Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Masterarbeit Informatik

Biologically Inspired Action Inference with Recurrent Spiking Forward Models

Manuel Traub

11. November 2019

Erstgutachter

Prof. Dr. Martin Butz
Kognitive Modellierung
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Zweitgutachter

Prof. Dr. Harald Baayen
Quantitative Linguistik
Seminar für Sprachwissenschaft
Universität Tübingen

Betreuer

Dr. Sebastian Otte
Kognitive Modellierung
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Traub, Manuel:
Biologically Inspired Action Inference with Recurrent Spiking Forward Models
Masterarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 15.05.2019 – 15.11.2019

Abstract

Artificial Intelligence (AI) has progressed to a point where intelligent systems already outperform humans on several different tasks, ranging from object recognition to complex board and video games. Still, we are only beginning to understand how learning in the mammalian brain works. Specifically how the brain solves the Temporal Credit Assignment problem, is poorly understood and probably significantly differs from Back-Propagation Through Time (BPTT), which is the standard way Recurrent Neural Networks (RNNs) are trained. Currently high performing RNNs like Long Short-Term Memories (LSTMs) are also a rough simplification of real Neural Networks. A more biologically plausible approximation can be achieved by Spiking Neural Networks (SNNs), which constitute the next generation of Artificial Neural Networks (ANNs). This thesis explores the application of a novel training algorithms called e-pop that is inspired by synaptic eligibility traces, which represent the accumulation of local information in the brain. Together with BPTT, a new SNN architecture, called Long Short-Term SNN (LSNN) that for the first time reaches the performance of LSTMs, is challenged with the task of controlling a many-joint robotic arm through the application of action inference by learning an appropriate forward model of the robotic arm dynamics.

Kurzfassung

Künstliche Intelligenz hat bereits begonnen den Menschen in verschiedenen Bereichen zu übertreffen, von Objekterkennung über Brettspiele bis hin zu Computerspielen. Trotz dieser Fortschritte sind die Mechanismen des Lernens im Säugetiergehirn weiterhin ein aktives Forschungsfeld. Probleme wie die Zuweisung von Fehlerinformationen über lange Zeitspannen hinweg sind erst ansatzweise verstanden und unterscheiden sich vermutlich stark von den Trainingsmethoden künstlicher Rekurrenter Neuronaler Netze (RNNs) wie Back-Propagation Through Time (BPTT). Zudem sind verwendete RNNs wie Long Short-Term-Memorys (LSTMs) eine starke Vereinfachung biologischer Neuronaler Netze. Biologisch plausibler hingegen ist die neuste Generation Neuronaler Netze, sogenannte Gepulste Neuronale Netzte (SNNs), welche Gegenstand dieser Arbeit sind. Benutzt wird eine neuartige SNN-Architektur namens Long Short-Term SNN (LSNN), welche als erste SNN-Architektur in der Lage waren die Performance von LSTMs zu erreichen. Trainiert werden diese LSNNs mithilfe von BPTT so wie einem neuen Lern-Algorithmus namens e-prop, dessen Arbeitsweise von synaptischen Eligibility-Tracen abgeleitet ist. Diese Eligibility-Traces, welche eine lokale Akkumulation von Information über die vergangene Aktivität einer Synapse modellieren, bilden die Basis einer biologisch plausibleren Alternative zu BPTT. Untersucht wird, ob LSNNs mithilfe von BPTT und e-prop in der Lage sind einen mehrgelenkigen Roboter-Arm zu steuern. Erreicht wird dies durch das Erlernen eines Vorwärtsmodelles, welches die Dynamik des Roboters vorhersagt und dann mittels des inversen Modells die Motorbefehle berechnet.

Contents

Chapter 1

Introduction

In recent years, Artificial Intelligence (AI) has started to surpass humans on a variety of different tasks, from image recognition over video games to complex board games like Go (Krizhevsky *et al.*, 2012; Mnih *et al.*, 2013; Silver *et al.*, 2016). These machine learning algorithms are inspired by biological neurons that enable the learning and processing of information in the mammalian brain.

Artificial neurons, which are the basis of current high performing Artificial Neural Networks (ANNs), approximate the behavior of a biological neuron by computing an activation function over a weighted sum of inputs. Even particular ANN structures are often inspired by structures found in the mammalian brain. Convolutional Neuronal Networks (CNNs) for example are loosely based on the visual cortex and employ several convolutional layers that learn increasingly complex visual features by mimicking the primate brains visual hierarchy (Cichy *et al.*, 2016).

While sophisticated reinforcement architectures for playing video games heavily relay on the interpretation of visual inputs through the use of CNNs, they are in essence feed forward networks that employ no recurrent connections between neurons (Mnih *et al.*, 2013). The human brain on the other hand is a highly recurrent neural network, capable of memorizing and processing past informations (Kar *et al.*, 2019; Kietzmann *et al.*, 2019). Recurrent ANNs like Long Short-Term Memory (LSTM) try to recreate such capabilities by recurrently connecting neurons within their networks. LSTMs for example use a gated recurrently connected memory cell that can hold information over extended periods of time (Hochreiter and Schmidhuber, 1997). This specialized architecture enables them to tackle challenging tasks like speech recognition, polyphonic music generation or recently mastering the professional e-sports game StarCraft II (Xiong *et al.*, 2018; Zhao *et al.*, 2018; Vinyals *et al.*, 2019).

While ANNs are in principle inspired by the workings of the brain, artificial neurons with a nonlinear activation function are a rough simplification of real neurons. In the brain, neurons do not produce real value outputs, instead they fire a so-called spike and thus effectively produce binary outputs. As shown in Figure 1.1, neurons collect incoming excitatory and inhibitory post synaptic potential (EPSP/IPSP) from presynaptic neurons. Once their internal action potential reaches a certain threshold, they produce a spike, which is then transmitted as EPSP or IPSP de-

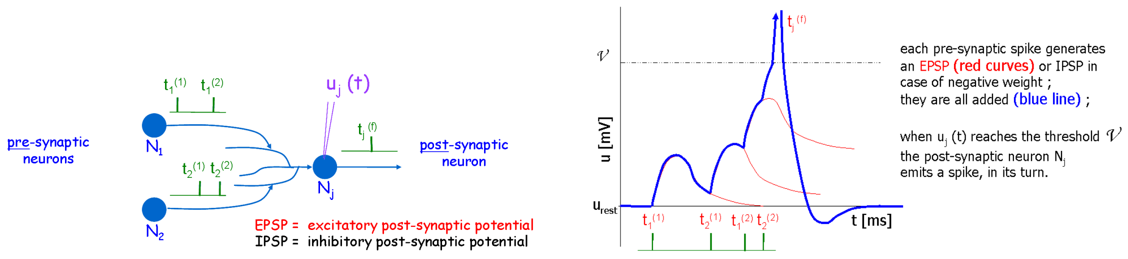

pre-synaptic neurons
N1
N2
t1(1) t1(2)
t2(1) t2(2)
uj (t)
tj(f)
Nj
post-synaptic neuron
EPSP = excitatory post-synaptic potential
IPSP = inhibitory post-synaptic potential
u [mV]
urest
t [ms]
each pre-synaptic spike generates an EPSP (red curves) or IPSP in case of negative weight ; they are all added (blue line) ;
when uj (t) reaches the threshold the post-synaptic neuron Nj emits a spike, in its turn.

Figure 1.1: Left: Presynaptic neurons spike and generate EPSP or IPSP depending on the synaptic weight. Right: Internal action potential of a neuron which increases through incoming EPSP until it reaches the neurons firing threshold. After the neuron spiked, it enters a refractory period where its action potential is reset. Figure adapted from Paugam-Moisy and Bohte (2012).

pending on the synaptic weight to post synaptic neurons. After a neuron has fired a spike, its action potential is reset and the neuron enters a refractory period in which it can not spike again. Classical Artificial neurons, also often called the second generation of ANNS, approximate the computation of biological neurons by assuming that neurons carry out computation by means of different firing rates. The real value in- and outputs to and from artificial neurons are an approximation of these firing rates. Since biological neurons also use the precise timing of individual spikes, this assumption does not hold (Maass, 1997). In order to improve the biological plausibility and accuracy, recently the third generation of neural networks, Spiking Neural Networks (SNN), has come into focus (Paugam-Moisy and Bohte, 2012). The difference to the second generation of ANNs is that instead of using real values as in- and outputs, neurons of the third generation only produce binary outputs, where a one is called a spike. The internal state of a spiking neuron includes an action potential, where the neuron integrates over weighted incoming spikes. Once this action potential reaches a certain threshold, the neuron itself produces a spike and its action potential is reset. After a spike, the neuron usually enters a refractory period in which it can not spike again.

One of the key advantages of SNNs over their second generation counterparts, is that they can be employed highly energy-efficiently in so-called neuromorphic hardware (Li *et al.*, 2018; Prodromakis and Toumazou, 2010; Liu and Delbruck, 2010). Neuromorphic chips like TrueNorth from IBM, that embodies a SNN of about 1 million neurons and 256 million synapses, have already been used to decode EEG signals, or driving a robot (Hwu *et al.*, 2017; Akopyan *et al.*, 2015; Nurse *et al.*, 2016).

While all spiking neurons try to model biological neurons more accurately than their second generation counterparts, there is a wide variety of different neuron types ranging in complexity and biological accuracy.

The Hodgkin-Huxley model is a very precise biophysically realistic model of a biological neuron, which models a neuron's membrane potential based on four coupled differential equations (Rinzel, 1990; Borges *et al.*, 2016). These equations account for sodium and potassium ion channels within a neuron's cell membrane that make up the membranes input and output current. From a computational perspective, simulating the Hodgkin-Huxley model takes a long time, since for each neuron several equations have to be solved. Also because numerical errors add up over time, a small step size has to be used during the simulation (Long and Fang, 2010).

Like the Hodgkin-Huxley model, the Izhikevich neuron is a precise model of a biological neuron, but uses a simplified set of two parameterizable differential equations. It simplifies the sodium and potassium ion channels into a general input current and uses a membrane recovery variable in order to model the neuron's refractory period. By using parameterizable equations, the Izhikevich neuron can simulate a variety of different neuron types from the mammalian brain (Izhikevich, 2003). Also since the model has an internal reset after each spike, numerical errors only add up from one spike to another and thus a significant greater step size than with Hodgkin-Huxley neurons can be used in a simulation.

The Leaky Integrate and Fire neuron (LIF) is a simple and computational cheap model of a biological neuron, but still capable of simulating the essential features of neural processing (Burkitt, 2006). LIF neurons are described by a single equation that is simple enough to make it mathematically analyzable. Also instead of using a differential equation, LIF neurons can be modelled directly in discrete time steps, which makes them especially suitable for computer simulations (Bellec *et al.*, 2019a).

Training of ANNs can be efficiently done with some variant of Gradient Descent and back-propagation or back-propagation through time for recurrent ANNs (Werbos *et al.*, 1990). For SNNs on the other hand, there exists no such universal training algorithm. Back-propagation requires a differential activation function, which is not possible with a non-continuous spiking function. So, in order to train SNNs, there exists a variety of different training algorithms. One of which is Spike-Timing-Dependent Plasticity (STDP), a learning rule based on the order of pre and post synaptic spikes. It has been demonstrated that STDP is one of the mechanisms for strengthening and weakening of synaptic connections between neurons in various species (Caporale and Dan, 2008). It is also known that this hebbian learning rule is at play in the visual cortex of primates (Huang *et al.*, 2014). Kheradpisheh *et al.* (2018); Mozafari *et al.* (2018, 2019) demonstrated that a simple STDP based learning rule enables Deep Convolutional Spiking Neural Networks to learn visual features and produce high accuracies for image classification tasks like MNIST and ETH-80.

Another biologically inspired training method is to use optimization strategies inspired by natural evolution called Evolutionary Algorithms (EA) (Tomassini, 1999; Kern *et al.*, 2004). Neuro-Evolution of Augmenting Topologies (NEAT) is one of such algorithms that is specifically designed for evolving the connectivifty of Neural Networks (Stanley and Miikkulainen, 2002). Like other EAs, NEAT employs the general principle of creating a population of individuals, neural networks that slightly differ in their connectivity, number of neurons and synaptic weights. Through mutations, NEAT slightly changes the weights of a mutating individual, inserts new neurons or creates a new synapse. Networks similar in structure can also be recombined, which means that the resulting network inherits the connectivity of the fitter one, but synaptic weights of synapses found in both networks are chosen at random. Through the optimization process, individuals compete against each other and only the best performing ones are allowed to create the next generation through mutations and recombinations. By using NEAT together with SpiNNaker, a fast neuromorphic supercomputer, Vandesompele *et al.* (2016) was able to teach a SNN to play the Atari game Pacman.

Recently Bellec *et al.* (2018) was able to train a so-called Long Short-Term Spiking Neural Network (LSNN) containing a mix of recurrently connected LIF- and ALIF neurons (neurons with and without an adaptive threshold) using Back-Propagation Through Time (BPTT). By using a pseudo-derivative in place for the non-existing derivative of the spiking function, LSNNs for the first time reach the performance of LSTMs. The usage of LIF neurons with an adaptive threshold particularly enables the back-propagation of error signals through several hundreds of time steps, avoiding the vanishing gradient problem similar to LSTMs that achieve this with their gated memory cells. In two follow up papers, Bellec *et al.* (2019a,b) also derived a biologically plausible learning rule called e-prop using its back-propagation approach on LSNNs. As shown in Figure 1.2, e-prop basically

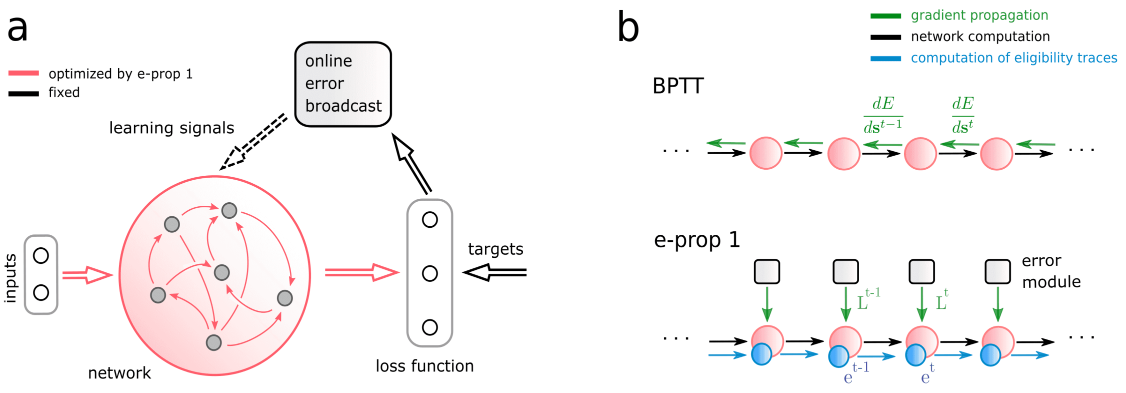


Figure 1.2: (**a**) Schematic view of an LSNN, recurrently connected Leaky Integrate and Fire neurons, with or without an adaptive threshold are trained by combining a local eligibility trace with an online learning signal computed as an error broadcast of the network error. (**b**) Comparison of information flow for Back-Propagation Through Time and e-prop. Image adapted from Bellec *et al.* (2019a).

uses BPTT in order to derive a local eligibility trace that can be computed forward in time and then combines it with a learning signal. When the learning signal is calculated using BPTT, e-prop is formally equivalent to BPTT and computes the same gradients. By approximating the learning signal through the use of past and present error information, a biologically plausible learning rule emerges which achieves accuracies comparable to BPTT.

One application where traditional RNNs, specifically LSTMs, achieve high performance is the control of a many-joint-robotic arm (Otte *et al.*, 2018, 2017b, 2016). For robotic arms with many degrees of freedom as shown in Figure 1.3, an LSTM can learn to predict the target pose of an arm given the input angles for each joint in a sequential order. By back-propagating the discrepancy between a desired goal position and its current estimation, an input gradient can be computed. This input gradient can then be used together with a gradient optimization method in order to adjust the joint angles and to smoothly steer the arm towards its goal position.

Traditional control algorithms like Rapid Exploring Random Trees or the Covariant Hamiltonian Optimization Motion Planner can need seconds to compute complex trajectories like those required by many-joint robotic arms (Kuffner and LaValle, 2000; Zucker *et al.*, 2013). Not only are SNNs implemented in neuromorphic hardware highly energy-efficient, they are also extremely fast and thus could enable the energy-efficient control of many-joint robotic arms in real-time. So the goal of this thesis is to explore whether biologically plausible SNNs, specifically LSNNs, could be trained to control the movement of such robotic arms through the process of action inference.

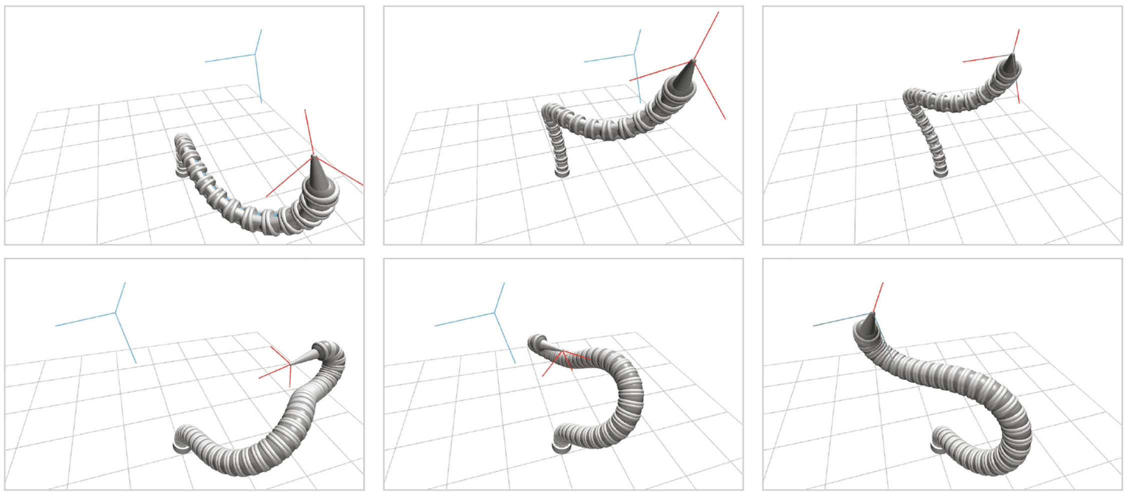


Figure 1.3: Action inference with a 20-joint robotic arm (**top**) and a 40-joint robotic arm (**bottom**). Image adapted from Otte *et al.* (2016).

Chapter 2

Foundations

This chapter derives the basic equations that build the foundation for the majority of experiments performed within this thesis. This includes the manly used neuron and network model, together with a variety of different training algorithms, ranging from BPTT to different flavors of the biologically plausible e-prop algorithm.

2.1 Network Model

The SNN architecture used in this thesis is based on the Long Short-Term Memory SNN (LSNN) developed by Bellec *et al.* (2019a, 2018). A LSNN is a simple 3-layer SNN with an input layer, followed by a recurrently connected hidden layer, and a readout layer. In the original LSNN, neurons from the input layer are spiking neurons and can only send a zero or one over their synapses, which multiplies the value with their weights. In this work, the spiking input neurons are replaced by artificial neurons that use the identity as activation function. They work the same way for spiking input data, but can also inject a real valued current into the network, which then is also weighted by the synaptic input weights. Neurons from the hidden layer are an ensemble of Leaky Integrate and Fire neurons (LIF), and Adaptive Leaky Integrate and Fire neurons (ALIF).

$$v_j^{t+1} = \alpha v_j^t + \sum_i w_{i,j}^{in} x_i^{t+1} + \sum_i w_{i,j}^{rec} z_i^t - z_j^t v_{thr} \tag{2.1}$$

The action potential or voltage v_j^t of an LIF neuron is updated according to Equation 2.1. Here x_i^{t+1} are the inputs from input neurons, z_i^t are hidden spikes and $\alpha = e^{-\Delta t/\tau_m}$ is a decay factor based on the membrane time constant τ_m and the length of one simulation time step Δt. After a spike z_j^t, the voltage is reset by subtracting the value of the firing threshold v_{thr}.

ALIF neurons use a threshold adaption value a_j^t in order to calculate an adaptive threshold $a_{j,thr}^t$.

$$a_j^{t+1} = \rho a_j^t + z_j^t \tag{2.2}$$

$$a_{j,thr}^t = v_{thr} + \beta a_j^t \tag{2.3}$$

This threshold adaption value increases with each spike z_j^t and slowly decays back to the base threshold v_{thr} based on the adaption decay factor $\rho = e^{-\Delta t/\tau_a}$, where τ_a is the adaption time constant. The adaptive threshold $a_{j,thr}^t$ is then computed using a constant threshold increase factor β as shown in Equation 2.3. The dynamics of ALIF, except for the adaptive threshold, evolve according to Equation 2.1. From Bellec *et al.* (2019a, 2018) it is not clear whether the reset of an ALIF neuron uses the adaptive threshold $a_{j,thr}^t$ or the base threshold v_{thr}, but neither version showed a significant improvement over the other regarding the simulations performed during this thesis, so the simple reset about the base threshold was used for the experiments performed within this thesis.

The effect of an adaptive threshold can be clearly seen in Figure 2.1, where an LIF and an ALIF neuron are both supplied with a constant input current, and the same base threshold. On the right side, the time between consecutive spikes increases for an ALIF neuron, while on the left side an LIF neuron spikes in regular intervals.

With the constant or adaptive threshold the spikes of LIF and ALIF neurons are calculated using the Heaviside step function as shown in Equation 2.4 and 2.5.

$$z_{j,LIF}^t = H\left(\frac{v_j^t - v_{thr}}{v_{thr}}\right) \tag{2.4}$$

$$z_{j,ALIF}^t = H\left(\frac{v_j^t - a_{j,thr}^t}{v_{thr}}\right) \tag{2.5}$$

The outputs of the LSNN are leaky integrators called readout neurons that sum up the network spikes from the current time step, which is due to the layer-wise computation of the network (Equation 2.6).

$$v_k^{t+1} = \alpha v_k^t + \sum_j w_{j,k}^{out} z_j^{t+1} \tag{2.6}$$

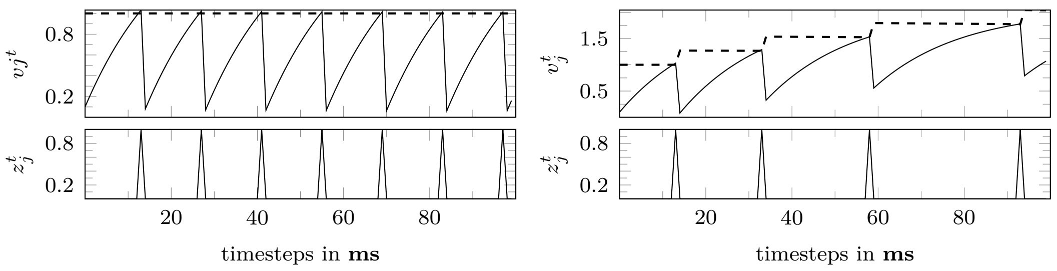


Figure 2.1: Comparison of LIF and ALIF neurons. **Left**: LIF neuron with a constant input current of 0.1 and a threshold of 1.0. **Right**: ALIF neuron with a constant input current of 0.1, a base threshold of 1.0 and a threshold increases constant of $\beta = 0.27$.

2.2 Back-Propagation Through Time

In this section, a mathematical framework is presented in order to formally derive the equations to update the synaptic weights of an LSNN using Back-Propagation Through Time (BPTT). It is assumed that for a given time step, the network's state can be described by the hidden state vectors $\mathbf{s}_j^t$ for each neuron and their corresponding observable output states z_j^t.

For LIF neurons this hidden state vector is one-dimensional and contains the voltage v_j^t, while for ALIF neurons it is two-dimensional and contains the voltage v_j^t together with the threshold adaption value a_j^t (Equation 2.7 and 2.8).

$$\mathbf{s}_{j,LIF}^t \overset{\text{def}}{=} v_j^t \tag{2.7}$$

$$\mathbf{s}_{j,ALIF}^t \overset{\text{def}}{=} \begin{pmatrix} v_j^t \\ a_j^t \end{pmatrix} \tag{2.8}$$

The network dynamics then evolve according to Equation 2.9, where $\mathbf{x}^{t+1}$ are the vector of network inputs at time step $t+1$ and $\mathbf{w}$ is the vector containing the network weights.

For each neuron the network function f then maps from the current hidden state $\mathbf{s}_j^t$ to the next hidden state $\mathbf{s}_j^{t+1}$, given the network's inputs, weights and observable states.

$$\mathbf{s}_j^{t+1} = f(\mathbf{s}_j^t, \mathbf{z}^t, \mathbf{x}^{t+1}, \mathbf{w}) \tag{2.9}$$

The next assumption is that a given error function $E(\mathbf{z}^1, ..., \mathbf{z}^T)$ only depends on the neuron outputs over a specific simulation time T.

For shorter equations the following simplified definitions are used.

$$\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} \overset{\text{def}}{=} \frac{\partial f}{\partial \mathbf{s}_j^t}(\mathbf{s}_j^t, \mathbf{z}^t, \mathbf{x}^{t+1}, \mathbf{w}) \tag{2.10}$$

It is worth mentioning, that there is an explicit distinction between partial derivatives and total derivatives of the network function f with respect to a certain variable. While ${}^{df}/_{d}$ denotes the total derivative, ${}^{\partial f}/_{\partial}$ stands for the partial derivative.

In order to use back-propagation to train an LSNN, one would need to calculate the derivative of the Heaviside step function, which is not defined. To overcome this problem, Bellec *et al.* (2019a) uses a pseudo-derivative h_j^t in place for the non-existing derivative of the Heaviside step function.

Using this pseudo-derivative, one can calculate the derivative of the the hidden state $\mathbf{s}_j^t$ by the observable state z_j^t for LIF (Equation 2.11) and ALIF (Equation 2.12 and 2.13) neurons (see Figure 2.2 for a visualization).

$$\frac{\partial z_j^t}{\partial v_{j,LIF}^t} \overset{\text{def}}{=} h_{j,LIF}^t = \gamma\ max\left(0, 1 - |\frac{v_j^t - v_{thr}}{v_{thr}}|\right) \tag{2.11}$$

$$\frac{\partial z_j^t}{\partial v_{j,ALIF}^t} \overset{\text{def}}{=} h_{j,ALIF}^t = \gamma\ max\left(0, 1 - |\frac{v_j^t - a_{j,thr}^t}{v_{thr}}|\right) \tag{2.12}$$

$$\frac{\partial z_j^t}{\partial a_j^t} \overset{\text{def}}{=} -\beta h_{j,ALIF}^t = -\beta\gamma\ max\left(0, 1 - |\frac{v_j^t - a_{j,thr}^t}{v_{thr}}|\right) \tag{2.13}$$

Using this derivatives, one can derive the basic equations for the error gradient of E with respect to a specific weight.

$$\frac{dE}{w_{i,j}} = \sum_t \frac{dE}{d\mathbf{s}_j^t}\frac{\partial \mathbf{s}_j^t}{w_{i,j}} \tag{2.14}$$

$$\boldsymbol{\delta}_j^t \overset{\text{def}}{=} \frac{dE}{d\mathbf{s}_j^t} = \frac{dE}{z_j^t}\frac{\partial z_j^t}{\partial \mathbf{s}_j^t} + \frac{dE}{d\mathbf{s}_j^{t+1}}\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} \tag{2.15}$$

$$\frac{dE}{z_j^t} = \sum_k \frac{dE}{d\mathbf{s}_k^t}\frac{\partial \mathbf{s}_k^t}{\partial z_j^t} + \sum_i \frac{dE}{d\mathbf{s}_i^{t+1}}\frac{\partial \mathbf{s}_i^{t+1}}{\partial z_j^t} + \frac{dE}{d\mathbf{s}_j^{t+1}}\frac{\partial \mathbf{s}_j^{t+1}}{\partial z_j^t} \tag{2.16}$$

The delta error $\boldsymbol{\delta}_j^t$ (Equation 2.15) just as the error function derived by the hidden spike (Equation 2.20) expand into the future and can be calculated using back-propagation.

2.2.1 Error Gradients for LIF Neurons

For LIF neurons the hidden state vector $\mathbf{s}_j^t$ is one-dimensional and equals v_j^t. So the gradients for input and recurrent weights compute to the sum of delta errors over time masked with the timing of presynaptic spikes for recurrent weights or weighted by the input value for input weights.

$$\frac{dE}{w_{i,j}^{in}} = \sum_t \boldsymbol{\delta}_j^t x_i^t \tag{2.17}$$

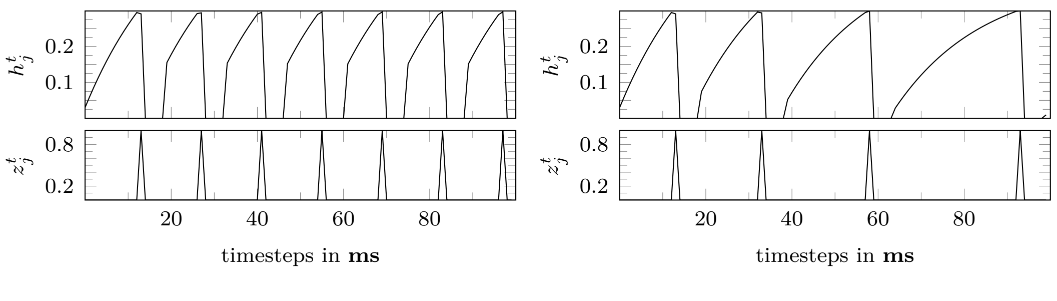


Figure 2.2: Comparison of LIF and ALIF neuron derivatives, with a derivative dumping factor of $\gamma = 0.3$. **Left**: LIF neuron with a constant input current of 0.1 and a threshold of 1.0. **Right**: ALIF neuron with a constant input current of 0.1, a base threshold of 1.0 and a threshold increases constante of $\beta = 0.27$.

$$\frac{dE}{w^{rec}_{i,j}} = \sum_t \boldsymbol{\delta}^t_j z^{t-1}_i \tag{2.18}$$

By replacing $\mathbf{s}^t_j$ with v^t_j in the equation for the delta error (Equation 2.15), and also inserting the pseudo-derivative one gets:

$$\begin{aligned}\boldsymbol{\delta}^t_j &= \frac{dE}{z^t_j}\frac{\partial z^t_j}{\partial v^t_j} + \frac{dE}{dv^{t+1}_j}\frac{\partial v^{t+1}_j}{\partial v^t_j} \\ &= \frac{dE}{z^t_j} h^t_j + \boldsymbol{\delta}^{t+1}_j \alpha\end{aligned} \tag{2.19}$$

The derivative of the error function with respect to the output spike then computes to the weighted sum over output delta errors from the present time step, plus weighted delta errors from recurrent neurons from the next time step, minus the delta error from the next time step multiplied with the firing threshold.

$$\begin{aligned}\frac{dE}{z^t_j} &= \sum_k \frac{dE}{d\mathbf{s}^t_k}\frac{\partial \mathbf{s}^t_k}{\partial z^t_j} + \sum_i \frac{dE}{dv^{t+1}_i}\frac{\partial v^{t+1}_i}{\partial z^t_j} + \frac{dE}{dv^{t+1}_j}\frac{\partial v^{t+1}_j}{\partial z^t_j} \\ &= \sum_k \boldsymbol{\delta}^t_k \frac{\partial \mathbf{s}^t_k}{\partial z^t_j} + \sum_i \boldsymbol{\delta}^{t+1}_i \frac{\partial \mathbf{s}^{t+1}_i}{\partial z^t_j} - \boldsymbol{\delta}^{t+1}_j v_{thr} \\ &= \sum_k w^{out}_{j,k}\boldsymbol{\delta}^t_k + \sum_i w^{rec}_{j,i}\boldsymbol{\delta}^{t+1}_i - \boldsymbol{\delta}^{t+1}_j v_{thr}\end{aligned} \tag{2.20}$$

2.2.2 Error Gradients for ALIF Neurons

For ALIF neurons the hidden state vector $\mathbf{s}^t_j$ is two-dimensional and contains v^t_j and a^t_j. But still, the gradients for input and recurrent weights compute to the sum of voltage delta errors over time masked with the timing of presynaptic spikes for recurrent weights or weighted by the input value for input weights.

$$\begin{aligned}\frac{dE}{w^{in}_{i,j}} &= \sum_t \begin{pmatrix} \frac{dE}{dv^t_j} & \frac{dE}{da^t_j} \end{pmatrix} \begin{pmatrix} \frac{\partial v^t_j}{w^{in}_{i,j}} \\ \frac{\partial a^t_j}{w^{in}_{i,j}} \end{pmatrix} \\ &= \sum_t \begin{pmatrix} \delta^t_{j,v} & \delta^t_{j,a} \end{pmatrix} \begin{pmatrix} x^t_i \\ 0 \end{pmatrix} \\ &= \sum_t \delta^t_{j,v} x^t_i\end{aligned} \tag{2.21}$$

$$\frac{dE}{w^{rec}_{i,j}} = \sum_t \delta^t_{j,v} z^{t-1}_i \tag{2.22}$$

The partial derivative of the next hidden state by the current one ${\partial \mathbf{s}_j^{t+1}}/{\partial \mathbf{s}_j^t}$ is now a 2x2 matrix for ALIF neurons.

$$\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} = \begin{pmatrix} \frac{\partial v_j^{t+1}}{\partial v_j^t} & \frac{\partial v_j^{t+1}}{\partial a_j^t} \\ \frac{\partial a_j^{t+1}}{\partial v_j^t} & \frac{\partial a_j^{t+1}}{\partial a_j^t} \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ 0 & p \end{pmatrix} \tag{2.23}$$

Inserting this state derivative matrix into the equation for the delta error gives a two-dimensional delta error containing a voltage delta error part and an adaption delta error part.

$$\begin{aligned} \boldsymbol{\delta}_j^t &= \frac{dE}{z_j^t}\frac{\partial z_j^t}{\partial \mathbf{s}_j^t} + \frac{dE}{d\mathbf{s}_j^{t+1}}\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} \\ &= \frac{dE}{z_j^t}\begin{pmatrix} h_j^t \\ -h_j^t\beta \end{pmatrix} + \begin{pmatrix} \alpha & 0 \\ 0 & p \end{pmatrix}\begin{pmatrix} \delta_{j,v}^t \\ \delta_{j,a}^t \end{pmatrix} \\ &= \frac{dE}{z_j^t}\begin{pmatrix} h_j^t \\ -h_j^t\beta \end{pmatrix} + \begin{pmatrix} \alpha\delta_{j,v}^t \\ p\delta_{j,a}^t \end{pmatrix} \end{aligned} \tag{2.24}$$

The error function derived with respect to the output spike then becomes a weighted sum of voltage delta errors over time from readout and hidden neurons. The future error of the neuron j is incorporated in the second part of the final equation, where the voltage delta error from the next time step weighted with the base threshold is subtracted and the adaption delta error from the next time step is added.

$$\begin{aligned} \frac{dE}{dz_j^t} &= \sum_k \frac{dE}{d\mathbf{s}_k^t}\frac{\partial \mathbf{s}_k^t}{\partial z_j^t} + \sum_i \frac{dE}{d\mathbf{s}_i^{t+1}}\frac{\partial \mathbf{s}_i^{t+1}}{\partial z_j^t} + \frac{dE}{d\mathbf{s}_j^{t+1}}\frac{\partial \mathbf{s}_j^{t+1}}{\partial z_j^t} \\ &= \sum_k \frac{dE}{d\mathbf{s}_k^t}\frac{\partial \mathbf{s}_k^t}{\partial z_j^t} + \sum_i \frac{dE}{d\mathbf{s}_i^{t+1}}\frac{\partial \mathbf{s}_i^{t+1}}{\partial z_j^t} + \frac{dE}{d\mathbf{s}_j^{t+1}}\begin{pmatrix} -v_{thr} \\ 1 \end{pmatrix} \\ &= \sum_k \boldsymbol{\delta}_k^t \frac{\partial \mathbf{s}_k^t}{\partial z_j^t} + \sum_i \boldsymbol{\delta}_i^{t+1}\frac{\partial \mathbf{s}_i^{t+1}}{\partial z_j^t} + \begin{pmatrix} \delta_{j,v}^{t+1} & \delta_{j,a}^{t+1} \end{pmatrix}\begin{pmatrix} -v_{thr} \\ 1 \end{pmatrix} \\ &= \sum_k \boldsymbol{\delta}_k^t w_{j,k}^{out} + \sum_i \begin{pmatrix} \delta_{i,v}^{t+1} & \delta_{i,a}^{t+1} \end{pmatrix}\begin{pmatrix} w_{j,i}^{rec} \\ 0 \end{pmatrix} - \delta_{j,v}^{t+1}v_{thr} + \delta_{j,a}^{t+1} \\ &= \sum_k w_{j,k}^{out}\delta_{k,v}^t + \sum_i w_{j,i}^{rec}\delta_{i,v}^{t+1} - \delta_{j,v}^{t+1}v_{thr} + \delta_{j,a}^{t+1} \end{aligned} \tag{2.25}$$

2.2.3 Error Gradients for Output Synapses

For readout neurons as defined in Equation 2.6, the delta error is an exponential decaying sum of future errors with respect to the particular readout neuron k.

$$\begin{aligned}\boldsymbol{\delta}_k^t &= \frac{\partial E}{\partial \mathbf{s}_k^t} + \frac{dE}{d\mathbf{s}_k^{t+1}} \frac{\partial \mathbf{s}_k^{t+1}}{\mathbf{s}_k^t} \\ &= \frac{\partial E}{\partial \mathbf{s}_k^t} + \boldsymbol{\delta}_k^{t+1} \alpha\end{aligned} \tag{2.26}$$

Using a summed squared error function $E = \frac{1}{2} \sum_{t,k} (v_k^t - u_k^t)^2$ with the target output u_k^t for readout neuron k at time step t, the delta error computes to a exponential smoothed difference between the readout neuron's output and the target signal.

$$\boldsymbol{\delta}_k^t = (v_k^t - u_k^t) + \boldsymbol{\delta}_k^{t+1} \alpha \tag{2.27}$$

The resulting gradient for output synapses is then the masked sum (by presynaptic spikes) of filtered readout errors.

$$\begin{aligned}\frac{dE}{dw_{ik}^{out}} &= \sum_t \frac{dE}{d\mathbf{s}_k^t} \frac{\partial \mathbf{s}_k^t}{w_{ik}^{out}} \\ &= \sum_t ((v_k^t - u_k^t) + \boldsymbol{\delta}_k^{t+1} \alpha) z_i^t \\ &= \sum_t \sum_{t' \geq t} (v_k^{t'} - u_k^{t'}) \alpha^{t'-t} z_i^t\end{aligned} \tag{2.28}$$

2.2.4 Error Gradients for Network Inputs

In order to use a gradient optimization technique for input values, the input gradient for a specific input and time step can be computed by weighting all hidden delta errors from that time step with the input synapses from the input in question.

$$\frac{dE}{dx_i^t} = \sum_j \frac{\partial \mathbf{s}_j^t}{\partial x_i^t} \frac{dE}{d\mathbf{s}_j^t} = \sum_j w_{i,j}^{in} \frac{dE}{d\mathbf{s}_j^t} = \sum_j w_{i,j}^{in} \boldsymbol{\delta}_j^t \tag{2.29}$$

2.3 E-Prop

The motivation of the e-prop algorithm is that BPTT is a biologically implausible way of solving the Temporal Credit Assignment problem (TCA) (Lillicrap and Santoro, 2019). Instead experimental data suggest that the brain solves TCA by combining local eligibility traces, which are accumulated information about a synapse's activation history, with neuromodulator based reward signals (Gerstner *et al.*, 2018).

The e-prop algorithm derives this principle of combining local eligibility traces with an online learning signal, by factoring the error gradients from BPTT into a sum of products between local eligibility traces and online learning signals.

$$\frac{dE}{dw_{i,j}} = \sum_t L_j^t e_{i,j}^t \tag{2.30}$$

Equation 2.30 shows this general decomposition of the error gradient for the synaptic weight $w_{i,j}$ into the sum of online learning signal L_j^t at time step t and the eligibility trace $e_{i,j}^t$ at the same time step. Eligibility traces here refer to all information that is currently available at the synapse. So it only considers past events, while the learning signal in this factorization contains the future errors.

$$\begin{aligned}
\frac{dE}{dw_{i,j}} &= \sum_t \frac{dE}{d\mathbf{s}_j^t}\frac{\partial \mathbf{s}_j^t}{\partial w_{i,j}} \\
&= \sum_t \left(\frac{dE}{dz_j^t}\frac{\partial z_j^t}{\partial \mathbf{s}_j^t} + \frac{dE}{d\mathbf{s}_j^{t+1}}\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t}\right)\frac{\partial \mathbf{s}_j^t}{\partial w_{i,j}} \\
&= \sum_t \left(\frac{dE}{dz_j^t}\frac{\partial z_j^t}{\partial \mathbf{s}_j^t} + \left(\frac{dE}{dz_j^{t+1}}\frac{\partial z_j^{t+1}}{\partial \mathbf{s}_j^{t+1}} + (\cdots)\frac{\partial \mathbf{s}_j^{t+2}}{\partial \mathbf{s}_j^{t+1}}\right)\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t}\right)\frac{\partial \mathbf{s}_j^t}{\partial w_{i,j}} \\
&= \sum_t \sum_{t' \geq t} \frac{dE}{dz_j^{t'}}\frac{\partial z_j^{t'}}{\partial \mathbf{s}_j^{t'}}\frac{\partial \mathbf{s}_j^{t'}}{\partial \mathbf{s}_j^{t'-1}}\frac{\partial \mathbf{s}_j^{t'-1}}{\partial \mathbf{s}_j^{t'-2}}\cdots\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^{t}}\frac{\partial \mathbf{s}_j^t}{\partial w_{i,j}} \\
&= \sum_t \sum_{t' \leq t} \frac{dE}{dz_j^{t}}\frac{\partial z_j^{t}}{\partial \mathbf{s}_j^{t}}\frac{\partial \mathbf{s}_j^{t}}{\partial \mathbf{s}_j^{t-1}}\frac{\partial \mathbf{s}_j^{t-1}}{\partial \mathbf{s}_j^{t-2}}\cdots\frac{\partial \mathbf{s}_j^{t'+1}}{\partial \mathbf{s}_j^{t'}}\frac{\partial \mathbf{s}_j^{t'}}{\partial w_{i,j}} \\
&= \sum_t \frac{dE}{dz_j^{t}}\frac{\partial z_j^{t}}{\partial \mathbf{s}_j^{t}} \sum_{t' \leq t} \frac{\partial \mathbf{s}_j^{t}}{\partial \mathbf{s}_j^{t-1}}\frac{\partial \mathbf{s}_j^{t-1}}{\partial \mathbf{s}_j^{t-2}}\cdots\frac{\partial \mathbf{s}_j^{t'+1}}{\partial \mathbf{s}_j^{t'}}\frac{\partial \mathbf{s}_j^{t'}}{\partial w_{i,j}} \\
&\stackrel{\text{def}}{=} \sum_t \frac{dE}{dz_j^{t}}\frac{\partial z_j^{t}}{\partial \mathbf{s}_j^{t}} \boldsymbol{\epsilon}_{i,j}^t \\
&\stackrel{\text{def}}{=} \sum_t \frac{dE}{dz_j^{t}} e_{i,j}^t \\
&\stackrel{\text{def}}{=} \sum_t L_j^t e_{i,j}^t
\end{aligned} \tag{2.31}$$

The derivation of the e-prop gradient calculation as shown in Equation 2.31 works by inserting the expansion of the delta error from Equation 2.15 into Equation 2.22. Through future expansion, one can rewrite the delta error part as a sum of products of future state derivatives. This product of future derivatives is defined to represent the identity for $t < t' - 1$ (Equation 2.32).

$$\frac{\partial \mathbf{s}_j^t}{\partial \mathbf{s}_j^{t-1}} \frac{\partial \mathbf{s}_j^{t-1}}{\partial \mathbf{s}_j^{t-2}} \cdots \frac{\partial \mathbf{s}_j^{t'+1}}{\partial \mathbf{s}_j^{t'}} \overset{\text{def}}{=} 1, \quad (if\ t < t' - 1) \tag{2.32}$$

The sum of products of future state derivatives multiplied with the hidden state derived by the synaptic weight can then be defined as what is called an eligibility vector ($\boldsymbol{\epsilon}_{i,j}^t$). The eligibility vector multiplied with derivation of the observable state by the hidden state is then defined as the eligibility trace. The remaining ${}^{dE}/_{z_j^t}$ would expand into the future, and is thus defined as the learning signal L_j^t.

$$\begin{aligned} \boldsymbol{\epsilon}_{i,j}^t &= \sum_{t' \le t} \frac{\partial \mathbf{s}_j^t}{\partial \mathbf{s}_j^{t-1}} \frac{\partial \mathbf{s}_j^{t-1}}{\partial \mathbf{s}_j^{t-2}} \cdots \frac{\partial \mathbf{s}_j^{t'+1}}{\partial \mathbf{s}_j^{t'}} \frac{\partial \mathbf{s}_j^{t'}}{w_{i,j}} \\ &= \boldsymbol{\epsilon}_{i,j}^{t-1} \frac{\partial \mathbf{s}_j^t}{\partial \mathbf{s}_j^{t-1}} + \frac{\partial \mathbf{s}_j^t}{w_{i,j}} \end{aligned} \tag{2.33}$$

Splitting the eligibility trace into a product of pseudo-derivative and eligibility vector is useful since the eligibility vector is determined by the recursively nested Equation 2.33 that can be computed incrementally forward in time.

2.3.1 Eligibility Traces for LIF Neurons

For LIF neurons, the eligibility trace is one-dimensional and is basically a product of pseudo-derivatives and filtered input spikes.

$$\boldsymbol{\epsilon}_{i,j}^{in,t+1} = \boldsymbol{\epsilon}_{i,j}^{in,t} \frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} + \frac{\partial \mathbf{s}_j^{t+1}}{w_{i,j}^{in}} = \boldsymbol{\epsilon}_{i,j}^{in,t} \alpha + x_i^{t+1} \tag{2.34}$$

$$\boldsymbol{\epsilon}_{i,j}^{rec,t+1} = \boldsymbol{\epsilon}_{i,j}^{rec,t} \frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} + \frac{\partial \mathbf{s}_j^{t+1}}{w_{i,j}^{rec}} = \boldsymbol{\epsilon}_{i,j}^{rec,t} \alpha + z_i^t \tag{2.35}$$

Equation 2.34 and 2.41 show the recursive calculation of eligibility vectors for input and hidden synapses.

$$e_{i,j}^{t+1} = \frac{\partial z_j^{t+1}}{\partial \mathbf{s}_j^{t+1}} \boldsymbol{\epsilon}_{i,j}^{t+1} = h_j^{t+1} \boldsymbol{\epsilon}_{i,j}^{t+1} \tag{2.36}$$

The eligibility trace is then given by Equation 2.36.

2.3.2 Eligibility Traces for ALIF Neurons

For ALIF neurons, the eligibility vector is two-dimensional and the state derivative is a 2x2 matrix. Using the state derivative as defined in Equation 2.23 leads to the same eligibility trace as for LIF neurons, since one gets the following eligibility vector, assuming $\boldsymbol{\epsilon}_{i,j}^{0} = 0$:

$$\begin{aligned}
\boldsymbol{\epsilon}_{i,j}^{t+1} &= \frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^{t}} \cdot \boldsymbol{\epsilon}_{i,j}^{t} + \frac{\partial \mathbf{s}_j^{t+1}}{\partial \theta_{ji}^{rec}} \\
&= \begin{pmatrix} \alpha & 0 \\ 0 & p \end{pmatrix} \cdot \begin{pmatrix} \epsilon_{i,j,v}^{t} \\ \epsilon_{i,j,a}^{t} \end{pmatrix} + \begin{pmatrix} z_i^t \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} \alpha\epsilon_{i,j,v}^{t} + z_i^t \\ p\epsilon_{i,j,a}^{t} \end{pmatrix} \\
&= \begin{pmatrix} \alpha\epsilon_{i,j,v}^{t} + z_i^t \\ 0 \end{pmatrix}
\end{aligned} \tag{2.37}$$

This results in the following eligibility trace, which is equal to that of LIF neurons:

$$\begin{aligned}
e_{ji}^{t+1} &= \frac{\partial z_j^{t+1}}{\partial \mathbf{s}_j^{t+1}} \cdot \boldsymbol{\epsilon}_{i,j}^{t+1} \\
&= \begin{pmatrix} h_j^{t+1} & -h_j^{t+1}\beta \end{pmatrix} \begin{pmatrix} \epsilon_{i,j,v}^{t+1} \\ \epsilon_{i,j,a}^{t+1} \end{pmatrix} \\
&= h_j^{t+1}(\epsilon_{i,j,v}^{t+1} - \beta\epsilon_{i,j,a}^{t+1}) \\
&= h_j^{t+1}(\epsilon_{i,j,v}^{t+1} - \beta 0) \\
&= h_j^{t+1}\epsilon_{i,j,v}^{t+1}
\end{aligned} \tag{2.38}$$

In order to compute a eligibility trace that reflects the adaptive behavior of the underlining neuron, instead of using the above derivation, Bellec *et al.* (2019a) inserts the definition of iz_j^t as given in Equation 2.5 into the equation for the adaptive threshold 2.3, which then leads to the following state derivative.

$$\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^{t}} = \begin{pmatrix} \frac{\partial v_j^{t+1}}{\partial v_j^t} & \frac{\partial v_j^{t+1}}{\partial a_j^t} \\ \frac{\partial a_j^{t+1}}{\partial v_j^t} & \frac{\partial a_j^{t+1}}{\partial a_j^t} \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ h_j^t & p - h_j^t\beta \end{pmatrix} \tag{2.39}$$

Using this extended state derivative from Equation 2.39 one gets the following two-dimensional eligibility vectors for input and recurrent synapses.

$$\begin{aligned}
\boldsymbol{\epsilon}_{i,j}^{in,t+1} &= \boldsymbol{\epsilon}_{i,j}^{in,t} \frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} + \frac{\partial \mathbf{s}_j^{t+1}}{w_{i,j}^{in}} \\
&= \begin{pmatrix} \alpha & 0 \\ h_j^t & p - h_j^t \beta \end{pmatrix} \cdot \begin{pmatrix} \epsilon_{i,j,v}^{in,t} \\ \epsilon_{i,j,a}^{in,t} \end{pmatrix} + \begin{pmatrix} x_i^{t+1} \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} \alpha \epsilon_{i,j,v}^{in,t} + x_i^{t+1} \\ h_j^t \epsilon_{i,j,v}^{in,t} + (p - h_j^t \beta) \epsilon_{i,j,a}^{in,t} \end{pmatrix}
\end{aligned} \tag{2.40}$$

$$\begin{aligned}
\boldsymbol{\epsilon}_{i,j}^{rec,t+1} &= \boldsymbol{\epsilon}_{i,j}^{rec,t} \frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} + \frac{\partial \mathbf{s}_j^{t+1}}{w_{i,j}^{rec}} \\
&= \begin{pmatrix} \alpha \epsilon_{i,j,v}^{rec,t} + z_i^t \\ h_j^t \epsilon_{i,j,v}^{rec,t} + (p - h_j^t \beta) \epsilon_{i,j,a}^{rec,t} \end{pmatrix}
\end{aligned} \tag{2.41}$$

This then leads to the one-dimensional eligibility trace for ALIF neurons:

$$\begin{aligned}
e_{i,j}^{t+1} &= \frac{\partial z_j^{t+1}}{\partial \mathbf{s}_j^{t+1}} \boldsymbol{\epsilon}_{i,j}^{t+1} \\
&= \begin{pmatrix} h_j^{t+1} & -h_j^{t+1} \beta \end{pmatrix} \begin{pmatrix} \epsilon_{i,j,v}^{t+1} \\ \epsilon_{i,j,a}^{t+1} \end{pmatrix} \\
&= h_j^{t+1} (\epsilon_{i,j,v}^{t+1} - \beta \epsilon_{i,j,a}^{t+1})
\end{aligned} \tag{2.42}$$

2.3.3 E-Prop 1: Learning Signals through Random Feedback Connections

To be biologically plausible, a learning signal can not rely on calculations of future errors. Therfore, e-pop 1 calculates an approximation of the true learning signal dE/dz_j^t (Equation 2.20) by ignoring future state errors of neuron j together with future errors from recurrently connected neurons. While this generally seems to work well, Bellec *et al.* (2019a) does not really ground this approximation, which could be problematic since it ignores all future influence of the current spike z_j^t.

$$L_j^t = \frac{dE}{z_j^t} \approx \sum_k \frac{dE}{d\mathbf{s}_k^t} \frac{\partial \mathbf{s}_k^t}{\partial z_j^t} \tag{2.43}$$

Equation 2.43 still comprises a problem for a biologically plausible learning signal, since the total derivative $dE/d\mathbf{s}_k^t$ still depends on future errors, but this problem can be resolved for a summed squared error function:

$$\begin{aligned}
\frac{dE}{dw_{i,j}} &= \sum_t L_j^t e_{i,j}^t \\
&\approx \sum_{t,k} \frac{dE}{d\mathbf{s}_k^t} \frac{\partial \mathbf{s}_k^t}{\partial z_j^t} e_{i,j}^t \\
&= \sum_{t,k} ((v_k^t - u_k^t) + \boldsymbol{\delta}_k^{t+1} \alpha) w_{j,k}^{out}) e_{i,j}^t \\
&= \sum_{t,k} \sum_{t' \geq t} (v_k^{t'} - u_k^{t'}) \alpha^{t'-t} w_{j,k}^{out} e_{i,j}^t \\
&= \sum_{t,k} \sum_{t' \leq t} (v_k^t - u_k^t) \alpha^{t-t'} w_{j,k}^{out} e_{i,j}^{t'} \\
&= \sum_t (\sum_k w_{j,k}^{out} (v_k^t - u_k^t)) \sum_{t' \leq t} \alpha^{t-t'} e_{i,j}^{t'}
\end{aligned} \tag{2.44}$$

By using the learning signal as defined in Equation 2.43, one gets a gradient calculation rule that only depends on past errors by turning the filtering of future errors into a filtering of past eligibility traces in Equation 2.44.

Since it is biologically implausible that errors would get weighted with the same value as the forward synapses, one can replace $w_{j,k}^{out}$ by a random feedback connection $w_{j,k}^{fb}$.

The derivative for output weights can also be calculated using past errors only:

$$\begin{aligned}
\frac{dE}{dw_{i,k}^{out}} &= \sum_t \frac{dE}{d\mathbf{s}_k^t} \frac{\partial \mathbf{s}_k^t}{w_{i,k}^{out}} \\
&= \sum_t ((v_k^t - u_k^t) + \boldsymbol{\delta}_k^{t+1} \alpha) z_i^t \\
&= \sum_t \sum_{t' \geq t} (v_k^{t'} - u_k^{t'}) \alpha^{t'-t} z_i^t \\
&= \sum_t \sum_{t' \leq t} (v_k^t - u_k^t) \alpha^{t-t'} z_i^{t'} \\
&= \sum_t (v_k^t - u_k^t) \sum_{t' \leq t} \alpha^{t-t'} z_i^{t'}
\end{aligned} \tag{2.45}$$

Equation 2.45 turns the filtering of future errors into a filtering of past hidden spikes.

2.3.4 Error Gradients for Network Inputs

Bellec *et al.* (2019a) does not apply the principles e-prop to input gradients, since input errors as derived in Equation 2.29 do not contain eligibility traces, which can only accumulated information about the activation history of a synapse.

Nevertheless an approximate input gradient can be derived that also neglects future errors in the same fashion as e-prop.

$$\frac{dE}{dx_i^t} = \sum_j \frac{\partial \mathbf{s}_j^t}{\partial x_i^t}\frac{dE}{d\mathbf{s}_j^t} = \sum_j w_{i,j}^{in}\left(\frac{dE}{dz_j^t}\frac{\partial z_j^t}{\partial \mathbf{s}_j^t} + \frac{dE}{d\mathbf{s}_j^{t+1}}\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t}\right) \tag{2.46}$$

By ignoring the future errors in Equation 2.46 one gets an approximate input error containing a learning signal similar to e-prop 1:

$$\frac{dE}{dx_i^t} \approx \sum_j w_{i,j}^{in}\frac{dE}{dz_j^t}\frac{\partial z_j^t}{\partial \mathbf{s}_j^t} = \sum_j w_{i,j}^{in} L_j^t \frac{\partial z_j^t}{\partial \mathbf{s}_j^t} \tag{2.47}$$

If one uses random feedback weights and the current output error to calculate the learning signal L_j^t, the approximate input error is a weighted sum over randomly weighted network errors multiplied with hidden (pseudo-)derivatives. So it only contains information about the network error at the present time step.

2.3.5 E-Prop 2 based Input Errors

Another way to calculate a biologically plausible input error is to use an approach similar to e-prop 2 where a separate LSNN (called error module) computes the learning signals L_j^t.

In order to train this error module that outputs learning signals, an error for this learning signal has to be derived. One way of doing this is to use so-called one shot learning, where an input error is calculated by using both the first LSNN together with the learning signals from the error module, and using the resulting gradient to update the inputs once.

$$\hat{x}_i^t = x_i^t - \sum_j \theta_{ji} L_j^t h_j^t \tag{2.48}$$

Equation 2.48 shows such an update rule, for an one shot learned input $\hat{x}_i^t$. Based on this equation, one can derive an error for the learning signal L_j^t.

$$\frac{dE}{dL_j^t} = \sum_i \frac{dE}{d\hat{x}_i^t}\frac{\partial \hat{x}_i^t}{\partial L_j^t} = -\sum_i \frac{dE}{d\hat{x}_i^t}\theta_{ji} h_j^t \tag{2.49}$$

The learning signal error as shown in Equation 2.49 contains a new error based on the updated input $\hat{x}_i^t$, which has to be computed. This is done by simply running the first LSNN again with the updated input and computing the input error for it.

Chapter 3

Emergence of STDP in E-Prop based Gradients

This chapter is somewhat separate from the rest of this thesis and shows an insight into e-prop, which was discovered after the majority of experiments were already performed and thus had no influence on experiments regarding the action inference of the many-joint robotic arm.

The e-prop algorithm as derived in Section 2.3 uses a very simple way of introducing a refractory period, by simply preventing the neurons from firing during a specified time window after the last spike. During this forced refractory period, the pseudo-derivatives are set to zero, but the neuron dynamics are otherwise unchanged.

By using a slightly more complex neuron model like the Izhikevich neuron which already has a refractory period built in, a Spike-Timing-Dependent Plasticity (STDP) based learning naturally emerges from eligibility traces.

STDP as already mentioned in the Introduction is a hebbian learning rule based on the order of pre- and postsynaptic spikes (Caporale and Dan, 2008). With STDP, the synaptic connection between two neurons is strengthened if the presynaptic neuron fires shortly before the postsynaptic one, implying a causal relationship between the pre- and postsynaptic spike. A weakening of the synapse occurs for events where the presynaptic neuron spikes shortly after the postsynaptic one, implying an uncorrelated spiking of the two neurons.

This chapter formally derives eligibility traces for Izhikevich neurons and experimentally shows the emergence of STDP influenced gradients when computed using e-prop. Also through simple modifications to the LIF neuron model the emergence of STDP based gradients in LSNNs is shown experimentally and mathematically.

3.1 Izhikevich Neuron

As already described in the Introduction, the Izhikevich neuron is a precise, but computationally cheap model of a biological neuron that uses two parameterizable differential equations.

For the derivation of eligibility traces, the typical parameter for those equations are used (Izhikevich, 2003).

$$v' = 0.04v^2 + 5v + 150 - u + I \tag{3.1}$$

$$u' = 0.004v - 0.02u \tag{3.2}$$

Equation 3.1 and 3.2 show the differential equations used by Izhikevich (2003) with the membrane voltage v and the recovery variable u. Once the membrane voltage crosses 30mV, a spike is emitted and v and u are reset in the following way:

```
if $v < 30mv$ then
    $v \leftarrow -65mV$
    $u \leftarrow u + d$
end if
```

After the reset, the neuron enters a refractory period where the voltage v goes to around 80mv as shown in Figure 3.1.

In order to derive an eligibility trace for Izhikevich neurons, the Equations 3.1 and 3.2 have to be modelled in discrete time steps, and also the reset has to be modelled within the new equations. To accomplish this, the variables $\tilde{v}_j^t$ and $\tilde{u}_j^t$ are introduced in order to replace v and u in the standard equations.

$$\tilde{v}_j^t = v_j^t - (v_j^t + 65)z_j^t \tag{3.3}$$

$$\tilde{u}_j^t = u_j^t + 2z_j^t \tag{3.4}$$

Now v and u can be computed in discrete time steps using Euler integration with a constant step size of δt.

$$v_j^{t+1} = \tilde{v}_j^t + \delta t(0.04(\tilde{v}_j^t)^2 + 5\tilde{v}_j^t + 140 - \tilde{u}_j^t + I_j^t) \tag{3.5}$$

$$u_j^{t+1} = \tilde{u}_j^t + \delta t(0.004\tilde{v}_j^t - 0.02\tilde{u}_j^t) \tag{3.6}$$

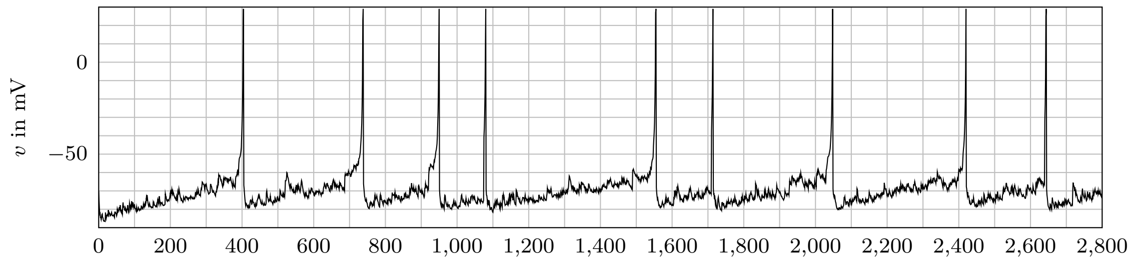


Figure 3.1: Membrane voltage of an Izhikevich neuron over time. Each simulation time step equals 0.25 milliseconds.

In order to derive the eligibility trace, the hidden state of Izhikevich neurons is now defined as a two-dimensional vector containing v_j^t and u_j^t.

$$\mathbf{s}_j^t = \begin{pmatrix} v_j^t \\ u_j^t \end{pmatrix} \tag{3.7}$$

The state derivative is then the following 2x2 matrix, which can be simplified assuming binary values for z_j^t.

$$\begin{aligned}
\frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} &= \begin{pmatrix} \frac{\partial v_j^{t+1}}{\partial v_j^t} & \frac{\partial v_j^{t+1}}{\partial u_j^t} \\ \frac{\partial u_j^{t+1}}{\partial v_j^t} & \frac{\partial u_j^{t+1}}{\partial u_j^t} \end{pmatrix} \\
&= \begin{pmatrix} 1 - z_j^t + 0.08\delta t(v_j^t - (v_j^t + 65)z_j^t)(1 - z_j^t) + 5\delta t(1 - z_j^t) & -\delta t \\ 0.004\delta t(1 - z_j^t) & 1 - 0.02\delta t \end{pmatrix} \\
&= \begin{pmatrix} 1 - z_j^t + 0.08\delta t v_j^t(1 - z_j^t) + 5\delta t(1 - z_j^t) & -\delta t \\ 0.004\delta t(1 - z_j^t) & 1 - 0.02\delta t \end{pmatrix} \\
&= \begin{pmatrix} (1 - z_j^t)(1 + (0.08v_j^t + 5)\delta t) & -\delta t \\ 0.004\delta t(1 - z_j^t) & 1 - 0.02\delta t \end{pmatrix}
\end{aligned} \tag{3.8}$$

Given this, the eligibility vector computes to the following form:

$$\begin{aligned}
\boldsymbol{\epsilon}_{i,j}^{t+1} &= \frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} \cdot \boldsymbol{\epsilon}_{i,j}^t + \frac{\partial \mathbf{s}_j^{t+1}}{\partial \theta_{ji}^{rec}} \\
&= \begin{pmatrix} (1 - z_j^t)(1 + (0.08v_j^t + 5)\delta t) & -\delta t \\ 0.004\delta t(1 - z_j^t) & 1 - 0.02\delta t \end{pmatrix} \cdot \begin{pmatrix} \epsilon_{i,j,v}^t \\ \epsilon_{i,j,u}^t \end{pmatrix} + \begin{pmatrix} \delta t z_i^t \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} (1 - z_j^t)(1 + (0.08v_j^t + 5)\delta t)\epsilon_{i,j,v}^t - \delta t \epsilon_{i,j,u}^t + \delta t z_i^t \\ 0.004\delta t(1 - z_j^t)\epsilon_{i,j,v}^t + (1 - 0.02\delta t)\epsilon_{i,j,u}^t \end{pmatrix}
\end{aligned} \tag{3.9}$$

As shown in Equation 3.9, the recovery eligibility vector $\epsilon_{i,j,u}^t$ is an exponential filter of the voltage eligibility vector just like for ALIF neurons. In contrast to ALIF neurons, the voltage eligibility vector $\epsilon_{i,j,v}^t$ is not separated from the recovery part. Instead, whenever an Izhikevich neuron spikes, its voltage eligibility vector is reset to the negative recovery eligibility vector.

In order to derive the final eligibility trace, a pseudo-derivative similar to the one for LIF neurons is used:

$$h_j^t = \gamma\ max(0, 1 - |\frac{v_j^t - 30}{30 - 2c}|) \tag{3.10}$$

With this pseudo-derivative, the neuron spike z_j^t derived by the hidden state $\mathbf{s}_j^t$ is then defined as:

$$\begin{pmatrix} \frac{z_j^t}{v_j^t} \\ \frac{z_j^t}{u_j^t} \end{pmatrix} \overset{\text{def}}{=} \begin{pmatrix} h_j^t \\ 0 \end{pmatrix} \tag{3.11}$$

The eligibility trace then simply computes to the pseudo-derivative times the voltage eligibility vector.

$$e_{ji}^{t+1} = \frac{\partial z_j^{t+1}}{\partial \mathbf{s}_j^{t+1}} \cdot \boldsymbol{\epsilon}_{i,j}^{t+1} = \begin{pmatrix} h_j^{t+1} & 0 \end{pmatrix} \begin{pmatrix} \epsilon_{i,j,v}^{t+1} \\ \epsilon_{i,j,a}^{t+1} \end{pmatrix} = h_j^{t+1} \epsilon_{i,j,v}^{t+1} \tag{3.12}$$

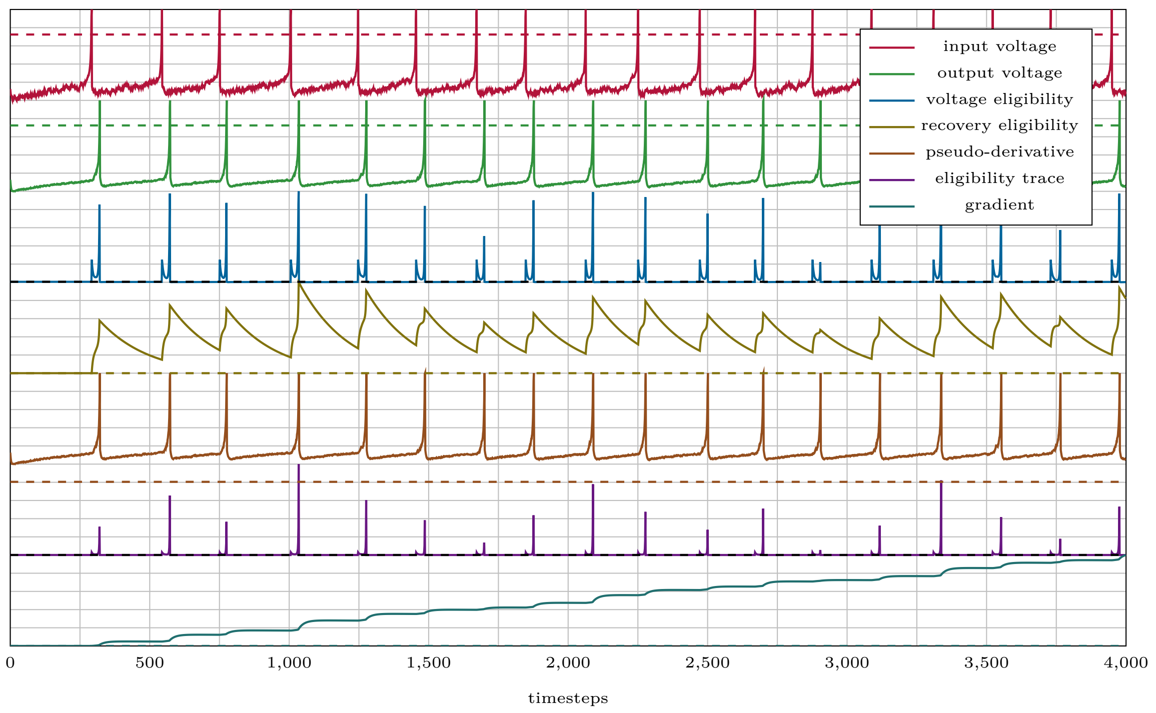


Figure 3.2: Simulation of a positive rewarded STDP behavior of two connected Izhikevich neurons, where the output neuron fires shortly after the input neuron. As a result, a gradient computed with a constant positive learning signal increases. Zero is marked with a dashed line in each plot.

To inspect the evolution of the derived eligibility traces, two Izhikevich neurons are weakly connected by a synapse and receive random inputs to simulate the behavior within a greater network.

To visualize the influence of the eligibility trace on the gradient, a constant positive learning signal is used, and the gradient is calculated based on Equation 2.31.

In the simulation shown in Figure 3.2, an artificial strengthening STDP behavior is introduced by using an overall lower random input current for the output neuron. An output spike shortly after an input spike is then ensured by steady increasing the random input current for the output neuron after the input neuron spiked. This positive STDP behavior is reflected within the increasing gradient.

In Figure 3.3 a synaptic weakening STDP behavior is induced, by using a higher random input current for the output neuron. An input spike shortly after an output spike is then ensured by steady increasing the random input current for the input neuron after the output neuron spiked.

In contrast to the positive STDP behavior, where the voltage and refractory

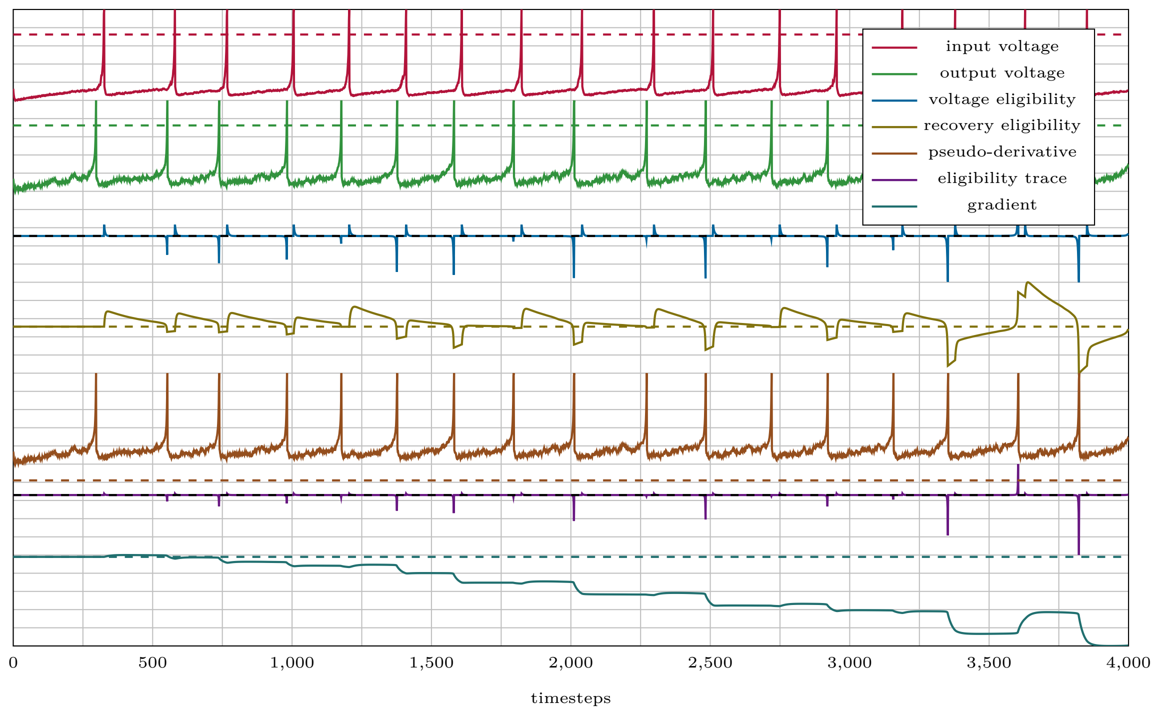


Figure 3.3: Simulation of a negative rewarded STDP behavior of two connected Izhikevich neurons, where the input neuron fires shortly after the output neuron. As a result, a gradient computed with a constant positive learning signal decreases. Zero is marked with a dashed line in each plot.

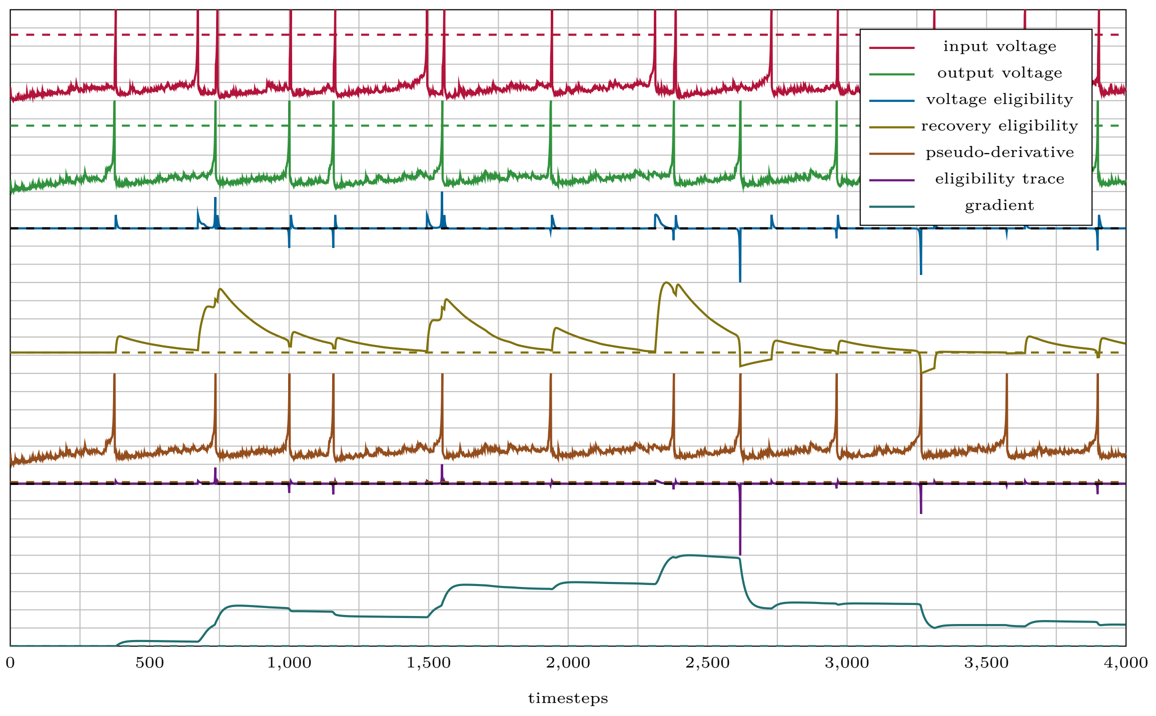


Figure 3.4: Simulation of two connected Izhikevich neurons with uncorrelated firings, where both positive and negative rewarded STDP events occur. As a result, a gradient computed with a constant positive learning signal tends to decay back to zero. Zero is marked with a dashed line in each plot.

eligibility are positive, here the voltage and refractory eligibility get negative and the tendency to weaken the synapse is clearly reflected in the decreasing gradient.

For neurons that spike uncorrelated, Figure 3.4 shows that the gradient fluctuates and tends to go back to zero.

While a general tendency to reflect STDP behavior in gradients based on the eligibility traces of Izhikevich neurons can be discovered, it is not completely obvious how specific STDP events emerge within the Izhikevich eligibility vector equations.

3.2 STDP-LIF Neuron

While a STDP influenced gradient can not be observed with the standart equations for LIF or ALIF neurons within an LSNN, by slightly modifying the original LIF equation (3.13), a clear STDP based eligibility trace emerges.

$$v_j^{t+1} = \alpha v_j^t + I_j^t - z_j^t v_{thr} \tag{3.13}$$

In order to compute such an eligibility trace that reflects the STDP behavior of the synapse connecting two LIF neurons, the LIF equation (3.13) has to be slightly altered into what is now called a STDP-LIF neuron:

$$v_j^{t+1} = \alpha v_j^t + I_j^t - z_j^t \alpha v_j^t - z_j^{t-\delta t_{ref}} \alpha v_j^t \tag{3.14}$$

Instead of using a soft reset about the fixed threshold, in Equation 3.14 the STDP-LIF neuron is hard reset to zero whenever it spikes, and whenever its refractory period ends (δt_{ref} being length of the refractory time period). Since the reset does include the voltage v_j^t it is now included into hidden state derivative and therefore into the computation of the eligibility trace:

$$\frac{\partial v_j^{t+1}}{\partial v_j^t} = \alpha - z_j^t \alpha - \alpha z_j^{t-\delta t_{ref}} = \alpha(1 - z_j^t - z_j^{t-\delta t_{ref}}) \tag{3.15}$$

$$\boldsymbol{\epsilon}_{i,j}^{t+1} = \frac{\partial \mathbf{s}_j^{t+1}}{\partial \mathbf{s}_j^t} \cdot \boldsymbol{\epsilon}_{i,j}^t + \frac{\partial \mathbf{s}_j^{t+1}}{\partial \theta_{ji}^{rec}} = \alpha(1 - z_j^t - z_j^{t-\delta t_{ref}})\boldsymbol{\epsilon}_{i,j}^t + z_i^t \tag{3.16}$$

$$e_{ji}^{t+1} = \frac{\partial z_j^{t+1}}{\partial \mathbf{s}_j^{t+1}} \cdot \boldsymbol{\epsilon}_{i,j}^{t+1} = h_j^{t+1} \boldsymbol{\epsilon}_{i,j}^{t+1} \tag{3.17}$$

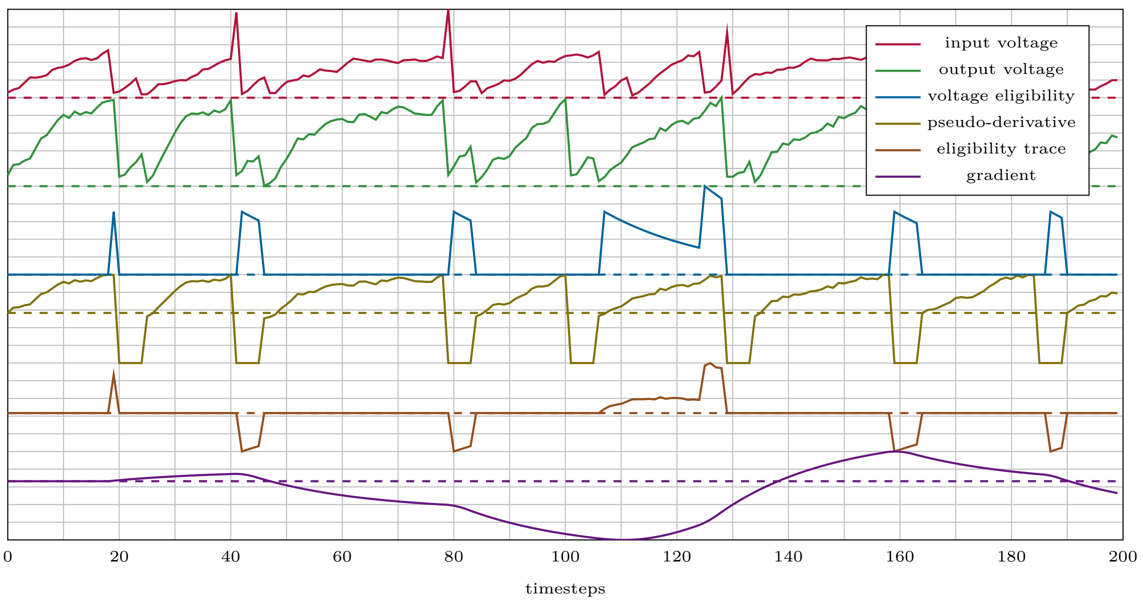


Figure 3.5: Simulation of two connected STDP-LIF neurons with uncorrelated spikes. STDP events are clearly shown in the eligibility trace, where first a positive rewarded output after input spike occurred, followed by two negative rewarded input after output spikes. The uncorrelated spiking is also reflected in the gradient, which tends to decay back to zero. Zero is marked with a dashed line in each plot.

This effectively resets the eligibility trace after a spike and after the refractory period.

An eligibility trace that reflects the STDP behavior of its synapse can now be introduced by using a constant negative pseudo-derivative during the refractory period and otherwise leave the neuron dynamics unchanged. Any incoming spike during the refractory period now produces a negative eligibility trace that persists for the time of the refractory period and has a negative influence on the gradient.

This STDP influenced gradient is shown in Figure 3.5 where two connected STDP-LIF neurons receive random input in order to produce uncorrelated spikes. Strengthening and weakening STDP events can be clearly identified in the eligibility trace and directly influence the resulting gradient. Thus by simple modifications to the LIF neuron, one can derive eligibility traces for STDP-LIF neurons that reflect the STDP behavior of the underlying synapse.

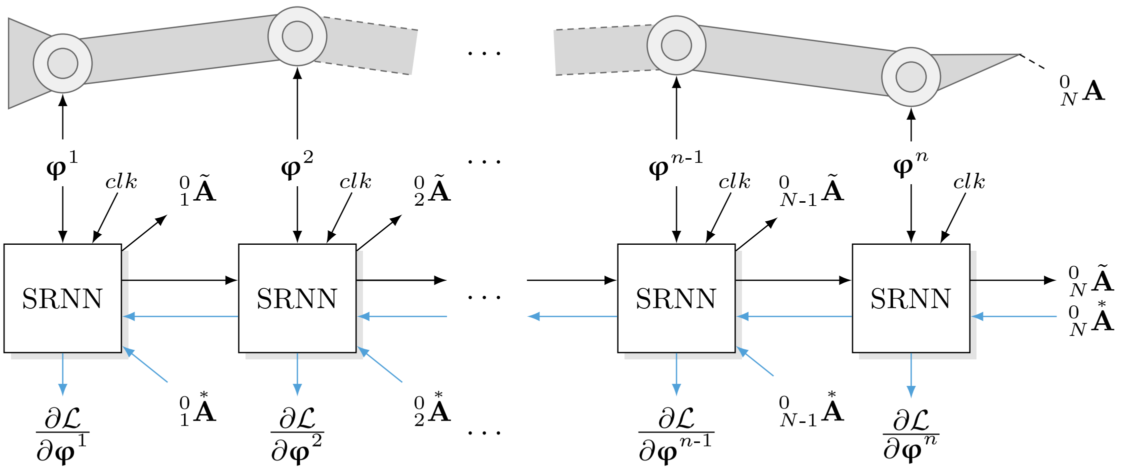
Since gradients which are computed using eligibility traces together with a back-propagated learning signal are mathematically equivalent to normal BPTT, it can be argued that BPTT itself facilitates STDP behavior.

Chapter 4

Predictive Forward Model

The basis for controlling the movement of a many-joint robotic arm through action inference is a forward model that learns to predict the endeffector pose of the arm, based on the angles for each joint. Input angles that direct the arm to a desired goal position can be inferred using such a trained forward model by back-propagating the discrepancy between the predicted and the target position. By using the back-propagated input errors, the joint angles can be adjusted using Gradient Descent to navigate the endeffector to reach a target position.

This chapter outlines the design, implementation and training of an LSNN capable of predicting the position and orientation of such a many-joint robotic arm.



$^{0}_{N}\mathbf{A}$
$\boldsymbol{\varphi}^1$
clk
$^{0}_{1}\tilde{\mathbf{A}}$
$\boldsymbol{\varphi}^2$
clk
$^{0}_{2}\tilde{\mathbf{A}}$
$\boldsymbol{\varphi}^{n\text{-}1}$
clk
$^{0}_{N\text{-}1}\tilde{\mathbf{A}}$
$\boldsymbol{\varphi}^n$
clk
SRNN
SRNN
SRNN
SRNN
$^{0}_{N}\tilde{\mathbf{A}}$
$^{0}_{N}\overset{*}{\mathbf{A}}$
$^{0}_{1}\overset{*}{\mathbf{A}}$
$^{0}_{2}\overset{*}{\mathbf{A}}$
$^{0}_{N\text{-}1}\overset{*}{\mathbf{A}}$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\varphi}^1}$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\varphi}^2}$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\varphi}^{n\text{-}1}}$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\varphi}^n}$

Figure 4.1: Draft of a Recurrent Spiking Forward Model. Within the time window τ, the network gets the joint angles φ^n and an optional clock (clk) signal as inputs and predicts the joint position $^{0}_{n}\tilde{A}$. Calculating the joint angles for a desired endeffector position $^{0}_{N}\overset{*}{A}$ is done by extracting a temporal gradient for the error signal $\mathcal{L}$, which represents the difference between a desired joint position $^{0}_{N}\overset{*}{A}$ and its current estimation $^{0}_{N}\tilde{A}$. Image adapted from Otte *et al.* (2017b).

4.1 Methods

The basic predictive forward model is outlined in Figure 4.1. The network is an LSNN with 2 inputs representing x and z angles (y being the up-direction) and 9 outputs representing the (x, y, z) position and orientation (up- and x-direction vector). Additionally the network can receive one or several clock inputs, in order to synchronize the output calculation and therefore improve the prediction.

The network computation is splitted into several time windows (τ). During each window the network receives the x and y angles of the currently processed joint φ^n, and outputs its position and orientation ${}^{0}_{n}\tilde{A}$.

The outputs are calculated as a mean of the readout neurons for a specific time $\tau_{out} < \tau$ at the end of a time window. If the network receives clock inputs, they are given during τ_{out}. In order to calculate the position and orientation for the endeffector of a robotic arm, the network is run for N time windows, and the joint angles are successively inputted into the network until it outputs the endeffector position and orientation ${}^{0}_{N}\tilde{A}$ during the last time window.

4.1.1 Implementation

For this thesis an LSNN was implemented using C++ 11 as main programming language and CUDA 9.2 for performance critical parts. As shown in Figure 4.2 an object orientated class hierarchy was used, where abstract blueprints for neurons, synapses, gradients, optimizers and networks define the base functionality of the framework.

The neuron base class for example defines access functions a subclass needs to implement in order to access the current action potential, or check whether a neuron has fired during the current time step. Also a function that returns the value of the pseudo-derivative needs to be implemented by a subclass.

A synapse class is composed of a presynaptic and postsynaptic neuron, and has to provide functionality in order to calculate the eligibility trace of the implemented synapse type, which is determined by the postsynaptic neuron type. Gradients are then computed for one synapse each, and need to provide a function that computes and returns this gradient. An optimizer class then uses a gradient in order to calculate a weight update for the synapse of that gradient.

The network class manages the interplay of neurons, synapses, gradients and optimizers. Therefore a mandatory update function has to be implemented in all subclasses in order to perform one simulation step for the class implementing it. This way, the BasicNetwork class computes one complete simulation time step by first updating input neurons and input synapses, then updating hidden neurons and hidden synapses followed by readout synapses and readout neurons. After the neuron and synapse updates, the gradients are updated and afterwards the optimizer's update functions are called in order to update the synaptic weights.

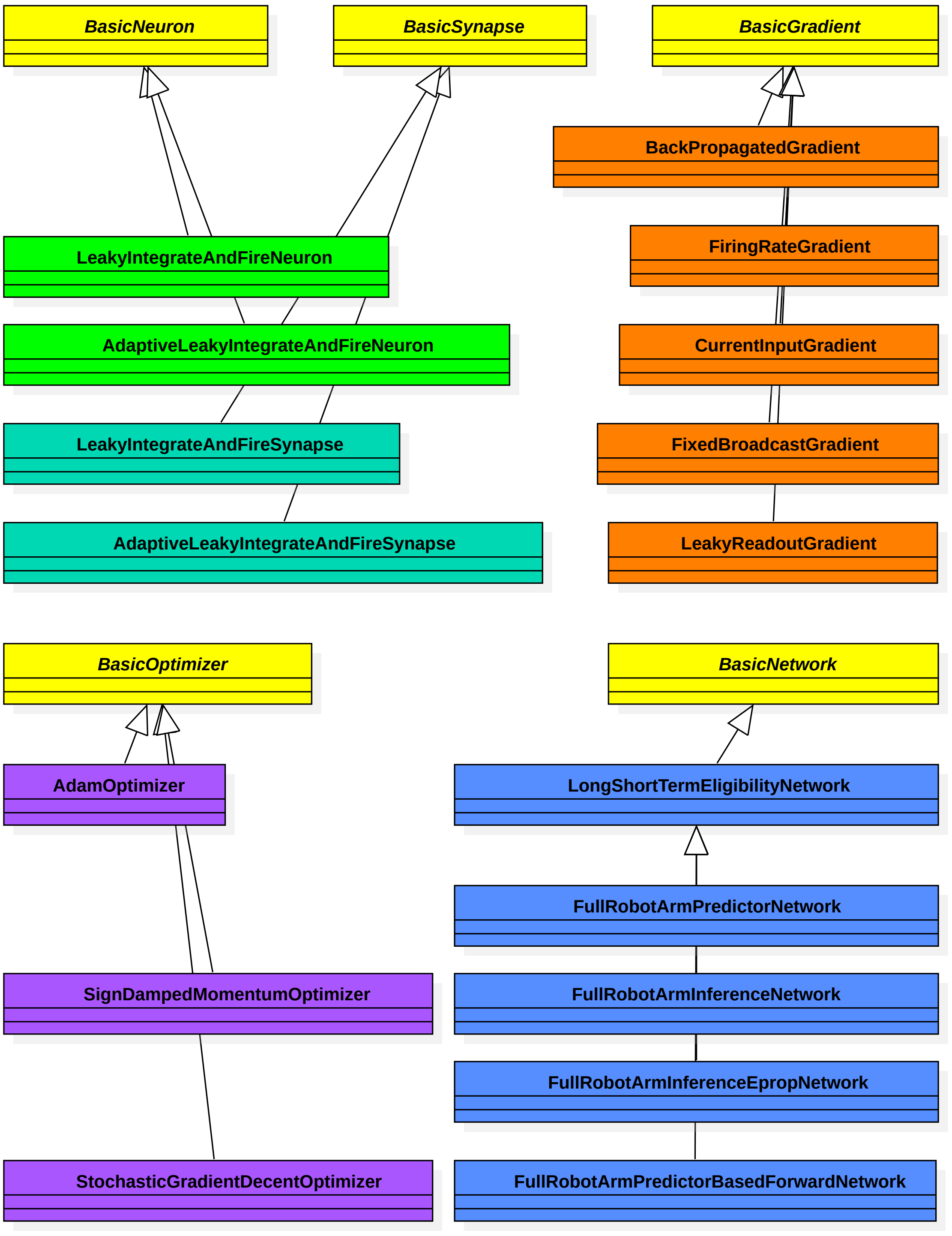

BasicNeuron
BasicSynapse
BasicGradient
BackPropagatedGradient
LeakyIntegrateAndFireNeuron
FiringRateGradient
AdaptiveLeakyIntegrateAndFireNeuron
CurrentInputGradient
LeakyIntegrateAndFireSynapse
FixedBroadcastGradient
AdaptiveLeakyIntegrateAndFireSynapse
LeakyReadoutGradient
BasicOptimizer
BasicNetwork
AdamOptimizer
LongShortTermEligibilityNetwork
FullRobotArmPredictorNetwork
SignDampedMomentumOptimizer
FullRobotArmInferenceNetwork
FullRobotArmInferenceEpropNetwork
StochasticGradientDecentOptimizer
FullRobotArmPredictorBasedForwardNetwork

Figure 4.2: Class diagram showing the most important classes of the implemented LSNN framework. **Yellow**: Abstract base classes defining blueprints and providing implementations of common functionality. **Green**: Neurons. **Cyan**: Synapses. **Orange**: Gradients. **Purple**: Optimizers. **Blue**: Networks.

The described abstract classes provide a flexible and extensible neural network implementation suitable for exploring different neuron and eligibility designs, like the STDP-LIF neuron or the Izhikevich neuron, which were introduced in the previous chapter.

In order to implement an LSNN, several subclasses are implemented, for example the LIF and ALIF neuron and their corresponding synapses.

The basic functionality of the LSNN is implemented in the class LongShortTermEligibilityNetwork from which specific network classes are derived to perform the different experiments within this thesis.

The described framework is optimized for readability and builds an LSNN in a simple and understandable way. Therefore it is especially not optimized for speed, since this would require storing and computing with large matrices, which are not as easy to understand and maintain as dedicated classes for neurons, synapses, gradients and optimizers.

In the used BasicNetwork class, a simple list off all neurons, synapses, gradients and optimizers is kept and updated. The connectivity of the network can then be defined by child classes. The dedicated classes then also allow an easier debugging.

In order to train an LSNN, a separate speed improved CUDA framework was developed, that implements the basic functionality of computing a fully connected LSNN simulation with up to 1024 hidden neurons.

Separate CUDA kernels for the different experiments performed in this thesis are implemented.

The main kernels for BPTT and eligibility based gradients are shown in Figure 4.3.

The redundant implementation of neurons, synapses and gradients provides a simple way of developing and testing specialized CUDA kernels by comparing GPU results to those computed with the CPU based framework.

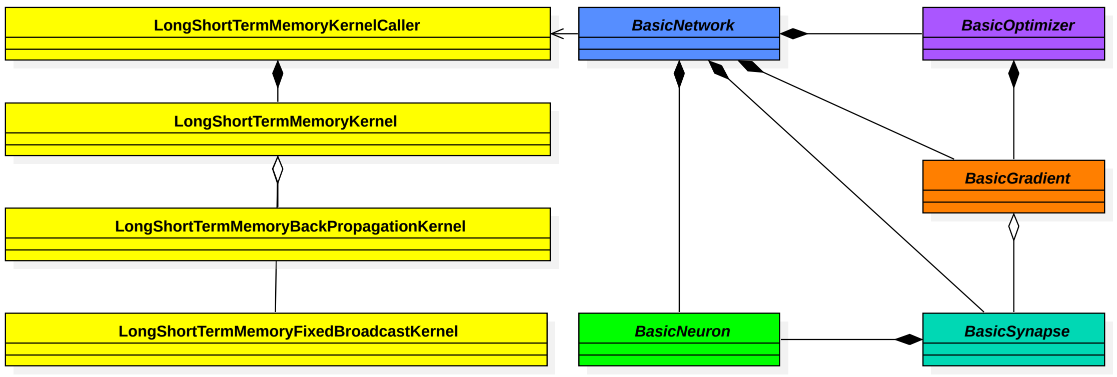

LongShortTermMemoryKernelCaller
LongShortTermMemoryKernel
LongShortTermMemoryBackPropagationKernel
LongShortTermMemoryFixedBroadcastKernel
BasicNetwork
BasicOptimizer
BasicGradient
BasicNeuron
BasicSynapse

Figure 4.3: Composition diagram: Performance critical computation is redundant implemented and can be run either on the CPU (**green**, **cyan**, **orange**) or on the GPU (**yellow**).

While an optimizer implementation would also benefit the parallel computing capabilities of the GPU, in tests the actual execution time of the optimizers on the CPU compared to the whole network simulation on the GPU was negligible and thus the optimizers were not implemented in CUDA.

4.1.2 Input Encoding

The simplest way to input data into an LSNN is to directly use real values and interpret them as an input "current" that is injected into the network. A more biologically plausible way is to encode the inputs using spikes. One way to do this is by using place codes as proposed in Bellec *et al.* (2018).

$$r_i = r_{max}\ e^{-100(\zeta_i - \zeta)^2} \tag{4.1}$$

In this place code, several input neurons are assigned to one input. Each of those input neurons has a value specific firing rate r_i which is determined by Equation 4.1, where ζ_i is the value the neuron i is tuned to.

4.1.3 Training

In order to train the predictive forward network, two types of 3D simulations were created, as shown in Figure 4.4.

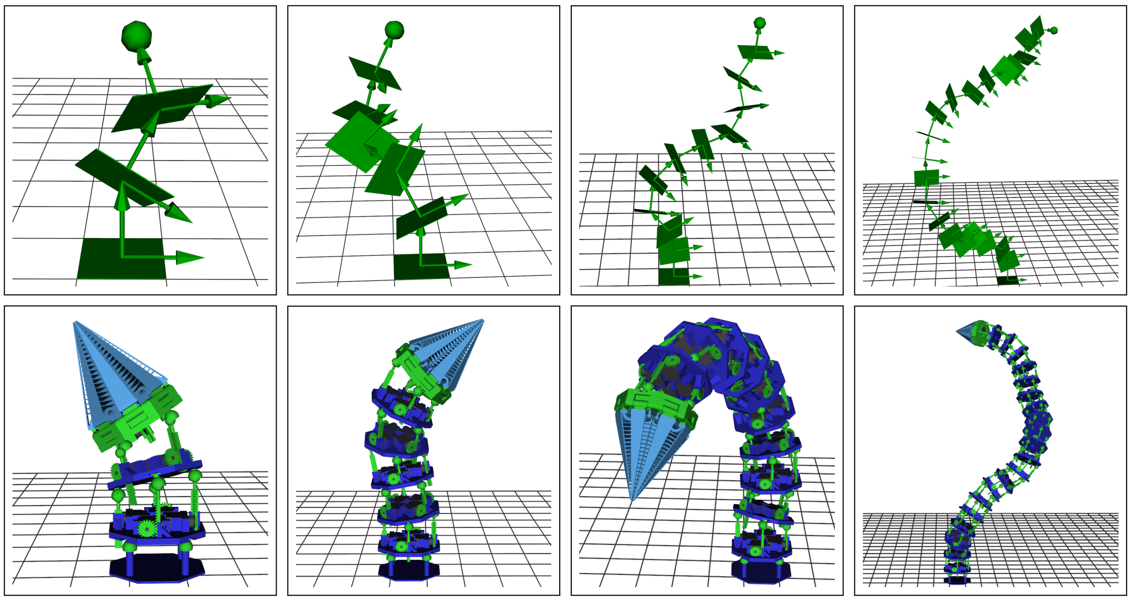


Figure 4.4: 3D-Simulations of a many-joint robotic arm with 2, 5, 10 and 20 joints. Mathematical simulation showing positions and orientations (top) and realistic CAD based simulation (bottom).

The simple 3D simulation is based on a mathematical model of an idealized many-joint robotic arm, with per joint x-, z-rotations and a y-translation. In this simulation the joint positions are represented by a flat cube, while an arrow embedded within the cube represents the relative x-direction and the arrow pointing to the next cube represents the relative y-direction (up-vector). The sphere at the top of the robot represents the endeffector position.

The second simulation is based on the CAD files of an actual robot arm, with a joint diameter of 100mm and an average joint distance of around 58mm. The movement of each joint is controlled by four small servo motors, that each drives a linear gear. This configuration allows for any x-, z-rotations of a joint with maximum angles of ± 20 degrees. The endeffector of the robot is a gripper controlled by another small servo.

While the CAD based simulation is not physically plausible, it does not account for masses, forces or object collisions, the general dynamic is restricted to ”plausible” movements. This means that object interactions of directly connected CAD parts are taken into account. So additionally to a fixed y-translation, a joint also performs small x- and z-translations based on the specific x- and z-angles in order to correct for CAD part interactions. Also the endeffector y-translation (around 126mm) is different from the joint y-translations (58mm).

In order to be comparable to the realistic model, the distance between two joints in the mathematical simulation is set to 80mm, while the joint's cube diameter is also set to 80mm.

During training, new target arm positions are generated for each batch by randomly choosing joint x- and z-angles and computing the corresponding joint positions and orientations. Also with a certain probability, edge-cases which are consecutive sharp turns in the same direction, are included into the training samples of each batch.

Orientation vectors are directly used as targets, while position coordinates are normalized by dividing them with the distance between two joints.

When computing input-output pairs, it is important to group the input angles with the output position and orientation that are directly affected by those inputs. Figure 4.5 shows this dependency.

A given joint's x- and z-rotation with angles (a_x^i, a_z^i) directly affects the directions v_x^i, v_y^i and the position p^{i+1}. So $(a_x^i, a_z^i) \rightarrow (p^{i+1}, v_x^i, v_y^i)$ is the easiest grouping that enables the learning of an input-output relationship.

The effect of any other input-output grouping is that the network has to remember the affected outputs longer than one time window and already gets new inputs while it calculates outputs depending on previous inputs. For example, if one chooses the naive input-output grouping of $(a_x^i, a_z^i) \rightarrow (p^i, v_x^i, v_y^i)$ the output p^i is actually computed using the angles (a_x^{i-1}, a_z^{i-1}). Thus the network has to remember those inputs while already getting the new inputs (a_x^i, a_z^i), which are needed in order to compute v_x^i, v_y^i.

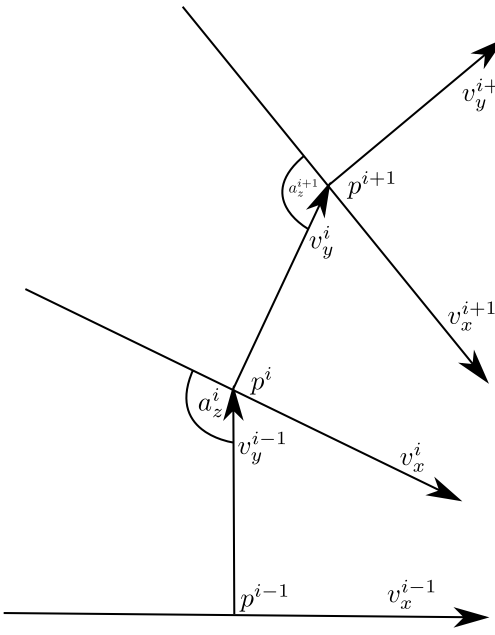


Figure 4.5: Schematic representation of the dependencies between input angles and output position and orientation. Using angles a_x^i (not shown) and a_z^i one can compute position p^{i+1} and the x- and y-orientation v_x^i, v_y^i.

4.2 Results

The Networks presented in this thesis were trained on either a Nvidia GTX-1070 Desktop-GPU or a Nvidia RTX-2070 Laptop GPU. While LSNNs with 1024 hidden neurons could theoretically be trained using the developed framework, due to the very long training time, only LSNNs up to 256 hidden neurons were considered for the majority of experiments.

In order to compare results, the main metric used in this thesis is an Euclidean-distance error $||\mathbf{E}||_2$, which for predictive forward networks is defined as the mean Euclidean-distance between the actual joint positions and the predicted joint positions.

$$||\mathbf{E}||_2 \overset{\text{def}}{=} \sum_i \sqrt{||\overset{*}{p}{}^i - p^i||_2} = \sum_i \sqrt{(\overset{*}{p}{}_x^i - p_x^i)^2 + (\overset{*}{p}{}_y^i - p_y^i)^2 + (\overset{*}{p}{}_z^i - p_z^i)^2} \tag{4.2}$$

The $||\mathbf{E}||_2$ error which is given in millimeters allows an intuitive comparison, and is a direct measure of the real accuracy of the compared models.

4.2.1 Training

If not stated otherwise, networks were trained using BPTT with a batch size of 128 and the Adam optimizer with a learning rate of 0.001.

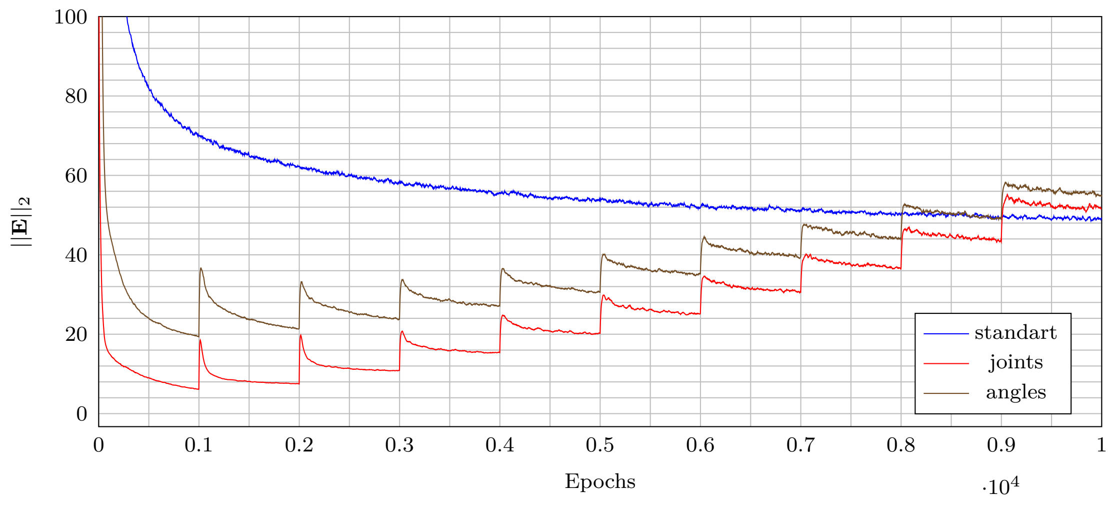


Figure 4.6: Comparison of curriculum learning with standard learning on an LSNN with 128 hidden neurons on a 10-joint robotic arm with max angles of 45 degrees. For angle based curriculum learning, the max training angles started by 10% of the final angle and increased about 10% every 1000 epochs. For joint based curriculum learning, learning started with one joint and increased about one joint every 1000 epochs. Plots were averaged of 5 training runs with independent initialized networks.

For the time window τ 12ms are used, which are 12 simulation time steps, and for the output decoding 7ms are used.

Since for each batch a new set previously not seen input-output pairs are computed, training epochs in this thesis refer to the number of weight updates performed during training.

In contrast to Otte *et al.* (2018), where curriculum learning was used to train the most accurate models, curriculum learning did not improve the prediction accuracy of an LSNN.

Figure 4.6 shows two separate curriculum learning approaches together with a standard training run. The angle based approach is adapted from Otte *et al.* (2018), and uses increasingly greater angles during training. The second approach steadily increases the number of joints of the robot arm, and therefore has the advantage of shorter simulation times during most of the training. Nevertheless, both approaches on average fail to reach the accuracy for a standard training run with full angles and joints from the start, regarding the same number of overall training epochs.

While e-prop manages to achieve reasonable accuracies when training a predictive forward LSNN (see Figure 4.7), BPTT generally outperforms e-prop, and was thus mainly used to train LSNNs.

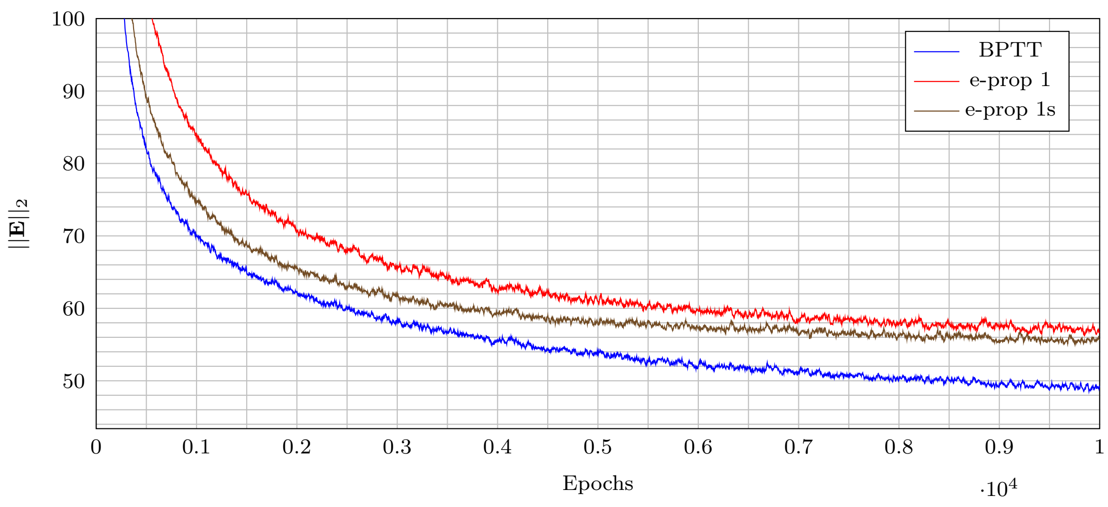


Figure 4.7: Comparison of e-prop with BPTT for a predictive forward LSNN with 128 hidden neurons on a 10-joint robotic arm with max angles of 45 degrees. While e-prop shows a slower convergence than BPTT, it also manages to achieve an acceptable accuracy. Using biologically plausible random feedback weights (e-prop 1) only slightly decreases the performance in comparison to symmetric feedback weights (e-prop 1s). Plots were averaged over 5 training runs using independently initialized LSNNs.

4.2.2 Input Encoding

Figure 4.8 (left) shows the comparison of an LSNN trained using a place code with 40 input neurons per input value, as described in Bellec *et al.* (2018).

Despite being biologically plausible, the place code shows a slower convergence and overall accuracy. Also since it uses more computational resources for other tests

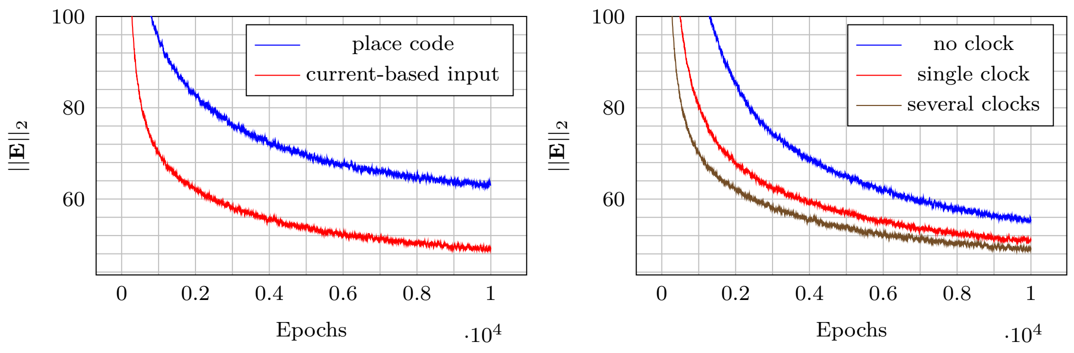


Figure 4.8: Left: Place code vs current-based input encoding for a predictive forward LSNN with 128 hidden neurons. **Right**: Effect of additional clock inputs on the performance of a predictive forward LSNN with 128 hidden neurons. Plots were averaged over 5 training runs using independently initialized LSNNs.

performed in this thesis, if not stated otherwise, the current-based input encoding is used.

The right plot of Figure 4.8 shows that a clock input also improves the overall performance of the predictive forward network. While a significant improvement can already be achieved by adding a third input neuron supplying a constant input current during output calculations within τ_{out}. The performance still slightly increases if one adds a separate clock neuron for each joint, which only supplies a clock input during the readout time for the pose estimation of that specific joint.

If not other wise stated, this approach of one clock neuron per joint is used in following experiments.

4.2.3 Accuracy

The accuracy of an LSNN predicting the position and orientation of the joints of a robotic arm heavily depends on the length of the simulated arm as well as on the maximum angles each joint can move as shown in Figure 4.9.

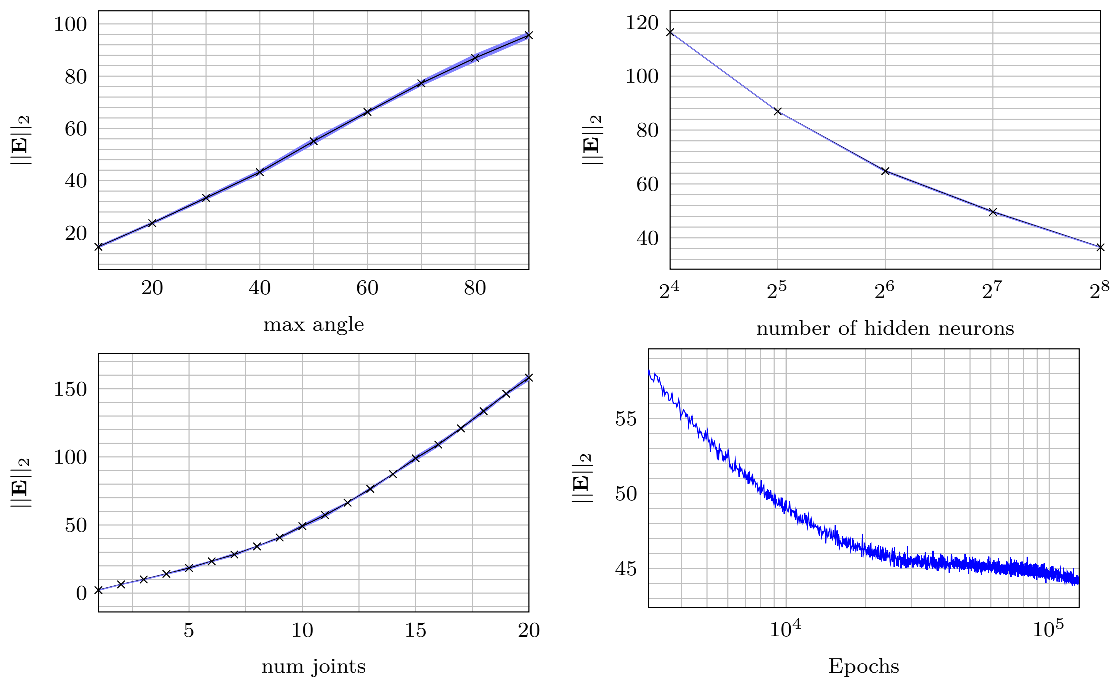


Figure 4.9: Evaluation of different predictive forward LSNNs. **Top**: Different number of angles/hidden neurons for a mathematical robot arm with 10 joints. **Bottom-Left**: LSNNs trained on mathematical arms with different number of joints. **Bottom-Right**: Euclidean-training error over 150,000 epochs, averaged over 5 runs. Confidence intervals are based on 5 training runs of LSNNs with 128 hidden neurons over 10,000 epochs and calculated using the Student-t ($\alpha = 5\%$) distribution.

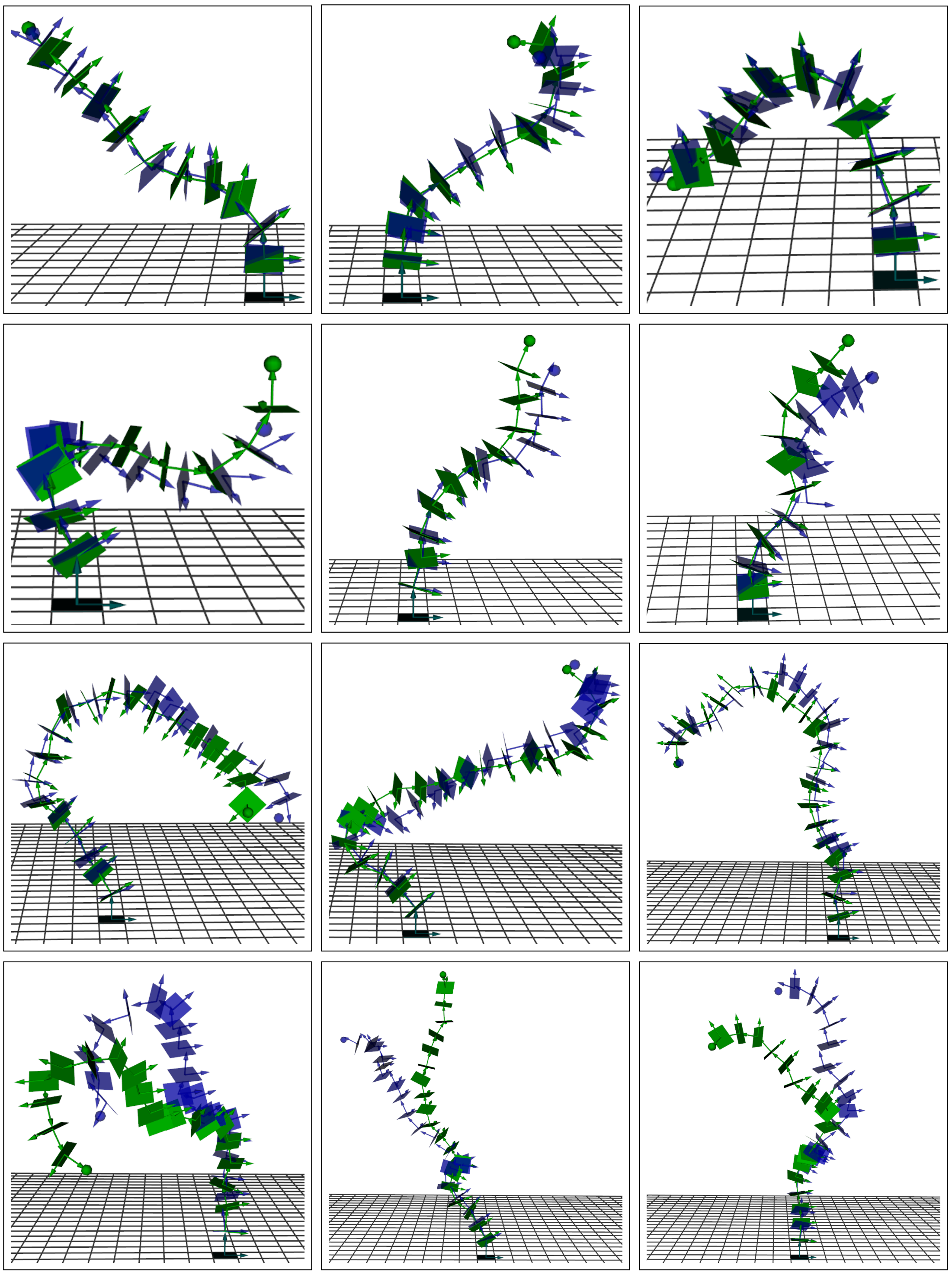


Figure 4.10: Trained predictive forward LSNN with 128/256 hidden neurons for 10-/20-joint arms with ±45 degree angles and final $||\mathbf{E}||_2$ accuracies of around 43mm/96mm. **From top to bottom**: Average predictions followed by fail-cases.

Also the number of hidden neurons used greatly affects the prediction accuracy. Acceptable performance can be achieved with 10,000 to 30,000 epochs of training, but the LSNN still continues to improve after 100,000 epochs as shown in Figure 4.9 (bottom-right).

While the network generally achieves a reasonable accuracy when trained with random arm positions, the performance significantly drops on edge-cases, as shown in Figure 4.10. The network usually fails on arm positions involving very sharp turns or when two consecutive joints have very dissimilar angles.

Another problem is that errors add up and increase for joints calculated later during the simulation. While the accuracy seems to decrease linearly with greater angles, this is not the case for an increased number of joints. Since the LSNN processes the joint angles sequentially, the prediction of the n-th joint depends on the prediction of the previous joint, therefore the predictions successively get worse for later joints. Nevertheless, on average accurate predictions can be achieved even for 20-joint arms (Figure 4.10 top row).

4.2.4 Realistic Simulation

Training an LSNN to predict the arm pose of a robotic arm based on the simulation of an actual robot using CAD files should be harder than the mathematical model, since interactions between the different CAD parts cause slight translations of the position of each joint depending on the x and z angles. Those translations amplify

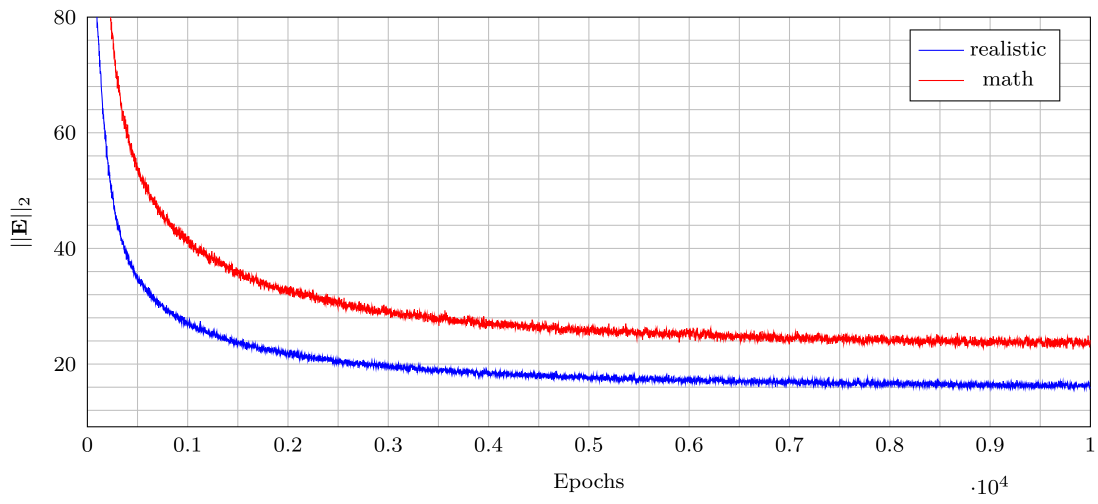


Figure 4.11: LSNN trained to predict the pose of a 10-joint realistic robotic arm simulation in comparison with an LSNN trained to predict the mathematical model of a 10-joint robotic arm with max angles of 20 degrees. 128 hidden neurons were used for both LSNNs. Plot shows the average error of 5 independent runs for each simulation type.

over the length of the robotic arm, and thus the endeffector reaches a significant different target for the same joint angles as the mathematical model. On the other hand, since the realistic arm can only use joint angles up to 20 degrees, the prediction should be easier as the mathematical model for significant higher angles. Also the distance between each joint is less than in the mathematical model (58mm compared to 80mm). This means that the realistic arm has a smaller effective range as an arm based on the mathematical model with the same number of joints and the same maximum angle of 20 degrees, which again could make it easier for the LSNN to predict the pose of the arm.

As it turned out, an LSNN has no problem learning the dynamics of the realistic model, and indeed exceed the accuracy of a mathematical model with the same number of joints and maximum angle degree, as shown in Figure 4.11.

Chapter 5

Inference

This chapter goes into the details of action inference using a biologically plausible Long Short-Term Spiking Neural Network (LSNN).

By calculating a temporal error gradient for the inputs of a forward model predicting the pose estimation of a many-joint robotic arm, it is possible to navigate through joint space, which enables the controlled movement of the robot to a desired goal position.

5.1 Methods

The main procedure of action inference uses BPTT, but this chapter also discusses an e-prop like biologically plausible approach and compares it to a direct model that calculates the joint angles given a desired goal position.

5.1.1 Back-Propagation Through Time

As described in Section 2.2.4, an input error for a given time step can be derived by weighting the back-propagated delta errors $^{d\mathbf{E}}/_{ds_i^t}$ with the corresponding input weights (see Equation 2.29).

In order to compute a meaningful input gradient that allows the control of a many-joint robotic arm through gradient descent, one needs to know the current pose and joint angles of the robotic arm. The next step is then to feed them successively into a pre-trained LSNN that predicts the pose estimation for each joint and outputs the endeffector position and orientation during its last time window. An input gradient that guides the endeffector towards a desired goal is then computed by supplying the LSNN with the goal position and an optional orientation as the endeffector target.

The error between the predicted pose and the desired target pose is back-propagated through time to derive an input error, which can then be minimized.

Through means of Gradient Descent or related methods, new input angles for each joint can be computed, which move the robotic arm closer to the goal position.

By repeating the described procedure with the newly computed joint angles, one successively moves the endeffector closer to its desired goal position.

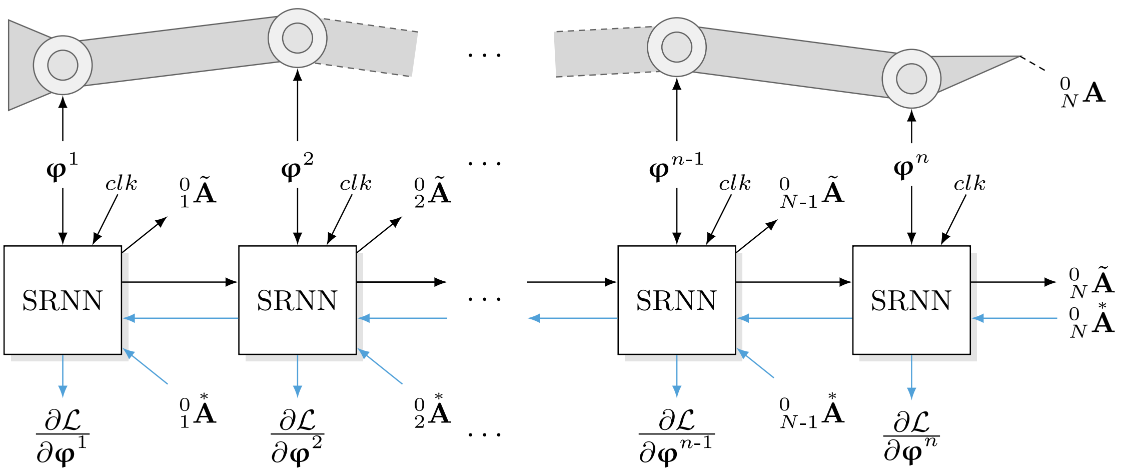

$\boldsymbol{\varphi}^1$
$\boldsymbol{\varphi}^2$
$\boldsymbol{\varphi}^{n\text{-}1}$
$\boldsymbol{\varphi}^n$
clk
${}^{0}_{1}\tilde{\mathbf{A}}$
${}^{0}_{2}\tilde{\mathbf{A}}$
${}^{0}_{N\text{-}1}\tilde{\mathbf{A}}$
${}^{0}_{N}\tilde{\mathbf{A}}$
${}^{0}_{N}\mathbf{A}$
${}^{0}_{N}\overset{*}{\mathbf{A}}$
${}^{0}_{1}\overset{*}{\mathbf{A}}$
${}^{0}_{2}\overset{*}{\mathbf{A}}$
${}^{0}_{N\text{-}1}\overset{*}{\mathbf{A}}$
SRNN
$\frac{\partial\mathcal{L}}{\partial\boldsymbol{\varphi}^1}$
$\frac{\partial\mathcal{L}}{\partial\boldsymbol{\varphi}^2}$
$\frac{\partial\mathcal{L}}{\partial\boldsymbol{\varphi}^{n\text{-}1}}$
$\frac{\partial\mathcal{L}}{\partial\boldsymbol{\varphi}^n}$

Figure 5.1: Draft of LSNN based action inference. Within the time window τ, the network gets the joint angles φ^n and an optional clock (clk) signal as inputs and predicts the joint position ${}^{0}_{n}\tilde{A}$. Calculating the joint angles for a desired endeffector position ${}^{0}_{N}\overset{*}{A}$ is done by extracting a temporal gradient for the error signal $\mathcal{L}$, which represents the difference between a desired joint position ${}^{0}_{N}\overset{*}{A}$ and its current estimation ${}^{0}_{N}\tilde{A}$. Image adapted from Otte *et al.* (2017b).

Using the described procedure, the inference accuracy is still limited by the accuracy of the predictive forward network. This problem can be resolved by either retraining the predictive forward model during the inference process or by using a target correction step as described by Otte *et al.* (2017b).

As the robotic arm moves towards a desired goal, during retraining, the predictive forward LSNN only gets the current arm pose as a training example. By using a small learning rate, the network overfits in a controlled manner towards an optimal prediction of the target pose. The difficulty of this approach is to choose a suitable retraining learning rate, so that the prediction significantly improves for the desired target pose, but the network does not overfit to the point that the learned dynamics of the robotic arm are destroyed.

In order to correct the prediction error, a modified target is used to calculate the input gradient. In a real robot, this modified target could theoretically be computed by using a visual feedback system that tells the difference between the predicted and actual endeffector pose of the robotic arm. In this thesis, the correction is calculated using the actual endeffector position from the simulation, which is calculated based on the joint angles.

$$p^N_{corrected} = p^N_{predicted} + \beta_{pos}(p^N_{target} - p^N_{actual}) \tag{5.1}$$

$$v^N_{x,corrected} = v^N_{x,predicted} + \beta_{rot}(v^N_{x,target} - v^N_{x,actual}) \tag{5.2}$$

$$v^N_{y,corrected} = v^N_{y,predicted} + \beta_{rot}(v^N_{y,target} - v^N_{y,actual}) \tag{5.3}$$

As described in Equations 5.1, 5.2 and 5.3, the modified endeffector target (Equation 5.4) is computed by using the distance between the actual position and orientation and the target position and orientation.

$$(p^N_{corrected}, v^N_{x,corrected}, v^N_{y,corrected}) \tag{5.4}$$

This distance is then multiplied with a position β_{pos} and orientation β_{rot} based scaling factor in the range $(0, 1)$ and then added to the prediction in order to calculate the corrected target.

For the actual inference process, two different optimizers are used. One being Adam together with a weight decay like regularizer that regularizes for smaller angles. The other is a modified Stochastic Gradient Descent with momentum, as described by Otte *et al.* (2017b).

$$\Delta\varphi^{t+1} = -\eta[\Theta^t]^2 \frac{d\mathbf{E}}{dx_i} + \mu\Delta\varphi^t \tag{5.5}$$

$$\Theta^{t+1} = \alpha\Theta^t + (1-\alpha)sgn\left(\frac{d\mathbf{E}}{dx_i}\right) \tag{5.6}$$

Equations 5.5 and 5.6 describe the used momentum optimizer. Here η is the learning rate and μ the scaling rate of the momentum term. The resulting weight update is scaled down depending on the oscillation of the gradient $[\Theta^t]^2$ (sgn represents the signum function).

5.1.2 Biologically Plausible Inference

While an LSNN can be trained using a biologically plausible training algorithm like e-prop 1 in order to predict the position and orientation of the joints of a many-joint robotic arm, an action inference is not directly possible with the e-prop 1 algorithm. This is due to the fact that e-prop 1 approximates the actual error gradient by only considering errors from the past and present time step.

For an action inference, the discrepancy between a desired goal position and its current estimation is back-propagated through time in order to derive an error gradient for the input angles. Since a predictive forward LSNN only outputs the endeffector position during the last time window τ^N, an error has to be back-propagated from this time window back to all previous time windows in order to derive meaningful input errors for the joint associated with that specific time window.

When using approximate input errors as descried in Equation 2.47, one can calculate a learning signal L^t_j as a weighted sum of network errors from the last network run, which does not require to use errors from the future.

$$\begin{aligned} \frac{dE}{dx_i^t} &\approx \sum_j w_{j,i}^{in} L_j^t \frac{\partial z_j^t}{\partial \mathbf{s}_j^t} \\ &= \sum_j w_{j,i}^{in} \frac{\partial z_j^t}{\partial \mathbf{s}_j^t} \sum_k w_{j,k}^{out} (v_k^T - u_k^T) \end{aligned} \tag{5.7}$$

Unfortunately, this error also does not include errors from the past activity of the underlying synapse, since it does not include an eligibility trace, and thus is a very limited gradient.

A better option is to use an e-prop 2 based approach where another LSNN computes the learning signals based on the previous activity of the predictive forward LSNN.

The basic principles of the e-prop 2 algorithm is that a second LSNN called error module gets the same inputs as the predictive forward LSNN and additionally receives the hidden spikes of the predictive forward model together with a target signal as input. The target signal in this case is the desired goal position.

During each time step, the readouts from the error module are used as an online learning signal for the input errors of the predictive forward model, incorporating past information into Equation 2.47.

While this enables a biologically plausible way of computing an input gradient, the error module itself should to be trained using BPTT.

In order to compute an error gradient that can train the error module, an error for the learning signal L_j^t has to be derived (see Equation 2.49). This is done by first running the predictive forward LSNN together with the error module (with a specific target) and calculating new input angles in an one shot learning fashion, then running the predictive forward LSNN again with the new inputs and calculating the error for this new inputs by using the desired goal as the endeffector target.

The errors for the new inputs can then be used to calculate an error for the learning signals of the error module.

5.1.3 Direct Inverse Model

Since one needs to compute an input error in order to train an error module that can compute input errors, at least one error module has to be trained using BPTT.

So instead of using the e-prop 2 error network, one might directly use the backpropagated input errors to train a direct inverse model that does not relay on the inputs and hidden states of a predictive forward model.

A direct inverse model in this thesis is an LSNN that gets a target endeffector position as input and has two outputs for the joint x- and z-angles. Over a simulation run the direct inverse model successively outputs the joint angles for a robotic arm during the decoding time windows τ_{out}^{n}, starting at the base and outputting the angles for the joint controlling the endeffector during the last time window τ_{out}^{N}.

The simple approach of training a direct inverse model with the joint angles of random arm poses, given their endeffector target, would produce conflicting gradients, since the same endeffector target can be reached using entirely different joint angles. Instead of using a predictive forward LSNN together with a target correction step as previously described, gradients directly based on the endeffector position can be computed. This is done by providing the direct inverse network with a desired endeffector position and computing the joint angles by running the direct inverse network for N time windows. The computed joint angles are then fed into a pre-trained predictive forward LSNN that predicts the pose of the robotic arm based on these angles. The discrepancy between the predicted endeffector target and the desired one can than be back-propagated to computed input errors for the angles of each joint. These input errors can then be directly injected into the direct inverse model to compute the gradients for the output, hidden and input weights.

5.2 Results

Action inference as implemented in this thesis needs a predictive forward LSNN as basis whether for training a direct inverse model or for classical inference using BPTT.

While the predictive forward model has to learn the general dynamics of the robotic arm, the actual prediction accuracy does not have that much of an influence on the inference accuracy. Since it can either be corrected mathematically, or through retraining of the predictive forward model for a desired target pose.

While retraining generally works, during experiments it turned out that when only retraining the readout weights of the predictive forward LSNN, higher learning rates and longer retraining periods could be used before the model overfits to the point that the learned dynamics are destroyed and the inference procedure no longer converges towards the desired goal.

In order to evaluate the inference process, the Euclidean-distance error $||\mathbf{E}||_2$ is redefined to capture the difference between a desired target position and the actual arm position of the endeffector.

$$||\mathbf{E}||_2 \overset{\text{def}}{=} \sqrt{||\overset{*}{p}^N - p^N||_2} = \sqrt{(\overset{*}{p}_x^N - p_x^N)^2 + (\overset{*}{p}_y^N - p_y^N)^2 + (\overset{*}{p}_z^N - p_z^N)^2} \tag{5.8}$$

Additionally, an orientation error is defined, which measures how much the inferred endeffector orientation differs with the target orientation.

$$\mathbf{E}_{rot} = \frac{180}{2\pi}\left[(acos\left(\frac{v_x^N \cdot \overset{*}{v}_x^N}{|v_x^N||\overset{*}{v}_x^N|}\right) + acos\left(\frac{v_y^N \cdot \overset{*}{v}_y^N}{|v_y^N||\overset{*}{v}_y^N|}\right)\right] \tag{5.9}$$

In Equation 5.9, the orientation error $\mathbf{E}_{rot}$ is defined as the average angle in degrees between the target up- and x-direction vector and inferred up- and x-direction vector.

Reachable inference targets are computed by using random angles for the robotic arm simulation and recording the endeffector target.

If not stated otherwise, the general inference procedure starts with an upright arm using zero angles and then iteratively computes the gradients and updates the angles accordingly for a fixed number of epochs.

5.2.1 BPTT based Inference

Inference using BPTT achieves the highest accuracies of all inference methods presented in this thesis, and is also able to control robotic arms with the most joints (50 for the CAD based model). Additionally, it is the only inference method capable

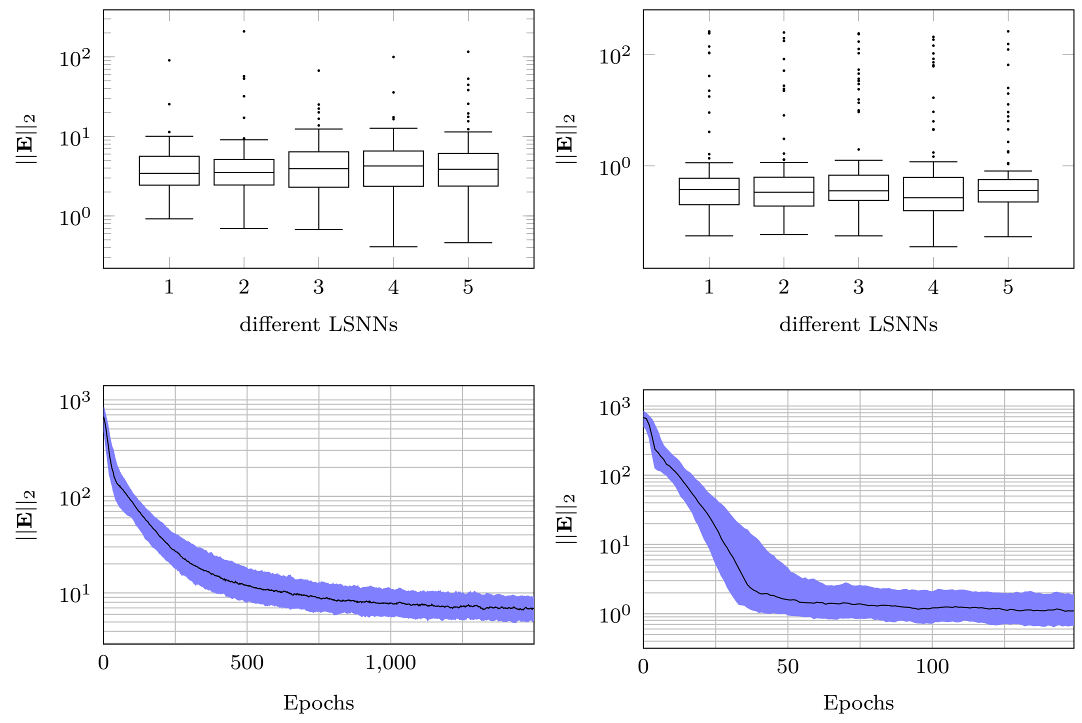


Figure 5.2: Comparison of Adam-based inference (**left**) and momentum-based inference (**right**) on a 10-joint robotic arm with maximum angles of 45 degrees. Predictive forward LSNNs used 128 hidden neurons and were trained for 150,000 epochs. **Top-row**: box plots for 5 different trained predictive forward models with 100 random inference targets each. **Bottom-row**: Median (black) and interquartile range (blue) over all 500 inference runs for each Adam and momentum based inference.

of optimizing not only the endeffector position, but also its orientation.

When training LSNNs in order to predict the pose of a many-joint arm using the mathematical or CAD based model, the final accuracies achieved do not vary much if the same number of hidden neurons and training epochs are used.

For BPTT based inference, the accuracy variance for different trained predictive forward LSNNs is even smaller, as shown in Figure 5.2, were no significant difference for the inference accuracy using separately trained predictive forward models can be noticed.

Figure 5.2 also shows the main differences between the two inference methods

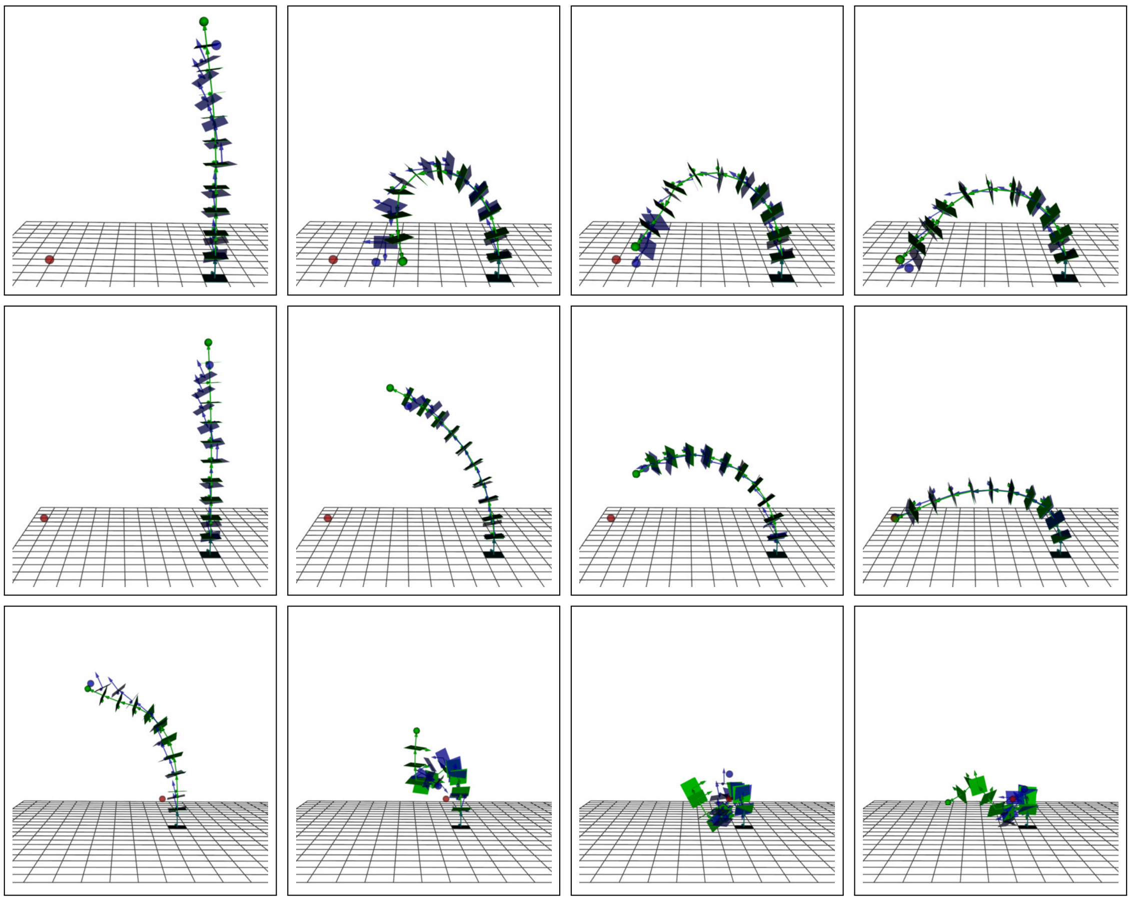


Figure 5.3: Inference comparison of the momentum-based approach (**top**) and the Adam-retraining based approach (**middle**) and a typical fail case (**bottom**) on a 10-joint mathematically ideal robotic arm with max angles of 45 degrees. The momentum-based approach precisely reaches its target with a sub millimeter precision, correcting the prediction error which is in the range of centimeters. Adam-based approach retrains the predictive forward LSNN to reach a sub centimeter prediction and inference accuracy. Inference fails on heavily twisted arms where the prediction breaks down.

(momentum- and Adam-based) on a 10-joint mathematical robotic arm with maximum joint angles of 45 degrees.

The Adam-retraining based approach on the left is about 10 times slower than the momentum based approach on the right. This is due to the fact that for Adam a much slower learning rate has to be used (0.01 compared to 0.1 for momentum). Since Adam normalizes the gradients, it effectively changes all joint angles with roughly the same magnitude, and therefore tends to twist the arm for larger learning rates. These findings align with those from Otte *et al.* (2018), where Adam was found to be suboptimal for action inference on many-joint robotic arms.

Also a strong regularizer that pulls the angles towards zero has to be used together with the Adam-based approach to future reduce the tendency to twist the arm.

As already mentioned, the LSNN can only be retrained for a small number of

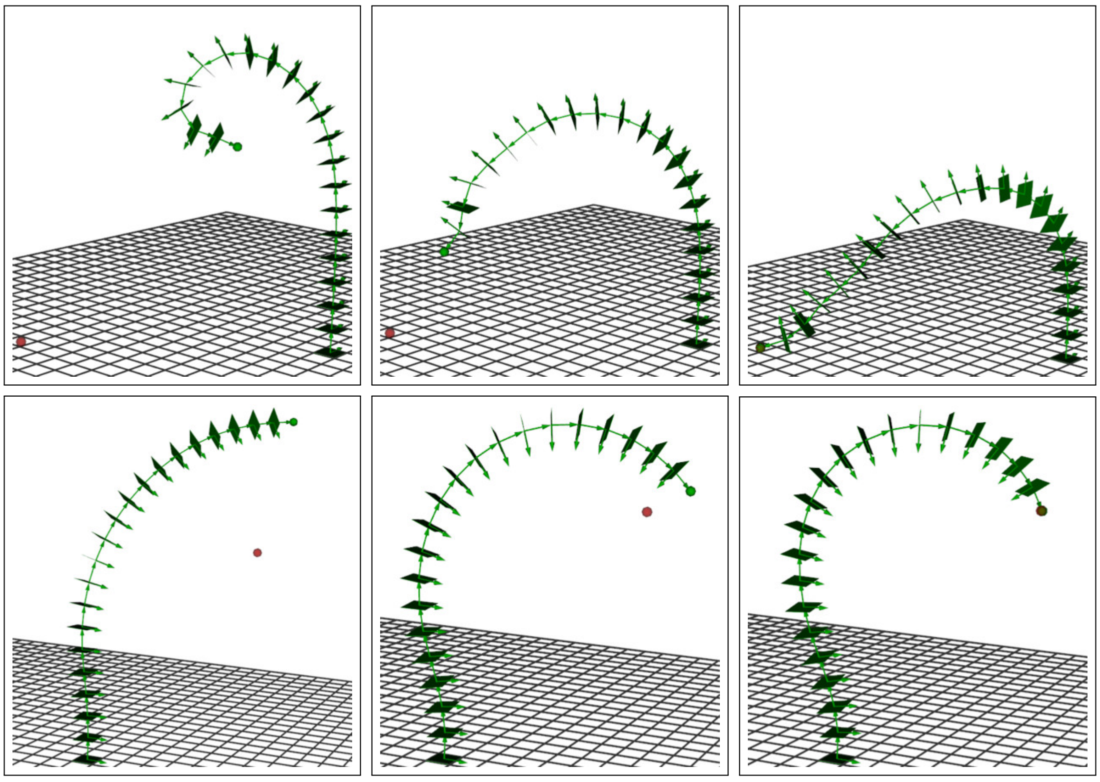


Figure 5.4: Inference comparison of the momentum-based approach (**top**) and the Adam-retraining based approach (**bottom**) on a 20-joint mathematically ideal robotic arm with max angles of 45 degrees. The momentum-based approach reflects the search through the joint space by wiggling the top of the arm into the direction of the target, and slowly pulling in later joints. The Adam approach moves all angles equal in magnitude and achieves a smooth path towards the goal by heavily regularizing the angles towards zero.

epochs in order to keep the learned dynamics intact, so the prediction accuracy which limits the Adam based approach can not get that much better. Still the Adam-based approach achieves a median sub centimeter accuracy and also fails less often than the momentum-based one.

The momentum-based approach on the other hand, which uses the mathematical prediction correction step, achieves a median sub millimeter accuracy and also converges much faster due to the greater learning rate that could be used.

Since both inference approaches produce fail cases in the range of several thousand millimeters, the median is a better metric than the average to reflect the typical accuracies.

The visual differences of the two approaches can be seen in Figure 5.3 and 5.4, where the Adam based approach produces much smoother arm poses and arm movements, which is due to the uniform joint modification and the strong regularizer. In the middle row of Figure 5.3 the predictive forward LSNN retraining can also be seen, where the prediction gets better as the arm moves towards the goal.

In contrast to the smooth Adam-based inference, the momentum-based approach reveals the underlying search through the joint space by wiggling around with the top of the arm and slowly pulling later joints towards the goal, also introducing sharp angles on joints near the endeffector.

Combining the two approaches repeatedly failed to produce good inference accuracies, since retraining seems to conflict with the prediction correction when used with the momentum optimizer. Also the prediction correction (without retraining)

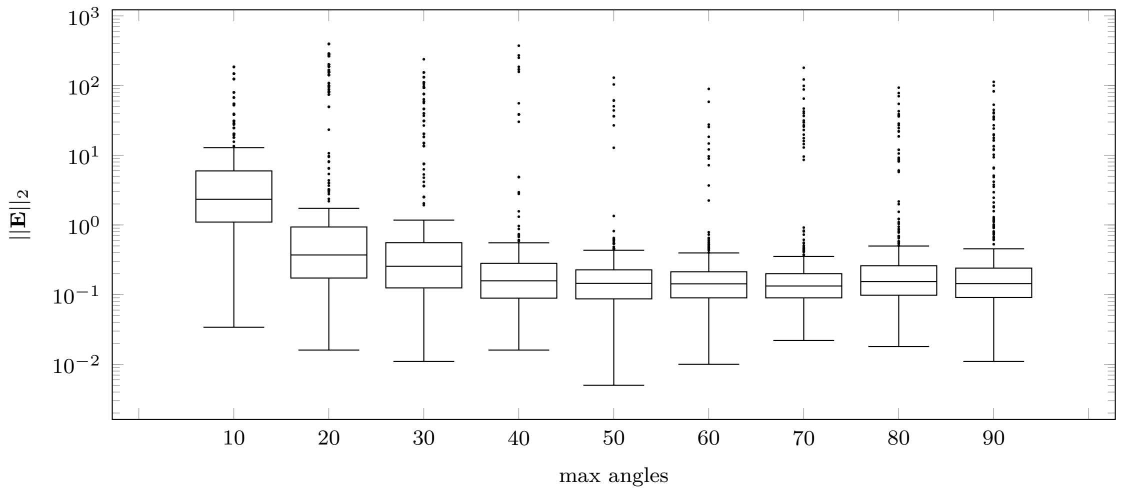


Figure 5.5: Comparison of momentum-based inference accuracy using predictive forward LSNNs for a 10-joint robotic arm with different max angles. Each experiment is based on 5 different predictive forward LSNNs with 128 hidden neurons separately trained over 10,000 epochs with independent random initial weights. With each trained LSNN, 100 inference runs over 500 epochs are calculated, and the minimum $||\mathbf{E}||_2$ are recorded.

did not work with the Adam optimizer. Futhermore using a regularizer that pulls the angles towards zero has not the wanted effect of smoothing the target pose or the general arm movement when used with the momentum based optimizer. Since the momentum optimizer computes non-uniform angle updates, the angles are either too strong regularized so that the arm does not reach its target, or too weak so that the top of the arm still wiggles around.

Typical fail cases for both inference approaches as shown in Figure 5.3 (bottom row) involve highly twisted arms, where the prediction breaks down and the gradient is unable to untwist the arm. By using a slightly smaller learning rate for a specific target on which the arm failed, or simply using the other inference method is in most cases enough to successfully reach the failed target on the second approach.

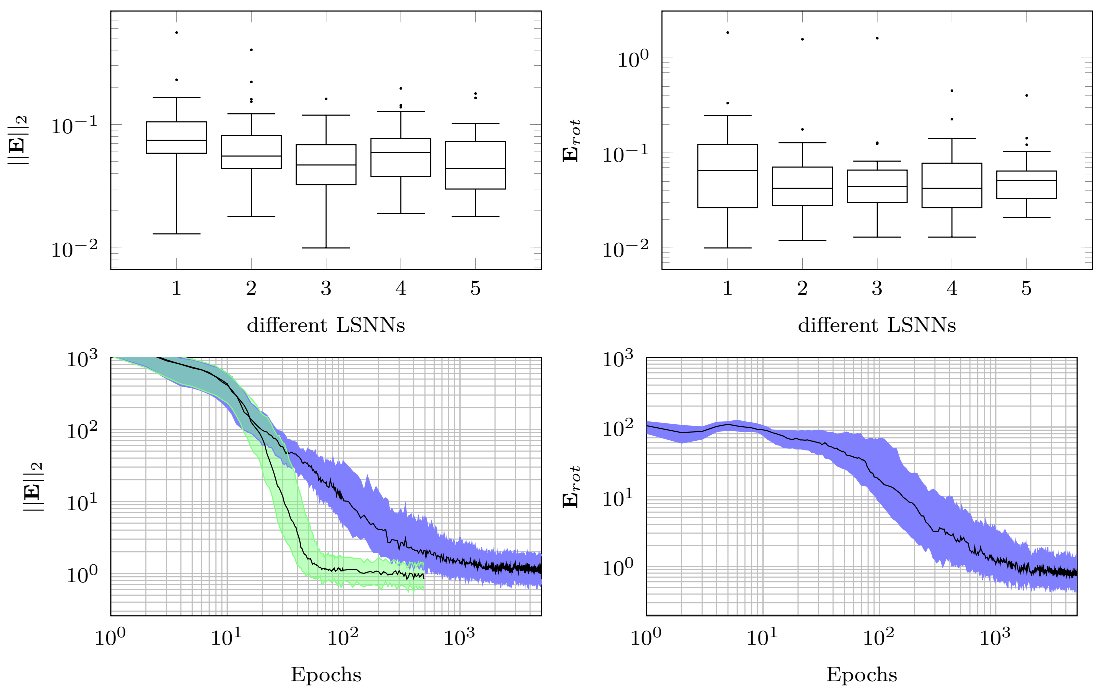


Figure 5.6: Momentum-based inference evaluation for a 20-joint robotic arm (mathematical simulation) with max angles of 45 degrees. 5 different LSNN were trained with 256 hidden neurons and evaluated on 25 different targets, each for 5000 inference epochs. Inference targets included endeffector orientation. **Top**: Minimum position (**left**) and rotation (**right**) accuracies observed over 5000 epochs for each of the 5 different trained LSNNs. **Bottom**: Median and interquartile-range for the position (**left**) and orientation (**right**) accuracies during the 125 inference runs. The median and interquartile-range for the position accuracy over 500 inference epochs, when orientation is not optimized, are plotted in light green.

.

Another interesting property is that the inference process tends to increase in accuracy for greater maximum allowed angles as shown in Figure 5.5. This is in direct contrast to the prediction that gets significantly worse the greater angles are allowed (see Figure 4.9).

In total, 45 LSNNs are trained, 5 for each max angle configuration. Except for the different angles all other parameters are kept the same and each LSNN is independently initialized with random weights. With each LSNN, 100 random targets are inferred over 500 epochs each, making a total of 500 random targets per max angle. An educated guess is that for the 10-joint arm used, the prediction, even for 90 degree angles, is still good enough, and the greater space of possible arm configurations for the same endeffector pose allows a greater accuracy during the inference process.

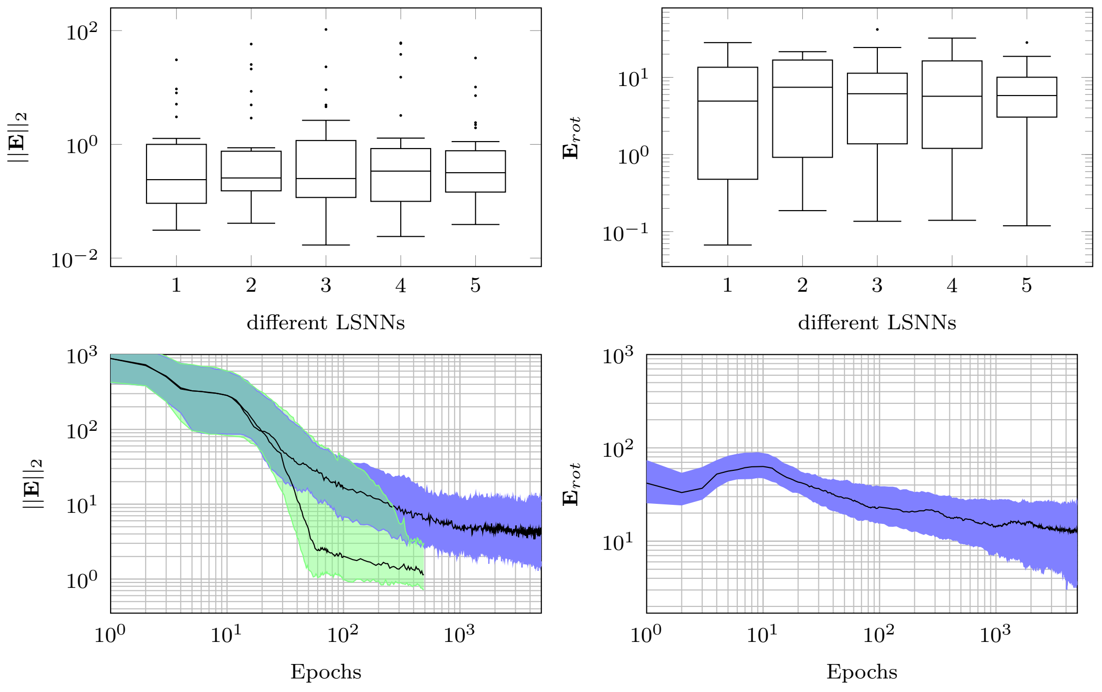


Figure 5.7: Momentum-based inference evaluation for a 25-joint CAD based realistic robotic arm. 5 different LSNN were trained with 256 hidden neurons and evaluated on 25 different targets each for 5000 inference epochs. Inference targets included endeffector orientation. **Top**: Minimum position (**left**) and rotation (**right**) accuracies observed over 5000 epochs for each of the 5 different trained LSNNs. **Bottom**: Median and interquartile-range for the position (**left**) and orientation (**right**) accuracies during the 125 inference runs. The median and interquartile-range for the position accuracy over 500 inference epochs, when orientation is not optimized, are plotted in light green.

.

When also considering the endeffector orientation during the inference process, the number of epochs it takes to reach a target position and orientation significantly increases, as shown in Figure 5.6.

Also probably due to the high regularization, the Adam-based approach was unable to orientate the endeffector at the target position into a target orientation.

While a mathematical robot model of a 20-joint arm with maximum joint angles of 45 degrees needed about 10 times more epochs, the minimum position accuracy still reached a sub millimeter level.

Also the desired orientation could be reached with a median angle between the desired and inferred up- and x-direction vectors of less than one degree.

For the realistic simulation when evaluated on a 25-joint arm, the orientation only reached a median minimum accuracy $\mathbf{E}_{rot}$ of less than 10 degrees. The position accuracy on the other hand still reached a minimum median accuracy of less than one millimeter, but also needed a lot more epochs to converge (see Figure 5.7).

One reason for the decreased orientation accuracy of the realistic arm model might be the fact that each joint is rotated about 45 degrees around the y-axis when compared to the previous joint, which could make it harder for the inference process to compute a sufficient gradient that takes this into account.

Inference using the CAD based realistic robot arm in general works better than

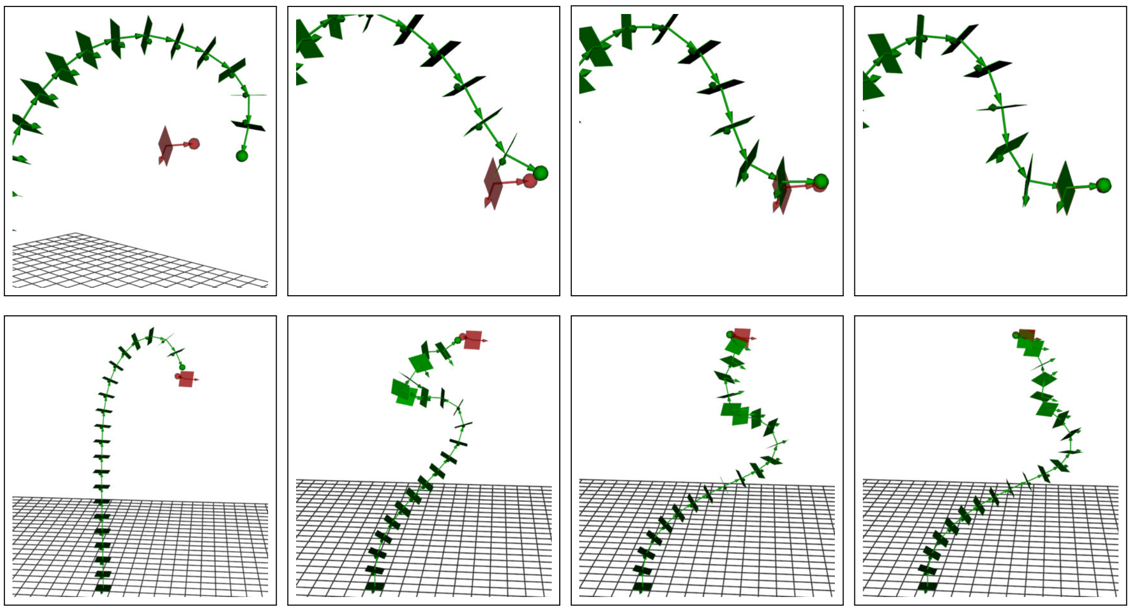


Figure 5.8: Momentum-based inference, with position and orientation endeffector targets on a 20-joint mathematical robotic arm with maximum angles of 45 degrees. **From left to right**: Arm approaches the target position and rearranges itself to fit the desired endeffector orientation.

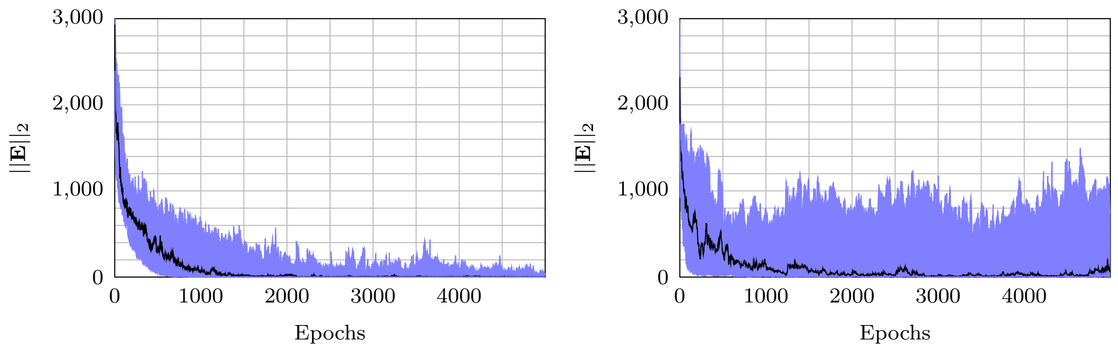


Figure 5.9: Comparison of momentum-based inference with long robotic arms. **Left**: Mathematical 40-joint robotic arm with max angles of 45 degrees. Predictive forward LSNN used 768 hidden neurons and was trained for around 180,000 epochs. **Right**: CAD based realistic robotic arm with 50 joints. Predictive forward LSNN used 512 hidden neurons and was trained for around 150,000 epochs. Each plot shows the median (black) and interquartile range (blue) for 25 random inference targets.

the mathematical arm with the default max angles of 45 degrees when only the endeffector position is inferred. This is mostly due to the fact that the realistic arm only allows for max angles of ± 20 degrees, and therefore more robust predictive forward models can be trained even for arms with 50 joints (100 degrees of freedom). Although the number of faile cases seem to increase with more joints, the 50-joint arm can precisely reach a target with the typical sub millimeter precision using the momentum optimizer (see Figure 5.10.

In Figure 5.9, a 50-joint CAD based robotic arm is compared with a 40 joints mathematical robotic arm using 45 degree angles.

While the 50-joint realistic arm reached a prediction accuracy of $||\mathbf{E}||_2 \approx 203mm$ and could infer angles to reach target poses using the momentum-based approach with a median accuracy of $||\mathbf{E}||_2 = 0.352$, a high number of faile cases can be seen in the high upper quantile $||\mathbf{E}||_2$ error.

The 40-joint mathematical arm on the other hand reached a prediction accuracy of $||\mathbf{E}||_2 \approx 197mm$ and a median inference accuracy of $||\mathbf{E}||_2 = 0.136$. Also the upper quantile clearly decays torwards zero.

The increased performance of the 40-joint arm could be due to the fact that 768 hidden neurons are used for the prediction LSNN instead of 512 for the 50-joint CAD based arm. Thise was done since the same arm could not infer tragets using the momentum-based approach with 512 hidden neurons.

Since each of the LSNNs needed several days to train and then several days to evaluate, only a single LSNN could be trained for each experiment. Still the evaluation of those LSNNs could be representative, since for smaller arms the variance between different trained predictive forward models was insignificant for the same number of epochs and hidden neurons.

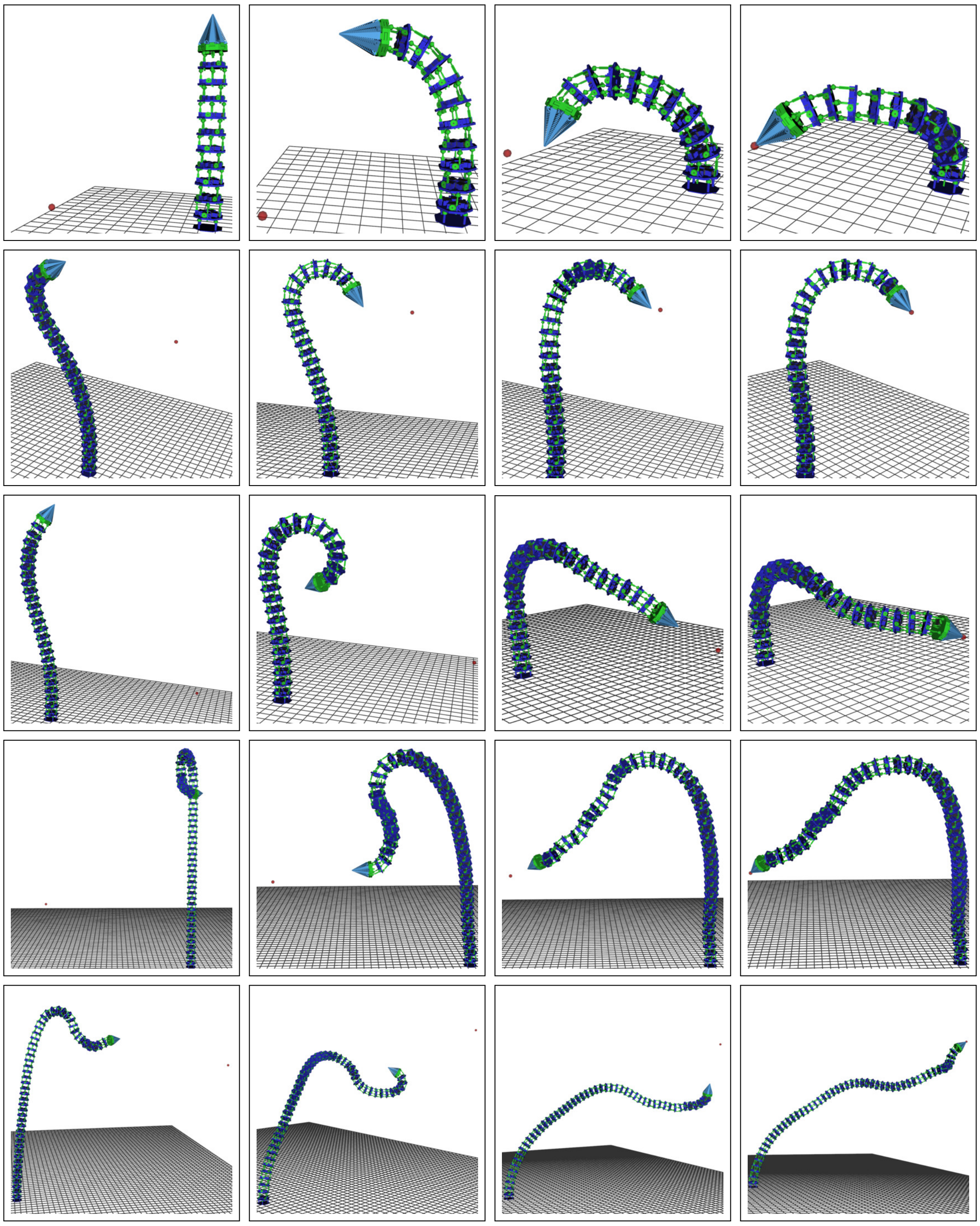


Figure 5.10: Momentum-based inference for a realistic robot arm simulation using CAD files. **From left to right**: Arm starts with zero angles and moves towards the red endeffector goal. **First-Row**: 10-joint arm. **Second- and Third-Row**: 25-joint arm. **Fourth- and Fifth-Row**: 50-joint arm.

5.2.2 E-Prop based Inference

As already mentioned, the e-prop 1 based inference approach has only access to present information in order to calculate an input error.

Experiments using the Adam-based approach completely failed to produce meaningful goal directed behavior using this e-prop 1 based error.

With the momentum-based approach some goal directed behavior could be observed, but the arm is far from reaching the desired goal positions as shown in Figure 5.11, where the arm has a median distance to the target of over 100mm.

Using the described one shot learning approach based on e-prop 2, the training data for an error module was calculated by using the endeffector position for a random arm pose as target and then starting with zero angles which are fed into a predictive forward model. The hidden spikes together with the input angles and the target are fed into the error LSNN and the computed errors are used to update the input angles. The predictive forward model is then run again with the new input angles, and the discrepancy between the endeffector position based on the newly computed angles and the target is used to compute a learning gradient for the error LSNN.

The new angles are also used as a starting point for the next iteration and The arm pose is reset back to zero angles randomly after an average of 10 consecutive iterations.

By initializing the error module with small readout weights (factor 0.001 smaller than normal), the angle updates are initially very small and converge over the training towards an one shot learning.

By design, the one shot learning uses a learning rate of 1.0 and a trained error module only slightly improves the first pose when run again as shown in Figure 5.12.

Also using a smaller learning rate and therefore taking smaller steps towards

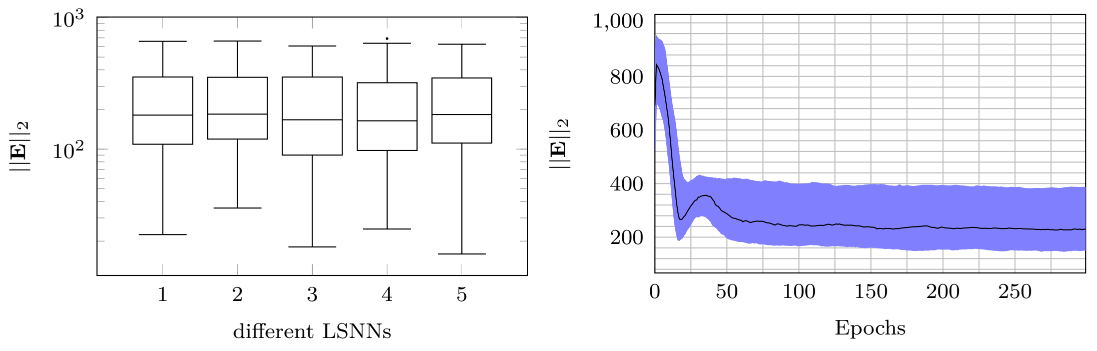


Figure 5.11: E-prop 1 like inference for 5 different LSNNs using 128 hidden neurons trained to predict the pose of a 10-joint robotic arm with a maximum of 45 degree angles. **Left**: Comparison of 100 inference runs for each independent trained LSNN. **Right**: Median (black) and interquartile range (blue) over all 500 inference runs.

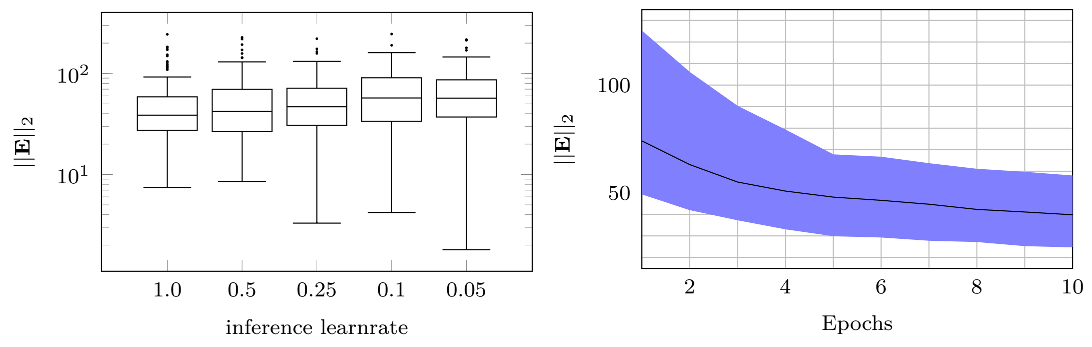


Figure 5.12: E-prop 2 based one shot like inference based on 5 different experiments with different predictive forward models and error module LSNNs each. Predictive forward LSNNs used 128 hidden neurons and the error module LSNNs used 256 hidden neurons. **Left**: Best inference accuracy for different learning rates evaluated using 25 independent random targets for each of the 5 error module LSNNs. The experiments for each learning rate used a total number of epochs based on the following formula: $Epochs = {}^{10}/_{\eta}$ **Right**: Median (black) and interquartile range (blue) over all 125 runs with a learning rate of 1.0 for each epoch the overall best accuracy (of that run) is recorded.

the goal did not improve the overall accuracy. It actually increased the minimal reached distances from the goal as shown in Figure 5.12.

While not as accurate as the BPTT based inference, the accuracy on the e-prop 2 based approach reaches a median accuracy below 10cm for a mathematical based 10-joint robotic arm with max angles of 45 degrees which is around four times better than the e-prop 1 based approach.

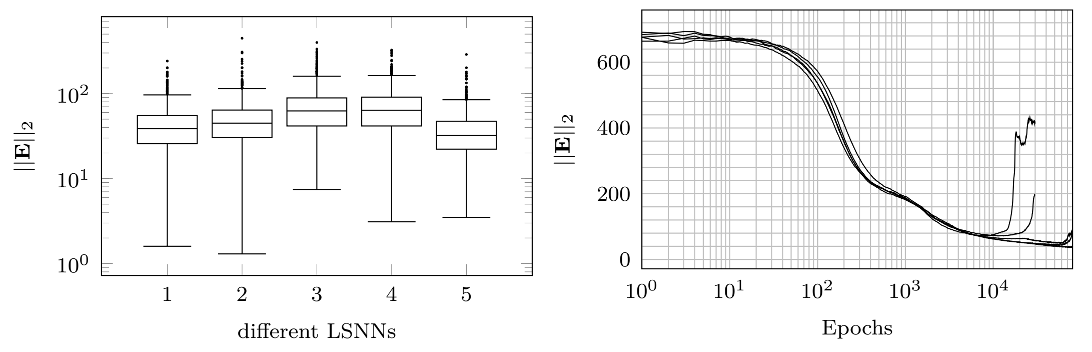


Figure 5.13: Direct inverse model evaluation based on 5 different direct inverse LSNNs that where each trained using a different predictive forward LSNN. Predictive forward LSNNs used 128 hidden neurons and direct inverse LSNNs 256 hidden neurons. **Left**: Accuracy for each direct inverse model, based on early stopping (the network weights that produced the lowest training error are used for evaluation). **Right**: Training error of the 5 different direct inverse LSNNs. Overfitting occurs in 4 networks after around 15,000, 20,000 60,000 and 65,000 epochs.

5.2.3 Direct Inverse Model

In contrast to the e-prop 2 inference, where an error module is trained to adjust the current joint angles in a way that moves the robot towards the desired goal position, with the direct inverse model joint angles are directly calculated based on a desired goal position. This implies that the model learns a mapping from 3D space to joint space, since no other input information are given.

The training of such direct inverse models tends to overfit at some point, where the arm learns a pose that has a short distance to the average endeffector training target.

Nevertheless, by using an early stopping approach, where the training is stopped before the model overfitts, the direct inverse LSNNs produce accuracies similar to the e-prop 2 based approach as shown in Figure 5.13.

Chapter 6

Robot

In order to test the inference process outside of a simulation within the real world, a robotic arm based on the CAD files used in the realistic simulation (Figure 6.1 and 6.2) is evaluated within this chapter.

The 8-joint robotic arm as shown in Figure 6.4 was designed using OpenSCAD and built from Polylactic Acid (PLA) plastic using a Fused Deposition Modeling (FDM) 3D-Printer (Boparai *et al.*, 2016; Wong and Hernandez, 2012; Kintel and Wolf, 2011; Swetham *et al.*, 2017).

Each joint is moved by four small servo motors that drive a horizontal linear gear. Each of those linear gears is connected over a ball-joint with the base of the next joint. By adjusting the height of the four linear gears each joint can approach any x-, z-angle combination within a range of $[-20, 20]$ degrees.

In order to compensate the change in position when approaching high angles, the ball joints are not directly connected with the base of the next joint, they rather

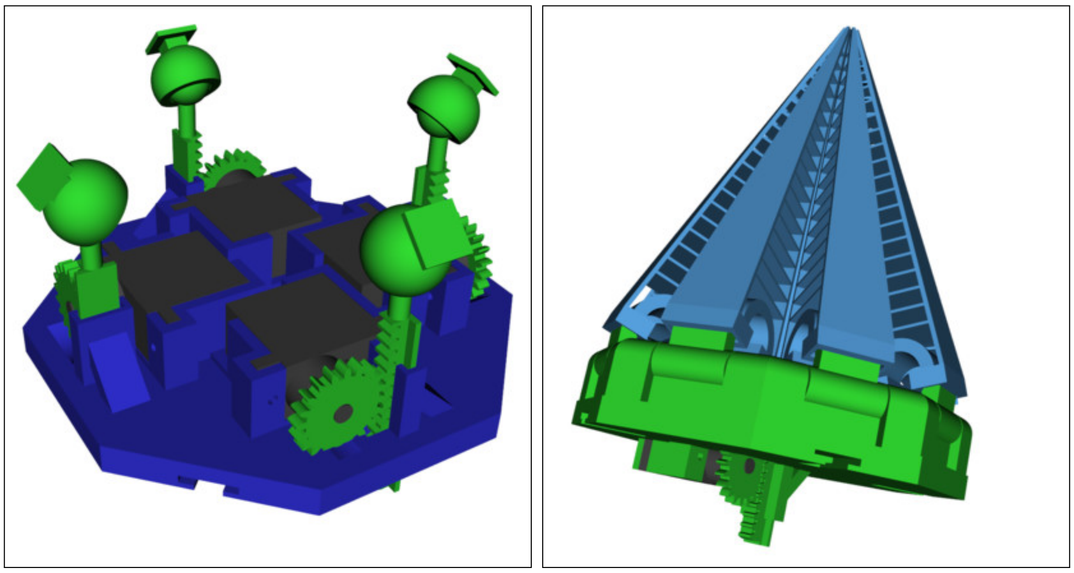


Figure 6.1: Left: single joint from the CAD based simulation. **Right**: Gripper from the CAD based simulation.

run in a linear track which points from the middle of the base to the outside.

Each joint is also rotated about 45 degrees around the y-axis relative to its previous joint, so that the linear gears form different joints do not collide with each other.

While there is a hole in the middle of each module to allow the routing of cables within the arm towards the base, one of the servo driver modules had to be positioned within the arm, since it was impractical to route the cables form the top 4 joints to the base.

The gripper on top of the robot is controlled by a single servo that opens or closes it.

Holding the robot upright in a neutral position, the 32 joint motors consume an average of about 15 Watt, while during movement, especially when returning from highly bend position where the gripper is far from the base, the motors can consume up to 40 Watt.

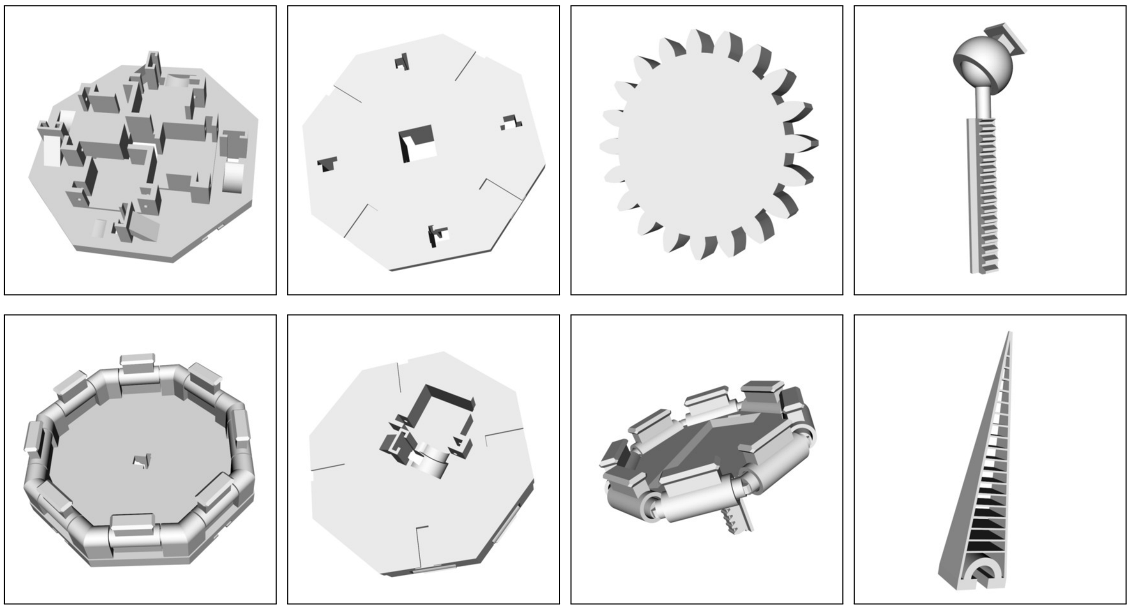


Figure 6.2: CAD files of the different parts for building the 8-joint robotic arm. **Top-Row**: Base unit for one joint containing four servo mounts and linear gear shafts. In the back view, the linear tracks, which are part of the dynamic connection between joints, are visible The normal gear can be turned a total of 180 degrees by the servo motors, in order to move the linear gear up and down. The ball-joint at the end of the linear gear can be printed in place and is thus inseparable connected with the linear gear. **Bottom-Row**: The gripper base- and movable-unit contains 8 connectors for mounting the 8 gripper arms. A servo mounted on the back of the gripper base unit controls the inner movable part of the gripper.

Since the Arduino microcontroller is very limited in working memory, it was not possible to directly run an LSNN on the board. Instead a program that enables the easy setting of the x- and z-angles for each joint and also accounts for the relative y-rotation of the joints was implemented. While it is in general possible to set the angles for each joint over a serial communication channel, for ease of usage, the angles for desired arm poses were directly copied into the code.

In order to evaluate the accuracy of the robot, the angles to reach 25 different targets were computed using BPTT based inference on an LSNN with 128 hidden neurons, trained to predict the pose of a 8-joint robotic arm with the CAD based simulation. The computed angles were then copied into the program code and uploaded onto the Arduino.

Each different arm pose was then approached from a neutral position (zero angles) and held for 30 seconds to allow manual measurements.

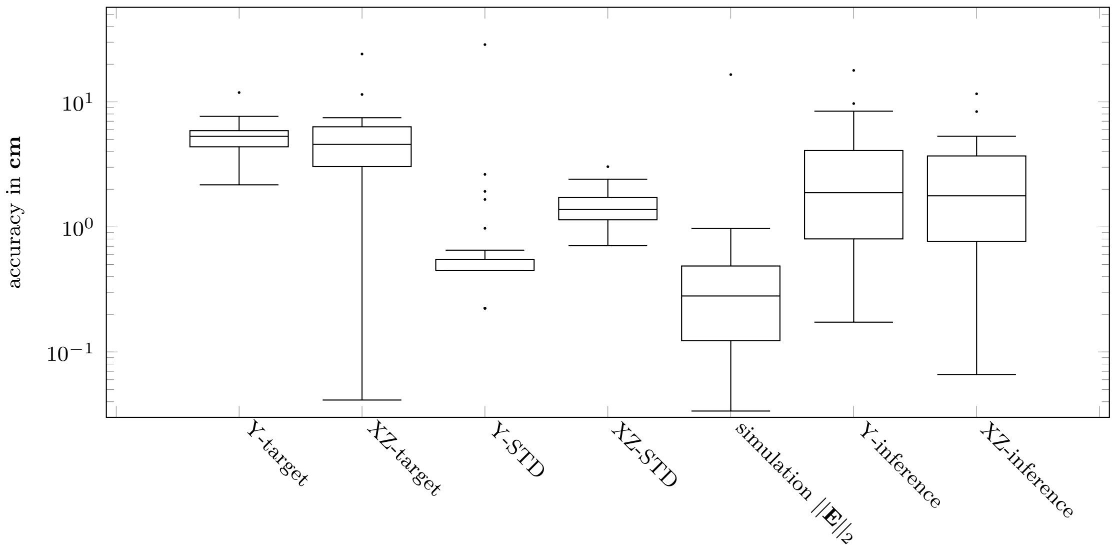


Figure 6.3: Evaluation of the robotic arm on 25 random poses. Each pose was approached and measured independently 5 times. From left to right: Error between the measured height (y-distance) and the target height. Error between the measured distance from the center (xz-distance) and the target distance form the center. Standard deviation of the height measurement for all 25 poses. Standard deviation of the xz-distance measurement for all 25 poses. The $||E||_2$ inference accuracy (in centimeter) of the inferred poses. Error between the measured height and the actual height calculated within the simulation based on the angles for each joint. Error between the measured xz-distance and the actual xz-distance calculated within the simulation based on the angles for each joint.

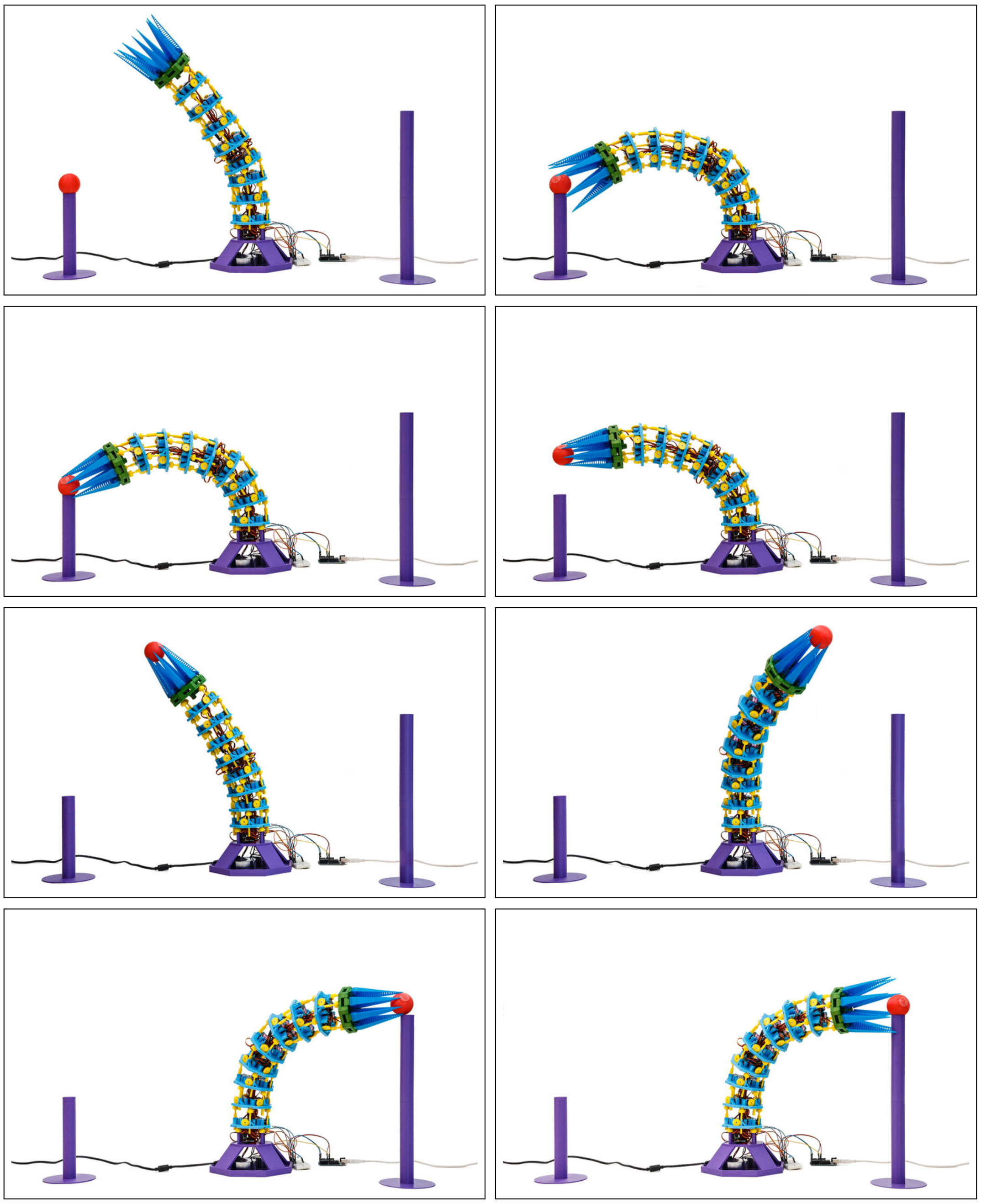


Figure 6.4: 8-joint robot moving a red ball from one socket to another. Angles for the two (socket-)targets were inferred using a CAD based prediction LSNN, together with a manual target correction step where the target was successively altered until the robot was able to successfully grab/release the ball. In order to approach the two targets, the inferred angles were interpolated from a neutral stand (zero angles). Images taken by Christoph Traub.

Each of the 25 poses were approached independently for 5 times and the height from the ground to the tip of the gripper, together with the distance from the ground (below the tip) to the base of the robot were recorded.

While the height could be measured very accurately with an average deviation below one centimeter, the distance from the ground to the center had a measurement uncertainty of one to three centimeters.

Nevertheless, when compared to the theoretical endeffector position based on the inferred angles, the robotic arm had a median accuracy of around two centimeters and a median accuracy of around five to six centimeters when compared to the actual target.

A slightly more complex task of moving a ball from one socket to another, as shown in Figure 6.4, could also be executed by using a manual target correction step, where the inference target for the two sockets were iteratively refined in order to compute the inferred angles which allowed the arm to grab and release the ball at the two target positions.

Chapter 7

Conclusion and Future Work

This thesis explored the application of recurrent Spiking Neural Networks (SNNs), specifically Long Short-Term SNNs (LSNNs), for action inference on many-joint robotic arms.

The training of a forward model that predicts the pose estimation in 3D space, given the x- and z-angles for each joint in a sequential manner, worked well, even for realistic robot arm models with up to 100 degrees of freedom.

While it is feasible to train an LSNN using the biologically plausible learn algorithm e-prop 1 and achieve high prediction accuracies, BPTT generally converges faster and achieves higher accuracies.

It could be shown that e-prop can produce error gradients that incorporate the Spike-Timing-Dependent-Plasticity of the underlying synapse, but the derived equations for Leaky Integrate and Fire neurons reset the eligibility trace, each time a spike occurs and therefore potentially destroys the aggregation of past error information within the eligibility trace. Nevertheless they show a fascinating property of the e-prop algorithm and are interesting for future research.

Inference based on BPTT worked especially well when only a specific position should be reached, the target could be approached with a sub millimeter precision with less than 150 angle updates.

For cases where the endeffector should also be orientated in a specific manner at the goal position, the inference process needed significantly more time but still achieved sub millimeter accuracy.

Future research using BPTT based inference could incorporate collision detection as used with LSTMs by Otte *et al.* (2018).

An odd trend could be discovered, where the position based inference, got more accurate the higher angles are allowed, although the prediction accuracy got significantly worse for larger angles. This trend was discovered for a mathematical arm with 10 joints, where the prediction accuracy even for 90 degree angles is still reasonable. It would be interesting to see whether this trend continues for longer arms.

While the evaluation of mathematical arms over 40 joints was mostly limited by the available GPU hardware, it remains an open question how much one can scale up the model, using larger networks that could be trained on more sophisticated hardware.

Biologically plausible inference possess a problem since it can not relay on future errors, and thus the simplistic e-prop 1 based approach repeatedly failed to reach a desired goal position.

Inference based on e-prop 2 involves a separate LSNN that acts as an error module and computes online learning signals in order to calculate input gradients. With these e-prop 2 based input gradients some sort of one short learning could be achieved, where a desired goal position can be inferred by a single pass of the prediction LSNN together with the error LSNN. Unfortunately, the accuracy achieved with this approach has still an error of several centimeters, and needs to be improved to be of practical usage.

A direct inverse model that learns a mapping from 3D (Endeffector) space to joint space produces a one shot inference with a comparable accuracy to the e-prop 2 based approach, but tends to overfit during learning.

A topic for future research could be the exploration of a hierarchical one shot model. Since the accuracy of the direct inverse and the e-prop 2 based inference is dependent on the length of the robotic arm, it is plausible to use several trained one shot models to significantly improve the overall performance by a hierarchical application.

This could be accomplished by calculating only the angles for the first joint with the LSNN that can compute the inference of the complete arm with N joints. By using a model for an arm with $N-1$ joints and staring at the inferred position of the first joint, one could refine the inferred position of the second joint, assuming a greater accuracy of the model for the shorter arm. Applying this principle one needs N models, but the computation could be carried out in only N time windows, since each model only has to compute the angles for its first joint.

For the training of the error LSNN it was assumed that BPTT is needed, but one may instead try to use the approximated e-prop 1 like input errors for training in order to get an error module that produces a slightly better input error and then use this error module to train another one. By repeating this procedure of training one error module with another, it could be possible to train an error module entirely in a biologically plausible way.

As demonstrated with the realistic CAD-based simulation, it is possible to train many-joint robotic arms and also achieve reasonable performance when applying the inferred angles without further processing on a real robot. By using a more sophisticated simulation that also models object collision and other forces that act on the robot, it should be possible to improve the accuracy of the real robot.

An interesting subject for future research would also be a target correction step for a real robot based on sensory or visual feedback.

Since the base joint has to carry the weight of the whole arm, the used robot with 8 joints and a gripper as endeffector is at the edge of what is possible using small servos. While scaling the model with stronger motors and more joints should be possible, another scenario where the used design could be extended with more joints, is using the robot not as an arm but more like a robotic snake. In this way, one joint does not have to carry the weight of all other joints above, but instead the robot could crawl over the ground, which would probably require less force from the servos than to lift up the whole arm. This would also be an interesting scenario to investigate goal-directed policy inference with LSNNs as described by Otte *et al.* (2017a).

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
BPTT	Back-Propagation Through Time
TCA	Temporal Credit Assignment
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
SNN	Spiking Neural Network
LSNN	Long Short-Term Spiking Neural Network
CNN	Convolutional Neuronal Network
EPSP	Excitatory Post Synaptic Potential
IPSP	Inhibitory Post Synaptic Potential
LIF	Leaky Integrate and Fire
ALIF	Adaptive Leaky Integrate and Fire
STDP	Spike-Timing-Dependent Plasticity
EA	Evolutionary Algorithm
NEAT	Neuro-Evolution of Augmenting Topologies

Bibliography

Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., Imam, N., Nakamura, Y., Datta, P., Nam, G.-J., *et al.* (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **34**(10), 1537–1557.

Bellec, G., Salaj, D., Subramoney, A., Legenstein, R., and Maass, W. (2018). Long short-term memory and learning-to-learn in networks of spiking neurons. In *Advances in Neural Information Processing Systems*, pages 795–805.

Bellec, G., Scherr, F., Hajek, E., Salaj, D., Legenstein, R., and Maass, W. (2019a). Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets. *arXiv preprint arXiv:1901.09049*.

Bellec, G., Scherr, F., Subramoney, A., Hajek, E., Salaj, D., Legenstein, R., and Maass, W. (2019b). A solution to the learning dilemma for recurrent networks of spiking neurons. *bioRxiv*, page 738385.

Boparai, K. S., Singh, R., and Singh, H. (2016). Development of rapid tooling using fused deposition modeling: a review. *Rapid Prototyping Journal*, **22**(2), 281–299.

Borges, R., Borges, F., Lameu, E., Batista, A., Iarosz, K., Caldas, I., Viana, R., and Sanjuán, M. (2016). Effects of the spike timing-dependent plasticity on the synchronisation in a random hodgkin–huxley neuronal network. *Communications in Nonlinear Science and Numerical Simulation*, **34**, 12–22.

Burkitt, A. N. (2006). A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological cybernetics*, **95**(1), 1–19.

Caporale, N. and Dan, Y. (2008). Spike timing–dependent plasticity: a hebbian learning rule. *Annu. Rev. Neurosci.*, **31**, 25–46.

Cichy, R. M., Khosla, A., Pantazis, D., Torralba, A., and Oliva, A. (2016). Comparison of deep neural networks to spatio-temporal cortical dynamics of human visual object recognition reveals hierarchical correspondence. *Scientific reports*, **6**, 27755.

Gerstner, W., Lehmann, M., Liakoni, V., Corneil, D., and Brea, J. (2018). Eligibility traces and plasticity on behavioral time scales: experimental support of neohebbian three-factor learning rules. *Frontiers in neural circuits*, **12**.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, **9**(8), 1735–1780.

Huang, S., Rozas, C., Trevino, M., Contreras, J., Yang, S., Song, L., Yoshioka, T., Lee, H.-K., and Kirkwood, A. (2014). Associative hebbian synaptic plasticity in primate visual cortex. *Journal of Neuroscience*, **34**(22), 7575–7579.

Hwu, T., Isbell, J., Oros, N., and Krichmar, J. (2017). A self-driving robot using deep convolutional neural networks on neuromorphic hardware. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 635–641. IEEE.

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks*, **14**(6), 1569–1572.

Kar, K., Kubilius, J., Schmidt, K., Issa, E. B., and DiCarlo, J. J. (2019). Evidence that recurrent circuits are critical to the ventral stream's execution of core object recognition behavior. *Nature neuroscience*, **22**(6), 974.

Kern, S., Müller, S. D., Hansen, N., Büche, D., Ocenasek, J., and Koumoutsakos, P. (2004). Learning probability distributions in continuous evolutionary algorithms–a comparative review. *Natural Computing*, **3**(1), 77–112.

Kheradpisheh, S. R., Ganjtabesh, M., Thorpe, S. J., and Masquelier, T. (2018). Stdp-based spiking deep convolutional neural networks for object recognition. *Neural Networks*, **99**, 56–67.

Kietzmann, T. C., Spoerer, C. J., Sörensen, L. K., Cichy, R. M., Hauk, O., and Kriegeskorte, N. (2019). Recurrence is required to capture the representational dynamics of the human visual system. *Proceedings of the National Academy of Sciences*, **116**(43), 21854–21863.

Kintel, M. and Wolf, C. (2011). Openscad, the programmers solid 3d cad modeller.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Kuffner, J. J. and LaValle, S. M. (2000). Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 995–1001. IEEE.

Li, Y., Wang, Z., Midya, R., Xia, Q., and Yang, J. J. (2018). Review of memristor devices in neuromorphic computing: materials sciences and device challenges. *Journal of Physics D: Applied Physics*, **51**(50), 503002.

Lillicrap, T. P. and Santoro, A. (2019). Backpropagation through time and the brain. *Current opinion in neurobiology*, **55**, 82–89.

Liu, S.-C. and Delbruck, T. (2010). Neuromorphic sensory systems. *Current opinion in neurobiology*, **20**(3), 288–295.

Long, L. and Fang, G. (2010). A review of biologically plausible neuron models for spiking neural networks. In *AIAA Infotech@ Aerospace 2010*, page 3540.

Maass, W. (1997). Networks of spiking neurons: the third generation of neural network models. *Neural networks*, **10**(9), 1659–1671.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mozafari, M., Kheradpisheh, S. R., Masquelier, T., Nowzari-Dalini, A., and Ganjtabesh, M. (2018). First-spike-based visual categorization using reward-modulated stdp. *IEEE Transactions on Neural Networks and Learning Systems*.

Mozafari, M., Ganjtabesh, M., Nowzari-Dalini, A., Thorpe, S. J., and Masquelier, T. (2019). Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks. *Pattern Recognition*, **94**, 87–95.

Nurse, E., Mashford, B. S., Yepes, A. J., Kiral-Kornek, I., Harrer, S., and Freestone, D. R. (2016). Decoding eeg and lfp signals using deep learning: heading truenorth. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 259–266. ACM.

Otte, S., Zwiener, A., Hanten, R., and Zell, A. (2016). Inverse recurrent models–an application scenario for many-joint robot arm control. In *International Conference on Artificial Neural Networks*, pages 149–157. Springer.

Otte, S., Schmitt, T., Friston, K., and Butz, M. V. (2017a). Inferring adaptive goal-directed behavior within recurrent neural networks. In *International Conference on Artificial Neural Networks*, pages 227–235. Springer.

Otte, S., Zwiener, A., and Butz, M. V. (2017b). Inherently constraint-aware control of many-joint robot arms with inverse recurrent models. In *International Conference on Artificial Neural Networks*, pages 262–270. Springer.

Otte, S., Hofmaier, L., and Butz, M. V. (2018). Integrative collision avoidance within rnn-driven many-joint robot arms. In *International Conference on Artificial Neural Networks*, pages 748–758. Springer.

Paugam-Moisy, H. and Bohte, S. (2012). Computing with spiking neuron networks. *Handbook of natural computing*, pages 335–376.

Prodromakis, T. and Toumazou, C. (2010). A review on memristive devices and applications. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pages 934–937. IEEE.

Rinzel, J. (1990). Discussion: Electrical excitability of cells, theory and experiment: Review of the hodgkin-huxley foundation and an update. *Bulletin of Mathematical Biology*, **52**(1), 3–23.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., *et al.* (2016). Mastering the game of go with deep neural networks and tree search. *nature*, **529**(7587), 484.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, **10**(2), 99–127.

Swetham, T., Reddy, K. M. M., Huggi, A., and Kumar, M. (2017). A critical review on of 3d printing materials and details of materials used in fdm. *Int J Sci Res Sci Eng Technol*, **3**, 353–361.

Tomassini, M. (1999). Parallel and distributed evolutionary algorithms: A review.

Vandesompele, A., Walter, F., and Röhrbein, F. (2016). Neuro-evolution of spiking neural networks on spinnaker neuromorphic hardware. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6. IEEE.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., *et al.* (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5.

Werbos, P. J. *et al.* (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, **78**(10), 1550–1560.

Wong, K. V. and Hernandez, A. (2012). A review of additive manufacturing. *ISRN Mechanical Engineering*, **2012**.

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., and Stolcke, A. (2018). The microsoft 2017 conversational speech recognition system. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5934–5938. IEEE.

Zhao, T., Ma, X., Ma, H., and Wang, Y. (2018). Happier: Hierarchical polyphonic music generative rnn.

Zucker, M., Ratliff, N., Dragan, A. D., Pivtoraiko, M., Klingensmith, M., Dellin, C. M., Bagnell, J. A., and Srinivasa, S. S. (2013). Chomp: Covariant hamiltonian optimization for motion planning. *The International Journal of Robotics Research*, **32**(9-10), 1164–1193.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum	Unterschrift