t if the membrane potential $V_j(t)$ is above its threshold $B_j(t)$ and the neuron is not in refractory state at time t. The refractory period for neuron j begins after a spike is emitted from neuron j and lasts a set amount of time steps t. During the refractory period $z_j(t)$ is set to 0. The firing threshold $B_j(t)$ of neuron j increases by $\frac{\beta}{\tau_{a,j}}$ each time neuron j spikes and decays back to the baseline threshold b_j^0 with the time constant $\tau_{a,j}$. $z_j(t)$ represents the binary spike train of the neuron j. $B_j(t)z_j(t)\delta t$ acts as a reset of the membrane voltage $V_j(t)$ after each spike.

4.2 Implementation on Loihi

In order to use an LSNN model on Loihi an implementation of the adapting LIF neuron architecture described in Eq. (4.3) and Eq. (4.4) is needed. Loihi does not provide an option to arbitrarily change the threshold of a compartment during runtime. The chip offers a homeostasis feature, which actually changes the threshold depending on the spike activity, but this feature is sadly not very useful for the purpose of achieving an adapting neuron. The homeostasis feature is mainly tailored to rate-based approaches and keeps the spiking activity between lower and upper bounds. It increases the threshold if the spiking frequency exceeds an upper limit and decreases the threshold if the activity falls below the lower bound. For an event-based model with low spike activities of the neurons, the homeostasis feature is not applicable. During the execution Loihi does also not allow arbitrary or conditionally changes of other state values, such as the membrane potential, except using SNIPs. As each adaptive neuron would have to be evaluated within a SNIP, which is costly, this option was not feasible. Therefore a different approach to achieve an adapting behavior of the threshold for the configured neurons had to be developed.

One feature of Loihi is the multi-compartment neuron. Compartments can be organized in a binary dendritic tree structure in which the root compartment is treated as somatic compartment. All other compartments of such a multi-compartment neuron are denoted as dendritic compartments. The individual compartments in this tree integrate incoming spikes as well as input from their leaf compartments and in turn pass their own activation towards the root compartment. To be specific a somatic compartment can add the membrane potential of a corresponding dendritic compartment to its own membrane potential at

every time step. The threshold of the neuron cannot be changed, but with a well-configured mutli-compartment neuron it is possible to change the membrane potential and therefore change the effective threshold margin. This feature is the key to configure an adaptive LIF neuron on Loihi.

4.2.1 Three compartment neuron model

An exact implementation of the adaptive LIF neuron from [17] on Loihi can be achieved with a three compartment neuron model as seen in Figure 4.1. The main neuron gets the input spikes and handles the regular membrane potential dynamics, but cannot generate spikes. The output neuron adds the membrane potential of the main neuron and the membrane potential of the decay neuron into its own membrane potential at every time step and spikes if the threshold is reached. The main neuron and the output neuron represent a regular LIF neuron. The decay neuron now represents the threshold adaptation. Every time the output neuron generates a spike, the decay neuron gets it also as input spike with a negative weight of the magnitude $\frac{\beta}{\tau_a}$. Additionally the decay of the membrane potential of the decay neuron has the time constant τ_a which is usually larger than the time constant τ_v of the decay of the main neuron. Therefore the decay neuron holds a negative membrane potential which is equal to the threshold change and this negative value is added to the regular membrane potential of the main neuron inside the output neuron at every time step. Thus, the effective threshold margin is increased every time the output neuron spikes.

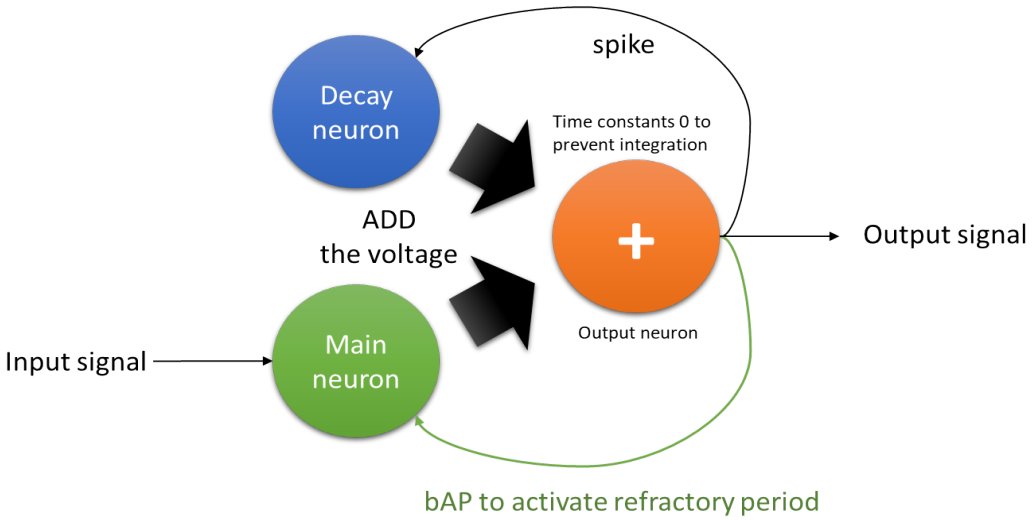


Figure 4.1 Illustration of the three compartment model to configure an adaptive LIF neuron on Loihi.

4.2.2 Two compartment neuron model

The major drawback of the three compartment neuron configuration for an adaptive LIF neuron is that it needs the memory of three neurons to represent one neuron. While this is no problem for small networks, it would not scale very well. Therefore a different more efficient model was developed, which needs only two compartments to configure an adaptive LIF neuron. The tradeoff is that it does not exactly resemble the adapting LIF model from [17] but has a very similar behavior. Figure 4.2 shows the concept of a two compartment neuron model to configure an adapting LIF neuron on Loihi.

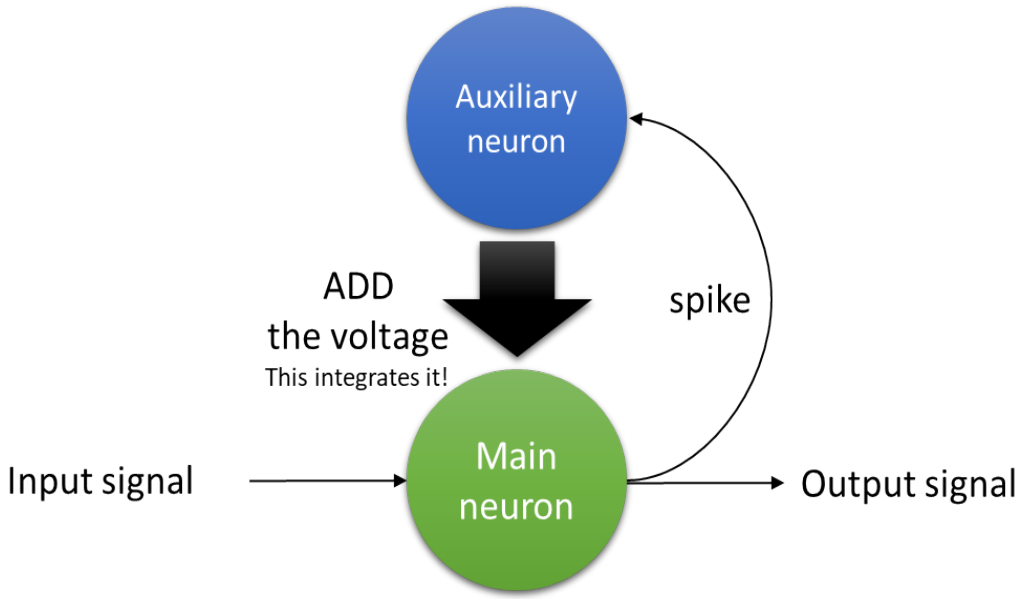


Figure 4.2 Illustration of the two compartment model to configure an adaptive LIF neuron on Loihi.

Two compartments are configured to imitate an adaptive threshold of a neuron. The main neuron is the regular input compartment, which receives external inputs and generates the output spikes. This compartment has a static threshold and elicit a spike if the membrane voltage is higher than this threshold. The second compartment is the auxiliary neuron which is responsible for keeping track of the adaptive threshold and its decay. If the main neuron spikes, the auxiliary neuron also gets an input spike using an inhibitory synapse which decreases its membrane potential. The time constant of the auxiliary neuron τ_a is usually much larger than the time constant τ_v of the main neuron. Therefore the membrane potential is decreasing slowly. The main neuron now adds the value of the membrane potential from the auxiliary neuron at every time step to its own membrane potential. This leads to an integration of the membrane potential of the auxiliary neuron in the main neuron and decreases its membrane potential. As the gap between the static threshold of the

main neuron and its membrane potential is increased by the auxiliary neuron, effectively the threshold is increased.

4.2.3 Analytical comparison

The two compartment concept for an adapting LIF neuron is different to the proposed model in [17]. Therefore an analytical comparison was made to quantity the differences and determine the inhibitory synaptic weight for the connection to the auxiliary compartment. It was assumed that the area under the curve of the adaptive threshold trace should be equal to stay close to the original model. The original model of [17] is described in Eq. (4.5) to Eq. (4.11).

$$\tau^u \dot{u} = -u + W s^{in} \longleftrightarrow u(t+1) = u(t)(1-\delta^u) + W s^{in} \tag{4.5}$$

$$\tau^v \dot{v} = -v + u \longleftrightarrow v(t+1) = v(t)(1-\delta^v) + u \tag{4.6}$$

$$s^{in} = \begin{cases} 1, & \text{if there is an input spike} \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

Eq. (4.5) and Eq. (4.6) describe the basic dynamics of a current-based leaky-integrate-and-fire neuron with one input synapse. The membrane potential or voltage v of a neuron is driven by the current u. Input spikes are described by the binary value s^{in} and W denotes the weight of the input synapse. The current as well as the membrane potential decay over time with their respective time constants τ^u and τ^v. The left side of $\longleftrightarrow$ shows the model in continuous time and the right side of $\longleftrightarrow$ denotes the discrete time equivalent.

$$v^{th} = v_0^{th} + \Delta v^{th} \tag{4.8}$$

$$\tau^{th} \Delta \dot{v}^{th} = -\Delta v^{th} + W_{th} s^{out} \longleftrightarrow \tau \Delta v^{th}(t+1) = \Delta v^{th}(t)(1-\delta^{th}) + W_{th} s^{out} \tag{4.9}$$

$$s^{out} = \begin{cases} 1, & \text{if } v \geq v^{th} \\ 0, & \text{otherwise} \end{cases} \tag{4.10}$$

Eq. (4.8) and Eq. (4.9) describe the dynamics of the threshold v^{th}. The baseline threshold v_0^{th} is increased by the adaptive threshold Δv^{th}, which varies depending on the spiking activity of the neuron. The adaptive threshold Δv^{th} is increased by the weight W_{th} every time the neuron generates a spike, i.e., the membrane potential v is greater or equal the threshold v^{th}. s^{out} is the binary information if the neuron generated a spike. Additionally, the adaptive threshold decays with a time constant τ^{th}.

$$\vartheta_A = v^{th} - v = v_0^{th} + \Delta v^{th} - v \tag{4.11}$$

The margin between the membrane potential v and the threshold value v^{th} is described by Eq. (4.11) and denoted by ϑ_A.

Eq. (4.12) to Eq. (4.18) describe the two compartment neuron model on Loihi.

$$\tau^u \dot{u}_1 = -u_1 + W_1 s_1 \longleftrightarrow u_1(t+1) = u_1(t)(1-\delta^u) + W_1 s_1 \tag{4.12}$$

$$\tau^{th} \dot{v}_2 = -v_1 + W_2 s_2 \longleftrightarrow v_2(t+1) = v_2(t)(1-\delta^u) + W_2 s_2 \tag{4.13}$$

$$\tau^v \dot{v}_1 = -v_1 + u_1 + v_2 \longleftrightarrow v_1(t+1) = v_1(t)(1-\delta^v) + u_1 + v_2 \tag{4.14}$$

$$s_1 = \begin{cases} 1, & \text{if there is an input spike} \\ 0, & \text{otherwise} \end{cases} \tag{4.15}$$

$$s_2 = \begin{cases} 1, & \text{if } v_1 \geq v^{th} \\ 0, & \text{otherwise} \end{cases} \tag{4.16}$$

Eq. (4.12) and Eq. (4.16) describe the basic dynamics of two compartment neuron with one input synapse on Loihi. The membrane potential of the main compartment v_1 is driven by the input current u_1 as well as the membrane potential of the auxiliary compartment v_2. Input spikes of the main compartment are described by the binary value s_1 and W_1 denotes the weight of the input synapse. The membrane potential of the auxiliary compartment v_2 is increased by the value of W_2 if the main compartment generated a spike, which is described by s_2. The current as well as the membrane potentials of the main and the auxiliary compartment decay over time with their respective time constants τ^u, τ^v and τ^{th}. Once more the model in continuous time is shown at the left side of $\longleftrightarrow$ and the at the right side of $\longleftrightarrow$ the model in discrete time is described.

$$v^{th} = v_0^{th} \tag{4.17}$$

Eq. (4.17) describes the dynamics of the threshold v^{th} which is equal to the baseline threshold v_0^{th}. The adaptive threshold is achieved through the dynamics of the main compartment and the auxiliary compartment.

$$\vartheta_B = v^{th} - v_1 = v_0^{th} - v_1 \tag{4.18}$$

The margin between the membrane potential v and the threshold value v^{th} is described by Eq. (4.18) and denoted by ϑ_B.

$$\vartheta_B = v_0^{th} v_1 \stackrel{!}{=} v_0^{th} + \Delta v^{th} - v = \vartheta_A \tag{4.19}$$

$$-v_1 \stackrel{!}{=} \Delta v^{th} - v \tag{4.20}$$

$$v(t) = \int_{\acute{t}}^{-\infty} e^{-\frac{\acute{t}}{\tau^v}} \sum_i (\acute{t} - t_i) W d\acute{t} \tag{4.21}$$

$$\sum_i (\acute{t} - t_i) \equiv s(t) \tag{4.22}$$

$$v_1(t) = \int_{\acute{t}}^{-\infty} e^{-\frac{\acute{t}}{\tau^v}} \left[\sum_i (\acute{t} - t_i) W_1 + \int_{\tilde{t}}^{-\infty} e^{-\frac{\tilde{t}}{\tau^{th}}} \sum_i (\tilde{t} - t_i) W_2 d\tilde{t} \right] d\acute{t} \tag{4.23}$$

In order to determine the difference between the original model and the Loihi model ϑ_A and ϑ_B are set equal as shown in Eq. (4.19) and substituted throughout Eq. (4.20) to Eq. (4.23). After transforming the equations into the Laplace space in Eq. (4.24) they can be solved and the difference between ϑ_A and ϑ_B is shown.

$$\begin{aligned}
V(s) &= A^v(s)S(s)W_1 \\
V_1(s) &= A^v(s)(S(s)W_1 + V_2(s)) \\
&= A^v(s)(S(s)W_1 + A^{th}(s) * S(s) * W_2) \\
&= A^v(s)S(s)W_1 + A^v(s)A^{th}(s)S(s)W_2 \\
\Delta V^{th}(s) &= A^{th}(s)S(s)W_{th} \\
\mathcal{L}(\vartheta_A - \vartheta_B) &= \mathcal{L}(\Delta v^{th}(t) - v(t) + v(t)) \stackrel{!}{=} 0 \\
&= \Delta V^{th}(s) - V(s) + V_1(s) = \\
&= A^{th}(s)S(s)W_{th} - A^v(s)S(s)W_1 + A^v(s)S(s)W_1 + A^v(s)A^{th}(s)S(s)W_2 \stackrel{!}{=} 0 \\
&A^{th}(s)W_{th} + A^v(s)A^{aux}(s)W_2 \stackrel{!}{=} 0 \\
&e^{-\frac{t}{\tau^{th}}}W_{th} + \int_0^t e^{-\frac{x}{\tau^v}} e^{-\frac{t-x}{\tau^{aux}}} dx W_2 \stackrel{!}{=} 0 \\
&e^{-\frac{t}{\tau^{th}}}W_{th} + e^{-\frac{t}{\tau^{aux}}} \int_0^t e^{-x(\frac{1}{\tau^v} - \frac{1}{\tau^{aux}})} dx W_2 \stackrel{!}{=} 0 \\
&\\
&\text{with } \gamma = \frac{1}{\tau^v} - \frac{1}{\tau^{aux}} \\
&e^{-\frac{t}{\tau^{th}}}W_{th} + e^{-\frac{t}{\tau^{aux}}}(-\frac{1}{\gamma}e^{-x\gamma}\Big|_0^t)W_2 \stackrel{!}{=} 0 \\
&e^{-\frac{t}{\tau^{th}}}W_{th} + e^{-\frac{t}{\tau^{aux}}}(-\frac{1}{\gamma}(e^{-t\gamma} - 1))W_2 \stackrel{!}{=} 0 \\
&e^{-\frac{t}{\tau^{th}}}W_{th} - \frac{W_2}{\gamma}(e^{-t(\gamma + \frac{1}{\tau^{aux}})} - e^{-\frac{t}{\tau^{aux}}}) \stackrel{!}{=} 0 \\
&e^{-\frac{t}{\tau^{th}}}W_{th} - \frac{W_2}{\gamma}e^{-\frac{t}{\tau^v}} - \frac{W_2}{\gamma}e^{-\frac{t}{\tau^{aux}}} \stackrel{!}{=} 0
\end{aligned} \tag{4.24}$$

The assumption is that τ^v is small compared to τ^{aux} and therefore $e^{-\frac{t}{\tau^v}} \approx 0$, which simplifies Eq. (4.24) to Eq. (4.25):

$$W_2 = -W_{th}\gamma \text{ if } \tau^{aux} \stackrel{!}{=} \tau^{th} \tag{4.25}$$

where W_2 denotes the weight of the auxiliary synapse i.e. the decrease of the membrane voltage after the main neuron spiked and W_{th} denotes the weight of the adaptive threshold change, i.e. the increase of the threshold after a spike. Under these assumptions the models

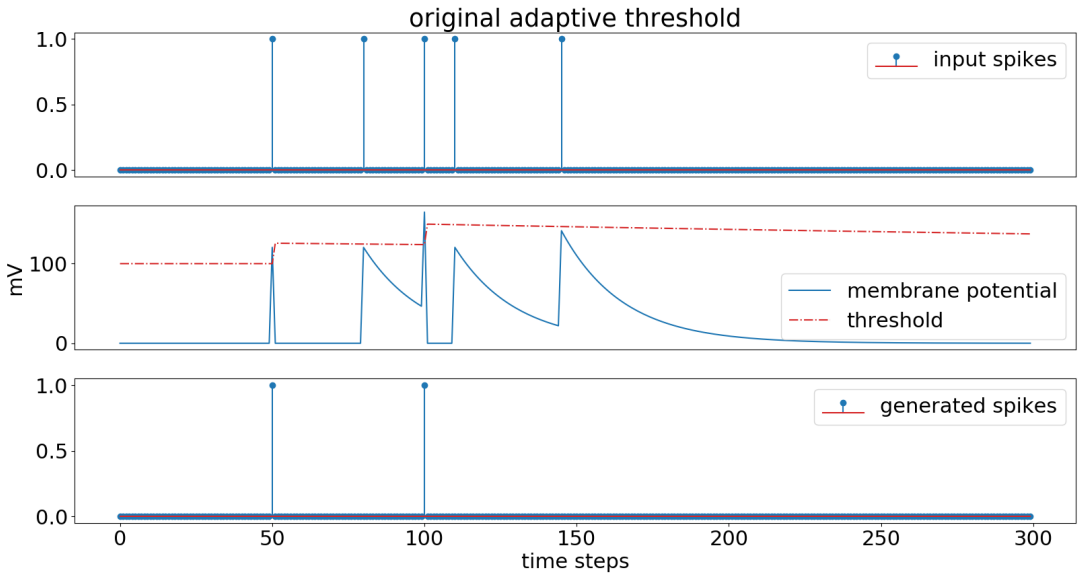
behave similar if the weight W_2 is chosen according to Eq. (4.25).

Figure 4.3 shows the original adaptive threshold from [17] and the concept of the two compartment model on Loihi. The weight parameter W_2 was set according to Eq. (4.25). It can be seen that the dynamics are clearly different, but the principle stays the same. After a neuron generated a spike, the threshold value is increased for the neuron over a given time period. As previously mentioned the threshold on the chip cannot be changed during runtime, thus the membrane potential needs to be modified accordingly. The dynamics of the implemented version on Loihi can be seen in Figure 4.4. This two compartment implementation of the adaptive LIF neuron was chosen to be used for developing an LSNN module on the Loihi chip.

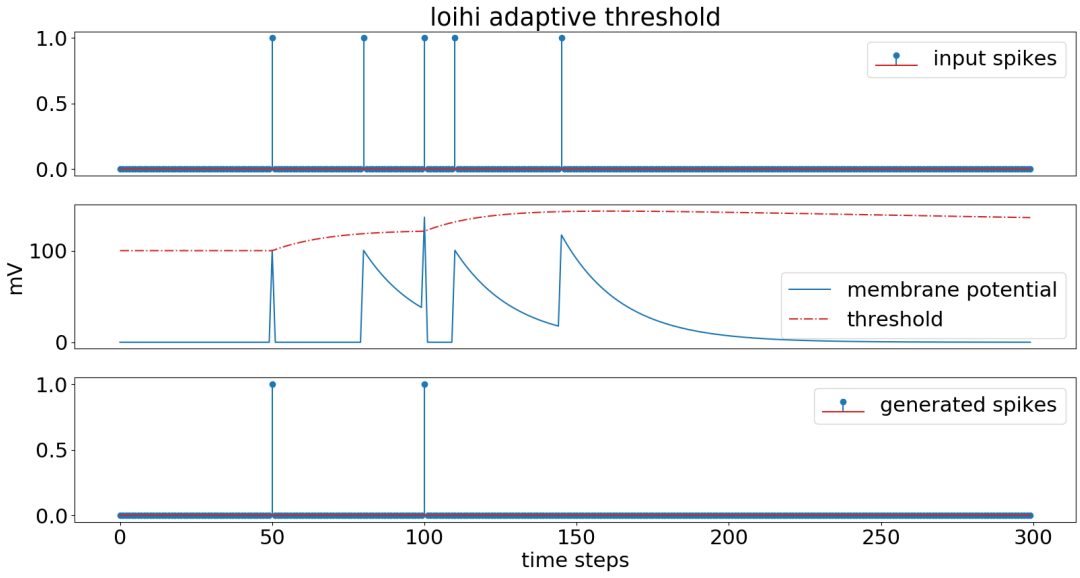
4.3 LSNN module

An LSNN module has been implemented on the Loihi chip in order to easily configure and use LSNNs. The module consists of python classes and is available on the NxCore level as well as the NxNet level. Originally the module was developed with the NxCore API and on this level it manually configures the registers according to certain network parameters, i.e. number of input, regular LIF, adaptive LIF and output neurons as well as their connection matrices. The main registers to configure are the compartment register for the neurons itself as well as the synapse and axon registers for their connections. It is important that the two compartments of the two compartment neuron model for the adaptive LIF neurons need to be next to each other in the compartment register. These details are encapsulated through a convenient interface, which just needs the network parameters and additional dynamic parameters, e.g., the time constants and thresholds of the neurons. In the NxNet version of the module the interface is the same, but the register configuration is done automatically by using NxNet based methods. For example, when configuring a multi-compartment neuron NxNet takes care of the correct placement of the compartments in the compartment register.

Listing 4.1 shows the method of the LSNN module to create an LSNN. The LSNN module constructs such a network, consisting of an input layer, a layer of recurrently connected regular and adaptive neurons as well as a linear output layer. The input layer receives input from a spike generator and the solution is read out from the activity of the output layer neurons using probes. The user is expected to provide both a stimulus and a set of input, recurrent and output weight matrices in order to use the module. Listing 4.2 demonstrates the configuration and execution of the network module, which configures an LSNN on Loihi.



(a)



(b)

Figure 4.3 The red dotted line in (a) shows the original adaptive threshold and the red dotted line in (b) shows the effective adaptive threshold of the two compartment model.

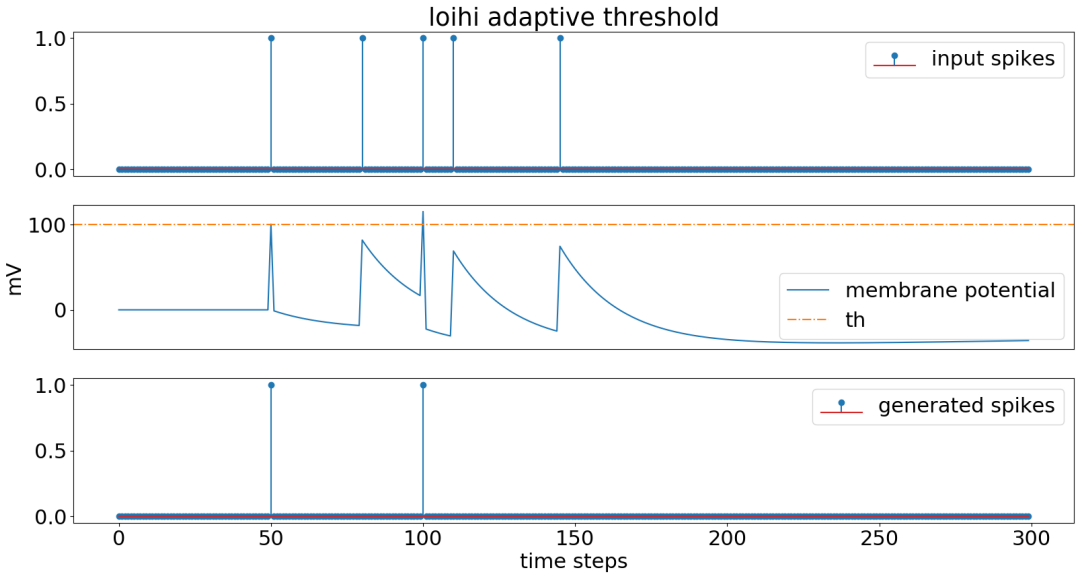


Figure 4.4 Dynamics of the implemented two compartment adaptive LIF neuron on Loihi. The threshold value cannot be changed, but the membrane potential can be decreased accordingly. The dynamics are equivalent to (b) in Figure 4.3

```
def createLSNN(numInput, numRegular, numAdaptive, numOutput,
    beta, tau_u, tau_v, tau_a, thr, refrac=0):
    """
    Creates a lsnn object.

    :param numInput:        Number of input neurons.
    :param numRegular:      Number of regular neurons.
    :param numAdaptive:     Number of adaptive neurons.
    :param numOutput:       Number of output neurons.
    :param beta:            Scaling constant of adaptive
       threshold.
    :param tau_u:           Membrane time constant current.
    :param tau_v:           Membrane time constant voltage.
    :param tau_a:           Membrane time constant of
       adaptive threshold.
    :param thr:             Baseline threshold value.
    :param refrac:          Refractory time delay.
    :return                 An LSNN object
    """
```

Listing 4.1 Description of the LSNN module.

```
## Setup LSNN
# Define layer sizes
numOutput = 2

# Create LSSN network
net = lsnn.createLSNN(numInput=2,
                      numRegular=3,
                      numAdaptive=4,
                      numOutput=numOutput,
                      beta=1.8,
                      tau_u=0,
                      tau_v=20,
                      tau_a=700,
                      thr=50*2**6)

## Define network weights and construct network
# Define input, recurrent and output weights
wIn = np.array(...)
wRec = np.array(...)
wOut = np.array(...)

# Load weights and construct network
net.loadWeights(wIn, wRec, wOut)
net.constructNetwork()

## Define spike generator
net.spikeGenerator(1, spiketimes=[10, 40])

## Activate network probes
for n in net.outputNodes:
    n.monitor.activateAllProbes()

## Run LSNN
numSteps = 500
net.run(numSteps)
net.disconnect()

## Visualize network activity
for i in range(0, numOutput):
   net.outputNodes[i].plot()
```

Listing 4.2 A small example how the LSNN module could be used.

5 TensorFlow simulation of LSNN-Loihi

In order to train an LSNN model running on Loihi a TensorFlow simulation was created. Loihi does not efficiently support supervised learning methods like BPTT. Changing the weights of the synapses according to some gradient would mean to reload the whole network on the chip after every iteration. Therefore it is not beneficial right now to use the chip for training with BPTT. TensorFlow on the other hand offers state of the art machine learning tools and methods, is well documented and adaptable. Furthermore the original LSNN model from [17] is implemented in TensorFlow and was used as foundation. So the TensorFlow framework was a good choice to implement a Loihi based LSNN model, which can be trained with BPTT. The trained network then could be transferred onto Loihi to run inference.

The TensorFlow LSNN module was modified to take care of the specific characteristics of the Loihi chip. Especially the dynamics of the neurons i.e. Eq. (3.1) and Eq. (3.2), the limited and discretized weight values, the two compartment model for the adaptive threshold, the overflow behavior of the dendritic accumulator and the specific precision of u and v were implemented. Additionally the training algorithm was adapted to use the finite precision Loihi model in the forward pass and the full precision model for the backward pass. After each training update the full precision model was rounded to the finite precision model to calculate the loss for the update. Figure 5.1 shows the process flow of the forward and backward pass during training.

5.1 Implementation

The dynamics of the neurons were implemented exactly as described in Eq. (3.1) and Eq. (3.2). The signs of the values were stored first, then the absolute values were cast and rounded to 64 bit integers and multiplied before the bit shift operation was applied. Then the values were cast to 32 bit floats again. As the TensorFlow cast and bit shift operation do not have gradients, a custom gradient for this calculation of the decay was implemented, being the gradient of a normal exponential decay with the same values and time constants.

Another important change was the precision of the weight matrix, i.e., the synaptic weight values. The maximum precision on the Loihi chip for the synaptic weights are 8 bits with fixed sign. Therefore the range is from -255 to 255. In comparison the original model uses a standard Gaussian weight distribution with mean 0 and standard deviation 1

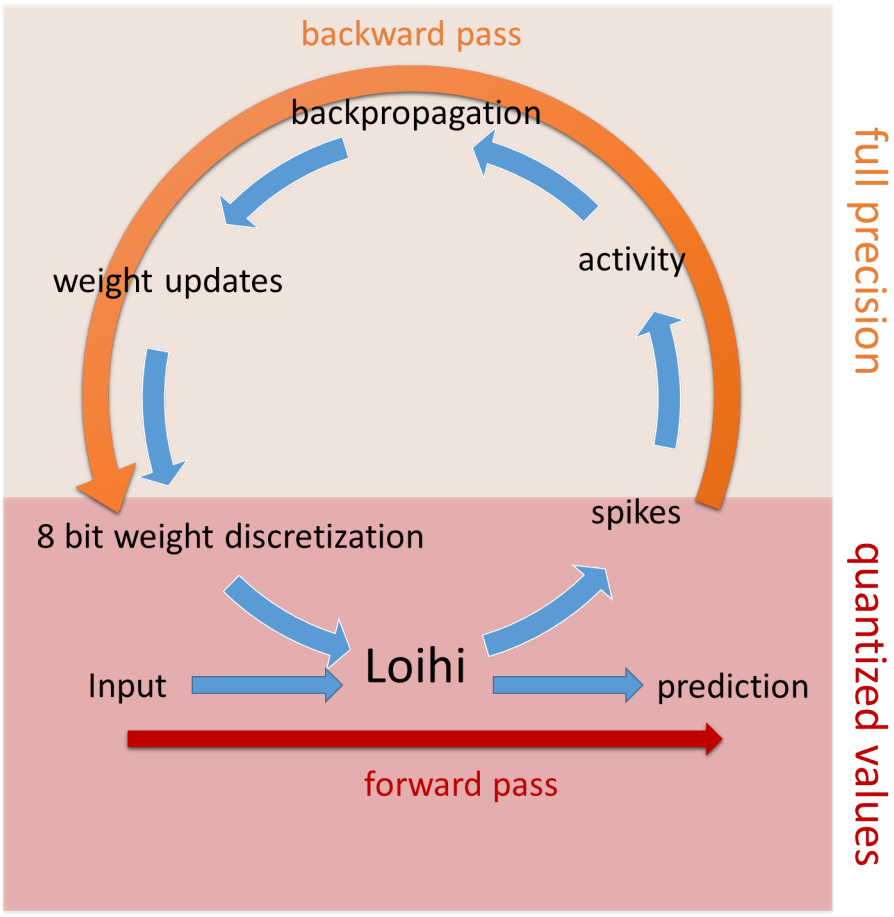


Figure 5.1 Process flow of the forward and backward pass during training. The forward path uses quantized values suitable for Loihi and the backward pass uses full precision values to determine the weight updates.

as initial values, with no further restrictions during updates. As the optimizers also rely on this distributions to calculate the updates, the backward pass kept the full precision. For the forward pass and especially the calculation of the loss the weights were quantized between -255 and 255. A simple ceil rounding as well as the option of stochastic rounding were implemented. Again a custom gradient was assigned to this operation, being the unchanged gradient (without this custom gradient, the gradient would become zero).

In TensorFlow the two compartment model was implemented by adding an additional state variable representing the second compartment to the neural cell model. The dynamics of the auxiliary state variable followed the description of the two compartment model as seen in the previous chapter 4.2.2. During the update of the membrane potential this additional negative value was added at every time step, if present.

Loihi showed an unexpected behavior if many input spikes to one neuron were present. The expectation was that there would be a limit to the maximum amount of input current in one time step, but this was not observed. Instead an overflow behavior was found. The dendritic accumulator had a finite precision of signed 16 bit, i.e., the maximum change of the current u per time step is 2^{15}. If this value was exceeded it got negative, it behaved like a signed integer. The exact dynamic of the dendritic accumulator is shown in Eq. (5.1):

$$u(t) = \left\{ \left[u(t-1) + 2^{n-1} + u_{input}(t) \right] \bmod 2^n \right\} - 2^{n-1}, \tag{5.1}$$

where $u(t)$ denotes the accumulated current of the dendrite which will be used in the neuronal dynamics and $u_{input}(t)$ denotes the the input current from all input spikes and their respective weights at time step t. The bit precision is described by n and is 16 bit for the dendritic accumulator on Loihi.

The finite precision of the current u and the membrane potential v was also implemented. These state variables have signed 24 bit precision on Loihi i.e. a value range between $-(2^{23}-1)$ to $(2^{23}-1)$. In the TensorFlow model this was achieved by clipping the values at these limits. In the end an exact representation of the behavior on the Loihi chip in the TensorFlow simulation was achieved. Thus, this model could be used to train a specific LSNN example with BPTT and convenient optimizers that TensorFlow offers on arbitrary hardware as well as take the trained parameters, load them on the chip and execute the network on Loihi.

5.2 Remarks to the adaptions

Some of the changes impacted the performance of the original model if the hyper-parameters were chosen poorly. The quantization of the synaptic weights was a major change and it was important to think about the consequences. For example, if the threshold of the neurons in the original model was chosen to be somewhere between 0 and 1, it would be very likely that, after rounding to the finite precision Loihi weights, the threshold would be in the middle of 0 and 255. This meant essentially that every weight value above the threshold was wasted, as it would behave exactly the same as any other weight above the threshold. Therefore the threshold should be chosen to resemble the highest weight of the quantized range i.e. 255, in order to not waste valuable precision. If the threshold would be chosen relative to other values, everything could be scaled accordingly to keep similar behavior. For the LSNN model in particular the threshold value and the beta hyper-parameter needed to be treated the same way, when working hyper-parameters of the original model were converted to the Loihi model.

Another remark concerns the refractory behavior of Loihi compared to the original LSNN model. In the original LSNN model the membrane potential was also able to change

due to incoming spikes during the refractory period. This was not possible on Loihi, as the membrane potential was fixed during a refractory period. Although this was not a problem to implement in the TensorFlow model, it certainly affected the performance of it. Therefore the refractory period was changed to one time step, i.e., no refractory period, for the experiments.

6 Application of LSNN-Loihi on sequential MNIST

The performance of LSNNs on Loihi was tested on a standard benchmark task. Sequential MNIST [18, 19] was chosen as task, because it requires the short term memory of the cells over a long time span in order to reach a high classification rate. It was also chosen to compare the performance of LSNN-Loihi with the original LSNN results from [17]. The model was trained off chip with TensorFlow and BPTT using GPUs or Xeon CPUs. After training, the learned weights were used to configure the LSNN network on Loihi and run inference on the chip to measure the classification rate.

6.1 Sequential MNIST

MNIST [40] refers to a database of handwritten digits from zero to nine. It has a training set of 60000 examples and a test set of 10000 examples from approximately 250 writers. Each digit is mapped on an image with 28x28 pixels and 256 gray levels. Some examples can be seen in Figure 6.1. Sequential MNIST refers to classification task where the pixels of the handwritten digits are presented sequentially, pixel by pixel. As an MNIST image has 784 pixels, this means each digit takes 784 time steps to process.

6.2 Input encoding

The gray values of the pixels from an MNIST image were encoded in spikes. 80 input neurons were used and each of them was associated with a particular threshold for the gray value. So there were 79 linear spaced thresholds between 0 and 256. Every second threshold refereed to an increasing gray value, while the others refereed to a decreasing gray value. If the gray value changes when switching from one pixel to the next and the gray value increases, every second input neuron from the last threshold to the next threshold generates a spike. This principle can be seen in (a) and (b) of Figure 6.2. The pixels marked red in (a) are the indicator for (b). The transition is four times from lower gray to higher gray values, which correspond to the steps in (b) and after that a two time transition to lower gray values. Furthermore the last input neuron becomes active after the presentation of all 784 pixels for 56 time steps, thus the presentation of one image takes 840 time steps. This last input neuron which generates a spike at every time step after the image presentation indicates to the network, that the image has ended and an output

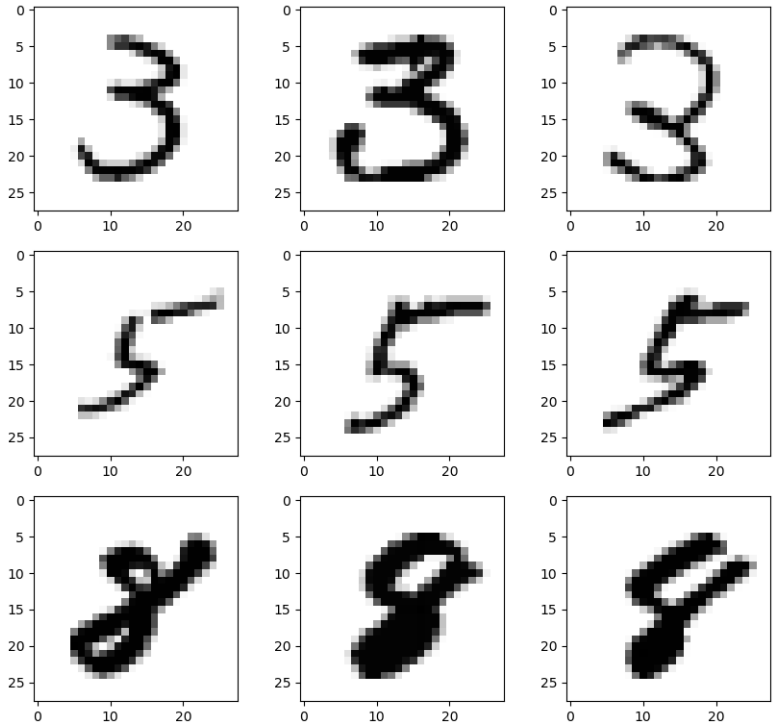
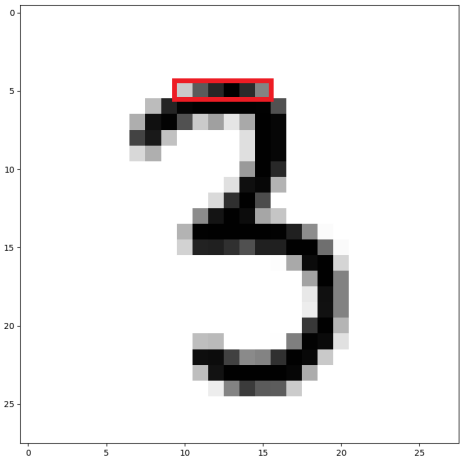


Figure 6.1 Sample digits of 3, 5 and 8 from the MNIST handwritten digit database.

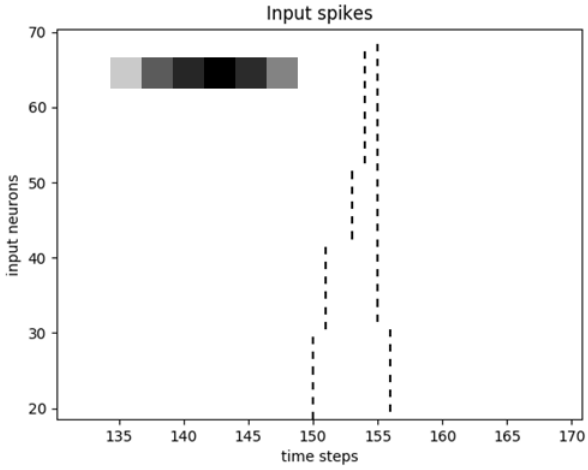
classification should happen. The firing of this special input neuron is shown at the top right of Figure 6.2c. Figure 6.2c also shows the complete input encoding of the MNIST picture from Figure 6.2a.

6.3 Network

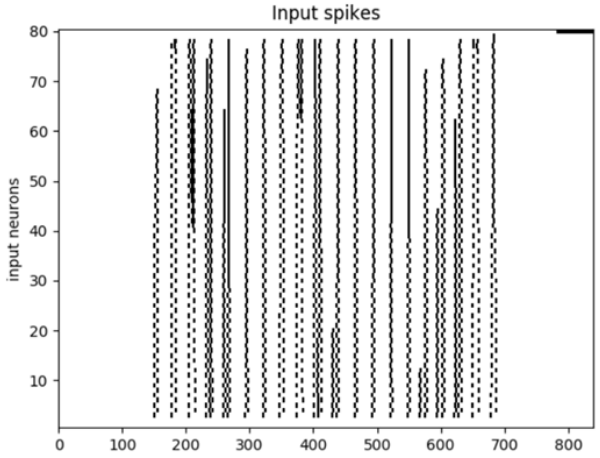
The chosen network structure and size was inspired from [17]. Figure 6.3 shows the structure of the used network. The regular spiking neurons were divided into excitatory neurons (75%) and inhibitory neurons (25%). All of the adaptive neurons were excitatory neurons. 140 regular spiking neurons and 100 adaptive neurons were chosen for the recurrent network. 80 input neurons were chosen to perform an input spike encoding of the images, and 10 output neurons were chosen corresponding to the 10 classes of the data. As the network was also trained using DEEP R [6], an overall connectivity of 20% was chosen. The detailed hyper-parameters which were used to train the Loihi LSNN can be



(a)



(b)



(c)

Figure 6.2 (a) handwritten MNIST digit, (b) the principle of exceeding and descending thresholds for the spike encoding and (c) the complete spike encoded MNIST digit.

seen in Table 6.1

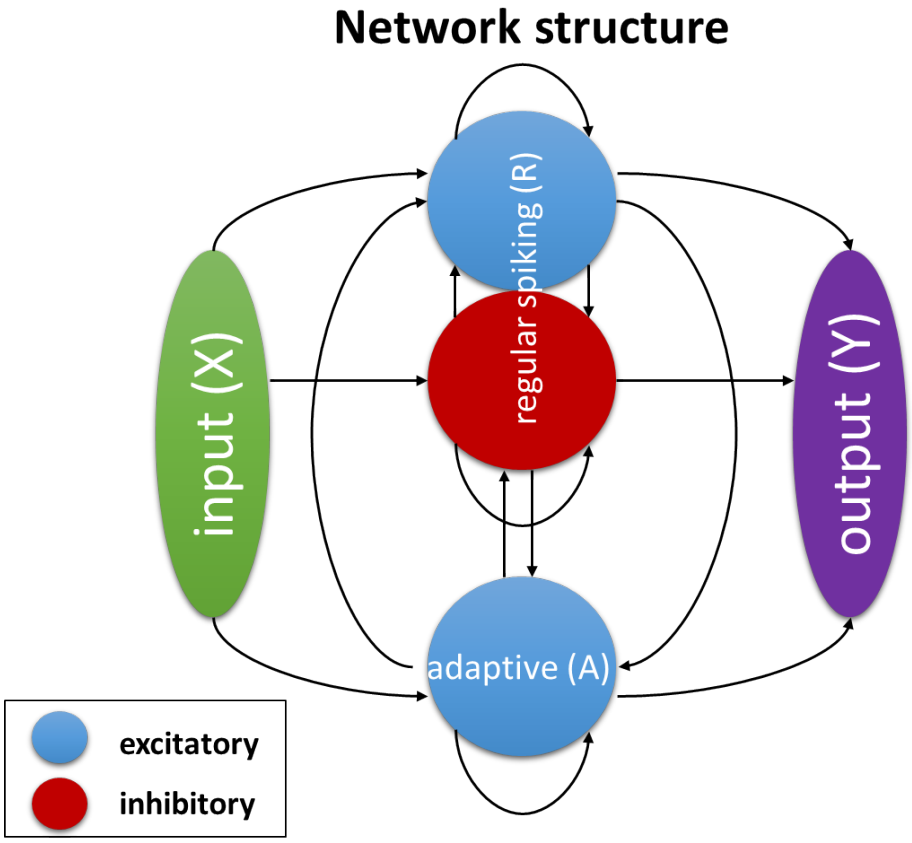


Figure 6.3 Structure of the Loihi LSNN network. The arrows show the recurrent connections between the different groups of neurons. The regular spiking neurons R consisted of 105 excitatory neurons and 35 inhibitory neurons. The adaptive neurons A were 100 excitatory neurons. The input population X and the output population Y consisted of excitatory and inhibitory neurons, depending on the training outcome of the network.

To compare the Loihi LSNN with the original model, a millisecond on the original model corresponds to one time step on Loihi. A more complex calculation was needed to get the corresponding beta and baseline threshold values for Loihi. This calculation is shown in Eq. (6.1):

$$p^{loihi} = p^{orig} \cdot 2^6 \cdot \frac{1}{1 - \exp(-\frac{1}{\tau_m})} \cdot \zeta, \tag{6.1}$$

where p^{loihi} stands for a hyper-parameter of the Loihi model, e.g., the beta value β or the baseline threshold value b^0. The corresponding hyper-parameter of the original model

Table 6.1 Hyper-parameters of the original LSNN model and the corresponding hyper-parameters on Loihi.

	original LSNN	Loihi LSNN
β	1.8	1390000
baseline threshold b^0	0.01	8128
refractory steps	5 ms	1
number of delays	10	1
τ_m	20 ms	20 time steps
τ_a	700 ms	700 time steps
scaling constant ζ	-	620

is denoted by p^{orig}. τ_m describes the time constant of the membrane potential and ζ is a scaling constant. First, every threshold on Loihi got multiplied by 2^6, as the input current got also multiplied by this factor Eq. (3.1). The next factor for the calculation was needed because a factor for the input current of the original model was removed. The factor was $1 - \exp(-\frac{1}{\tau_m})$. The last factor was the scaling constant ζ, which was used for scaling the threshold so it corresponds to a certain synaptic weight on Loihi. It should be remembered that Loihi has finite precision and therefore an exact adoption was not possible, e.g. the beta value did not follow exactly Eq. (6.1).

6.4 Results

The classification was performed at the last time step i.e. time step 840 of an image presentation. Each output neuron denoted a digit. The output neuron which had the highest membrane potential on this last time step defined the predicted class of the image. The cross entropy error between the softmax of the output neuron potentials and the target label was minimized, in order to train the network. ADAM with an initial learning rate of 0.01 was used as optimizer for the training. The learning rate was decayed every 2500 iterations by a factor of 0.8.

Table 6.2 shows the results for the sequential MNIST task on Loihi. Different mini-batch sizes and rounding options for the forward/backward pass weight rounding were tried during training. Figure 6.4 show a comparison of the classification accuracy for different neural network models.

Table 6.2 Results on the sequential MNIST task on Loihi.

mini-batch size	iterations	rounding	# runs	mean	std	max
32	100000	ceil	2	76.04%	6.10%	82.09%
512	100000	ceil	2	92.24%	0.25%	92.49%
1024	100000	ceil	3	92.74%	0.26%	93.10%
1024	100000	stochastic	2	84.89%	1.47%	86.35%
1500	35800	ceil	1	-	-	94.07%
1500	60000	ceil	1	-	-	93.42%
1500	100000	ceil	2	93.59%	0.38%	93.97%
2048	100000	ceil	2	92.05%	1.33%	93.38%

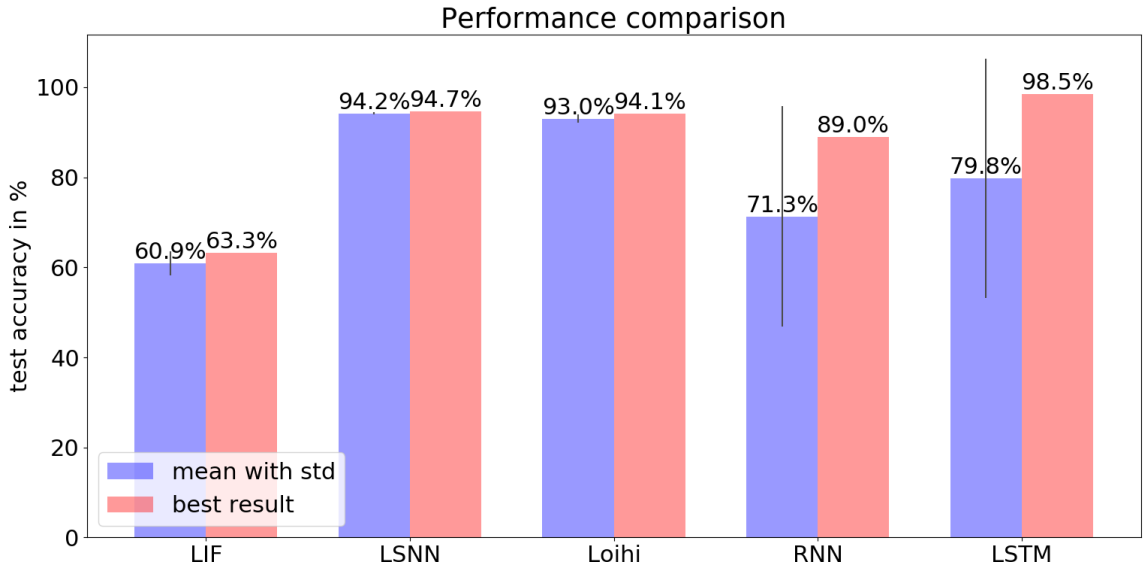


Figure 6.4 Comparison of the classification results for different models of neural networks. LIF denotes a normal SNN and LSNN denotes to the original model from [17]. Loihi shows the results of this thesis, albeit only the results from Table 6.2 with a batch size greater or equal to 1024 are considered for this comparison. To compare the results with ANNs, the classification accuracies of a fully connected RNN and a LSTM network are also shown.

7 Discussion

The results show that a neuromorphic hardware implementation of LSNNs on Loihi is feasible and yield similar results as a software implementation. Furthermore the results show that the limitations like quantization due to lower precision and memory restrictions of the Loihi neuromorphic hardware chip can be overcome. The main performance difference between the the Loihi LSNN and the original LSNN is that the Loihi LSNN needs around three times the training time to achieve similar classification rate. Although Table 6.2 shows one experiment which actually yielded one of the highest achieved classification rates on just 35800 iterations. This is on par with the results of [17] on the same training duration. Nevertheless, most experiments needed the longer training duration to achieve similar performance. This might be due to the quantization of the weights and therefore less options to get to good performing parameters in combination with the hard boundaries of these parameters. It could also be that for this specific implementation different hyperparameters would improve duration of the training.

Additionally the results of Table 6.2 suggest that a bigger mini-batch size is beneficial for the Loihi LSNN. This could also be associated with the decay of the learning rate in combination with the longer training duration. As a bigger mini-batch presumably yields a more accurate gradient, it might be needed to get faster near a local optima before the learning rate gets too low. Thus, finding an optimal ratio between batch size and learning rate could improve the training duration. A more general approach of this was actually investigated in [41]. Although one has to be careful to not forget the idea of stochastic gradient descent and that it can actually take advantage of small mini-batches.

Stochastic rounding was also tried in order to improve performance. As seen in Table 6.2 it did not really do well. It was surprising that the stochastic rounding approach did not reach the performance of the simple ceiling rounding approach. [42] suggested that stochastic rounding might be beneficial when dealing with low-precession fixed-point computations, but it could not be observed during this experiments. It might need an even longer training duration to improve the classification rate.

The average neuron firing rate in [17] was centered around 20 Hz. Due to the removal of the refractory period in the Loihi LSNN this firing rate increased to around 50 Hz. A higher regularization factor, which was introduced in [17] could be used to decrease the average firing rate. This was not investigated extensively during the experiments, but it seemed that a higher firing rate was beneficial during training. It might be interesting to introduce a

growth factor for the regularization of the spiking activity and increase it during training in a similar manner as the learning rate.

It also has to be noted that the classification was performed differently. In the original model the values of the output neurons over the last 56 time steps, so the values during the output cue, were averaged before the highest value and therefore the class label was determined. In the Loihi LSNN only the value at the last time step was used. Although the model should be able to learn it just as well, it still could impact the performance. The trainable bias of the output neurons of the original model was also omitted, due to Loihi not having an option to directly add an output bias on chip. Once again, the model should not suffer from this in theory, but it might be worth to investigate the impact of it. Especially, if more complex tasks should be performed.

The results from Table 6.2 all used the hyper-parameters from Table 6.1. There the scaling constant ζ actually corresponds to a weight of 127 according to 6.1, which means that 128 quantization steps were wasted for this experiments. At first this was due to too high scaling, which yielded in saturation of the membrane potentials of the output neurons. Therefore a smaller value ζ was used which did not saturate the output neurons and therefore performed better. The mistake of the scaling was fixed at some point, but the ζ value was not changed. As the experiments took several days on a GPU or even weeks on a Xeon CPU a complete restart would have exceeded the time period for this thesis. One experiment, which used the full range was performed and it did not yield better results. At the end, the Loihi LSNN model nearly reached the performance of the original model.

8 Conclusion

In this thesis a novel neuromorphic hardware architecture was investigated and an inventive software model, the LSNN, was implemented on the actual hardware chip Loihi. It was shown how certain features of Loihi could be used to overcome architectural limitations without impairing the performance too much. Therefore, a multi-compartment neuron model on Loihi was developed, which was included in an LSNN module for Loihi. Furthermore, a TensorFlow simulation of Loihi LSNN was implemented to be able to train the LSNN module on conventional hardware only and afterwards the trained parameters were ported on the chip to run the model. Finally, the developed modules were used to perform a classification task. Sequential MNIST, a standard benchmark task, was used as application to assess the performance of the Loihi LSNN. The performance was compared to the original LSNN model and the Loihi LSNN reached similar classification accuracies. Importantly, this showed that LSNN can be implemented on the neuromorphic hardware chip Loihi without losing its essential and performance optimizing properties.

This means that there is a good chance that other applications of the LSNN architecture can be also configured and executed on Loihi. [17] showed that LSNNs are capable of classifying the TIMIT data set. Therefore LSNNs on Loihi might prove useful for speech processing, as this is a perfect application field for spiking neural networks, as well as small, energy efficient hardware chips. Imagine a smart device, like a mobile phone, which does not need internet access for their virtual assistants, but only an energy efficient neuromorphic chip.

Bibliography

[1] Josh Merel, Yuval Tassa, Dhruva Tb, Sriram Srinivasan, Jay Lemmon, Ziyu Wang, Greg Wayne, and Nicolas Heess. Learning human behaviors from motion capture by adversarial imitation. *arXiv*, Jul 2017.

[2] Dayong Wang, Aditya Khosla, Rishab Gargeya, Humayun Irshad, and Andrew H. Beck. Deep Learning for Identifying Metastatic Breast Cancer. *arXiv*, Jun 2016.

[3] Yigit Demir, Yan Pan, Seukwoo Song, Nikos Hardavellas, John Kim, and Gokhan Memik. *Galaxy: a high-performance energy-efficient multi-chip architecture using photonic interconnects*. ACM, Jun 2014.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, Jun 2015.

[5] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv*, Jun 2017.

[6] Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert A. Legenstein. Deep rewiring: Training very sparse deep networks. *CoRR*, abs/1711.05136, 2017.

[7] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable Bounds for Learning Some Deep Representations. *PMLR*, pages 584–592, Jan 2014.

[8] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, Dec 1997.

[9] Dharmendra S Modha. Introducing a brain-inspired computer. 2008.

[10] Steve Furber and Steve Temple. Neural systems engineering. *J. R. Soc. Interface*, 4(13):193–206, Apr 2007.

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

[12] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34(10):1537–1557, Oct 2015.

[13] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, 38(1):82–99, Jan 2018.

[14] Steve K. Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S. Modha. Backpropagation for Energy-Efficient Neuromorphic Computing, 2015. [Online; accessed 23. Oct. 2018].

[15] Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, et al. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proc. Natl. Acad. Sci. U.S.A.*, 113(41):11441–11446, Oct 2016.

[16] Peter U. Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Jul 2015.

[17] Guillaume Bellec, Darjan Salaj, Anand Subramoney, Robert Legenstein, and Wolfgang Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. *arXiv*, Mar 2018.

[18] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *CoRR*, abs/1504.00941, 2015.

[19] Rui Costa, Ioannis Alexandros Assael, Brendan Shillingford, Nando de Freitas, and TIm Vogels. Cortical microcircuits as gated-recurrent neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 272–283. Curran Associates, Inc., 2017.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(9):1735–1780, Nov 1997.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia,

Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

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

[23] M. Foster and C.S Sherrington. A textbook of physiology. With C.S. Sherrington. Part 3. The central nervous system, 1897.

[24] R.L. Rapport. *Nerve Endings: The Discovery of the Synapse*. W.W. Norton, 2005.

[25] Christof Koch and Idan Segev. *Methods in neuronal modeling : from ions to networks*. Cambridge, Massachusetts: MIT Press, 2 edition, 1999.

[26] L.F Abbott. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin*, 50(5):303 – 304, 1999.

[27] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533 EP, Oct 1986.

[28] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[29] Yoshua Bengio, Dong-Hyun Lee, Jörg Bornschein, and Zhouhan Lin. Towards biologically plausible deep learning. *CoRR*, abs/1502.04156, 2015.

[30] Carver Mead. *Analog VLSI and Neural Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[31] C. Lin, A. Wild, G. N. Chinya, Y. Cao, M. Davies, D. M. Lavery, and H. Wang. Programming spiking neural networks on intel's loihi. *Computer*, 51(3):52–61, March 2018.

[32] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in python. *Frontiers in Neuroinformatics*, 2:5, 2008.

[33] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. Pynn: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2009.

[34] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[35] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, May 2013.

[36] Werner Vogels. Bringing the magic of amazon ai and alexa to apps on aws. `https://www.allthingsdistributed.com/2016/11/amazon-ai-and-alexa-for-all-aws-apps.html`, 2016. Accessed: 2018-11-18.

[37] Pranav Khaitan. Chat smarter with allo. `https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html`, 2016. Accessed: 2018-11-18.

[38] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[39] OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018. Accessed: 2018-11-18.

[40] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[41] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[42] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.