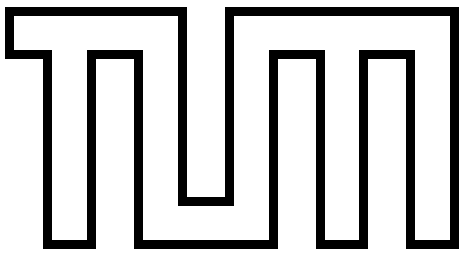


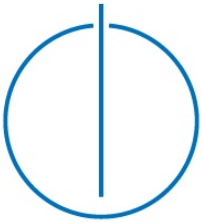
Technical University of Munich

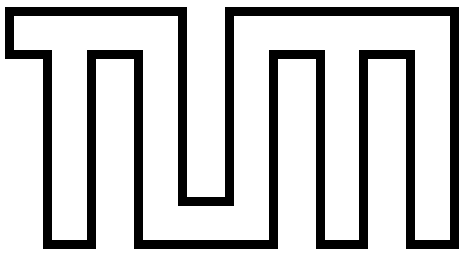
Department of Informatics

Master's Thesis in Robotics, Cognition, Intelligence

Training Neural Networks for Event-Based End-to-End Robot Control

Claus Meschede





Technical University of Munich

Department of Informatics

Master's Thesis in Robotics, Cognition, Intelligence

Training Neural Networks for Event-Based End-to-End Robot Control

Trainieren neuronaler Netzwerke zur eventbasierten end-zu-end Robotersteuerung

Author: Claus Meschede, B.Sc.

Supervisor: Prof. Dr.-Ing. habil. Alois Knoll

Advisor: Zhenshan Bing, M.Sc.

Submission: 28/07/2017

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 28th July 2017

(Claus Meschede)

Abstract

In the recent past, reinforcement learning algorithms such as Deep Q-Learning (DQN) have gained attention due to their promise of solving complex decision making tasks in high-dimensional state spaces. Despite the successes, the amount of neurons and connections in deep Artificial Neural Networks (ANN) makes applications computationally expensive, time-consuming and energetically inefficient, which is particularly problematic in real-time applications and mobile devices, e.g. autonomous robots. Event-based Spiking Neural Networks (SNN) performing computations on neuromorphic hardware could help solve some of these problems, although it is not clear yet how to train them efficiently, especially in reinforcement learning tasks. However, building energy-efficient mobile robots in the future could greatly benefit from SNNs with reinforcement learning capabilities.
Therefore, a simulated lane following task was implemented in this thesis using a Dynamic Vision Sensor (DVS) for event-driven data processing in order to provide a testing ground for training algorithms. Hereby, the problem was approached from two different directions. First, a policy was learned using the DQN algorithm and transferred afterwards using supervised learning such that the control output could be computed by a SNN. Second, based on a Reward-modulated Spike-Timing-Dependent-Plasticity (R-STDP) learning rule a SNN was trained directly in the same task.
Both approaches in this work have been shown to successfully follow the desired path in different scenarios with varying complexity. While after training the R-STDP-based controller shows a better performance, it lacks the reward-prediction capabilities needed for more complex decision making tasks. Until such an algorithm is found, indirect training using classical reinforcement learning methods could provide a practical workaround.

Kurzfassung

Reinforcement Learning Algorithmen, wie z.B. Deep Q-Learning (DQN), haben in den vergangenen Jahren aufgrund ihrer Fähigkeit selbst in hochdimensionalen Zustandsräumen Lösungsstrategien zu finden an Aufmerksamkeit gewonnen. Möglich wurde dies durch die Verwendung neuronaler Netzwerke (ANN), deren Anwendung jedoch oft zeit-, rechen- sowie energieintensiv ist und besonders bei Echtzeit-Anwendungen oder mobilen Anwendungen, z.B. autonomen Robotern, zu Problemen führen kann. Impulsbasierte neuronale Netwerke (SNN), die auf spezieller neuromorpher Hardware gerechnet werden können, könnten dabei helfen diese Herausforderungen zu lösen. Bisher fehlen allerdings die geeigneten Algorithmen, um diese effizient zu trainieren. Besonders in klassischen Reinforcement Learning Problemen könnten zukünftige Anwendungen hier von neuartigen Algorithmen profitieren.

Um unterschiedliche Algorithmen zu testen und zu vergleichen, wurden daher in dieser Arbeit verschiedene Szenarien simuliert, in denen ein Roboter mithilfe eines event-basierten Dynamic Vision Sensors (DVS) lernen soll der Spur einer Straße zu folgen. Dabei wurde das Problem aus zwei unterschiedlichen Richtungen betrachtet. Im ersten Fall wurde eine Steuerungsstrategie mithilfe des DQN Algorithmus gelernt und anschließend auf ein impulsbasiertes SNN übertragen. Im zweiten Fall wurde ein SNN direkt trainiert unter Verwendung einer vom Gehirn inspirierten Lernregel, der sogenannten Reward-modulated Spiking-Timing-Dependent-Plasticity (R-STDP).

Beide Algorithmen waren in der Lage Steuerungsstrategien in Szenarien mit unterschiedlicher Komplexität zu lernen und der Spur zu folgen. Die Steuerung des Roboters mithilfe des R-STDP Algorithmus wies dabei das präziseres Steuerungsverhalten auf, allerdings lassen sich aufgrund der fehlenden Schätzung zukünftiger Belohnungen bisher keine komplexeren Aufgaben lösen. Bis zur Entwicklung eines solchen Algorithmus könnte indirektes Training mithilfe klassischer Reinforcement Learning Methoden eine Alternative bieten.

Contents

Glossary

ANN	Artificial Neural Network (2. generation neural network)
CC	Classical Conditioning
CNN	Convolutional Neural Network
CS	Conditioned Stimulus
DQN	Deep Q-Network
DVS	Dynamic Vision Sensor
EPSP	Excitatory Post-Synaptic Potential
IF	Integrate-and-Fire neuron (without leaky part)
IPSP	Inhibitory Post-Synaptic Potential
LIF	Leaky-Integrate-and-Fire neuron
MDP	Markov Decision Process
NEST	Neural Simulation Technology
OC	Operant Conditioning
ROS	Robot Operating System
R-STDP	Reward-modulated Spike-Timing-Depdendent-Plasticity
SNN	Spiking Neural Network (3. generation neural network)
STDP	Spike-Timing-Depdendent-Plasticity
TD	Temporal Difference (Learning)
US	Unconditioned Stimulus
V-REP	Virtual Robot Experimentation Platform

List of Figures

List of Tables

1. Introduction

In recent years the idea of autonomously operating robots performing complex tasks has become a realistic prospect for the future, e.g. in the form of self-driving cars, space exploration robots, collaborative industrial robots or cleaning and service robots. The fast pacing technological advancements of the digital age can be attributed to a variety of sources, arguably the most important ones being improvements on the hardware and algorithmic side as well as cheap sensors available through mass production.
On the data side, Artificial Neural Networks (ANN) inspired by the inner workings of biological brains, have become powerful tools for complex computational problems such as image classification. Today, Convolutional Neural Networks (CNN) mimicking connectivity patterns of the visual cortex in animals have reached near-human performance in many machine learning problems [1]. In order to operate within the real world, robots need to make sense of their environment through sensors that typically deliver high-dimensional data. Due to hardware progress and refined training algorithms, deep network architectures today can be used to extract highly non-linear functions from training data, where classical algorithmic approaches often fail to handle the complexity involved in these problems. For control tasks and their inherent lack of training data, the theoretical framework of reinforcement learning has become a promising way of training neural networks. Recently, neural networks have successfully been trained to learn policies for complex control tasks such as playing Atari games [2] or robot control [3]. While reinforcement learning algorithms using neural networks remain an interesting research topic, capable algorithms are already available today.
Despite the hardware progress that made large neural networks applicable to real world problems, the high computational demands of these networks still take a toll. First, training artificial neural networks is time consuming and can easily take multiple days for state-of-the-art architectures. Executing large, trained networks is computationally expensive as well and typically produces high response latencies. Second, performing computations with large networks on traditional hardware usually consumes a lot of energy as well.

In self-driving cars for example, this results in computational hardware configurations consuming a few thousand Watts standing in stark contrast to the human brain that only needs around 20 Watts of Power [4]. Especially in mobile applications where real-time responses are important and energy supply is limited, these are considerable disadvantages.

A possible solution to some of these problems could be given by event-based neural networks that mimic the underlying mechanisms of the brain much more realistically. In nature, information is processed using impulses or spikes, making seemingly simple organisms able to perceive and act in the real world exceptionally well and outperform state-of-the-art robots in almost every aspect of life. While computations within conventional ANNs can be understood as approximation of the impulse rate over time, event-based networks or Spiking Neural Networks (SNN) perform computations based on individual spikes taking the precise timing into account as well. This approach could help processing information more efficiently both in terms of energy and data, e.g. using neurally inspired hardware such as the SpiNNaker board [5] or Dynamic Vision Sensors (DVS) [6]. Moreover, in contrast to classical reinforcement learning algorithms, information can be processed in continuous time facilitating low latency responses without the need for time-steps [7].

On the other hand, training these kind of networks is notoriously difficult, especially when it comes to deep network architectures. Since error backpropagation mechanisms commonly used in ANNs can't be directly transferred to SNNs due to non-differentiabilities at spike times, there has been a void of practical learning methods. On the level of single synapses, experiments have shown that the precise timing of pre- and post-synaptic spikes seems to play a crucial part in the change of synaptic efficacies [8]. With these Spike-Timing-Dependent-Plasticity (STDP) learning rules, networks have been trained in various tasks, e.g. digit recognition [9]. However, it is not clear how the brain assigns credit in deep hierarchies as efficiently as backpropagation does, although Lee et al. [10] have shown a practical work-around implementation of backpropagation in SNNs and recently Bengio et al. [11] provided an explanation attempt of connecting known biological learning mechanisms to backpropagation.

In nature, rewards seem to be playing a crucial role for the development of life in general. Evolution has programmed life forms of any kind to acquire rewards, and do it in the best possible way. To do so, the brain needs to identify the reward value of objects for survival and reproduction, and then direct the acquisition of these reward objects through learning, approach, choices, and positive emotions. Very similar questions arise in robotics, when there is often no data to learn from and a robots needs to adapt their behavior from experience only. For a long time, studying reward systems

has been done separately in computer- and neuroscience. It is only recently that the theoretical framework of Reinforcement Learning could be connected to its biological neural implementation [12]. Therefore, some research has been done trying to implement biologically plausible reinforcement learning algorithms based on experimental findings in SNNs. A learning rule that incorporates a global reward signal in combination with STDP is called Reward-modulated Spike-Timing-Dependent-Plasticity (R-STDP) [13][14].
Despite the shortcomings of todays event-based neural networks, Kaiser et al. [15] proposed a lane following task as testing ground for training algorithms, which is both a challenging and technically relevant task while being simple enough for basic simulations. They presented a framework for a neurorobotic vehicle simulation where images were provided by a simulated DVS. Meant as a proof of concept, a simple Braitenberg vehicle controller [16] was built based on a static SNN that performed the task. The framework was based on the open-source robot simulator Gazebo [17], using ROS [18] for communication between different modules, and NEST [19] for simulating SNNs. Performing event-based computations end-to-end, starting from sensors through neural networks to motor control, could greatly benefit from powerful learning rules and lead to practical applications in the future, e.g. energy-efficient mobile robots.
Therefore, in this work, a similar lane following task was implemented and adapted in order to test different training algorithms for SNNs. In the beginning, the task was reproduced using the robot simulator V-REP [20] and communicating via ROS as middleware as well. In order to train event-based neural networks in this task, the problem was approached from two different ends. First, in an indirect training setup, a conventional ANN was trained in a classical reinforcement learning setting using the Deep Q-Learning (DQN) algorithm published by Mnih et al. [2]. Afterwards, the learned policy was transferred using an indirect approach for supervised learning [21], such that the control output could be computed by a SNN. Second, the Braitenberg vehicle controller by Kaiser et al. [15] was reproduced with manually set weights and tested on the track. Finally, by replacing the static connections with R-STDP synapses [22], the network was trained directly learning the synaptic weights all by itself.
The thesis is structured as follows: In chapter 2, a short introduction into the theoretical basis of classical reinforcement learning, ANNs and SNNs, as well as DVS devices is given. A literature review, summarizing and condensing previously published approaches of robot control using SNNs is given in chapter 3. Chapter 4 describes the simulation environment for the lane following task, as well as implementation details of different controllers. In chapter 5, the controllers are tested, discussed and compared in different scenarios. Finally, in chapter 6, the work is summarized and potential future work is outlined.

2. Theoretical Background

This chapter is meant as a short summary of the theoretical foundations as well as the vocabulary that is used in the following chapters. For an in-depth introduction into reinforcement learning, it is recommended to have a look at Sutton and Barto [23]. An overview of SNNs can be found in [24] and [25].

2.1. Reinforcement Learning

For nervous systems found in humans and animals, rewards play a significant role for learning, approach behavior, choices and emotions. They make us eat, drink, mate and control higher level actions leading to possible future rewards. By evolutionary selection our brains have developed mechanisms to learn, select, approach and consume rewards in the best possible way. This means identifying objects, stimuli, events and activities that lead to the best rewards.
These underlying reward functions cannot be explained by environmental properties alone but the particular behavioral reactions they induce. Sensory receptors translate the energy from environmental events, such as vision, touch, pain, hearing, smell and taste into action potentials and send them to the brain. The incoming signals are then processed and the brain generates subjective preferences that reflect on specific environmental stimuli, objects, events, situations and activities as rewards. For approach behavior and decision-making, the brain needs to value possible rewards and choose between them. In this context, the value of a specific action or object can be described as the potential effects of a reward on survival and reproduction and is implemented in the brain in various neural reward signals.
Based on the idea of learning from interaction and external reward signals, a theoretical framework was developed in computer sciences that formalizes reward problems and gives a mathematical description to work with. Rather than defining a set of learning methods, reinforcement learning in this context can be understood as a characteristic problem definition. As opposed to supervised learning where examples are provided by

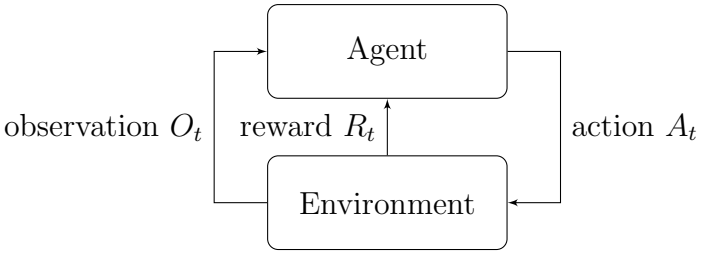


Figure 2.1.: Interaction between agent and environment.

a knowledgeable supervisor, reinforcement learning agents try to maximize a numerical *reward* that is given by the environment. In order to do that, they need to learn how to act optimally in different situations and how actions may affect possible subsequent rewards [23].

This iterative process (see Fig. 2.1) of observing the environment, acting on it and receiving rewards at each time step can be described as *Markov Decision Process (MDP)*. In order to understand how problems can be formulated as MDP and solved using reinforcement learning algorithms, it is important to understand the basic underlying concepts and definitions as well.

Hereby, *states* correspond to the information that is used by an agent to determine what happens next. Formally, it is a function of the agents history of observations, actions and rewards. If the agent can directly observe the state of the environment without uncertainty, the environment is called *fully observable*.

At each time step, the agent receives a feedback signal, indicating how well the agent is doing. Goal of the agent is to maximize this cumulative *reward* over all future time steps. In biological systems, for instance, this could be interpreted as pleasure or pain.

Formally, a MDP is a Markov Process with rewards and decisions. It is an environment in which all states fulfill the Markov Property, i.e. state transitions only depend on the current state and not on the history of all visited states before. Therefore, MDPs are usually defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $p^a_{s,s'} \in \mathcal{P}$ is the probability that action a in state s at time t will lead to state s' at time $t+1$, $r^a_{s,s'} \in \mathcal{R}$ is the immediate reward received after transitioning from state s to state s', due to action a and γ is the discount factor representing the difference in importance between future rewards and present rewards.

In order to achieve its goal of maximizing rewards, the agent has to learn some optimal behavior mapping states to actions. This behavior, called *policy* π, can either be a deterministic or a stochastic distribution over all actions. This corresponds to a set of stimulus-response rules or associations in psychology.

With these definitions, a *value function* under a policy π can be defined predicting future

rewards. This is important for evaluating states and therefore to select between actions:

$$v_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots | s] \tag{2.1}$$

In reinforcement learning problems, rewards are usually sparse in time and delayed. To execute an optimal sequence of actions leading to future rewards, the agent therefore needs to incorporate all future rewards into previously seen states, instead of only the next one. Since for many problems predicting all future rewards is not feasible, the discount factor γ can be used to effectively restrict this look ahead in time. This problem is usually referred to as *credit assignment problem.*
Reformulating this equation into a recursive form leads to the following formula, called *Bellman expectation equation*:

$$v_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s') | s] \tag{2.2}$$

In the optimal case where following a policy π^* maximizes all future rewards, the *Bellman optimality equation* can be formulated as

$$v_{\pi^*}(s) = \max_{a \in \mathcal{A}} \left(r^a_{s,s'} + \gamma \sum_{s' \in S} p^a_{ss'} v_{\pi^*}(s') \right). \tag{2.3}$$

In general, this non-linear function can't be solved in closed form, therefore many iterative solution methods have been developed, e.g. *Value Iteration, Policy Iteration, Q-Learning or SARSA.*
The Bellman equations illustrate two common problem definitions in reinforcement learning. First, given a policy π an agent usually needs to evaluate the future to reason about its actions. Second, the goal of course is to optimize the future by finding the best policy. These closely related problems are called *prediction* and *control*, respectively.
Furthermore, in most cases the agent's knowledge about the environment is incomplete. There are many problems in which it doesn't know the transition probabilities from one state to another in advance. Predicting what the environment will do next is called modeling in reinforcement learning. A model predicts the next state and reward when acting on the environment. Some methods solving the Bellman optimality equation like Value Iteration and Policy Iteration need to know about transition probabilities in advance. Other methods, e.g. Q-Learning and SARSA, can learn about the environment without having an exact model. Therefore, these algorithms are called *model-free.* Moreover, some other methods try to learn a model from experience that can be used for planning tasks.

Lastly, solving the Bellman optimality equation iteratively means learning while interacting with the environment and incorporating these experiences immediately into future decisions on which action to choose next. This means the agent has to gather information about the environment before it can use this experience to optimize its behavior. These two opposing goals pose a challenge that is commonly referred to as *exploration-exploitation* trade-off.

2.1.1. Temporal Difference Learning (TD)

A method for iteratively solving the aforementioned Bellman equation (see Eq. 2.1 - 2.3) is by sampling experiences. These experience samples can be used to update estimates of a value function towards a sampled target. The main difference of these methods is the way in which these targets are constructed. *Monte Carlo* methods, for example, use complete sampled episodes until a terminal state has been reached and update every visited state in retrospect towards a target based on Equation 2.1. A big advantage of learning directly from raw sampled experiences is that a model of the environment's dynamics is not needed. *Temporal difference (TD)* learning methods construct the updating target for value functions slightly different, such that they are still model-free while not having to wait for a final outcome as Monte Carlo methods do. Even though both methods use experience to update their value function estimate based on what happens after that visit, TD methods need to wait only until the next time step and not until the end of an episode. This often turns out to be a critical consideration in many applications and can be achieved by combining sampled experiences with previous estimates into an updating target following the Bellman expectation equation (Eq. 2.2). Using an updating method that is in part based on an existing estimate is called *bootstrapping*. The simplest TD method looking only one step ahead from state s to state s' is called *TD(0)* and updates its value function estimates for each state according to the following formula with the learning rate α and the discount factor γ:

$$v(s) \leftarrow v(s) + \alpha\Big[\underbrace{\underbrace{\underbrace{r_{t+1}}_{\text{sample}} + \gamma \cdot \underbrace{v(s')}_{\text{estimate}}}_{\text{target}} - v(s)}_{\text{prediction error}}\Big] \tag{2.4}$$

For a fixed policy π, TD methods have been proved to converge to the correct value function v_π. While this property holds for Monte Carlo methods as well, TD methods have usually been found to converge faster than constant-rate Monte Carlo methods on

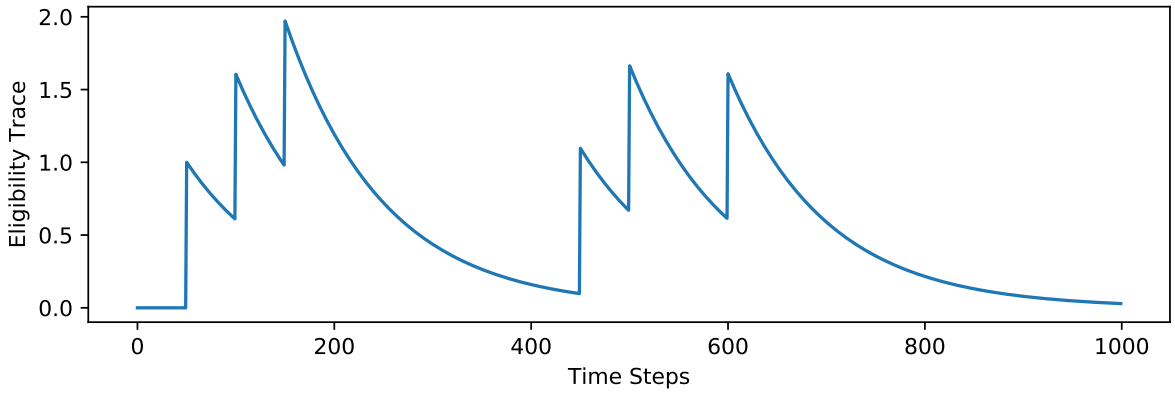


Figure 2.2.: Example eligibility trace according to Equation 2.6. It was computed for 1000 time steps with a decay factor $\alpha = 0.99$ and state visits at time steps 50, 100, 150, 450, 500, 600.

stochastic tasks [23].
So far, the TD(0) algorithm can predict a value function for a fixed policy π in a fully incremental, on-line fashion, but it doesn't solve the control problem finding an optimal policy π^*. This is usually done by using some variation of generalized *policy iteration* where most methods primarily differ in their approaches to the prediction problem. For instance, one could choose an action a at each time step following an ϵ-*greedy* policy resulting in an algorithm called *SARSA*. Given $\epsilon \in [0, 1]$, following this policy means choosing an action with probability $1-\epsilon$ that maximizes future rewards or choose a random action otherwise. This algorithm also has been proven to converge to the optimal policy π^* if some specific conditions are satisfied [23].

2.1.2. Eligibility Traces

In Equation 2.4, the reward prediction error of the TD(0) algorithm is computed by looking one state ahead in time. For propagating distal rewards back in time to solve the credit assignment problem, states have to be revisited many times before the algorithm will converge. In other cases the algorithm may fail to predict the correct value function at all.
A possible solution for this problem could be to extend the updating target in the TD(0) algorithm to more steps. Instead of sampling only one step in the MDP, the prediction error could be computed using several or even all future rewards. The problem with this approach is that in practice looking multiple states ahead means having to wait multiple steps for value updates. In fact, computing the TD error using all future rewards is

equivalent to the previously discussed Monte Carlo methods.
Therefore, instead of looking ahead in time by computing the updating target with several steps of delay, it is also possible to look back in time and give credit to previously visited states. This is achieved by introducing a weighted track of visits that combines a frequency heuristic assigning credit to the most frequently and a recency heuristic assigning credit to the most recently visited states. Using this so-called Eligibility trace, received rewards can be credited to states several steps back in time:

$$E_0(s) = 0 \tag{2.5}$$

$$E_t(s) = \alpha E_{t-1}(s) + 1(S_t = s) \tag{2.6}$$

Interestingly, under certain conditions it can be shown that the forward view algorithm looking one or more steps ahead and the backward view algorithm where credit is assigned using the eligibility trace in fact amount to the identical sum of offline updates [23]. In Figure 2.2 a hypothetical eligibility trace for a single state is shown as a function over time steps. An algorithm that makes use of this principle is called TD(λ), where $\lambda \in [0; 1]$ defines the extent of the look ahead.

2.1.3. Deep Q-Learning

Instead of using the value function as defined in Equation 2.1, many algorithms make use of the slightly different *action-value function* that predicts all future rewards when performing a certain action $a \in \mathcal{A}$ at a state $s \in \mathcal{S}$:

$$q_\pi(s, a) = \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s, a] \tag{2.7}$$

As before, for deterministic policies this equation can be recursively reformulated into another version of the Bellman expectation equation:

$$q_\pi(s, a) = \mathbb{E}_\pi[r_{t+1} + \gamma q_\pi(s', \pi(s'))|s, a] \tag{2.8}$$

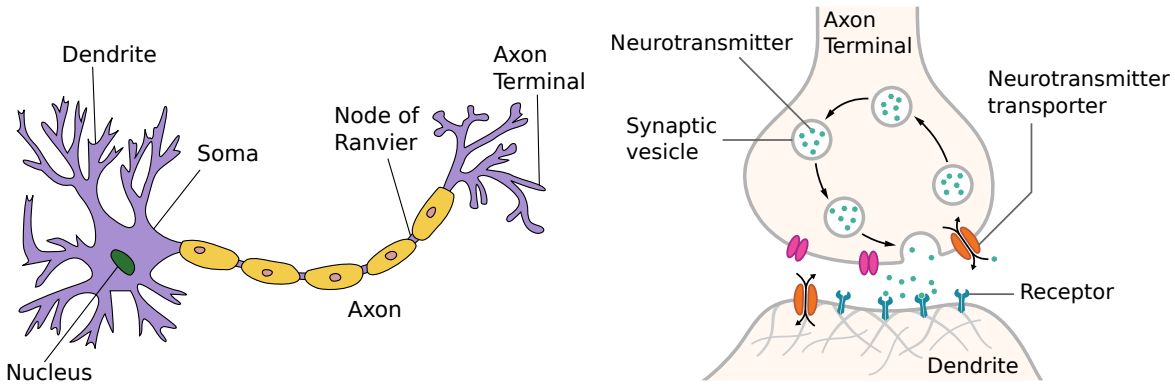
By iteratively updating the action-values according to the Bellman equation, reinforcement learning algorithms can solve MDP problems with guaranteed convergence of q to its optimal value q_* for each state-action pair.
If the optimal value $q_*(s', a')$ was known for all possible actions a' at the next time step, an optimal strategy maximizing all future rewards could intuitively be formulated as

$$q_*(s, a) = \mathbb{E}_{s'}[r_{t+1} + \gamma \max_{a'} q_*(s', a')|s, a] \tag{2.9}$$

An algorithm that makes use of this strategy by combining the Bellman equation (Eq. 2.8) with an ϵ-greedy policy is called *Q-Learning* [26]. As other value iteration algorithms do, it uses Equation 2.9 for iterative updates of the action-values. Since this updating procedure does not need any transition probabilities from one state to another, this algorithm is model-free. Furthermore, this *off-policy* algorithm can learn an optimal strategy, even if the sampled experiences are following a different policy, which is a particularly useful property for training neural networks.
Unfortunately, while this approach works very well for reinforcement learning problems with small and discrete state and action spaces, it is bound to fail for problems with many states and actions. Following the curse of dimensionality the number of states can quickly explode for many problems, making approaches without any form of generalization practically impossible.
Therefore, instead of estimating the action-values for each state-action pair separately, linear or non-linear function approximators can be used to generalize over states. Parametrized by θ, such function approximators can be trained by minimizing the mean-squared error in the Bellman Equation 2.8:

$$L(\theta) = \mathbb{E}\left[\left(q_{action}(s, a|\theta) - (r(s, a) + \gamma q_{target}(s', \pi(s')|\theta))\right)^2\right] \tag{2.10}$$

Unfortunately, training non-linear function approximators has been proven difficult in the past. In contrast to other linear approximators, there are no theoretical performance guarantees and training tends to be unstable. In part, this is due to correlations between subsequent experience samples that lead optimization algorithms into local minima, rather than global ones. Another problem here are the correlations between prediction and updating target of the loss function, because both of them depend on θ.
Mnih et al. [2][27] have addressed these problems with two novelties in the learning process. First, they introduced an *experience buffer*. In this buffer, experienced states are stored. For training of the neural network, randomly drawn sets of experiences can be used for stochastic gradient descent optimization of the loss function. This is possible due to the off-policy nature of the Q-Learning algorithm and decorrelates subsequent states. Second, two separate networks are used for choosing actions for sampling and estimating the action-values in the loss function. Hereby, the parameters of q_{target} are only updated periodically to reduce correlations.
Later, Lillicrap et al. [3] extended this idea to an actor-critic reinforcement learning architecture, which allowed for continuous action space formulations.



(a) Neuron [30] (b) Synapse [31]

Figure 2.3.: Structure of the nervous system.

2.2. Artificial Neural Networks

2.2.1. Biological Background

While todays understanding of the human brain remains rather incomplete and challenging, some insights into our neural structure have been made over the past couple of decades. Since the initial discovery of neurons as basic structure of the nervous system by Santiago Ramón y Cajal at the beginning of the twentieth century, a rough concept of how neurons might work has been developed. At the very basis, neurons can be understood as simple building blocks processing incoming information in the form of short pulses of electrical energy into output signals. By connecting neurons to huge networks, complex dynamics emerge that can process information and make sense of our world. This basic concept can be found all over nature, from simpler organisms like jellyfish with a couple of thousand neurons to humans with an estimated number of 86 billion neurons on average in our nervous system [28]. Typically, a single neuron in our brain is connected to over 10.000 other neurons [29].

Figure 2.3a shows the structure of a typical neuron of the human brain embedded in a salty extra-cellular fluid. Incoming signals from multiple dendrites alter the voltage of the neuronal membrane. When a threshold is reached, the cell body or soma sends out an action potential - also called spike or pulse - itself. This process of generating a short (1ms) and sudden increase in voltage is usually referred to as spiking or firing neuron. It is followed by a short inactive period called refractory period, in which it can't send out other spikes regardless of any incoming signals.

Once the membrane potential threshold has been reached and the neuron fires, the

generated output spike is transmitted via the axon of a neuron. These can grow quite long and branch out to a multitude of other nervous cells at the end. Every cubic millimeter of our cortex contains up to 4 kilometers of them [32]. To allow the electrical signals being quickly propagated down a neuron, the coated axon has periodic breaks called nodes of Ranvier that basically function as amplifiers for the traversing signals [25].
At the end of an axon, the axon terminal, incoming signals are transmitted to other nervous cells, such as other neurons or muscular cells. While for a long time these so-called synapses were thought to simply transfer signals from one neuron to another, there is now overwhelming evidence that synapses are in fact one of the most complicated part of a neuron. On top of transmitting information, they work as signal pre-processor and play a crucial part in learning and adaption for many neuroscientific models. When a traversing spike reaches an axon terminal, it can cause a synaptic vesicle to migrate towards the presynaptic membrane, as shown in Figure 2.3b. Vesicles are spherical containers that can store and release different kinds of neurotransmitters, the chemicals that transmit signals across chemical synapses. At the presynaptic membrane, the triggered vesicle will fuse with the membrane and release its stored neurotransmitters into the synaptic cleft filled with the extra-cellular fluid. After diffusing into this gap, neurotransmitter molecules have to reach a matching receptor at the postsynaptic side of the gap and bind with them. Directly or indirectly, this causes postsynaptic ion-channels to open or close. The resulting ion flux initiates a cascade that traverses the dendritic tree down to the trigger zone of the soma, changing the membrane potential of the postsynaptic cell. Therefore, different neurotransmitters can have opposing effects on the postsynaptic neurons excitability, thus mediating the information transfer. These effects, making postsynaptic cells either more or less likely to fire action potentials are called excitatory postsynaptic potential (EPSP) or inhibitory postsynaptic potential (IPSP), respectively. The dependence of postsynaptic potentials on different amounts and types of neurotransmitters released and the resulting number of ion-channels activated, is often in short referred to as synaptic efficacy. After a while, neurotransmitter molecules are released again from their receptors into the synaptic cleft and either reabsorbed into the presynaptic axon terminal or decomposed by enzymes in the extra-cellular fluid.
The properties, characteristics and capabilities of synapses as signal pre-processors, e.g. chances of vesicle deployment or regeneration and amount of receptors, aren't fixed, but always changing depending on the short and long term history of its own and outside influences. Neuro-hormones in the extra-cellular fluid can influence both the pre- and postsynaptic terminals temporarily, i.e. by enhancing vesicle regeneration or blocking neurotransmitters from activating ion-gate receptors. Psychoactive drugs and neurotoxins, for example, can change properties of neurotransmitters being released and reabsorbed in

a synapse. All these effects, changing the influence of incoming spikes on the postsynaptic membrane potential are usually referred to as synaptic plasticity and form the basis of most models of learning in neuro- and computer-sciences.
While most of the neurons in our nervous system are connected via axon-dendrite connections (axodendritic synapse), other arrangements exist as well, e.g. axon-soma (axosomatic synapse), axon-axon (axoaxonic synapse) or axon-bloodstream (axosecretory synapse).

2.2.2. From McCulloch-Pitts to Backpropagation

In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a theoretical paper on how neurons might work describing a simple neural network model using electrical circuits [33]. Capable of performing mathematical operations with boolean outputs, these first generation neural networks fire binary signals once a threshold of summed incoming signals is reached in a neuron. They have been successfully applied in powerful artificial neural networks such as multi-layer perceptrons and Hopfield nets [34]. With the advent of more powerful computing, this concept has later been extended by introducing continuous activation functions, e.g. sigmoid or hyperbolic tangent functions, to handle analog in- and outputs as well. In fact, it has been proven that sufficiently large neural networks with continuous activation functions can approximate any analog function arbitrarily well by altering the network information flow through their synaptic weights [35]. The most commonly used and powerful supervised learning algorithm that takes advantage of continuous activation functions by using gradient-descent on an error function is called backpropagation [36].
While second generation neurons do not model electrical pulses that have been described in their biological counterparts, their analog information signals can actually be interpreted as an abstracted rate coding. Over a certain period of time, an averaging window mechanism can be used to code pulse frequencies into analog signals giving these models a biologically more plausible meaning.

2.2.3. Spiking Neural Networks

Following its biological counterpart, a third generation of neural networks has been introduced that directly communicates by individual sequences of spikes. Instead of using abstracted information signals, they use pulse coding mechanisms that allow incorporating spatial-temporal information that would otherwise be lost by averaging

over pulse frequencies. It becomes clear that these neural network models, referred to as Spiking-Neural-Networks (SNN), can be understood as an extension to second generation neural networks and can in fact be applied to all problems solvable by non-spiking neural networks. In theory it has been shown that these models even are computationally more powerful than perceptrons and sigmoidal gates [37].
Due to their functional similarity to biological neurons [38], SNNs have become a scientific tool for analyzing brain processes, e.g. helping to explain how the human brain can process visual information in an incredibly short amount of time [39]. Moreover, SNNs promise solutions for problems in applied engineering as well, e.g. fast signal-processing, event detection, classification, speech recognition, spatial navigation, motor control or sensor fusion [24]. In combination with event-based sensors such as DVSs, these networks promise power efficient, low latency alternatives to second generation neural networks, e.g. for applications in robotics [10].
In the literature, many different mathematical descriptions of spiking neural models have been proposed, processing excitatory and inhibitory inputs using internal state variables. One of the most widely used models is the so-called Leaky-Integrate-and-Fire (LIF) model that can be easily explained by the principles of electronics. These models are based on the assumption that the timing of spikes rather than the specific shape carries neural information. The sequences of firing times are called spike trains and can be described as

$$S(t) = \sum_t \delta(t - t^f), \tag{2.11}$$

where $f = 1, 2, \ldots$ is the label of a spike and $\delta(\cdot)$ is a Dirac function with $\delta(t) \neq 0$ for $t = 0$ and $\int \delta(t)dt = 1$.
Passing a simplified synapse model, the incoming spike train will trigger a synaptic electric current into the postsynaptic neuron. This input signal induced by a spike train $S_j(t)$ can in a simple form be described by the exponential function

$$i(t) = \int_0^\infty S_j(s - t) \exp(-s/\tau_s)ds. \tag{2.12}$$

Here, τ_s denotes the synaptic time constant. This synaptic transmission can be modeled by low-pass filter dynamics (Fig. 2.4B).
The postsynaptic current then charges the LIF neuron model increasing the membrane potential u according to

$$C\frac{du}{dt}(t) = -\frac{1}{R}u(t) + \left(i_0(t) + \sum w_j i_j(t)\right). \tag{2.13}$$

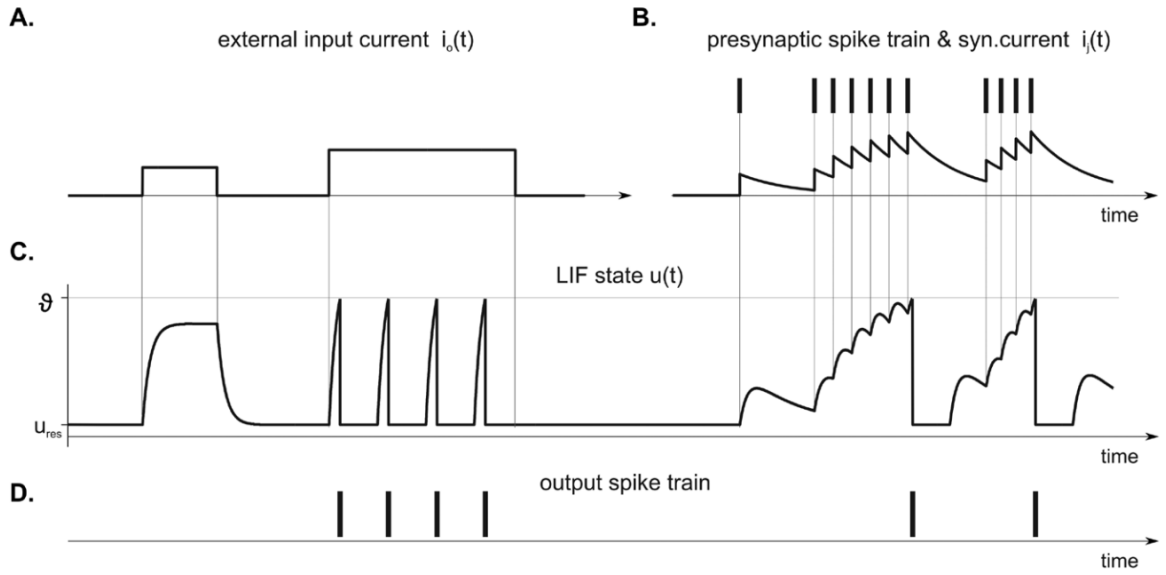


Figure 2.4.: Ponulak and Kasinski [24] show a LIF unit plotted over time for an exemplary external input current $i_0(t)$ (A) and presynaptic spike train (B). (C) shows the membrane potential u reaching its threshold ϑ and producing output spikes (D).

In terms of electronics, this corresponds to charging a capacity C that slowly "leaks" voltage over time, depending on the resistance R. If $R \to \infty$, the LIF model reduces to the simple Integrate-and-Fire (IF) unit. $i_0(t)$ denots an external current driving the neural state, $i_j(t)$ is the input current from the j-th synaptic input and w_j represents the strength of the j-th synapse. Once the membrane potential u reaches a certain firing threshold ϑ, the neuron fires a single spike and its membrane potential is set back to $u_{res} < \vartheta$. Usually, this spiking event is followed by a refractory period in which the neuron stays inactive and can't be charged again (Fig.2.4C and Fig. 2.4D).
A rather good analogy illustrating this concept is a balloon that is being inflated. To represent the energy potential leaking away, one could imagine a small hole in it. Once the air pressure inside of the balloon reaches a certain limit, it will burst.

2.2.3.1. Unsupervised Learning

One of the earliest theories in neuroscience explaining the adaption of synaptic efficacies in the brain during the learning process was introduced by Donald Hebb in his 1949 book *The Organization of Behavior* [40]. With his seemingly simple learning rule that is often summarized by the phrase *"Cells that fire together, wire together"*, Hebbian processes can reorganize the connections within neural networks and, under certain conditions, result

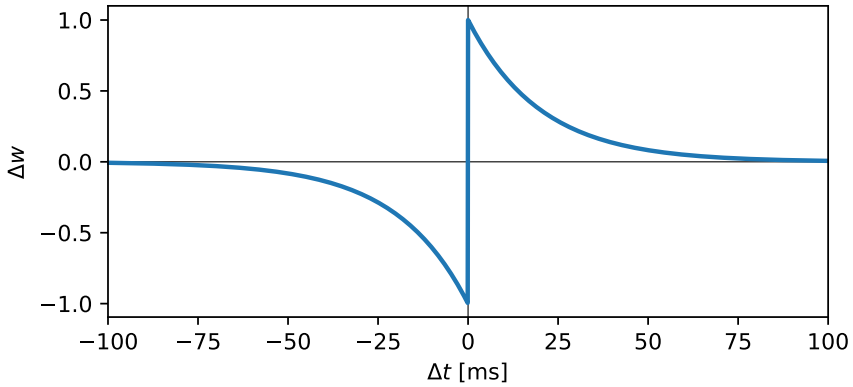


Figure 2.5.: Learning curve of the STDP mechanism with $A_+ = 1.0$ and $A_- = 1.0$. τ_+ and τ_- are set to $20\,ms$. The same parameters were used for the simulations in this work as well (Tab. A.4).

in the emergence of new functions. Because of the absence of direct goals, correction functions or a knowledgeable supervisor, this kind of learning is usually categorized as unsupervised learning [41]. Learning based on Hebb's rule has been successfully applied to problems such as input clustering, pattern recognition, source separation, dimensionality reduction, formation of associative memories, or formation of self-organizing maps [41].
However, while in his early work Hebb interpreted the neuronal activity in his formula as neural firing rate, he was also aware of the shortcomings of this model. First, this formula did not account for depression and had to be adapted in later experimental and theoretical work. Second, because of the causal and temporal dependencies of two neurons, one cannot have caused or taken part in firing the other, if they fire exactly at the same time. Indeed, later experiments showed that the synaptic plasticity is influenced by the exact timing of individual spikes, in particular by their order [42][43]. If a presynaptic spike preceded a postsynaptic spike, a potentiation of the synaptic strength could be observed, while the reversed order caused a depression. This phenomenon has been termed Spike-Timing-Dependent-Plasticity (STDP) or anti-STDP for the exact opposite impact and explains the activity-dependent development of nervous systems. In other words, neural inputs that are likely to have contributed to the neurons excitation are strengthened, while inputs that are less likely to have contributed are weakened. After some time, this process often leads to some remaining, important connections while many other synaptic weights reduce to zero. Remaining inputs are therefore usually correlated in time. Biologically, experiments could attribute STDP at least in part to mechanisms involving postsynaptic receptors, although it remains unclear to what extent other mechanisms are contributing as well [44].
In the past, different mathematical models of STDP have been proposed, e.g. by Gerstner and Kistler [45]. For this work, the weight update rule under STDP as a function of the

time difference between pre- and postsynaptic spikes was defined as

$$\Delta t = t_{post} - t_{pre} \tag{2.14}$$

$$STDP(\Delta t) = \begin{cases} A_+ e^{-\Delta t/\tau_+}, & \text{if } \Delta t \geq 0 \\ -A_- e^{\Delta t/\tau_-}, & \text{if } \Delta t < 0 \end{cases}, \tag{2.15}$$

with A_+ and A_- representing positive constants scaling the strength of potentiation and depression, respectively. τ_+ and τ_- are positive time constants defining the width of the positive and negative learning window. This learning curve is shown in Figure 2.9.
Based on a STDP learning rule, Diehl and Cook [9] have shown a SNN for digit classification that learns without presenting any labels. The network relying on commonly used mechanisms, i.e. exponential synapses with STDP, reached an accuracy of 0.95 on the MNIST benchmark [46].

2.2.3.2. Supervised Learning

In non-spiking neural networks, many successes in recent years can be credited to finding ways to efficiently learn from labeled data. This type of learning, where a neural network mimics a known outcome from given data is called supervised learning. A variety of different neuroscientific studies have shown that this type of learning can also be found in the human brain, e.g. in motor control and motor learning [47][48]. But despite of the extensive exploration of these topics, the exact mechanisms of supervised learning in biological neurons remain unknown. Several models have been proposed how this might work, either by using activity templates to be reproduced [49][50][51] or error signals to be minimized [52][48]. In the nervous system, these teaching signals might be provided by sensory feedback or other supervisory neural structures [53]. Unfortunately, most of these methods were designed for single-layer networks with limited capabilities.
Alongside efficient supervised learning algorithms, the successes of non-spiking neural networks can also be assigned to their powerful structure of up to thousands of layers, capable of approximating high-dimensional functions in unseen precision [54]. The most widely used algorithm for this task has been backpropagation [36], where a global error function is minimized. Due to the continuous nature of second generation activation functions, this error is differentiable and can be used in gradient descent algorithms. However, spiking neural networks usually don't have this continuous property.
In the literature, there are different proposed solutions to this problem, one of which is to use different data representations between training and processing. For training, Diehl et al. [21] used a conventional non-spiking neural network that can be trained regularly

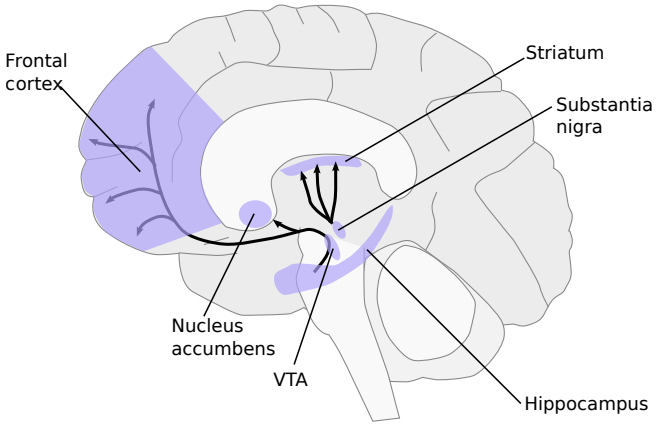


Figure 2.6.: Major dopamine pathways. As part of the reward pathway, dopamine is manufactured in nerve cell bodies located within the ventral tegmental area (VTA) and is released in the nucleus accumbens and the prefrontal cortex. The motor functions of dopamine are linked to a separate pathway, with cell bodies in the substantia nigra that manufacture and release dopamine into the dorsal striatum. [55]

with backpropagation. Using a conversion algorithm, the obtained weights can then be translated into an equivalent SNN. Another solution to this problem is to tweak and simplify the mathematical structure of SNNs in such a way, that stochastic gradient descent algorithms like backpropagation can be used. Lee et al. [10] achieved this by treating the mebmrane potentials of spiking neurons as differentiable signals, where discontinuities at spike times are considered as noise.

2.2.3.3. Reinforcement Learning

The key substrates for the brains function in reward are specific neuronal signals that occur in a limited number of brain structures, e.g. the ventral tegmental area (VTA) [12] (Fig. 2.6). The primary, homeostatic or pleasurable reward functions are innate and determined by the physiology of the body and its brain that emerged from evolutionary selection.

Dopamine neurons forming the midbrain dopaminergic cell groups are crucial for executive functions, motor control, motivation, reinforcement and rewards. The comparatively few neurons are confined in groups to a few, relatively small brain areas and synthesize and release dopamine. Dopamine is an organic chemical that has several important functions in the brain and body, e.g. as neurotransmitter or as chemical messenger outside the central nervous system. The somata of these dopamine neurons produce enzymes that

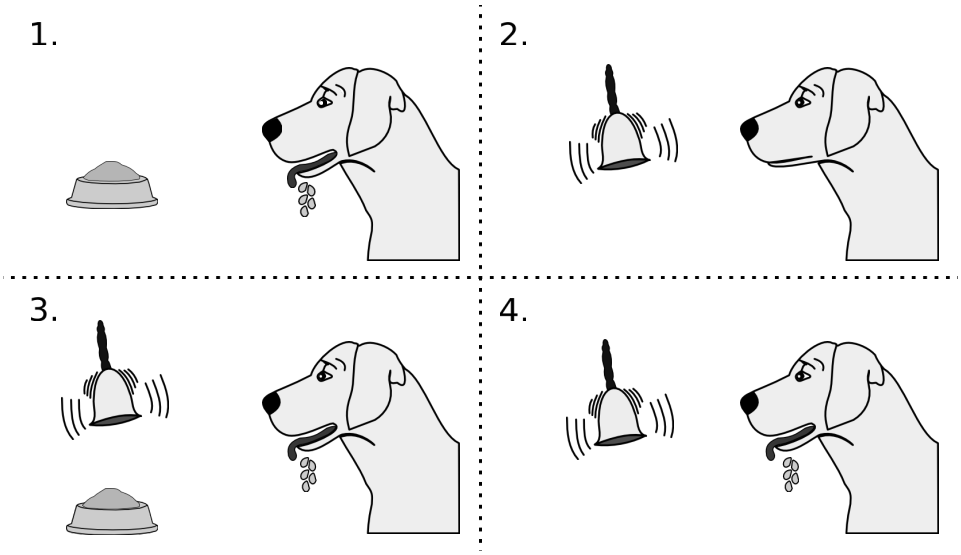


Figure 2.7.: Classical Conditioning by Pavlov. (1) Before Conditioning, the dog starts to salivate when it sees food. (2) When a bell rings, the dog doesn't salivate. (3) During Conditioning the bell is rung shortly before the food is presented. (4) After conditioning, the dog will start salivating if the bell is rung. [58]

can be transmitted via the projecting axons to their synaptic destinations. The Axons project to many other brain areas along the so-called dopaminergic pathways where most of the dopamine is eventually produced. This physiological process of regulating diverse populations of neurons with one or more chemicals emitted by a given neuron is known as neuromodulation. Other examples of neuromodulators in the brain include serotonin, acetylcholine or histamine. Most types of neurological rewards increase the level of dopamine in the brain, thus stimulating the dopamine neurons [12]. For initiating specific actions high levels of dopamine result in early action selections, because the amount of dopamine sets a threshold in the brain. In fact, many diseases as Parkinsons can be linked to malfunctioning dopaminergic systems [56], Furthermore, dopamine functions as teaching signal, altering the neural structure in such a way that responses will get easier to evoke in the future [57].

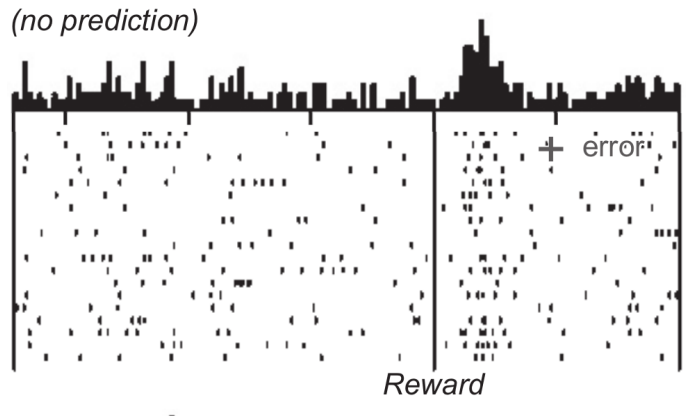
Evolution has given humans and animals learning capabilities to efficiently assess information and act on it. It enables organisms to adapt to their environment and thus live in more widely varying situations to acquire more food and mating partners. The selection of certain behaviors beneficial for survival is deeply connected to their experienced past and expected future rewards. These concepts are represented in Pavlovian and operant learning.

In Pavlov's famous experiment [59], a dog salivates to a bell when a sausage often follows, but it does not salivate just when a bell rings without consequences. From the dog's point

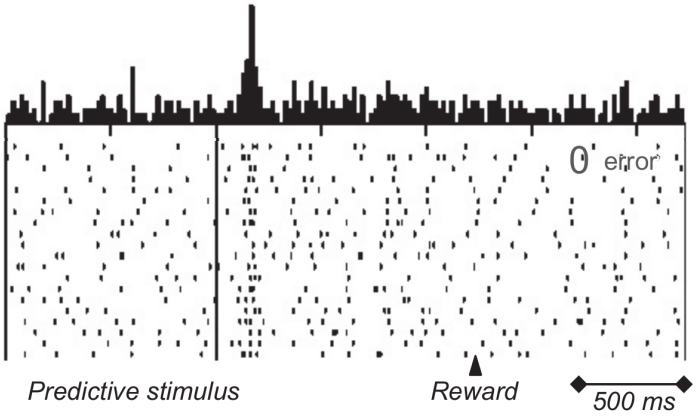
of view, the bell in this experiment has become a sausage predictor. This concept of an external stimulus that becomes a predictor and evokes an internal expectation is knwon as Pavlovian learning and constitutes one of the core concepts of learning theory (Fig. 2.7). It sets up predictions on stimuli indicating that a reward is going to occur this time and also contains information about the reward. In operant learning, on the other hand, rewards require own actions. In another experiment by Thorndike [60] for example, a cat runs around a cage and, among other things, presses a lever and suddenly gets some food, leading the cat to pressing again, and again, coming back for more. The cat's behavior in this context becomes causal for obtaining rewards. Thus, operant learning represents a mechanism by which we act on the world to obtain more rewards. In eliciting goal-directed action, it increases the possibilities to obtain rewards and enhances the chances for survival and reproduction. In these experiments, positive reinforcers increase and maintain the strength of the behavior that leads to them.

As described in Section 2.1.1 and formalized in Equation 2.4, learning can be understood as updating an estimate towards a prediction error. This formal concept has been studied in several neurophysiological studies in single neurons in monkeys. These conducted experiments allow for well detailed, quantitative behavioral assessments while controlling confounds from sensory processing, movements and attention. Results from tests with several consecutive conditioned stimuli suggest that dopamine neurons code the TD prediction error exactly as the predicted value changes between the stimuli. In such tasks, dopamine responses match closely the temporal profiles of TD prediction errors that increase and decline with sequential variations in the discounted sum of future rewards (Fig. 2.8). Thus dopamine neurons code reward value at the time of conditioned and unconditioned stimuli relative to the reward prediction at that moment [61][62]. However, in a study by Matsumoto and Hikosaka [63] it was shown that dopamine neurons do not behave as uniformly as previously thought with varying responses to positive and negative stimuli. In humans, dopamine-dependent prediction errors have been shown to guide decisions as well [64]. Moreover, dopamine has also been shown to modulate spike-timing-dependent plasticity (STDP), although the exact spike-timing and dopamine requirements for induction of long-term potentiation (LTP) and long-term depression (LTD) are still unclear [65][66][67].

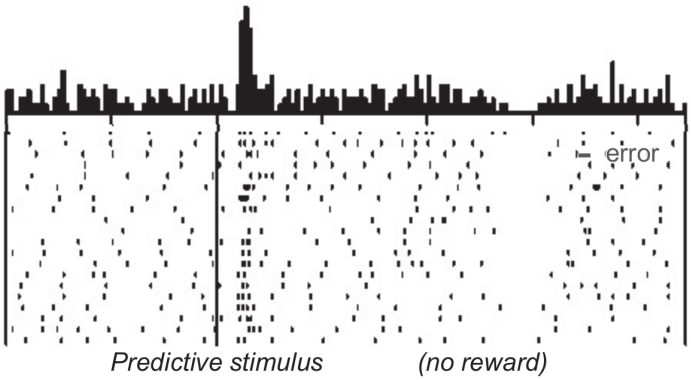
Despite the growing understanding of how reward-based learning might be implemented in the neural structure of the brain, there are still lots of open questions. E.g. it is not clear which biological mechanisms dopamine neurons use to compute TD prediction errors, since it involves the concept of "previous steps", thus requiring some form of memory. Neurons communicate through action potentials in continuous time, which is not easily transferable to the framework of MDPs. Biologically plausible time-steps are generally



(a) + error: Reward without predictive stimulus



(b) 0 error: Reward following predictive stimulus



(c) - error: No reward after predictive stimulus

Figure 2.8.: In a study on single dopamine neurons in monkeys Schultz et al. [61] presented visual stimuli on a computer screen and used a liquid as reward. After performing experiments similar to Pavlov's experiment on classical conditioning, it could be observed that the neural activity of these dopamine neurons seems to encode a reward prediction error closely matching errors from the formal TD model.

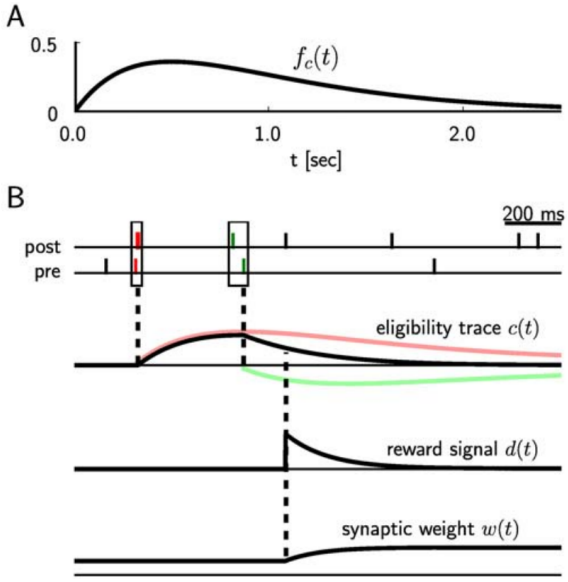


Figure 2.9.: In an exemplary figure, Legenstein et al. [13] showed the scheme of reward-modulated STDP according to Equations 2.15 - 2.18. (A) Eligibility function $f_c(t)$, which scales the contribution of a pre/post spike pair (with the second spike at time 0) to the eligibility trace $c(t)$ at time t. (B) Contribution of a pre-before- post spike pair (in red) and a post-before-pre spike pair (in green) to the eligibility trace $c(t)$ (in black), which is the sum of the red and green curves. According to Equation 2.19 the change of the synaptic weight w is proportional to the product of $c(t)$ with a reward signal $d(t)$.

difficult to envision and there seems to be no evidence for that either. From an evolutionary point of view, continuous time mechanisms simply seem more favourable. Furthermore, it remains unknown how the brain is capable of using these seemingly simple low-level mechanisms to learn complex functions for an abstracted view of the world. As a result of these incompatibilities, publications of reward-based learning in Spiking Neural Networks have been focusing on the loose ends of the neuroscientific understanding of the brain: How can neurons compute and learn from TD prediction errors in continuous time? What are the mechanisms of global reward signals, e.g. in the form of synthesized dopamine, modulating plasticity processes like STDP?

A simple learning rule combining models of STDP and a global reward signal was proposed by Izhikevich [68] and Florian [14]. In this work, a synapse model based on Potjans et al. [22] was used. It is based on the standard rule for STDP, modelling the synaptic weight change in dependence on the time difference between firing times of pre- and post-synaptic neurons (Eq. 2.15). Instead of instantaneously applying these weight changes to synapses as done in STDP, these changes are collected in an eligibility trace. Under consideration of one of the core problems in reinforcement learning, the credit assignment problem, this function can be understood as a method to capture the likelihood of a single firing synapse

of being responsible for a following reward. The more positively reinforcing events occur closely in time before a reward event, the higher the probability of these events to have something to do with the reward. The eligibility trace of a synapse can be defined as

$$\dot{c}(t) = -\frac{c}{\tau_c} + STDP(s_{post} - s_{pre})\delta(t - s_{pre/post})C_1 \tag{2.16}$$

where the contribution of all spike pairings with the second spike time at time $t - s_{pre/post}$ is modelled (Fig. 2.9A) with τ_c being the time constant of the synaptic eligibility trace, δ the Dirac delta function, s_{pre} and s_{post} the spike times of the contributing neurons and C_1 a constant coefficient.
In parallel to previously discussed findings regarding dopamine neurons in humans and monkeys, the reward signal can be modeled as function of the spike times of neuromodulatory neurons as well:

$$\dot{n}(t) = -\frac{n}{\tau_n} + \frac{\delta(t - s_n)}{\tau_n}C_2, \tag{2.17}$$

where τ_n denotes the time constant of the neuromodulatory signal, δ the Dirac delta function, s_n the spike times of the neuromodulatory neurons and C_2 a constant coefficient. Following the observations in neurons of a base firing rate, the global reward can be expressed as the neuromodulatory signal deviation from its mean value $\bar{b}$.

$$r(t) = n(t) - \bar{n} \tag{2.18}$$

By combining the eligibility trace with the global reward signal, a simple learning rule can be defined changing the weight of a synapse in proportion to the product of eligibility trace and reward signal.

$$\dot{w}(t) = c(t) \cdot r(t) \tag{2.19}$$

Since this model shown in Figure 2.9B can lead to unbounded growth, the weights are usually clipped at lower and upper boundaries. It was used in [14] to solve the XOR problem, a classical benchmark problem for artificial neural network training. With a fully connected feed-forward neural network with 60 input neurons, 60 hidden neurons and 1 output neuron, the modulated STDP learning rule was able to correctly solve this task in 99.1% of 1000 experiments. In another experiment, a feed-forward network with 100 input neurons directly connected to 100 output neurons could learn to approximate a target firing-rate pattern. This was done by rewarding or punishing the change in distance between actual and desired output pattern. The network was able to learn given output patterns consistently within a short amount of time.

As discussed earlier, one of the difficulties involving Temporal Difference learning in Spiking Neural Networks is that typical reinforcement learning formulations rely on discrete descriptions of states, actions and time. Spiking Neural Networks, on the other hand, communicate with spike events defined in continuous time. Potjans et al. [69] addressed this problem in an actor-critic reinforcement learning framework using spiking neurons. This kind of architecture consists of some critic neurons, predicting the value of an agent's state, and some actor neurons defining a policy. States are represented by neurons corresponding to single states. For learning and encoding value functions and policies, this architecture essentially uses single layer spiking neural networks to map the state input to a value or policy function. Therefore, the value function is encoded in the synaptic weights from state-value synapses and the policy is determined by state-action synapse weights. In an actor-critic architecture, both modules can in some sense learn from each other. While the critic solves the prediction problem, the actor uses this knowledge to solve the control problem. Potjans et al. [69] represented each state by a population of 40 neurons and each action is represented by a single neuron, which corresponds to discrete space representations. Learning the synaptic weights is performed with the TD(0) algorithm, but instead of using the discrete formulation (Sec. 2.1.1), the formula is reformulated to work in continuous time using so-called activity traces and thresholds. Using the TD(0) algorithm to learn single synaptic weights can be seen as supervised learning with a self-imposed correction signal. Although this approach gives some flexibility to apply TD learning in spiking neural networks, it can be shown that it is in fact mathematically equivalent to the discrete time formulation. Therefore, this approach does not really solve the problem of using TD learning in continuous spaces, but it was shown that it could successfully solve a simple 5x5 grid-world task where an agent had to navigate to a goal point.

A more advanced approach for using TD learning in Spiking Neural Networks was shown by Frémaux et al. [7]. States are represented by place cells arranged on a grid. A continuous representation is achieved by interpolating between these place cells using a so-called tuning curve modulating the firing rates of near-by cells. While this approach is capable of representing continuous spatial information, it is bound to fail for higher dimensional spaces since the number of place cells is growing exponentially with each dimension. The place cells are then connected to the critic neurons evaluating future rewards. Similarly, each place cell is connected to one of 180 actor neurons, each representing a different direction of motion. To obtain a final motion direction, these neurons are regarded as vectors which can be summed up to obtain a moving direction. It can be shown that such coding schemes as population vector coding allow a discrete number of neurons to code for a continuum of actions [70]. To ensure a clear choice of actions, a N-winner-takes-all

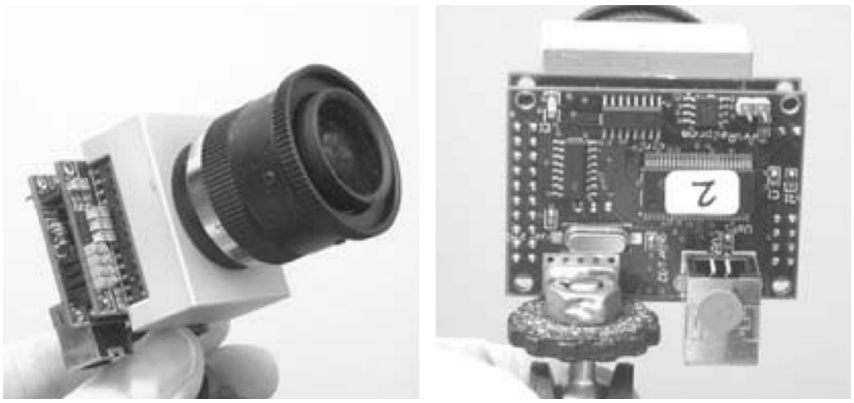
lateral connectivity scheme is used.
Using this similar basic architecture, a truly continuous-time value function is formulated which can be used to define a continuous temporal difference error as well. These continuous formulations allow for deriving a learning rule that is based on minimizing an error function by gradient descent. Interestingly, the synaptic learning rule that is derived from theoretical considerations on how to implement Temporal Difference learning in a continuous time space is very similar to previous STDP learning rules based on experimental evidence, where instead of a global reward signal the TD prediction error is used to modulate plasticity.
Using this architecure and learning rules, a water-maze navigation task is solved successfully, in a number of trials consistent with animal performance. Furthermore, it was applied to solve the acrobat and the cartpole problems, two complex motor control tasks.
Both, algorithms based on continuous time formulations of Temporal Difference learning as well as algorithms based on a modulated version of Spike-Timing-Dependent Plasticity have been shown to be capable of learning synaptic weights to solve simple tasks in low-dimensional spaces. Furthermore, results from theoretically derived TD learning rules and STDP learning rules based on experimental evidence seem to indicate a connection between these rules, which could lead to a biologically plausible unified learning rule.
While such a learning rule could explain how the brain uses prediction errors to adapt complex networks and learn new behaviours, many of the underlying mechanisms remain unclear. It is, for example, still not known how single neurons can compute prediction errors, even though the neuronal activity of dopamine neurons could be predicted by the theoretical TD model. Despite models presented earlier, the underlying plasticity rules for learning predictions remain unclear as well. Especially the multiple ways in which neuromodulators can interact with neural activity combined with complex network structures suggest many different mechanisms of interplay between them, few of them well understood. While most model networks consist of homogeneous neurons connected to each other in a feed-forward structure, the brain in contrast features highly recurrent networks of various neuron types. How learning of the recurrent connections can be achieved by STDP under neuromodulation, while maintaining balanced network activity, remains an open question as well.

2.3. Dynamic Vision Sensor

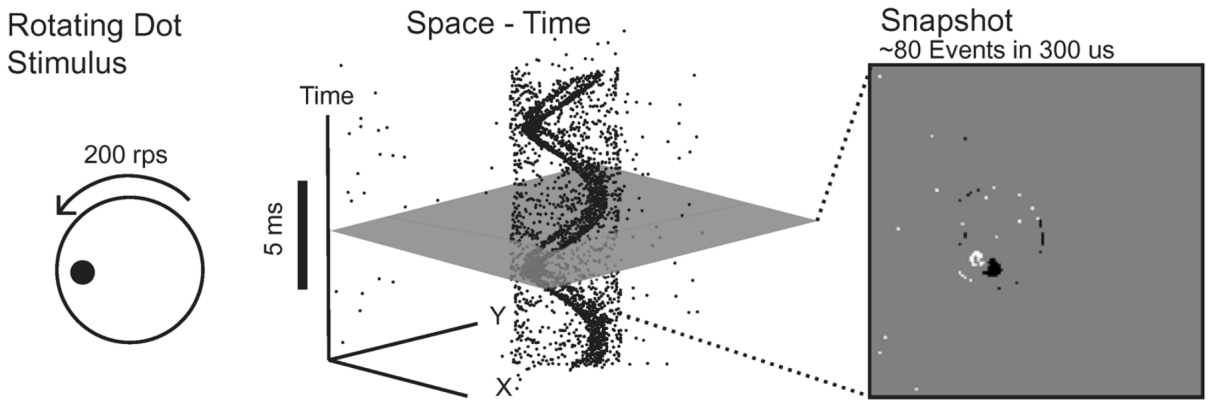
Since the emergence of machine vision as a field of research in the late 1960s, video data has usually been understood as a series of constant rate image frames. While many years of research have produced low cost, high resolution recording devices and efficient algorithms solving difficult computer vision problems, there are still some drawbacks regarding this frame-based architecture. Due to the redundancy of information - pixels produce data even if intensity values don't change - output data for high frame-rate applications can quickly grow to massive sizes. Furthermore, the technical design of frame-based image recording devices often comes with limits in dynamic range.
Therefore in recent years a new concept of image sensor called Dynamic Vision Sensor (DVS) has been developed giving up the frame-based paradigm. Inspired by biological vision systems, these devices generate sparse, event-based output that represents the positive and negative relative luminance change of a scene. Hereby, the event threshold can be understood as slow light adaption parameter in biology [71]. By letting the pixels operate asynchronously sending out events nearly instantaneously on an address-event bus, no complete image frames exist in the system. The output consists of a stream of precisely-timed, individual events representing local changes in temporal contrast. Due to the technical design of these devices, image data can be produced with low latency and good temporal resolution as well as high contrast sensitivity and dynamic range. As a consequence of the sparse image data representation, Dynamic Vision Sensors typically show very low power consumption while producing less output-data.
In the literature, DVS devices have been used for pose estimation [72], pencil balancing [73] or digit recognition [10]. In Table 2.1 and Figure 2.10, some specifications and image examples of a Dynamic Vision Sensor are shown published by Lichtsteiner et al. [6].

Table 2.1.: Specifications of a Dynamic Vision Sensor by Lichtsteiner et al. [6]

Dynamic range:	$120dB$
Contrast threshold mismatch:	2.1%
Pixel array size:	128x128 pixels of $40\mu m \times 40\mu m$
Photoreceptor bandwidth:	$\geq$ 3 kHz
Event saturation rate:	1 million events per second
Power consumption:	23mW
Minimum latency:	$15\mu s$



(a) Vision sensor system.



(b) Events produced over time by a rotating dot stimulus.

(c) Eye: $\sim$ 7500 events in 300 ms

Figure 2.10.: Dynamic Vision Sensor by Lichtsteiner et al. [6]. Images taken under natural lighting conditions with either object or camera motion.

3. A Taxonomy of Robot Control using Spiking Neural Networks

For a couple of years now, solving control tasks using Spiking Neural Networks has been an interest in various fields of research. In the medical community for example, SNNs could be used for building brain-controlled prosthetic devices in the future. In neuroscience, building and simulating simplified brain models on computers has become a powerful tool for testing hypotheses on information processing, synaptic plasticity and learning. Similarly, getting insights into the underlying neural mechanisms that control the behavior of biological organisms, e.g. insects, has been an ongoing research topic in biology as well. Last but not least, by using biologically inspired sensors, computational devices and algorithms roboticists hope to gain advantages e.g. in power efficiency as well.
In the control tasks presented in this chapter, the network is usually supposed to learn a function mapping some state input to a control or action output. When successfully learned, the network should be able to perform simple tasks such as wall following, obstacle avoidance, target reaching, lane following, taxis behavior or food foraging. In most cases, the network input directly comes from the robot's sensors that range from binary sensors, e.g. olfactory, to multi-dimensional continuous sensors, such as cameras. In other cases, the input can be pre-processed data, e.g. coming from EEG data. Similarly, the output can range from one-dimensional, binary behavior control to multi-dimensional continuous ouput values, e.g. for motor control, as well.
Initially, solving simulated control tasks was done by manually setting network weights, e.g. in [74], [75] and [76]. While this approach is able to solve simple behavioral tasks such as wall following [77] or lane following [15], it is usually only feasible for very small network architectures with few weights.
Therefore, a variety of training methods for SNNs in control tasks has been proposed in the past. Instead of focusing on criteria such as field of research, biological plausibility or specific task, this chapter is meant as a classification of published algorithms into the basic underlying training mechanisms from a robotics and machine learning perspective. In the first part of this chapter, some implementations of SNN control are introduced that use

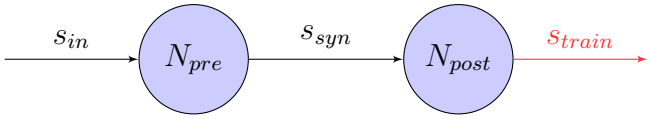


Figure 3.1.: Supervised Hebbian training of a synapse: Weight of the synapse between pre- and post-synaptic neurons, N_{pre} and N_{post}, is adjusted by the timing of pre-synaptic spike-train s_{syn} and external post-synaptic training signal s_{train}.

some form of Hebbian-based learning. In the second part, publications are shown that try to bridge the gap between classical reinforcement learning and spiking neural networks. Finally, in the last section some alternative methods on how to train and implement spiking neural networks are discussed.

3.1. Hebbian-Based Learning

As discussed in Section 2.2.3, learning rules first described be Hebb that rely on the precise timing of pre- and post-synaptic spikes seem to play a crucial part in the emergence of highly non-linear functions in spike-based neural networks. Therefore, different biologically plausible learning rules such as STDP have been proposed in many publications. While all Hebbian-based algorithms depend on the spike timing, the exact learning rules can differ, e.g. using different STDP learning curves. Moreover, training these networks can be achieved in different ways as well, as shown in the following sections.

3.1.1. Supervised Training

A simple way of training SNNs for control tasks is by providing an external training signal, that adjusts the synapses in a supervised learning setting. As shown in Figure 3.1, when an external signal is induced into the network as post-synaptic spike-train, the synapses can adjust their weights, for example using learning rules such as STDP. After an initial training phase, this will cause the network to mimic the training signal with satisfactory precision. While this approach provides a simple, straight-forward way for training networks, it is dependent on an external controller. Especially for control tasks involving high-dimensional network inputs, this may not be feasible.
Carrillo et al. [78] used this basic approach to train a spiking model of the cerebellum to control a robotic arm with 2 degrees of freedom in a target reaching task taking joint angles and speeds, as well as target position as inputs. In contrast to other STDP learning rules, only long term depression was externally induced by a training signal,

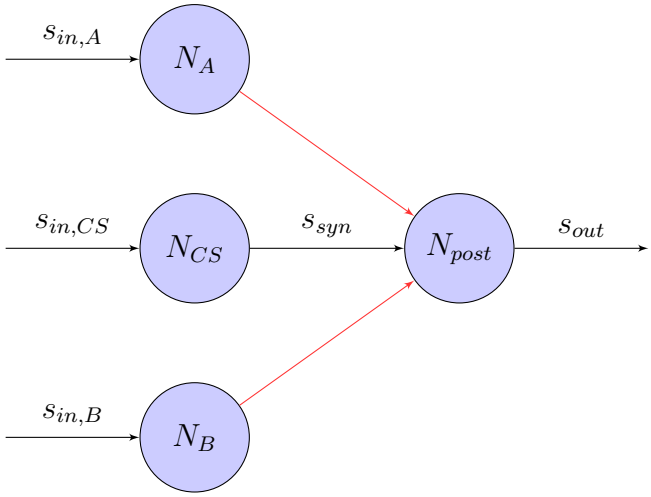


Figure 3.2.: Classical Conditioning with STDP synapse between N_{pre} and N_{post}: An unconditioned stimulus (US) A or B causes the post-synaptic neuron N_{post} to fire. The conditioned stimulus (CS) firing shortly before its associated US will adjust its weights such that N_{post} will fire even in the absence of US. Due to the Hebbian learning rule, the synaptic weight is unchanged when the other, unrelated stimulus causes N_{post} to fire.

which relied on the motor error, namely the difference between desired and actual state. In a similar experiment, Bouganis and Shanahan [79] trained a single-layer network to control a robotic arm with 4 degrees of freedom in 3D space. As inputs, joint angles and the spatial direction of the end-effector were used, while outputs consisted of four motor command neurons. The training signal was computed using an inverse kinematics model of the arm, adjusting the synaptic weights with a symmetric STDP learning rule.

3.1.2. Classical Conditioning

In his famous experiment on classical conditioning discussed in Section 2.2.3.3, Pavlov's dog learns to associate an unconditioned stimulus (US), in this case food, and a conditioned stimulus (CS), a bell, with each other. While it is not clear how the high-level stimuli given in his experiment are processed within the brain, the same learning principle can be used for training SNNs on a neural level as well. Figure 3.2 shows how a STDP synapse can learn to associate an US and a CS provoking a response even in the absence of the US.

Following this principle, bio-inspired robots can learn to associate a CS, e.g. sensory information, with an US that functions as an external reinforcer. That way, robots can learn to follow the desired behavior based on sensory inputs. Arena et al. [80][81][82] showed how classical conditioning can be used in an obstacle avoidance and target reaching

task. In a SNN with two output motor neurons, distance and vision sensors function as CS, while contact and target sensors work as US causing an unconditioned response. In a similar experiment, Cyr and Boukadoum [83] carried out different classical conditioning tasks in a controlled virtual environment using infrared, ultrasound and visual neurons as CS and vibration neurons as US. Wang et al. [84][85] constructed a controller that stimulates two motor neurons as US. A single-layer SNN using proximity sensor data as CS input is then trained in tasks such as obstacle avoidance and target reaching. Iwadate et al. [86] used light sensors in a target reaching task to punish wrongful behavior. Jimenez-Romero et al. [87][88][89] implemented a virtual ant that learns to associate olfactory sensor input with different behaviors trough a single-layer SNN. The robot is able to learn to recognize rewarding and harmful stimuli as well as simple navigation in a simulated environment.
In order to successfully learn such behavioral tasks, some unconditioned stimulus has to be given for every relevant conditioned stimulus the robot should learn. This also means that the robot will not learn to associate stimuli that are delayed in time. Taken together, using classical conditioning for robot control basically means constructing an external controller that provides unconditioned stimuli for every relevant state input, which may not be feasible in many tasks.

3.1.3. Operant Conditioning

While Classical Conditioning is concerned with passively associating conditioned and unconditioned stimuli with each other, Operant Conditioning (OC) consists of associating stimuli with responses and actively changing behaviors thereafter. Conceptually, operant conditioning is closely related to reinforcement learning and its agent-environment interaction cycle (Fig. 2.1). Instead of developing a formal mathematical model, operant conditioning has been mainly researched in biological and psychological domains. Despite advances in the understanding of operant conditioning, it is still not clear how this type of learning is implemented on a neural level.
In this context, Cyr et al. [90][91] developed a spiking OC model that consists of an input feeding cue neuron, an action neuron and a predictor neuron receiving rewards or punishments. With this simple basic architecture and learning rules such as habituation and STDP, they were able solve simple OC related tasks in a simulated environment, such as pushing blocks.

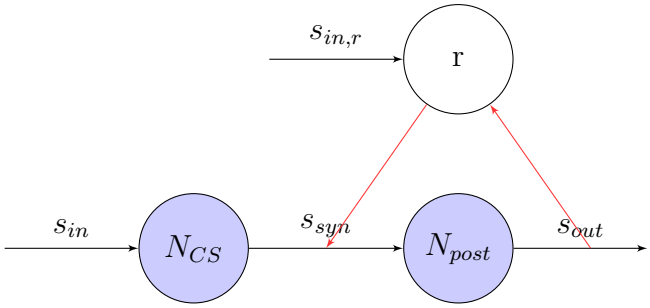


Figure 3.3.: Reward-modulated STDP synapse between N_{pre} and N_{post}: Depending on the post-synaptic output spike-train, a reward r is defined that modulates the weight change of the synapse.

In another publication by Dumesnil et al. [92][93] a reward-dependent STDP learning rule was implemented on a robot to allow for OC learning and demonstrated in a maze task. Interestingly, the developed learning rule closely resembles previously discussed modulated STDP learning rules, which will be discussed in the next section.

3.1.4. Reward-Modulated Training

In Figure 3.3 the previously discussed learning rule for reward-based training is shown. Inspired by dopaminergic neurons in the brain, effects of STDP events are collected in an eligibility trace and a global reward signal induces synaptic weight changes. In contrast to supervised training as discussed in the previous sections, rewards can be attributed to stimuli, even if they are delayed in time. This can be a very useful property for robot control, because it might simplify the requirements of an external training signal leading to more complex tasks.
In the literature, a variety of algorithms has been published using this basic learning architecture for training. While they are all based on the same mechanism, the reward signal can be constructed in different ways.

3.1.4.1. Rewarding Specific Events

Probably the most straight-forward implementation of reward-based learning resembling classical reinforcement learning tasks has been shown in several publications that use rewards associated to specific events. Evans [94] trained a simple, single-layer SNN in several food foraging tasks consisting of 4 input sensor neurons and 4 output motor neurons. In a separate network, other reward-related sensor neurons stimulated a dopaminergic neuron that in turn modulated the synaptic weight change. With this

simulation setup, the robot was able to learn food-attraction behavior and subsequently unlearn this behavior when the environment changed. By shifting the dopamine response from the primary to a secondary stimulus, the robot was able to learn even with large temporal distance between correct behavior and reward. Helgadottir et al. [95] described a robotic platform for learning target approach behavior. They used a biologically inspired sensory-motor network architecture with populations of color receptor neurons connected to a single extrinsic neuron. An external flash signal reinforced the robot to move forward in the target direction. Skorheim et al. [96] simulated a grid-based foraging task using the surrounding cells as visual input layer and each possible movement direction as output layer. Successful food acquisition was rewarded and the recently active synapses associated with this response were strengthened. Faghihi et al. [97] showed a SNN model of a fruit fly that is able to execute both first and second order conditioning. In a simple task the simulated fly learns to avoid getting close to an olfactory target emitting electric shocks. Furthermore, the same behavior can be transferred to a secondary stimulus that gets associated to the primary stimulus without emitting electric shocks itself.

3.1.4.2. Control Error Minimization

As opposed to rewarding specific events, dopamine-modulated learning can also be used in an optimization task to minimize an objective function. This is usually achieved by strengthening or weakening the connections that led to changes in the objective function based on their eligibility traces.
Clawson et al. [98] used this basic architecture to train a SNN to follow a trajectory. The network consisted of lateral state variables as inputs, a hidden layer and an output layer population decoding the lateral control output. Learning is achieved by minimizing the error between actual and desired output, which is provided by an external linear controller.

3.1.4.3. Indirect Control Error Minimization

For some potential applications of SNNs, e.g. neuroprosthetic devices implanted in the brain, direct manipulation of synaptic weights might not be possible. Therefore, an indirect approach of training SNNs was shown by Foderaro et al. [99] that induces changes in the synaptic efficacy through input spikes generated by a separate critic SNN. This external network was provided with control input as well as feedback signals and trained using a reward-based STDP learnung rule. By minimizing the error between control output and optimal control law it was able to learn adaptive control of an aircraft. Similar ideas

were presented by Zhang et al. [100][101], Hu et al. [102] and Mazumder et al. [103] who trained a simple, virtual insect in a target reaching and obstacle avoidance task.

3.1.4.4. Metric Minimization

The same principle can also be applied to minimize a global metric, which might be easier to construct and calculate than an external controller.
Chadderdon et al. [104] proposed a spiking-neuron model of the motor cortex which controlled a single-joint arm in a target reaching task. The model consisted of 144 excitatory and 64 inhibitory neurons with proprioceptive inputs cells and output cells controlling the flexor and extensor muscles. A global reward or punishment signal was given depending on the change of hand-target distance. Neymotin et al. [105] and Spüler et al. [106] extended this architecture later. Similarly, Dura-Bernal et al. [107] used a biomimetic cortical spiking model composed of several hundred spiking model-neurons to control a two-joint arm. With proprioceptive sensory input (muscle lengths) and muscle excitation output, the network was trained by minimizing the hand-target distance. Kocaturk et al. [108] extended the same basic architecture in order to develop a brain-machine interface. Extracellularly recorded motor cortical neurons provide the network inputs used for prosthetic control. By pressing a button, the user can reward desired movements and guide the prosthetic arm towards a target. Using a miniaturized microprocessor with resistive crossbar memories implemented on a two-wheeled differential drive robot, Sarim et al. [109][110] showed how an STDP-based learning rule could lead to target approaching and obstacle avoidance behavior. Although in this case learning was implemented using if-then rules that relied on distance changes from target and obstacles, it is conceptually identical to reward-modulated learning. This can easily be seen by exchanging the if-rules with a reward of $+1$ or -1.

3.1.4.5. Reinforcing Associations

Chou et al. [111] introduced a tactile robot that uses a network architecture inspired by the insular cortex. As in classical conditioning, a dopamine-modulated synaptic plasticity rule was used to reinforce associations between conditioned and unconditioned stimuli.

3.2. Reinforcement Learning

In the previous section a variety of approaches was presented for training SNNs based on Hebbian learning rules. This was done either by providing a supervised training signal through an external controller or by using a reward-based learning rule with different ways of constructing the reward. In general, all of these approaches have been trained in tasks that don't require looking very far ahead.
In classical reinforcement learning theory on the other hand, learning to look multiple steps ahead in a MDP is one of the main concerns. Therefore, several algorithms have been published combining SNNs with classical reinforcement learning algorithms.

3.2.1. Temporal Difference

In Section 2.1 a learning rule looking one or more steps forward in time was introduced called temporal difference learning. Likewise, two algorithms were discussed in Section 2.2.3.3 that used a spiking network architecture to extend TD learning to continuous time space. Hereby, Potjans et al. [69] and Frémaux et al. [7] used place cells to represent the state space in a MDP and single-layer SNNs for state evaluations and policies. Both algorithms were able to learn to navigate in a simple grid-world after some training.
With a similar approach, Nichols et al. [112] presented a robot controller inspired by the control structures of biological systems. In a self-organizing, multi-layered network structure, sensory data coming from distance and orientation sensors was gradually fused into state neurons representing distinct combinations of sensory inputs. On top, each individual state neuron was connected to 3 output motor neurons. By fusing the sensory input into distinct state neurons and connecting them to action neurons, a simplified TD learning rule could be used to set each synaptic weight in the last layer individually. Performance of this controller was demonstrated in a wall-following task.
While these state representations work very well for relatively small state spaces, they are usually bound to fail for larger, high-dimensional state spaces. In fact, these approaches can conceptually be seen as a SNN implementation of table-based Q-learning where each synapse represents a single action-value.

3.2.2. Model-based

Although for robot control tasks as shown in this thesis, model-free reinforcement learning methods seem favorable, two recent publications are at least worth mentioning that

presented SNN implementations of model-based reinforcement learning algorithms.
Rueckert et al. [113] presented a recurrent spiking neural network for planning tasks that was demonstrated on a real robot in an obstacle avoidance task. Friedrich and Lengyel [114] implemented a biologically realistic network of spiking neurons for decision making. The network uses local plasticity rules to solve one-step as well as sequential decision making tasks.

3.3. Other

3.3.1. Evolutionary Algorithms

Natural evolution has produced a multitude of organisms in all kinds of shapes with survival strategies optimally aligned to environmental conditions. Based on these ideas a class of algorithms has been developed finding problem solutions by mimicking elementary natural processes called evolutionary algorithms [115]. Initially, a random population of individuals (encoded problem solutions) is created and evaluated using a predefined fitness function. Then, following the "survival of the fittest" paradigm, a new generation of individuals is created based on the most promising solutions of the previous generation. Usually, this is either done by mutation where small random changes are applied to individuals, or by crossover where "genes" are exchanged between individuals. Generally, evolutionary processes can be understood as some form of gradient descent optimization. Therefore, a typical problem using these algorithms is getting stuck in local minima. In applications in robot control, evolving SNNs have been shown to work well in mostly static environments. Due to the training principle of trial and error, there are usually difficulties in dynamically changing environments.
Floreano and Mattiussi [116][117] showed a vision-based controller in an irregularly textured environment that navigated without hitting obstacles. The predefined SNN consisted of 18 sensory input receptors connected to 10 fully-connected hidden neurons and 2 motor output neurons. Using static synaptic weight values, the algorithm was used to search the space of connectivity by genetically evolving only signs of weights. With a population of 60 individuals, fitness was evaluated by summing up over motor speeds at every time step, and new generations were created using one-point crossover, bit mutation and elitism. Hagras et al. [118] later extended this approach to evolve SNN weights as well using adaptive crossover and mutation probabilities. They were able to evolve good SNN controllers in a small number of generations in a wall following scenario. Howard and Elfes [119] presented a quadrotor neurocontroller that performed a hovering tasks in

challenging wind conditions. With a feed-forward network taking the differences between current position and target position as input and pitch, roll and thrust as output, weights and topology were evolved minimizing the spatial error. In a target reaching and obstacle avoidance tasks using binocular light sensors and proximity sensors, Batllori et al. [120] evolved a SNN by minimizing the control error in order to mimic an external controller signal. Markowska-Kaczmar and Koldowski [121] used a feed-forward network architecture of predefined size to control a toy car. Based on speed, localization and road boarder input signals, the network controlled speed regulation and turn direction and evolved its weights using a genetic algorithm.

3.3.2. Self-Organizing Algorithms

Alnajjar and Murase [122] formulated a synaptic learning rule that enforced connections between neurons depending on their activity. With this self-organization algorithm that resembles other Hebbian-based learning methods, they were able to learn obstacle avoidance and simple navigation behavior.

3.3.3. Fuzzy Logic

Kubota [123] implemented a hybrid SNN and Fuzzy Logic controller and demonstrated it learning obstacle avoidance and target-following behavior.

3.3.4. Liquid State Machine

Burgsteiner [124], Probst et al. [125] and Arena et al. [126] showed how liquid state machines can be trained for robot control tasks.

4. Methodology

In this chapter different controllers are presented in a lane following task using a Dynamic Vision Sensor. The basic simulation environment that all controllers share is shown in the first section. Afterwards, implementation details of different controllers are discussed.

4.1. Simulation Environment

In order to provide a simple and flexible environment to test and validate different algorithms, the cross-platform robotics simulator V-REP [20] was used in this thesis. V-REP is a simulator with an integrated development environment that is free for educational entities, such as students at schools and universities. With its distributed control architecture it can control each object individually via embedded scripts.
Developing capable real-world robots usually means integrating many different software and hardware parts into a functional system. Often one has to rely on parts from various manufacturers and code written by other developers. Therefore, defining standardized interfaces and functionalities has become an important part of robotics. The most widely used software framework for robot software development is called Robot Operating System (ROS) [18] and provides an operating system-like functionality on a heterogeneous computer cluster, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

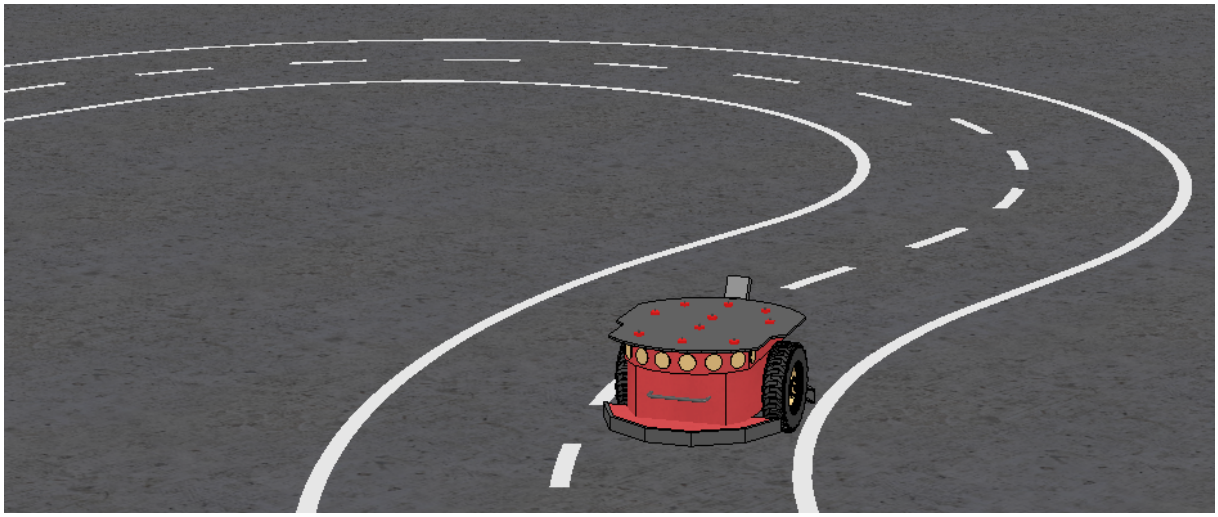


Figure 4.1.: Robot task: Right lane following.

For communicating with computer programs outside of the simulator, V-REP provides a ROS interface plugin that can be used to create ROS nodes for publishing and subscribing to topics. In this work, communication between the V-REP environment and the controller was done using ROS messages that were defined in a non-threaded child script attached to the robot model and written in Lua [127]. Child scripts in V-REP are simulation scripts attached to scene objects and called at every simulation step. Some implementation details will be discussed in the next sections. In Table 4.1 the simulation settings in V-REP used in this thesis are shown.

Table 4.1.: V-REP simulation settings

Dynamics engine:	Bullet 2.78
Dynamics settings:	Accurate
Simulation time step:	50 ms

4.1.1. Right Lane Following Task

In Chapter 3 some implementations of robots using SNNs were discussed. These models were shown to learn tasks such as wall following, lane following, target reaching, obstacle avoidance, food foraging, adaptive aircraft control or grid-world navigation. As inputs, most of these robots were using simple, low-dimensional sensory receptors, e.g. boolean olfactory- or photo-receptors, proximity sensors or ultrasound sensors. Other networks used proprioceptive information to control prosthetic arms in closed loop systems. In most robotic implementations, output information was decoded and used for motor or

muscular control, e.g. left and right motor of a 2-wheeled robot or muscular systems of flexion and extension for steering wheels or arms. Other implementations let the network control high-level behavioral decisions or directional decisions in grid-based navigation tasks.
Recently, Kaiser et al. [15] proposed a different task of using SNNs for robot control. In their paper, a general framework for a simulated right lane following application was given. In contrast to previous algorithms, high-dimensional data coming from a DVS was used as input for the network to control a muscular steering wheel model. For roboticists this is an interesting application due to several reasons. First, in order interact within real-world scenarios robots need to learn how to make sense of complex information. Therefore, algorithms are needed that have the ability to handle high-dimensional data from vision sensors. It has been shown that SNNs can solve classical machine learning tasks such as image classification by converting gray-scale pixel values into Poisson spike rates [9]. Second, due to their event-based nature, combining SNNs with DVS data seems like a more natural fit. Recently, Lee et al. [10] showed that SNNs can handle event-based image data as well and, in fact, produce higher classification accuracies than with converted gray-scale images, even though robotic applications seem more difficult to handle. Third, the advantages of DVSs such as their low latency and high dynamic range make them interesting for a variety of real-world applications including autonomous driving in challenging lighting conditions.
Based on the proposed framework, a similar task was implemented in this thesis in order to test and validate different learning algorithms in different lane following scenarios with varying complexity. Further details regarding the specific scenarios implemented are discussed in Section 5.1.

4.1.2. Pioneer P3-DX Robot

Figure 4.2 shows the V-REP model of a Pioneer P3-DX [128] robot that was used for the lane following task. The 45 cm long Pioneer robot is a versatile, 2-wheeled mobile robot mostly found in research and it comes with 8 front and 8 rear ultrasonic sensors. Instead of using the on-board ultrasonic sensors, a DVS camera was attached to the front of the robot. More details on the DVS device are discussed in the next section.
Attached to the robot model in V-REP is a child script that handles the communication between simulator and controller. In order to control the robot, the following subscribers are defined in the script executing callback functions. For motor control "/leftMotorSpeed" and "/rightMotorSpeed" messages are directly set as target velocities for both motors. Furthermore, boolean messages published under the topic "/resetRobot"

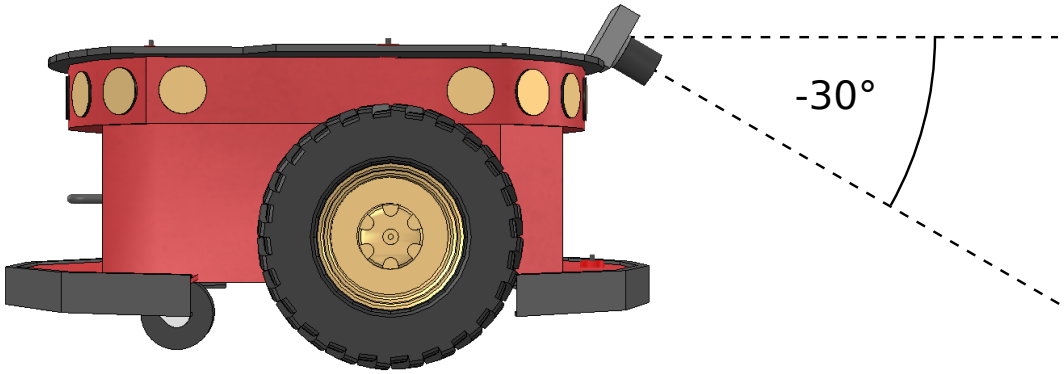
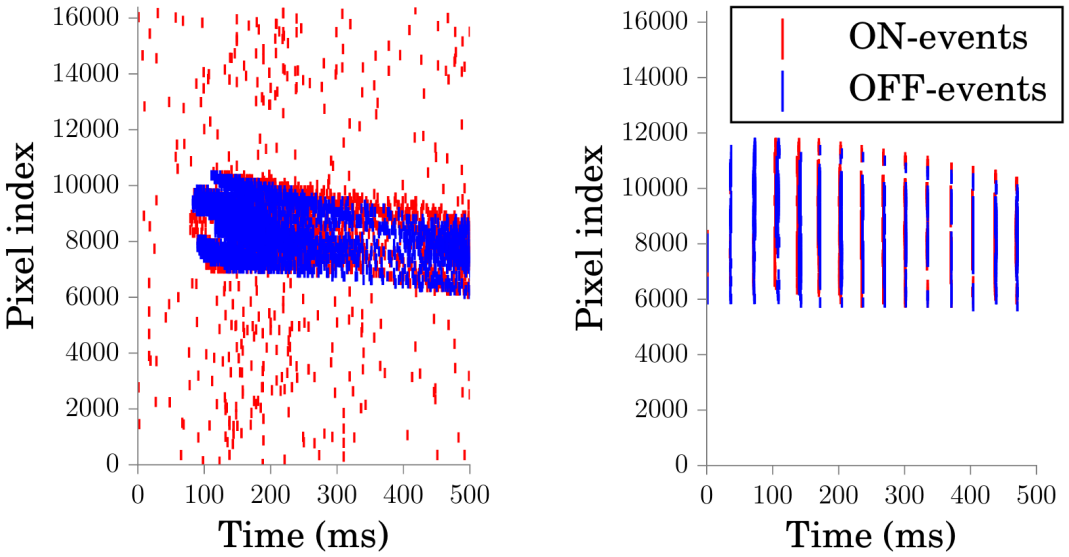


Figure 4.2.: Pioneer P3-DX robot with attached Dynamic Vision Sensor (DVS).

will place the robot to its starting position on the inner or outer lane depending on the value. This is useful for resetting the robot during learning and letting it learn to follow the right lane in both directions. For evaluation purposes as well as training, it comes in handy to have some kind of metric determining the performance of an algorithm. For this reason, the child script also publishes the exact position and orientation of the robot in a topic called "/transformData" at every simulation time-step. With a mathematical track model this data was used to calculate the distance of the robot to the lane center as well as its projected lane position.

4.1.3. DVS Simulation

As discussed in Section 2.3, Dynamic Vision Sensors are biologically inspired vision sensors with a continuous output of independent pixel events and a temporal resolution in the order of microseconds. Unfortunately, the same properties make it somewhat difficult to simulate as well. Therefore, Mueggler et al. [129], Orchard et al. [130] and Serrano-Gotarredona and Linares-Barranco [131] have previously tried to let a real DVS device observe a screen in order to convert images into an event-based data streams. In this thesis, however, the DVS camera is simulated within V-REP using the DVS128 model by iniLabs [132] that comes with the simulator. This model produces events by saving camera images from the previous time-step in a buffer and computing the pixel intensity change between buffer image and current image. An event is created if the pixel change is larger than a certain threshold, which was set to 10% in this work. The threshold is equivalent to the dvs bias or to the slow light adaption parameter in biology [15].



(a) Real DVS events (b) Simulated DVS events

Figure 4.3.: Kaiser et al. [15] compared the generated address events of a real DVS to simulated one. Both cameras are observing the same stimuli of a ball entering the field of view. The 2 dimensional image structure is flattened to a 1 dimensional pixel index. ON-events are drawn in red, OFF-events are drawn in blue. (a) Address events generated by a real DVS. The events are all independent and have individual time-stamps. (b) Address events generated by a DVS simulation from a webcam video stream. The events are only at given time-steps, when a new frame is received.

Simulating the DVS camera by comparing consecutive image frames unfortunately means that pixel events are not independent anymore and will be emitted in batches at every time-step with the same time stamp. Therefore, the simulation time-step length (Tab. 4.1) determines the dvs time resolution as well. Usually, this will set the DVS time resolution to a much lower frequency than a real device, which is a drawback of this kind of DVS simulation. In Figure 4.3, Kaiser et al. [15] have shown how real DVS events compare to simulated DVS events, when observing the same stimuli. Despite the obvious differences they argue that a robust SNN should be able to cope with such inputs nonetheless. By all means, perceiving discrete frames as fluid motion seems to be no problem for the human brain at all.

In accordance to the real DVS128 sensor by iniLabs, data from the DVS camera is transmitted based on the Address Event Representation (AER) protocol [133] with a bus width of 15. Hereby, 7 bits comprise the X address and 7 bits the Y address of an event. 1 bit is used for the polarity (ON- or OFF-event). Furthermore, a time-stamp of every event can be added to the protocol.

For the purpose of this work, the transmitted data was simplified to event address only

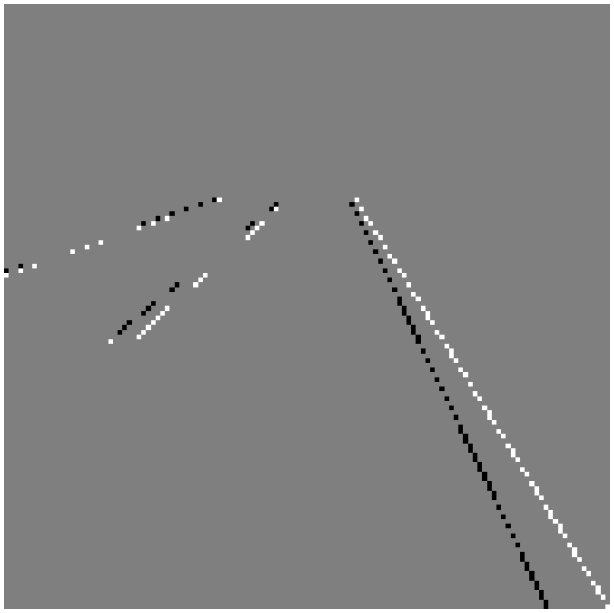


Figure 4.4.: DVS frame from a single time-step of the V-REP simulation. Black and white pixels show ON- and OFF-events, respectively.

and published in topic called "/dvsData". Furthermore, in order to reduce noise in the images, the DVS camera can only detect the road markings and other deliberately placed objects in the simulation. Intensity changes on the ground for example are ignored. A single DVS image frame from the simulation is shown in Figure 4.4.

4.2. Controllers

In the previous section, the lane following simulation was presented, including a model of the Pioneer robot and a DVS device. Moreover, it was shown how the simulation can be controlled using ROS topics for incoming motor speed and reset commands, as well as outgoing DVS and position information. In this section, several algorithms are discussed learning to control the robot using ANNs as well as SNNs.

4.2.1. DQN

Based on the Deep Q-Learning (DQN) algorithm discussed in Section 2.1.3, a controller was implemented using conventional rate-based neural networks that learned how to follow the right lane in a classical reinforcement learning setting. Therefore, at first it is explained

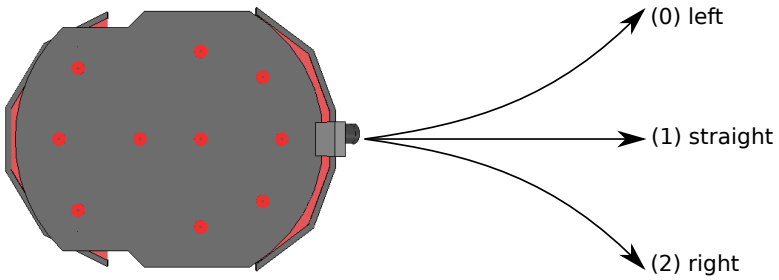


Figure 4.5.: Action space in lane following task with three discrete actions: (0) Turn left: Set left motor speed to $v_s - v_t$ and right motor speed to $v_s + v_t$. (1) Go straight: Set left and right motor speed to v_s. (2) Turn right: Set left motor speed to $v_s + v_t$ and right motor speed to $v_s - v_t$.

how the task of following the right lane can be translated into a Markov Decision Process (MDP). Afterwards, some details about the implementation of the algorithm are given.

4.2.1.1. Lane Following as MDP

Markov Decision Processes are usually defined as a 5-tuple of actions, states, transition probabilities, rewards and discount factor (Sec. 2.1). While the transition probabilities can be ignored when using model-free reinforcement learning algorithms such as Q-learning, other components of the MDP have to be carefully chosen to ensure fast and stable learning.
Figure 4.5 shows three discrete actions that the robot can choose from in this task. It can go straight, letting left and right motors run at the same speed, or it can take a turn by adding and subtracting speed to both motors depending on the desired direction.
As discussed before, goal of the robot is to follow the right lane using data from the DVS only. While in similar reinforcement learning tasks using conventional cameras (e.g. Mnih et al. [2] or Lillicrap et al. [3]), scaled images could be directly used as state input for the MDP, this is more difficult using a DVS device. First, in order to reduce the computational complexity of the task, images were reduced to a lower resolution in this work as well. Second, due to the event-based nature of the DVS data, image frames coming from the simulation do not always contain sufficient information for the network to make meaningful decisions. Therefore, the state input was computed by condensing information of several consecutive DVS frames into a single image. As shown in Figure 4.6, this is done by dividing the original 128×128 DVS frames into small 4×4 regions and counting every event over consecutive frames regardless of the polarity. Furthermore, the image is cropped at the top and bottom resulting in a 32×16 image. While this

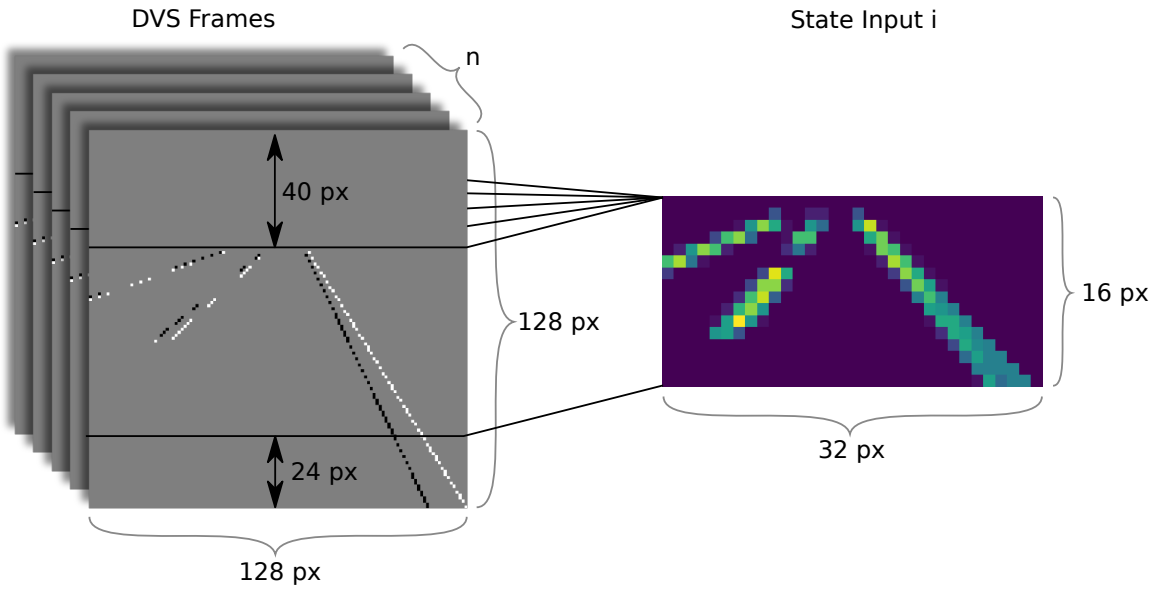


Figure 4.6.: Conversion of consecutive DVS frames into state input for reinforcement learning. This is done by dividing the original 128×128 DVS frames into small 4×4 regions and counting every event over consecutive frames regardless of the polarity. Furthermore, the image is cropped at the top and bottom resulting in a 32×16 image. In the DQN algorithm, a binary version of image i was used as state input.

is a substantial reduction in dimensionality, it is still a problem challenging enough. To further increase the performance of the algorithm, the final DQN state input $s_{M\times N}$ used in this thesis is a binary version of the state input $i_{M\times N}$ only containing ones and zeros:

$$s_{m,n} = \begin{cases} 0, & \text{if } i_{m,n} = 0 \\ 1, & \text{if } i_{m,n} > 0 \end{cases} \tag{4.1}$$

In the simulation, DVS frames are calculated and published every $50\,ms$ (with every simulation time-step). Actions in the MDP, on the other hand, are executed every $500\,ms$. Therefore, during one step in the MDP, DVS frames are stored in a first-in-first-out (FIFO) queue of length 10. At every time-step of the MDP, the last 10 DVS frames are then converted into the final state input.

Rewards play a crucial role in reinforcement learning and define the goal of an agent. In this work, the robot is supposed to learn to follow a lane staying as close to the center as possible. Figure 4.7 shows the reward that is given at every time-step of the MDP. It is defined as a Gaussian distributed function of the lane-center distance. As the model-free DQN algorithm learns from experience samples with a one-step lookahead, it is beneficial for learning to use a reward that is well distributed over the state space and monotonically

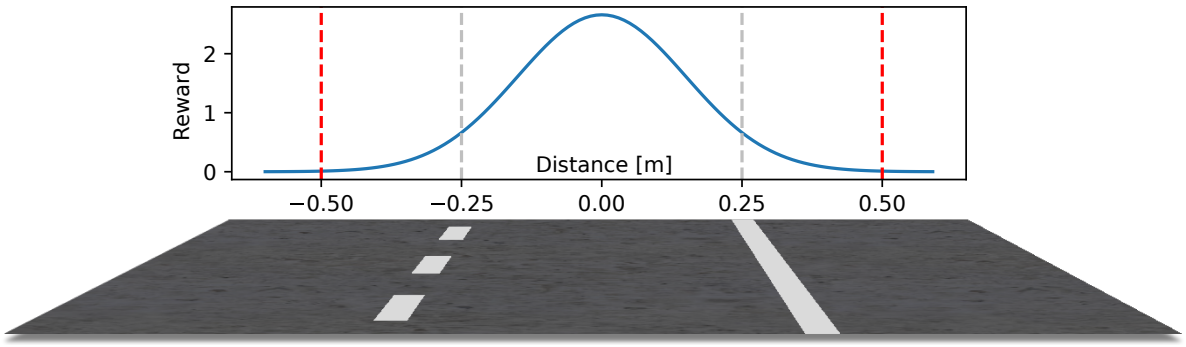


Figure 4.7.: Reward given in the right lane following task. It is defined as a Gaussian distribution over the distance to the center of the lane with a standard deviation of $\sigma = 0.15$ and mean at 0. The lane markings are $0.25\,m$ away from the lane-center. If the robot will go further than $0.5\,m$ from the lane-center, episodes are terminated and the robot will be positioned at its starting position.

increasing towards the goal. This ensures that the robot will learn to navigate in the direction of the goal, even if it has not been there yet. Besides DVS data, the simulator publishes position data of the robot every $50\,ms$ as well. With a mathematical model of the lane center in both directions, this data is used to calculate the exact distance of the robot to the lane-center and the resulting reward. Again, this is done every $500\,ms$ using the latest published position data.
If the robot reaches a position where its distance to the lane-center is greater than $0.5\,m$, training episodes are terminated and a reset message is raised that causes the simulator to place the robot to its starting position on the opposite lane. Letting the robot alternate between both lane directions increases its experienced states and results in a more generalized policy after learning. In reinforcement learning, the extent to which an agent takes expected future rewards into account is usually controlled by a discount factor. Although the lane-following task does not necessarily need looking ahead many steps, the discount factor was set to 0.99, therefore potentially being able to solve tasks that involve a little bit more foresight as well.

4.2.1.2. Controller 1: DQN

In Figure 4.8 the communication of the DQN controller components is shown. All components were run on Ubuntu 14.04 LTS, the controller, environment handler and the experience buffer were written in Python 2.7.
The environment handler manages all communication between the DQN controller and the V-REP simulation discussed in Section 4.1. While the controller is implemented on the

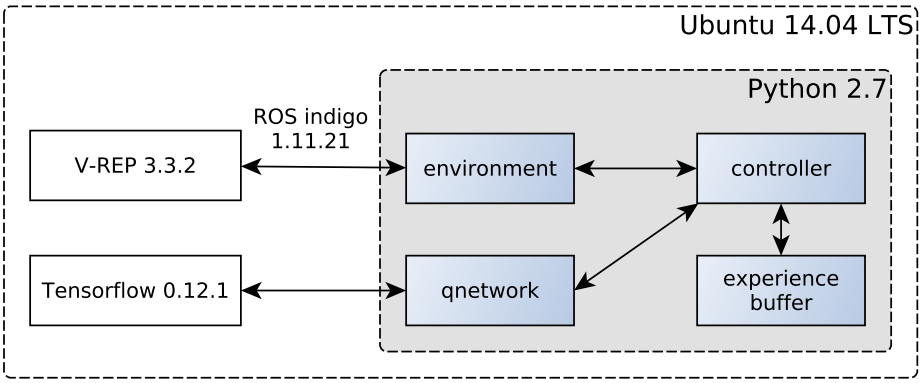


Figure 4.8.: Communication between components of the DQN controller.

MDP level, the environment handler translates between ROS messages and states, actions and rewards. Specifically, it handles topic publishers and subscribers. Discrete MDP actions are translated into left and right motor speed and published on their respective topics (Fig. 4.5). In order to reset the robot, the environment handler publishes a boolean message that places the robot to its starting position switching lanes after each reset. Furthermore, it subscribes to the DVS data and defines a callback function that stores the data in a FIFO-queue. When environment is called to perform a discrete action, the stored DVS data is converted into the binary state image and given back to the controller. Moreover, it subscribes to the robots position data as well. Using a predefined model of the lane center in both directions, the reward shown in Figure 4.7 is calculated and given back to the controller as well.

One of the main improvements of DQN with respect to earlier algorithms is the addition of an experience buffer, which stores transitions from the MDP and samples random experiences for stochastic gradient descent steps. First, this decorrelates the data and prevents the algorithm from getting stuck in local minima. Second, by fixating the size of the buffer and throwing out older experience samples, it mitigates between early random exploration and later exploitation of knowledge.

The neural network that was used by the DQN controller was implemented using Tensorflow [134], an open source machine learning software library developed by Google. Tensorflow provides a Python interface that can be used for high-level handling of neural networks while core functions are computed using efficient kernel implementations. Furthermore, in combination with the parallel computing platform CUDA [135] developed by Nvidia, it provides a simple tool for fast GPU computing. In many previous classification tasks as well as control tasks using DQN, Convolutional Neural Networks (CNN) have come to be the preferred network architecture for image data. This is due

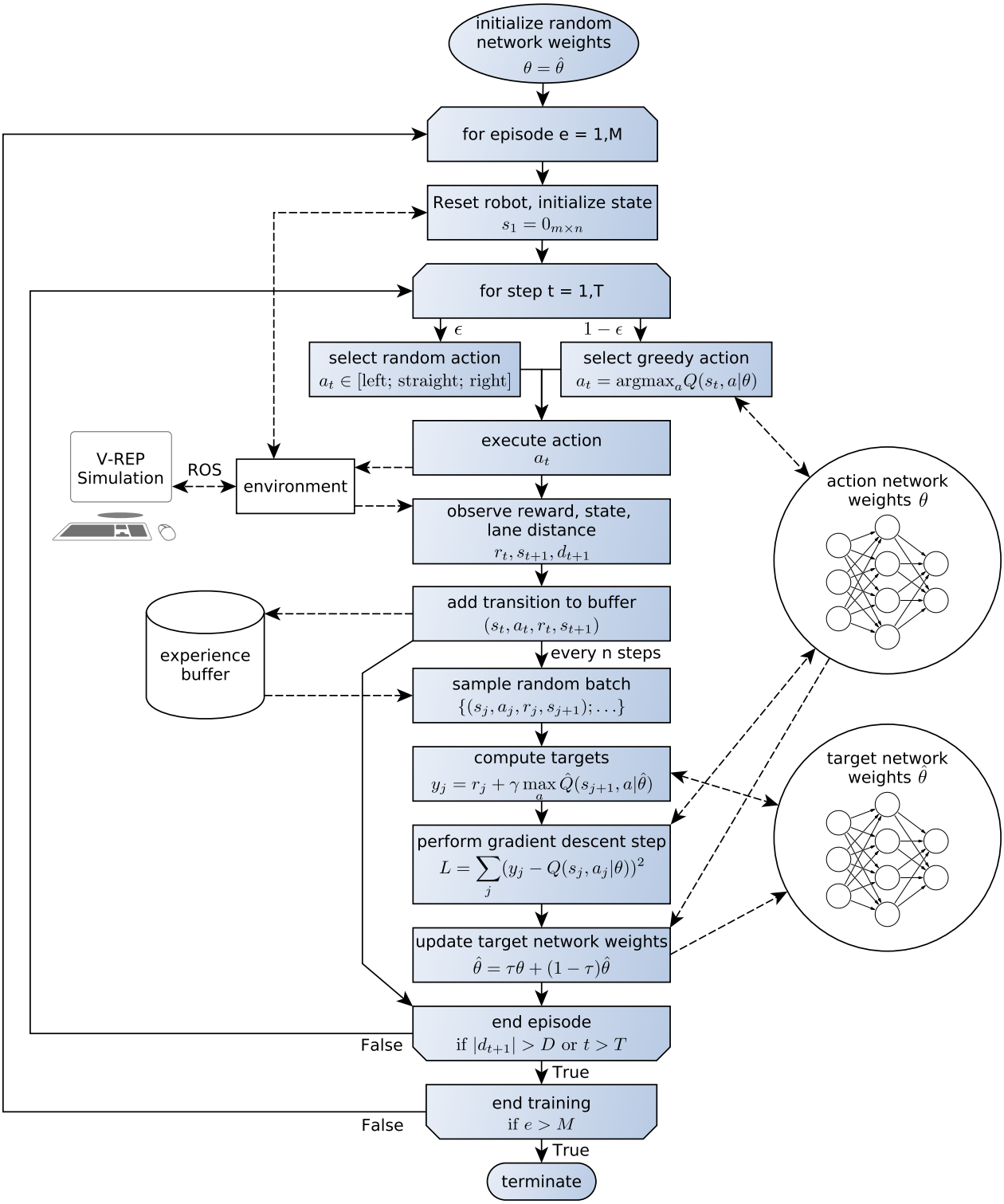


Figure 4.9.: Flowchart of the DQN algorithm.

to the fact that CNNs make use of spatial information of the input data, which can lead to considerable performance gains in many tasks. In this work however, a fully connected feed-forward network architecture using rectified linear units (ReLU) as activation was chosen in order to insure comparability to similar published SNNs (Diehl et al. [21], Lee et al. [10]). The network takes the binary state image as input, resulting in $32 \times 16 = 512$ input neurons. It consists of 2 hidden layers with 200 neurons each and 3 output neurons representing the discrete actions. Training is performed using the stochastic optimization algorithm Adam [136].

Figure 4.9 shows a detailed flow chart of the DQN algorithm discussed in Section 2.1.3 and the communication with external components. In the beginning, the action network is initialized with random weights and copied to the target network. Each episode of the training procedure begins with a reset of the robot to its starting position, changing lanes each episode. Hereby, the initial state input is a vector of zeros.

At each time step, actions are chosen following an ϵ-greedy policy. This means with a probability of $\epsilon \in [0; 1]$ the agent will randomly select an action. Otherwise, it will select the action with the highest action-value. In the beginning, ϵ was set to 1 ensuring pure exploratory behavior. After a predefined number of time steps, ϵ then linearly decreased to its end value close to zero.

Chosen actions are sent to the environment handler, which will communicate with the simulator getting back reward, next state image and the distance to the lane-center. Moreover, each transition is stored in the experience buffer. Every n steps, the actual training step is performed by randomly sampling transitions from the experience buffer. Using the target network for calculating the updating targets, the loss function is then constructed in order to perform a stochastic gradient descent step on the action network. In the end of each training step, the target network weights are slowly updated towards the action network weights with $\tau \in [0; 1]$ and $\tau \lll 1$ [3].

Training episodes end if the robot goes beyond the maximum distance to the lane-center or if the maximum number of steps in an episode is reached. The latter mechanism guarantees that the robot will experience both directions of the road, even if it has already learned a good policy following the lane. The overall training procedure is ended either after a predefined number of episodes or total training steps.

4.2.1.3. Controller 2: DQN-SNN

The DQN algorithm presented in the previous section handles event-based data by storing consecutive DVS frames and batch processing them at every step in the MDP. Clearly, this approach can not be the ideal mechanism for handling this kind of data as it annihilates some of the advantages that make the DVS sensor powerful in the first place, e.g. its temporal resolution. Furthermore, calculations using conventional ANNs are generally computationally expensive and add latency to the control system. Spiking Neural Networks could help solve these problems. First, time plays as crucial role in these networks, which makes them suitable for handling data streams, e.g. from a DVS device, without the need for batch processing. Second, their neurally inspired design could lead to efficient hardware for power efficient computing in the future.
Unfortunately, the DQN algorithm relies on precise estimations of action-values using backpropagation for training. Due to their event-based nature, spikes have to be decoded somehow in order to get real values which makes it very difficult to get network output with similar precision. Moreover, the non-differentiability of spike events makes it very difficult to use training mechanism such as backpropagation, even though Lee et al. [10] have recently shown how this could be achieved anyhow. In classification tasks on the other hand where precise output spike decoding is not necessary, SNNs have repeatedly shown to generate good results even when compared to conventional ANNs, e.g. Lee et al. [10], Diehl and Cook [9], Diehl et al. [21]. Therefore, in this section it is shown how the previously learned DQN policy can be used to create an artificial labeled dataset for supervised learning in order to train a SNN. In other words, a SNN is trained approximating the policy given be the Q-network of the DQN algorithm.
A simple and effective training method for classification tasks has been proposed by Diehl et al. [21]. Based on the fact that simple Integrate-and-Fire neurons in SNNs behave very similar to rectified linear units (ReLU) in conventional ANNs, an indirect training method can be used for training:

1. Create artificial dataset by labeling stored states with actions.
2. Train conventional ANN with no hidden layer biases and ReLu activation functions.
3. Normalize weights.
4. Transfer weights to SNN with IF-neurons and perform control task.

For training ANNs using stochastic gradient descent on the prediction error, the DQN algorithm stores every single transition in the previously discussed experience buffer. This makes it very convenient to use the same data for training the SNN as well. Therefore, at

Algorithm 1 Model-Based Normalization

```
for layer in layers:
    max_pos_input = 0
    for neuron in layer.neurons:                ▷ Find maximum input for this layer
        input_sum = 0
        for input_wt in neuron.input_wts:
            input_sum += max(0, input_wt)
        end for
        max_pos_input = max(max_pos_input, input_sum
    end for
    for neuron in layer.neurons:                ▷ Rescale all weights
        for input_wt in neuron.input_wts:
            input_wt = input_wt / max_pos_input
        end for
    end for
end for
```

Figure 4.10.: Model-based normalization algorithm for converting ANNs into SNNs according to Diehl et al. [21].

first, all stored state images as shown in Figure 4.6 are labeled using the pre-trained action network from DQN. While the previously trained action network used the binary input images $s_{M\times N}$, it is important to note that for training SNNs the non-binary images $i_{M\times N}$ as shown are used. Furthermore, the pixel values $i_{m,n}$, describing the number of spike events in the same 4×4 window over consecutive DVS frames, are scaled to $\hat{i}_{m,n} \in [0; 1]$ by dividing every value by the maximum pixel value $i_{max} = \max_{j,m,n}(i^j_{m,n})$ in the whole dataset. Therefore, the input values can be interpreted as spike firing rates making the network transferable to a SNN.

In the next step, the labeled dataset is used to train a conventional ANN. The fully-connected feed-forward network consists of an input layer with bias and $32\times16 = 512$ input neurons, a hidden layer without any bias and 200 neurons, and an output layer also without any bias and 3 output neurons. It was implemented in Tensorflow using dropout [137] and the Adam [136] optimizer to get good classification accuracy.

After training, the weights can be transferred to a SNN with IF neurons that matches the previous network architecture. In order to avoid inputs to neurons that exceed the firing threshold in single simulation time step, the weights are normalized for each layer separately beforehand. This can be achieved by scaling weights such that the maximal weighted input to a neuron in each layer equals the firing threshold in oder to avoid

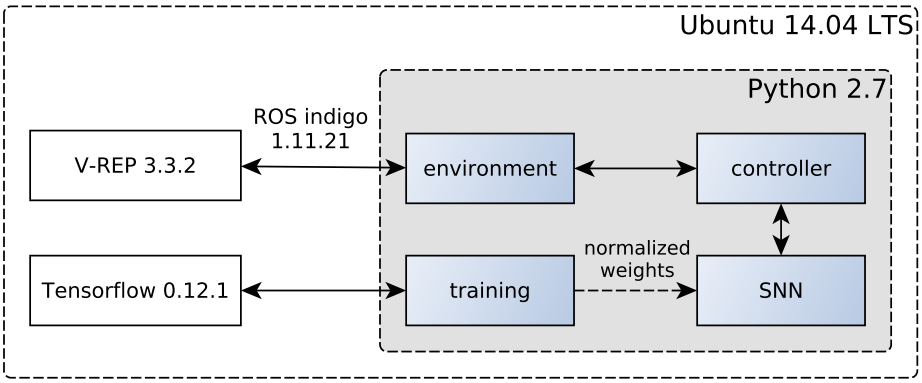


Figure 4.11.: Communication between components of the DQN-SNN controller.

information loss during network simulation and insure minimal accuracy reductions in the SNN with transferred weights.

The normalization algorithm is shown in Figure 4.10. For the first layer, the bias is needed to handle input vectors consisting of zeros only. Without any bias, the network output would be always zero as well, regardless of its weights. In multi-layered SNNs, external input currents introducing biases to deeper layers are difficult to envision, because they have to be scaled somehow matching the firing rates coming from previous layer neurons. For the first layer though, the bias can be interpreted as an additional input current with a constant firing rate of 1.

The SNN with simple IF neurons was implemented in Python based on the Matlab implementation by Diehl et al. [21], which uses a time step based approach in order to propagate spikes through the network and update membrane potentials. As mentioned earlier, the V-REP simulator publishes DVS data every $50\,ms$. The data is then scaled to $[0; 1]$ causing Poisson input neurons [138] to fire for $50\,ms$ as well. The scaling factor can be roughly estimated by dividing the maximum pixel value v_{max} by the number n of consecutive DVS frames used in the data set. Therefore, if a Poisson neuron fires with its maximum frequency over n simulation steps, it can be interpreted as the maximum firing rate of 1 in the data set.

The information from consecutive DVS frames is propagated through the SNN over time. When a neuron fires and sends a spike to the next layer, it increases the membrane potential of the next layer's neurons. Therefore, information is stored in the membrane potentials and it takes some time to generate output spikes. As a consequence, this means that output spikes are generated sparsely in time, leaving simulation steps with no spike output at all. In order to generate a control signal, even if there are no output spikes during a simulation step, a trace was implemented for each action. The action trace z_t^a

accumulates output spikes s_t for each action respectively and decays over time with a factor $c \in [0; 1]$. The action with the highest trace value is eventually chosen at every simulation step:

$$z^a_{t+1} = c \cdot z^a_t + s_t \tag{4.2}$$

$$a_t = \text{argmax}_a(z^a_t) \tag{4.3}$$

Figure 4.11 shows all components that have been used for the SNN controller. The communication with the simulator is managed using the same environment handler as before.

4.2.2. R-STDP

Training a neural network with DQN to learn a policy and transferring the policy to a SNN by creating an artificial labeled dataset is cumbersome and it introduces some loss in the training process. Furthermore, this approach ignores one of the main strength that SNNs bring compared to conventional ANNs, which is their ability to take the precise timing of spikes into account and not just the averaged rate. Therefore, ideally one would take the DVS data feeding it directly into a SNN and use some reinforcement learning algorithm to train the weights accordingly. Unfortunately, as discussed in Chapter 3, a suitable algorithm with reinforcement learning capabilities for sequential decision making tasks has yet to be found and constitutes an ongoing research topic.
In order to serve as a basis for further investigations, Kaiser et al. [15] proposed a simple Braitenberg vehicle controller for the right lane following task. Introduced in his book "Vehicles", Braitenberg [16] presented simple mobile agents with a different number of sensors and motors. Depending on simple static connection schemes between sensors and motors, these agents exhibit simple animal-like behavior, such as turning towards or away from a sensory stimulus, e.g. in the form of light.
In this section, the steering wheel model and the components implemented for the Braitenberg vehicle controller are discussed first. Afterwards, it is shown how this approach can be adapted such that it learns how to follow the lane based on a R-STDP rule instead of setting weights manually.

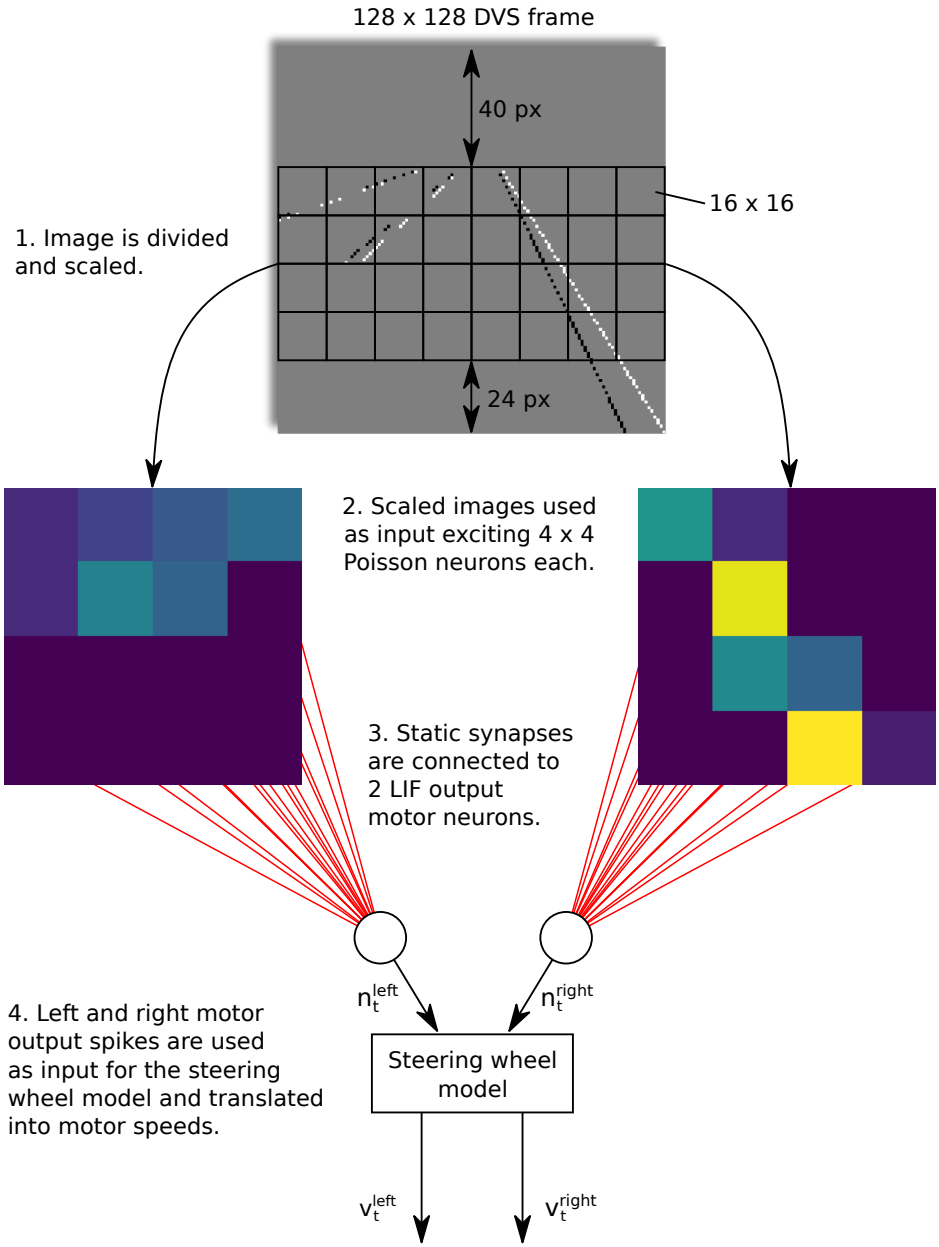


Figure 4.12.: Network architecture of the Braitenberg vehicle implementation using DVS frames as input.

4.2.2.1. Controller 3: Braitenberg

The control architecture of the Braitenberg vehicle implemented in this work is shown in Figure 4.12. It is based on the idea that neural activity, excited by DVS events from the left or right side of the image, will cause left and right motor to increase speed turning the robot away from neurally active directions. Therefore, the robot will move keeping the activity at a local minimum and follow the lane enclosed by left and right markings. The DVS images are cropped at the top and bottom the same way as before. The remaining image is then divided into two 4×4 images, where each pixel value describes the number of all ON- and OFF-events in a 16×16 window of the original image.
As discussed in Section 4.1.3, the simulator sends DVS events in batches all with the same time stamp. In order to feed them effectively into a SNN they have to be "spread" again in time across a simulation time step. Therefore, instead of generating spikes directly from DVS events, the pixel values of the 4×4 images are used to excite Poisson neurons to fire with the respective rate during a simulation time step.
The Poisson neurons are connected to 2 Leaky-Integrate-and-Fire (LIF) neurons using static synapses and manually set weights. The number of output spikes n_t^{left} and n_t^{right} of each motor neuron is counted during each simulation step and used as input for the steering wheel model.
In a classic Braitenberg mobile agent, the motor neuron activity is directly used for motor speed control. Kaiser et al. [15] however proposed a steering wheel model based on an agonist-antagonist muscle system that is more suitable for car control. The same model was implemented in this work as well with only one change. Instead of steering angles, turn speeds are computed and added or subtracted for left and right motor (Fig. 4.5). First, the output spike count is scaled by the maximum possible output:

$$m_t^{left/right} = \frac{n_t^{left/right}}{n_{max}} \in [0;1], \text{ with } n_{max} = \frac{T_{sim}}{T_{refrac}}, \tag{4.4}$$

where T_{sim} denotes the simulation time step length and T_{refrac} describes the refractory period length of the LIF neuron. Based on the difference of the normalized activities m_t^{left} and m_t^{right} and a turn constant c_{turn}, the turn speed is defined as

$$S_t = c_{turn} \cdot a_t, \text{ with } a_t = m_t^{left} - m_t^{right} \in [-1;1]. \tag{4.5}$$

Furthermore, in order to ensure slower speed in turns, the overall speed is controlled according to

$$V_t = -|a_t| \cdot (v_{max} - v_{min}) + v_{max}, \tag{4.6}$$

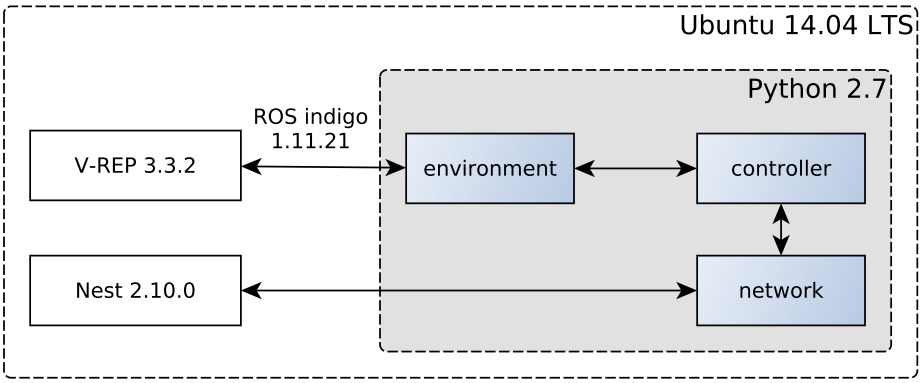


Figure 4.13.: Communication and components of the Braitenberg vehicle controller and the R-STDP controller.

where v_{min} and v_{max} are predefined speed limits. Since controlling a car is generally a continuous process, overall and turn speed were smoothened based on the activities:

$$v_t = c \cdot V_t + (1 - c) \cdot v_{t-1}, \tag{4.7}$$

$$s_t = c \cdot S_t + (1 - c) \cdot s_{t-1}, \tag{4.8}$$

$$\text{with } c = \sqrt{\frac{(m_t^{left})^2 + (m_t^{right})^2}{2}} \tag{4.9}$$

Finally, the control signals for the left and right motor were computed by

$$v_t^{left} = v_t + s_t \text{ and } v_t^{right} = v_t - s_t. \tag{4.10}$$

In Figure 4.13 all components of the Braitenberg vehicle controller are shown. The environment handler manages all communication with the simulator and converts DVS frames into left and right 4×4 images. Moreover, the steering wheel model is implemented in the environment handler as well. The left and right SNN was implemented using the neural simulation tool NEST [19] and its Python API pynest.

4.2.2.2. Controller 4: R-STDP

Instead of dividing the input data and feeding it into two separate networks with static weights, this part of the Braitenberg control architecture can be exchanged with a single SNN. Figure 4.14 shows the changes that have been made to the network. Overall communication between components has not been changed (Fig. 4.13). As before, the input data is scaled and used for excitation of Poisson neurons, this time in a single network with $8 \times 4 = 32$ input neurons. The input layer is connected to two LIF output neurons in an "all to all" fashion using R-STDP synapses. The number of output spikes n_t^{left} and n_t^{right} in each simulation step again serves as input for the Steering Wheel model presented in the previous section. In principle, this amounts to the same basic network architecture as the Braitenberg vehicle controller, where missing synaptic connections from the other half of the image can be interpreted as zero weight connections.
The network was again implemented in NEST using the stdp dopamine synapse model that comes with the simulator for all connections. Based on Izhikevich [68] and Potjans et al. [22], a low-pass filtered version of the spike rate of a user-specific pool of neurons constitutes the dopaminergic signal that modulates spike-timing-dependent plasticity as shown in Section 2.2.3.3. In order to reduce the complexity involved in the control task, the reward signal in this work was directly set at each simulation time step instead of doing it indirectly by exciting a pool of dopaminergic neurons first. While this approach may be less biologically plausible, it simplifies the control task without changing the basic underlying dynamics of the problem, which makes it a good starting point. The reward signal given at each simulation time step is shown in Figure 4.15. It is defined for each motor with opposite signs linearly depending on the robot's distance to the lane-center. When the robot is right from the lane-center and should turn left to the center, connections that led the right motor neuron to fire are strengthened, connections that led the left motor neuron to fire are weakened. On the opposite side of the lane-center this process is turned around. Over time, the robot should learn to associate certain input stimuli with left or right turns and act accordingly. These considerations lead to the following rewards for left and right motor neuron connections with d being the distance to the lane-center and c_r a constant scaling the reward:

$$r_{left/right} = -/+ (d \cdot c_r) \tag{4.11}$$

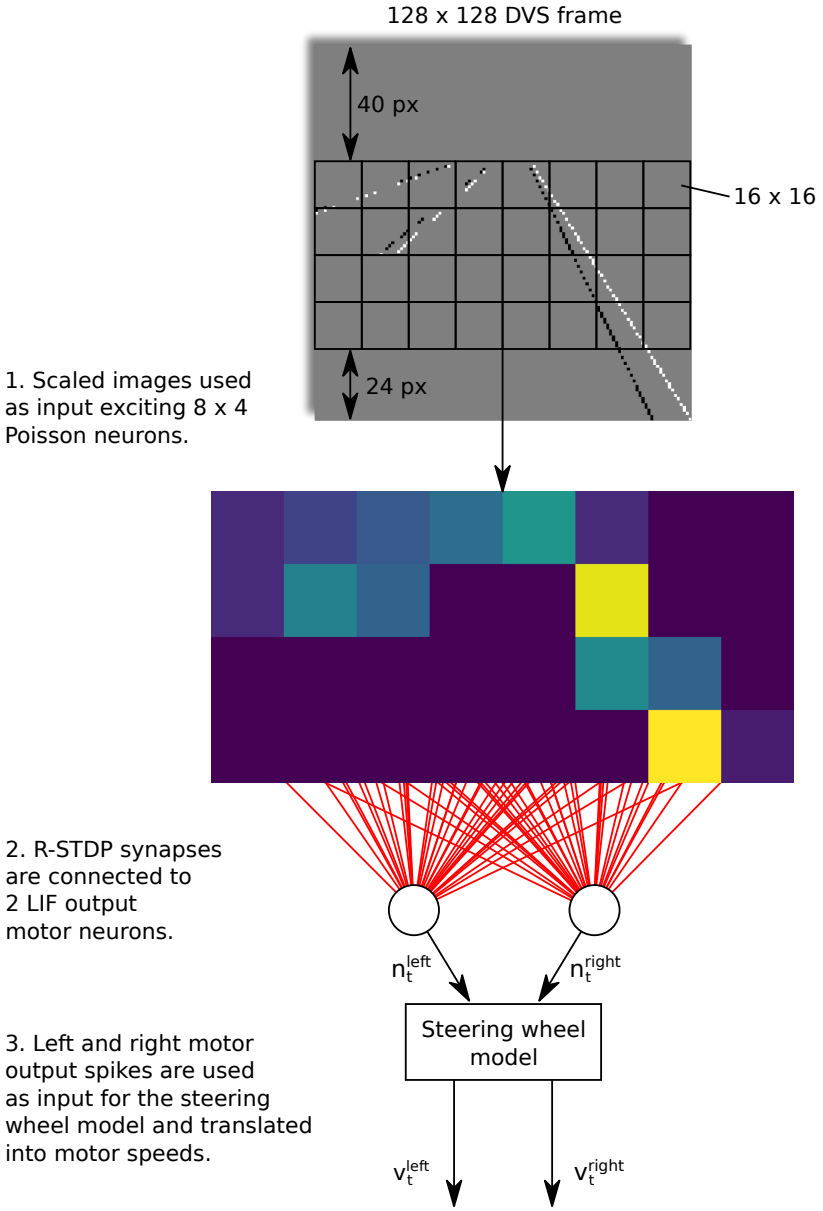


Figure 4.14.: Network architecture of the R-STDP implementation using DVS frames as input.

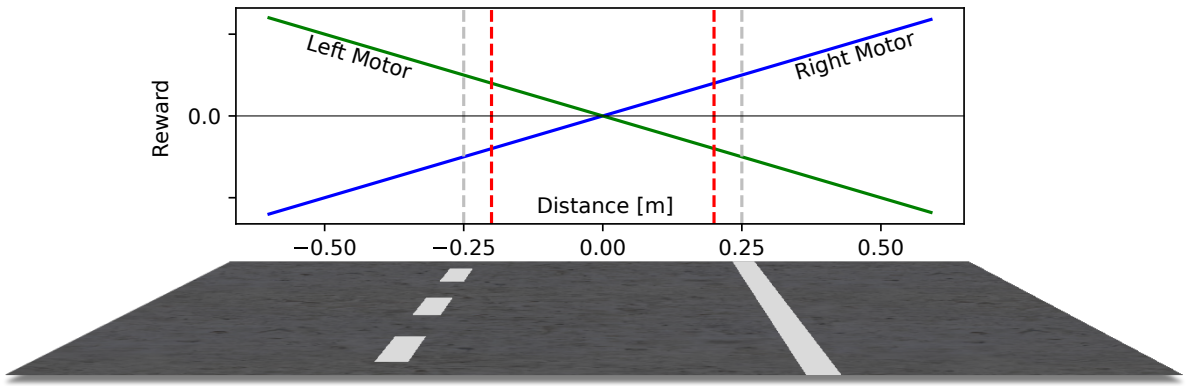


Figure 4.15.: Reward given by the R-STDP controller: It is defined for each motor individually as a linear function of the lane-center distance scaled by a constant c_r. The lane markings are $0.25\,m$ away from the lane-center. If the robot will go further than $0.2\,m$ from the lane-center, episodes are terminated and the robot will be positioned at its starting position.

5. Discussion

With the lane-following task in mind, 4 controllers for the Pioneer robot were presented in the previous chapter approaching the problem from two different directions. While the last chapter was concerned with the basic principles and implementation details of these controllers, training parameters as well as algorithm performance were left out. Therefore, in this chapter the controllers are tested in different lane-following scenarios and discussed afterwards. Simulation parameters as well as additional training details are given in Appendix A and B.

5.1. Lane-Following Scenarios

The first scenario that the controllers were trained on is shown in Figure 5.1. It consists of a circular course with a two-lane road. The road is comprised of two solid lines and a uniformly dashed line in the middle. From the starting position onwards, the outer lane can be divided into 6 sections, namely *(A) straight*, *(B) left*, *(C) straight*, *(D) left*, *(E) right* and *(F) left*. Similarly, the inner lane can be divided into sections *(C) straight*, *(B) right*, *(A) straight*, *(F) right*, *(E) left* and *(D) right*. During training, the robot will switch between inner and outer lane at each reset. Therefore, it will experience both left and right turns equally. Furthermore, it will experience turns with different radii as well.
In the simulator the scenario was constructed such that the course position of the robot (position of the robot orthogonally projected to the lane-center) as well as the lane-center distance (orthogonal distance of the robot to the lane-center) could be easily computed using the positional data from the simulator and the known lengths and radii. These values were used for formulating rewards during training and evaluating the performance of the controllers afterwards.
Based on the same layout and dimensions, a second scenario has been implemented testing the algorithms on a different road pattern where the left and right solid lines are missing. In a third scenario, two different road patterns had to be learned in parallel (Fig. 5.2).

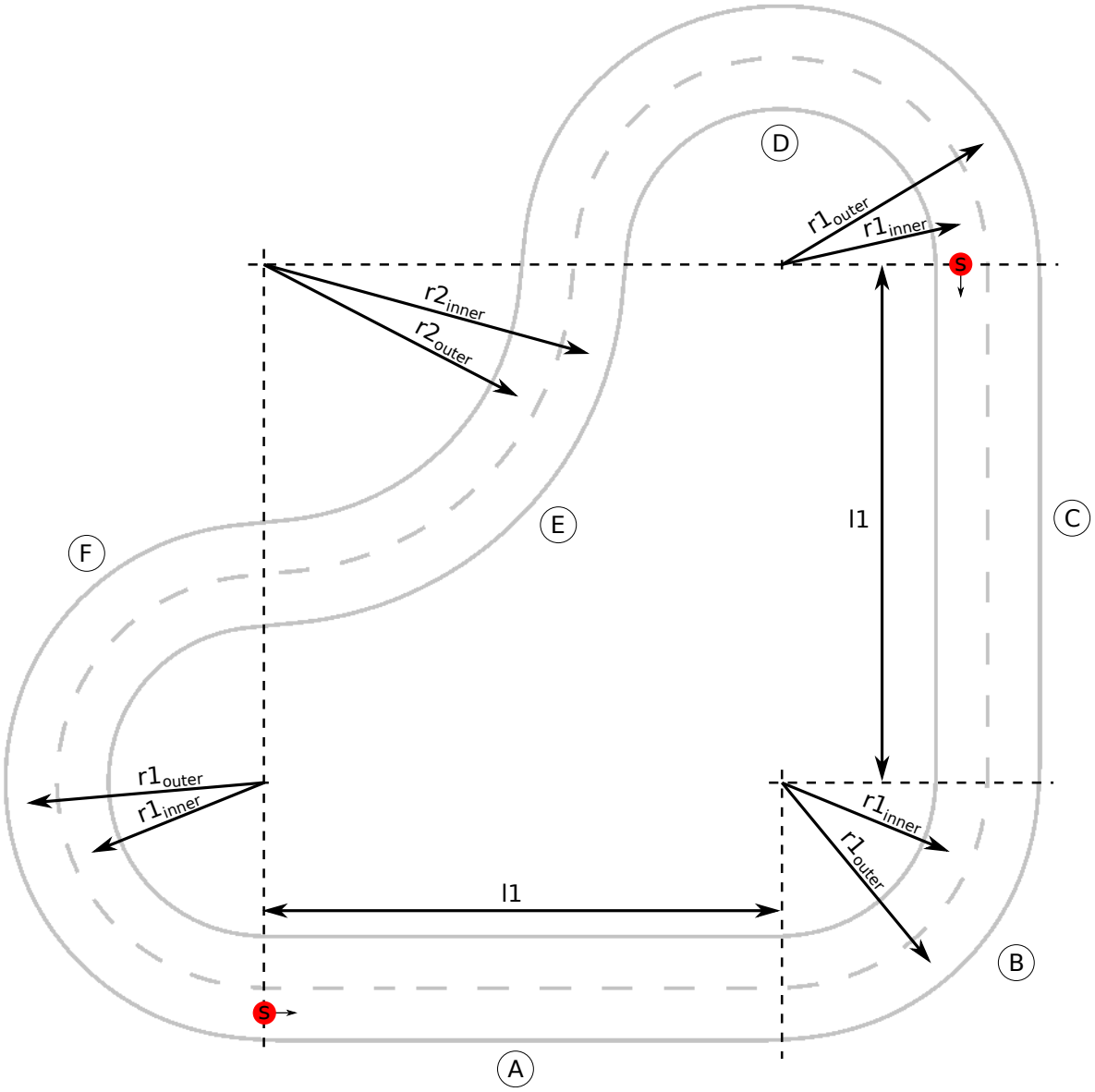
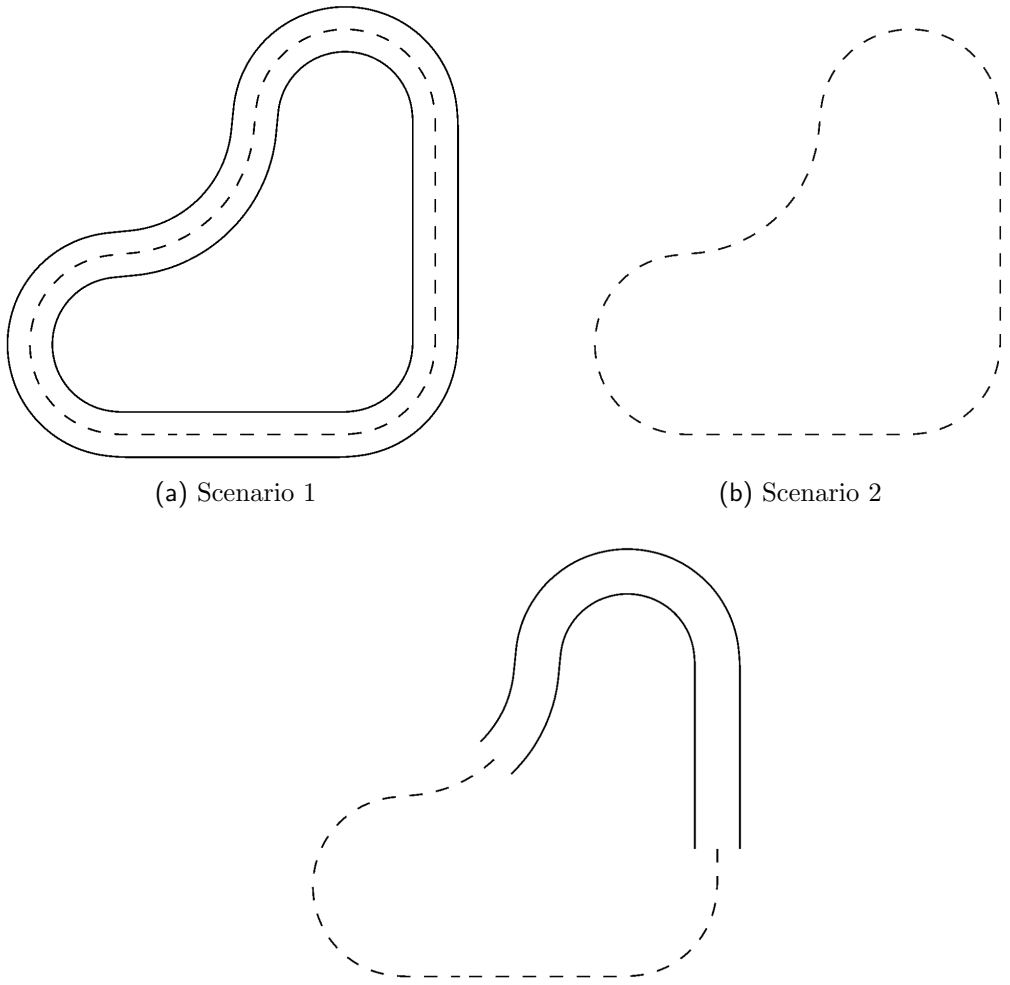


Figure 5.1.: Scenario 1: The simple lane-following scenario consists of a road with 2 lanes and 6 different sections A, B, C, D, E and F. Starting positions are marked with "s". Dimensions: $r1_{inner} = 1.75m$, $r2_{inner} = 3.25m$, $r1_{outer} = 2.25m$, $r2_{outer} = 2.75m$, $l1 = 5.0m$



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

Figure 5.2.: Different lane following scenarios. In scenario 1 and 2 the road markings are following a single pattern. In scenario 3, two different patterns have to be learned in parallel.

5.2. Training Details

5.2.1. Controller 1: DQN

In Figure 5.3 the training progress of the DQN algorithm in scenario 1 is shown. Training parameters are given in Table A.1. In the beginning, the robot will randomly choose actions. Episodes are terminated once these random actions lead the robot beyond the $0.5\,m$ lane-center distance threshold. Therefore, steps and rewards in the beginning are randomly distributed at a low level. Even though the ϵ-greedy policy constantly increases

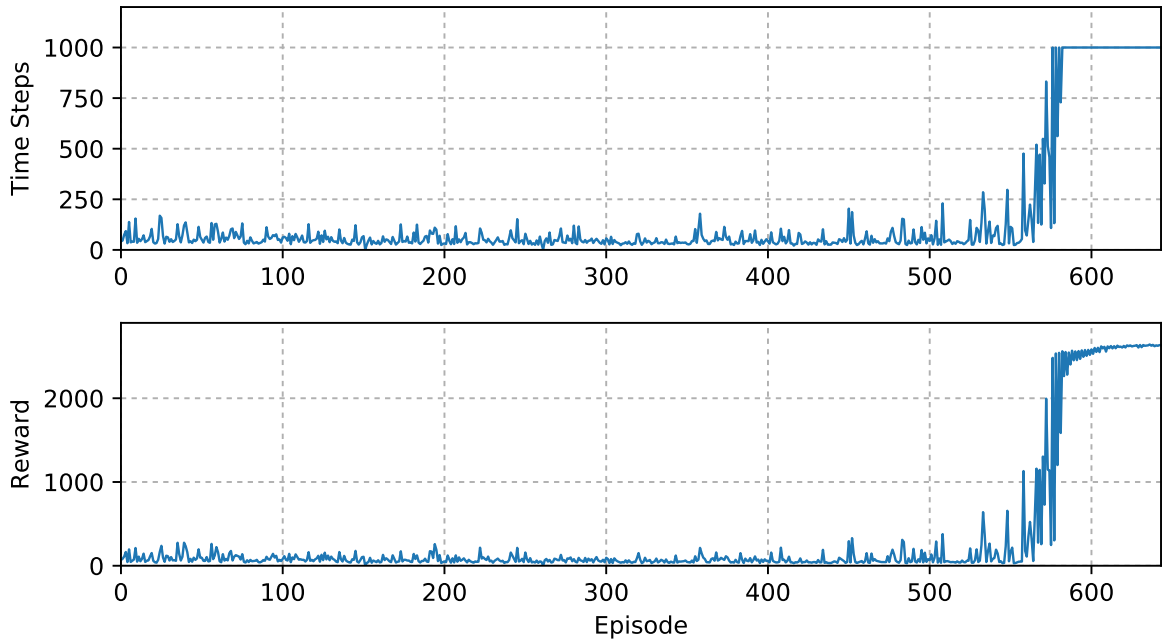


Figure 5.3.: Scenario 1: Time steps and rewards for each training episode of the DQN controller. After 500 episodes, the robot learns to follow the lane without causing a reset. Episodes are limited to 1000 steps. The accumulated rewards collected in 1000 steps still improve after the robot has reached the step limit.

the chance of choosing the action with the highest action-value, the robot doesn't show any learning effect until episode 400. At around episode 300, steps and rewards are actually decreasing, because the robot is following a policy that is not optimal yet. After approximately 580 episodes, the robot has learned to follow the lane without causing a reset. In order to ensure experiences from both lanes, even if the robot has successfully learned to follow them, episodes are terminated after 1000 steps and the robot is placed on the other side of the road. After 580 steps the accumulative rewards over 1000 steps are still slowly increasing approaching a reward maximum.
Similarly, the algorithm learns a control strategy in scenario 2 as well. Due to the reduced complexity in the state images, effective learning already begins after 300 episodes (Fig. B.1). In the third scenario, on the other hand, the algorithm failed to learn a stable policy. Figure 5.4 shows the episode lengths and rewards in 170000 time steps total. The starting positions in scenario 3 are placed such that the robot experiences both road patterns from the beginning on. At around episode 800, the robot learns to follow the lane, even completing laps and reaching the time step limit at times. Unfortunately, it does not learn a generalized policy that works for both lanes. Once the algorithm figures out how to take a turn or a transition section from one pattern to another, it seems to have detrimental

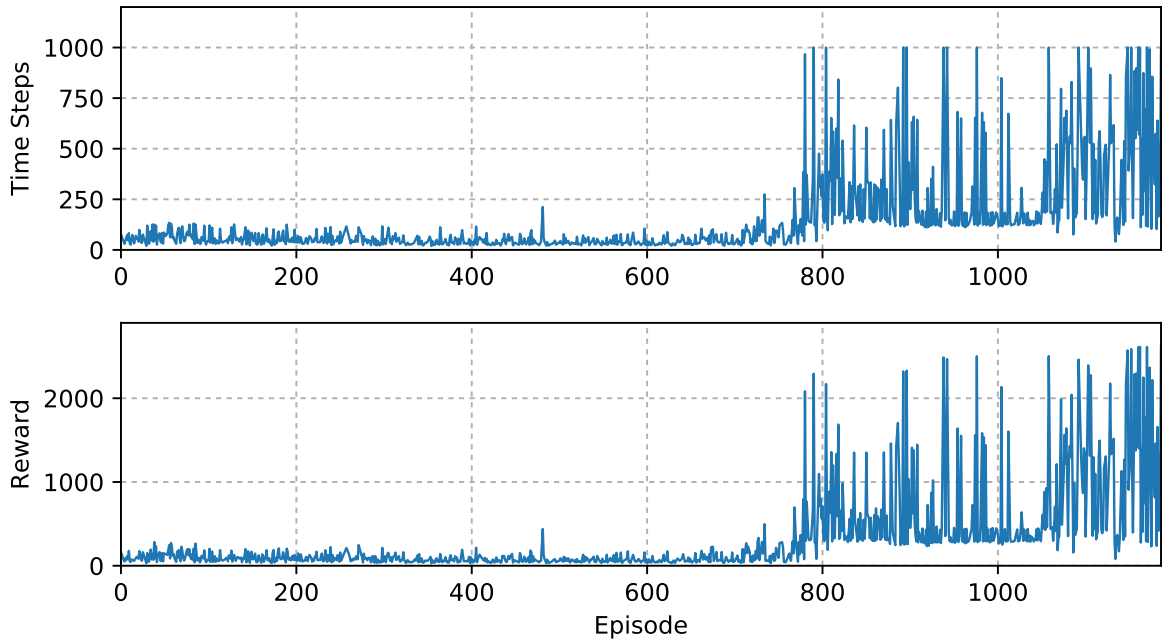


Figure 5.4.: Scenario 3: Time steps and rewards for each training episode of the DQN controller. The algorithm failed to learn a stable policy after 1186 episodes and 170000 steps.

effects to its behavior in other situations. Even though the average reward over several episodes increases towards the end, the algorithm never reaches the time step limit in consecutive episodes. Taken together, the algorithm in scenario 3 optimizes its behavior but fails to reach a global reward maximum.

5.2.2. Controller 2: DQN-SNN

For the policy transfer from the DQN action network to a SNN, an artificial dataset was created using the state samples stored in the experience buffer. During training of the DQN controller in scenario 1, 98990 state samples were visited and stored. At the beginning of the training procedure, the robot experiences many states that are far from the optimal lane-center position. Once it has learned to follow the lane, on the other hand, it will experience only states close the lane-center. For the policy transfer these states are much more important, because the robot controlled by the SNN will likely never see those "poor" states far from the lane-center. Therefore, it is important to train the SNN on a dataset with mostly "good" states. This was achieved by letting the robot run and collect states for a while after successfully learning a good policy to ensure a favorable distribution of states in the dataset. Using the previously trained DQN action network,

Figure 5.5.: Scenario 1: Static connection weights to the left and right motor neuron of the Braitenberg vehicle controller.

all states were labeled with actions and an ANN (Sec. 4.2.1.3) was trained reaching a classification accuracy of 93.05%. Further training and network parameters are shown in Table A.2. Afterwards, the network weights were normalized and transferred to a SNN with the same architecture performing the robot control task. In scenario 2, 100236 states could be classified with an accuracy of 91.71% following the same procedure. Due to the fact that the DQN algorithm could not successfully learn a stable policy in the third scenario, the DQN-SNN controller was only implemented for scenario 1 and 2.

5.2.3. Controller 3: Braitenberg

In a classic Braitenberg vehicle, the activity of sensory inputs steers the agent towards stimuli or away from stimuli depending on the connection scheme. In scenario 1, the robot is supposed to follow the lane without crossing the solid line on the right or the dashed line in the middle of the road. Therefore, if the robot deviates from the lane-center, the motor

neuron activities should increase or decrease such that the robot adjusts its direction accordingly. Figure 5.5 shows the weights of the synaptic connections to the left and right motor neuron. If a line in the robot's vision gets closer to the bottom center of the image, the related motor neuron activity will be increased while the opposite motor neuron's activity will be decreased. If the robot gets close to the solid line on its right side, for example, left and right motor neurons will de- and increase their firing rate, respectively, causing the robot to turn to the left. The same principle applies for the opposite side as well. The network weights have been chosen manually by trial and error. Parameters are given in Table A.3. This controller has only been applied in scenario 1.

5.2.4. Controller 4: R-STDP

In order to let the controller learn the network weights by itself instead of choosing the network weights from theoretical considerations and by trial and error, the static connections of the Braitenberg vehicle controller were replaced with R-STDP synapses. However, the training parameters of the R-STDP controller have to be carefully chosen to ensure successful learning.
First, the reward constant as discussed in Equation 4.11 has to be determined. The constant defines the magnitude of weight changes during learning and turned out to be a crucial part for the learning process. If the constant is too low, learning will take too much time and it might be difficult to see any progress at all. If it is too high, on the other hand, learning gets increasingly instable and the robot won't learn anything. The reward constant value was chosen manually after several trials (Tab. A.4). Second, the initial network weights are critical for learning as well. In this work, weights were initialized uniformly at a relatively low value of 200. The weights have to be larger than zero, because both motor neurons must be excited from the beginning in order to induce weight changes following the R-STDP learning rule. In the best case the initial weight values are as close as possible to their final values after learning. Therefore, the initial weight value has been set to an estimated mean value of the weights after learning. Furthermore, the weights were clipped to $[0; 3000]$ only allowing excitatory synaptic connections.
In Figure 5.6 the training progress of the R-STDP controller in the first scenario is shown. Specifically, it shows the termination position of the robot at each trial when it exceeds the lane-center distance of $0.2\,m$ causing a reset. Moreover, the changes of the synaptic weights are shown over the course of the simulation. A simulation step is equivalent to $50\,ms$ both for the simulation of the SNN as well as the robot simulator itself. In the beginning of the training procedure, the robot will go straight forward, because all connection weights for both motor neurons have been set to the same value. Therefore, during the first 10000

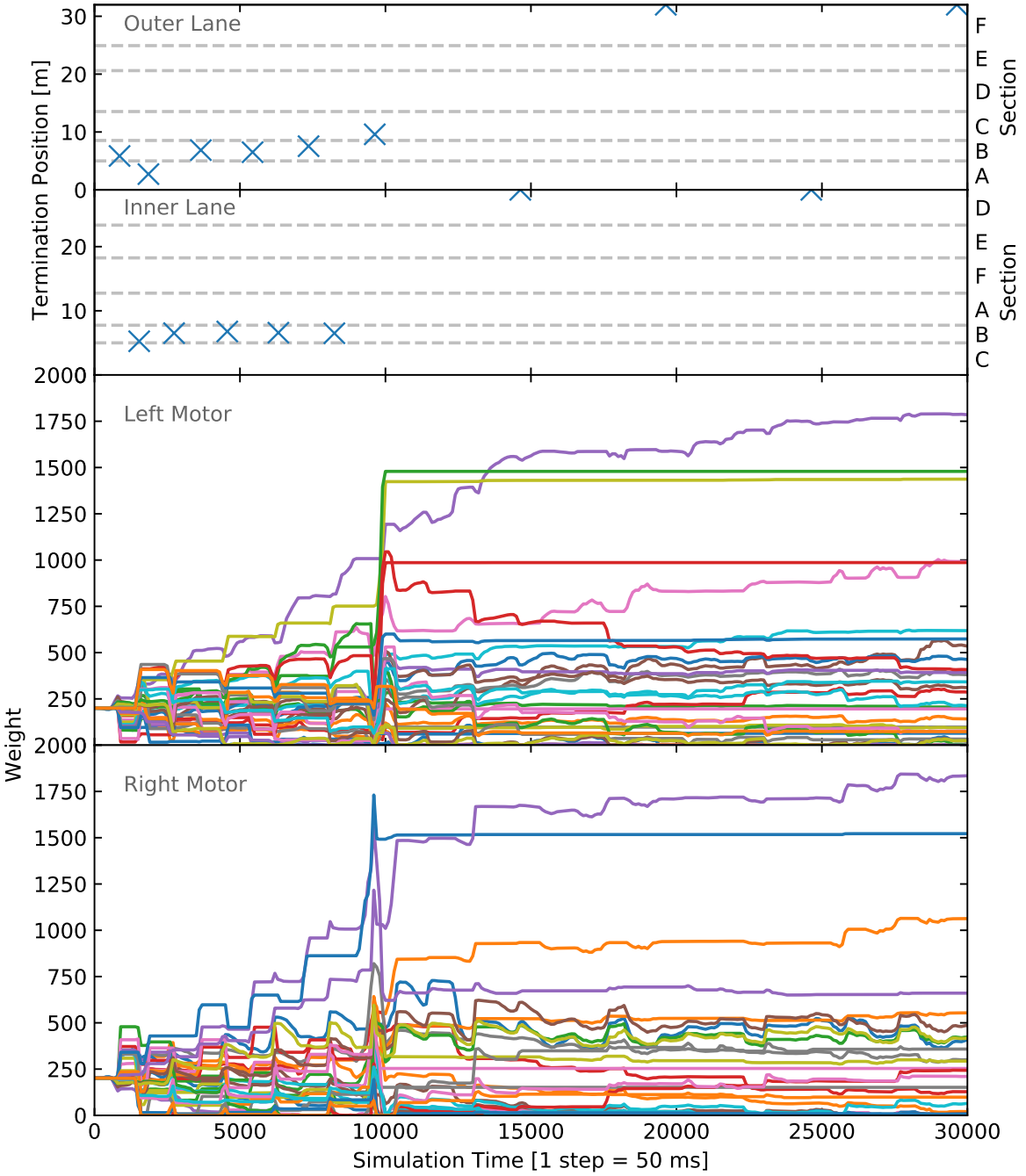


Figure 5.6.: Scenario 1: Learning progress of the R-STDP controller. The termination position when the robot causes a reset and the network weights are shown over the number of simulation steps ($1\,step = 50\,ms$). During the first 10000 simulation steps, the robot causes resets at each trial in the first turn in both directions (Termination in section B). Afterwards, it has successfully learned how to follow the lane only causing a reset when a lap is completed.

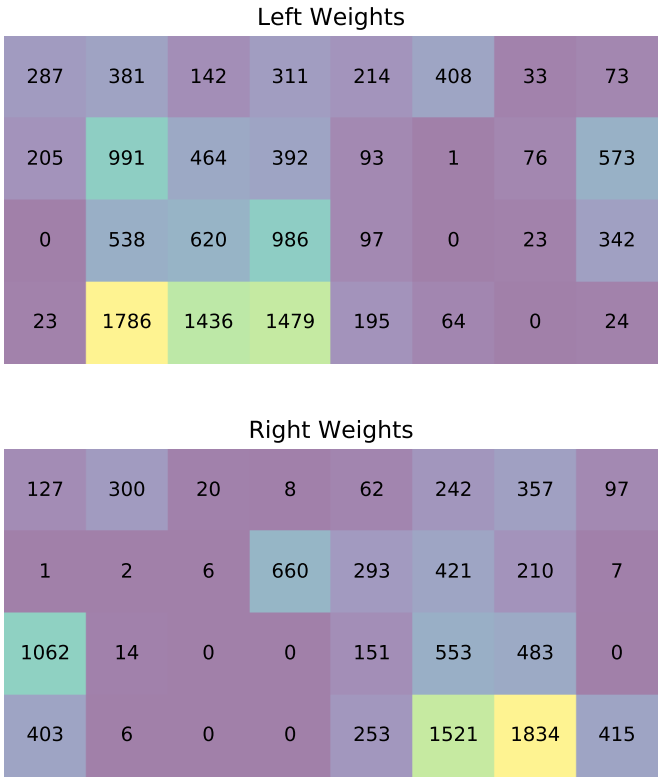


Figure 5.7.: Scenario 1: Learned connection weights to the left and right motor neuron of the R-STDP controller after 30000 simulation steps.

simulation steps, trials are mostly terminated at the first turn in both directions when the robot misses the turn and the lane-center distance exceeds $0.2\,m$. Each time the robot misses a turn, it will periodically induce high reward values in the beginning changing the synaptic weights. Shortly before step 10000, the robot has learned to take the turn, but it still deviates from the optimal lane-center position. Consequently, the high reward over a longer period of time causes a significant change in the connection values. Evidence for that can also be found by looking at the termination position on the outer lane shortly before step 10000 that lies beyond the first turn. Afterwards, the controller follows the lane in both directions without causing a reset. Episodes are only terminated once the robot has completed a lap. Following both lanes close to the optimal lane-center position means low reward values as well. Therefore, the weight changes after step 10000 are considerably smaller than before. The learned weights after 30000 simulation steps are shown in Figure 5.7. Interestingly, the connection weights resemble the theoretically derived weights of the Braitenberg controller with very low values at one half of the image and increasing values from the top corner to the bottom center at the other half of the image. Furthermore, it

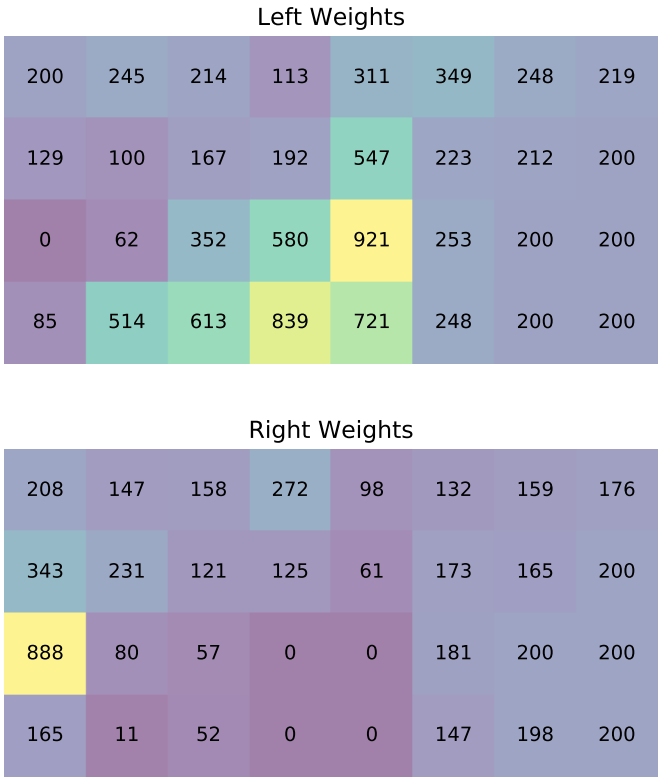


Figure 5.8.: Scenario 2: Learned connection weights to the left and right motor neuron of the R-STDP controller after 30000 simulation steps.

can be seen that left and right motor neurons seem to be triggered mostly through middle and right road line enclosing the lane.

The training progress of the R-STDP controller in scenario 2 is shown in Figure B.2 and seems very similar to the first scenario, completing the first full lap in less than 5000 simulation steps. The weights of the controller network after 30000 simulation steps are shown in Figure 5.8. While the networks weights on the left side from both motor neurons resemble the connection weights learned in scenario 1, it can easily be seen that the weights on the right side have been left unchanged, due to the missing lines in this scenario and the consequential lack of activity during training.

Figure 5.9 shows the learning progress during training in scenario 3. First, learning a successful control strategy takes considerably more time than in the first two scenarios. The obvious explanation for this is that scenario 3 incorporates two different road patterns making the environment more complex. Therefore, the controller has to distinguish between a higher number of different situations as well slowing down the learning procedure. Moreover, due to the simple fact that the robot does not encounter certain

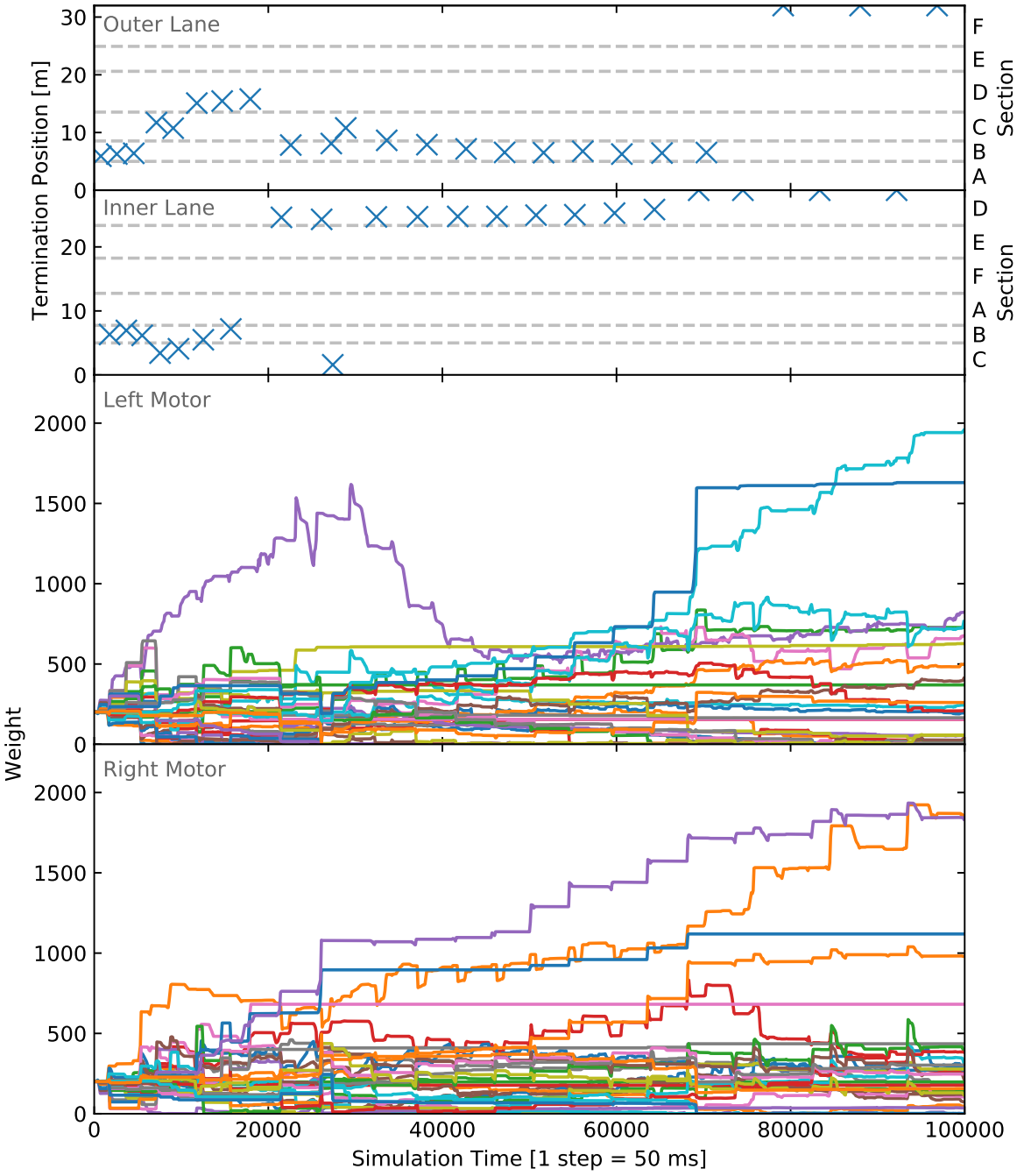


Figure 5.9.: Scenario 3: Learning progress of the R-STDP controller. The termination position when the robot causes a reset and the network weights are shown over the number of simulation steps ($1\,step = 50\,ms$). After an initial learning phase, the robot is mostly reset in sections B and D until laps are completed on both lanes after approximately 75000 steps.

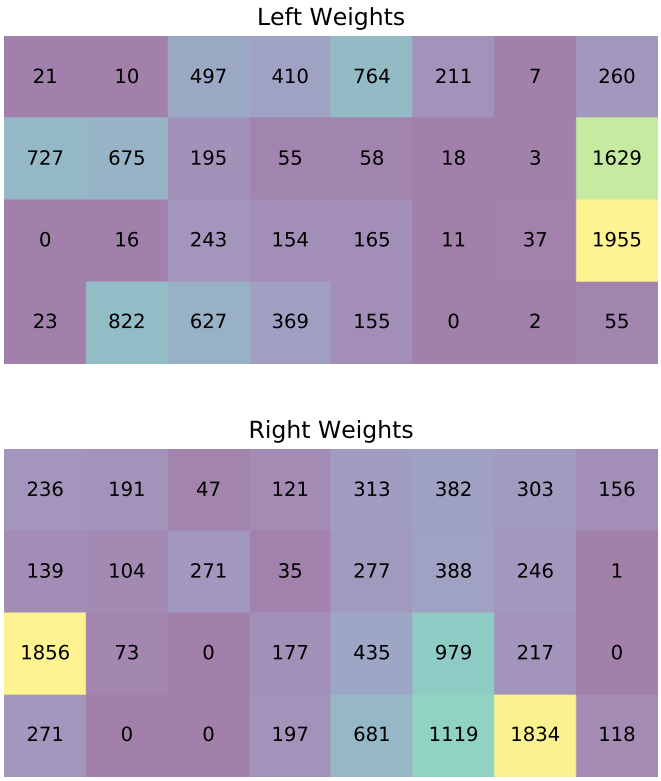


Figure 5.10.: Scenario 3: Learned connection weights to the left and right motor neuron of the R-STDP controller after 100000 simulation steps.

situations until it has learned how to get there, it will only start learning a generalized control strategy that works for both lanes towards the end. In the left motor plot it can be seen that some weights might even be increased in the beginning and decreased again afterwards. After an initial learning phase until approximately step 20000, the controller is mostly reset in sections B (outer lane) and D (inner lane). When the weights have adapted sufficiently after approximately 75000 steps, the robot finishes the laps on both lanes. In Figure 5.10 the learned weights after 100000 steps are shown. In comparison to the first scenario the weight patterns seem very similar, which makes sense considering the fact that the road pattern in scenario 1 is the combination of both road patterns in scenario 3.

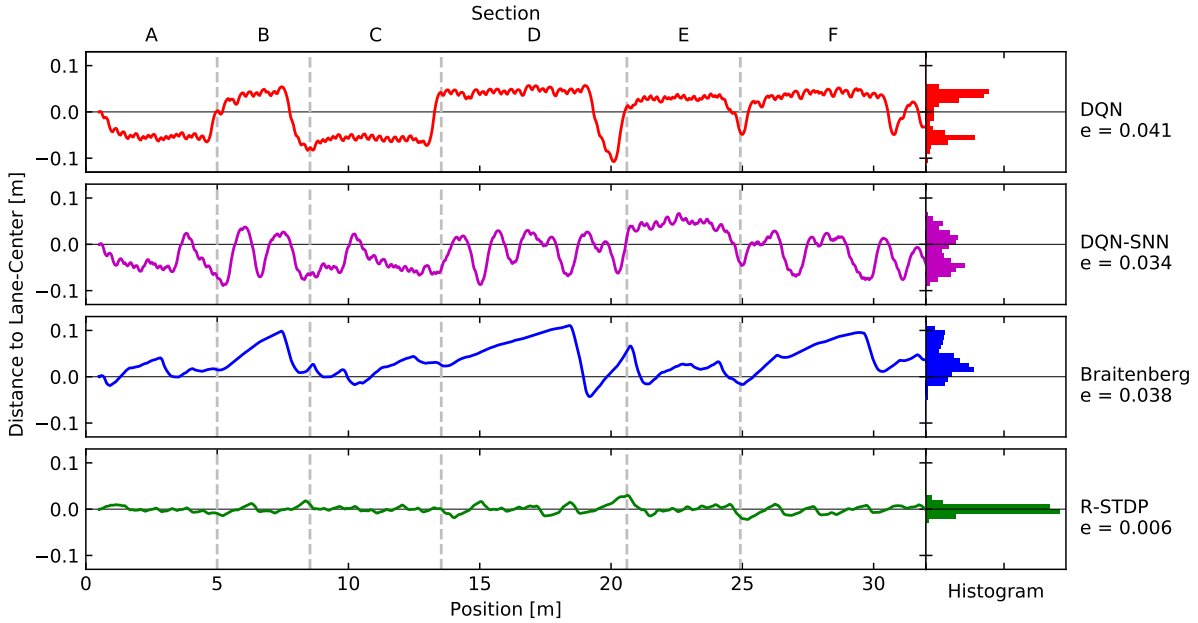


Figure 5.11.: Scenario 1: Comparison of different controllers on the outer lane. The deviation from the lane-center is shown over the robot position projected to the lane-center. Positive lane-center distances correspond to deviations to the right side, negative distances to the left side. Course sections are marked by vertical dashed lines (A=straight, B=left, C=straight, D=left, E=right, F=left). On the right side, error distributions for all controllers as well as mean errors e (mean distance to the lane-center) are shown.

5.3. Performance & Comparison

In the beginning, all 4 controllers were successfully trained and tested in scenario 1 without any changes being made. While the Braitenberg controller was only implemented for the first scenario, the remaining controllers learned a control strategy for the second scenario as well. Only the R-STDP controller, however, learned a stable control strategy for the third scenario. In order to get comparable performance metrics for each controller, they were evaluated completing one lap on the outer lane in each scenario. Figure 5.11 - 5.13 show the deviation of the robot from the lane-center over the projected course position during one lap for each successful controller. Moreover, the course is divided into the 6 sections shown in Figure 5.1. The robot path representation as a projection to the lane-center line allows for a numerical analysis of the controllers performance. Specifically, the error distribution (distance to the lane-center) can be shown in the form of a histogram as well as the mean error for each controller.

At first, the DQN controller was trained and tested in scenario 1. The behavior of the robot very clearly depends on the section that it is in. In the straight sections A and C

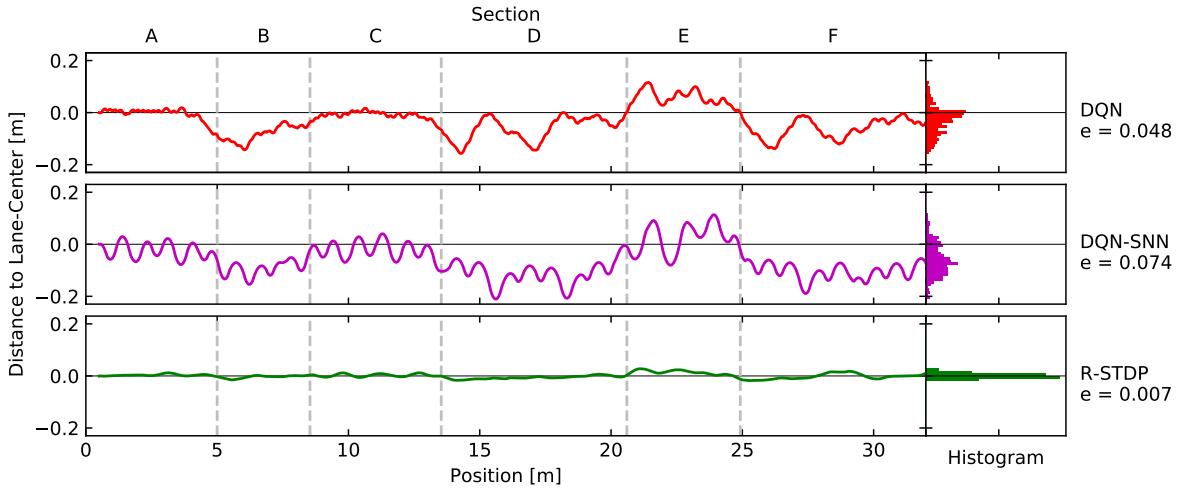


Figure 5.12.: Scenario 2: Comparison of different controllers on the outer lane. The deviation from the lane-center is shown over the robot position projected to the lane-center. Positive lane-center distances correspond to deviations to the right side, negative distances to the left side. Course sections are marked by vertical dashed lines (A=straight, B=left, C=straight, D=left, E=right, F=left). On the right side, error distributions for all controllers as well as mean errors e (mean distance to the lane-center) are shown.

of the first scenario, the robot exhibits a tendency to the left. In the left turn sections B, D and F as well as the right turn section E, the robot tends to the right side of the lane. While the controller does not optimally minimize the lane-center distance over the whole course, it seems to be very stable with a constant deviation during each section. This behavior can also be seen in the error histogram with two peaks at both sides of the lane-center. In contrast to the other controllers, the DQN algorithm leads to the highest numerical error with a mean deviation of $e = 0.041\,m$. Interestingly, even though the MDP was defined with 3 discrete actions *left*, *straight* and *right*, the algorithm only chooses between turning left and right after training. In the second scenario, the DQN algorithm shows a higher mean error, which can be explained by the reduced information in the state images. Especially in turns on the outer lane, when the robot sees only a very small part of the dashed line, there are only a few pixels containing any information. During the straight sections A and C, on the other hand, the robot follows the lane very close to its center having enough information for a near-optimal control strategy. As discussed in the previous section, the DQN algorithm does not learn a stable policy in the third scenario that combines two different road patterns. During training, however, it manages to complete full laps and reaches the time step limit several times proving that it can in fact learn a good policy with different road patterns. The problem here seems to be a general policy that works for both lanes handling the full complexity of the task. Considering other reinforcement learning tasks have successfully been solved using DQN

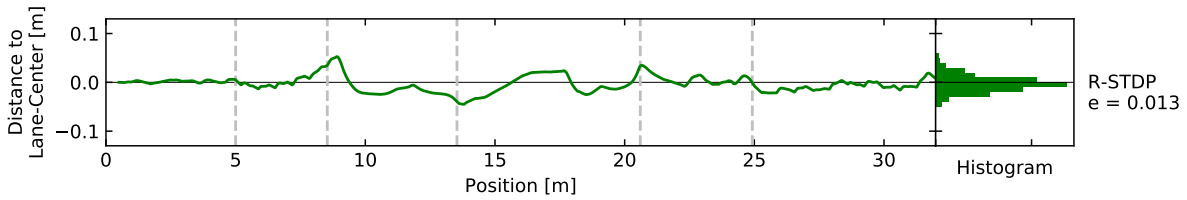


Figure 5.13.: Scenario 3: R-STDP controller on the outer lane. The deviation from the lane-center is shown over the robot position projected to the lane-center. Positive lane-center distances correspond to deviations to the right side, negative distances to the left side. Course sections are marked by vertical dashed lines (A=straight, B=left, C=straight, D=left, E=right, F=left). On the right side, the error distribution as well as the mean error e (mean distance to the lane-center) are shown.

(e.g. playing Atari games in [2]), it seems likely that this is mainly due to the simple network architecture that has been implemented in this work failing to evaluate states accurate enough in order to learn a stable policy.

Following the DQN controller, a SNN was trained in order to approximate the policy learned by the DQN algorithm. Hence, when looking at the performed lap, the DQN-SNN controller exhibits some similarities to the DQN controller, e.g. its left tendency in straight sections (A and C) or its right tendency in right turns (E). Overall the controller seems more unstable with a lot more oscillatory behavior, especially in left turns (sections B, D and F). When looking at the histogram, the error distribution of the DQN-SNN controller looks like a smoothened version of the DQN controller. Moreover, the mean error of the transferred SNN controller is surprisingly lower than the one of the original DQN controller. One explanation for this interesting behavior could be the decision frequency that is much higher as before. For every decision the DQN controller collects consecutive frames in order to have enough data and combines them into one single state image. In this work, states images were composed of 10 DVS frames and decision were made every $10 \times 50\,ms = 500\,ms$. The SNN, on the other hand, does not have to accumulate DVS frames beforehand. The network architecture will combine the data in the membrane potentials over time. Therefore, the network output can be read every $50\,ms$ without having to wait for 10 simulation steps, although it takes some time until enough data has been propagated through the network to produce meaningful output spikes. In fact, in many time steps during the simulation the SNN won't produce any output spike at all, which is why action traces were used to ensure a control signal even if there are no output spikes. Considering the loss that has been introduced when training the SNN on an artificial state-action dataset, the performance of the controller still seems pretty good.

In the second scenario, the SNN controller shows heavy oscillatory behavior. Again, this can probably explained by the reduced amount of information in the image data due

to the missing lines. If less events are created and fed into the network, it takes longer until the information gets propagated through the network and generates an output spike. Therefore, the frequency in which the network can make decisions is much lower resulting in this unstable behavior.
Next, the Braitenberg controller was evaluated performing the same lap. While the controller successfully finishes the course, it can be seen quite clearly that it strongly tends to the right side of the lane, which can be explained by the field of vision of the robot. In the right half of the DVS images, the robot usually only sees the right solid line. In the left half, however, the robot sees the left solid line as well as the dashed middle line of the road leading to a higher number of detected events and a higher activity of the left motor neuron eventually. This will shift the robot to the right until is has reached a balance in the activity of the motor neurons. Even in the right turn (section E) the robot is mostly right from the lane-center. In left turns the distance to the lane-center grows until a point is reached where previously unstimulated neurons with high weights are now excited. These will push the right motor neuron activity leading to a movement correction back to the center. This can be seen in all 3 left turns (sections B, D and F). The value of the controller's mean error during the performed lap is comparable to the first two controllers.
The purpose of the Braitenberg controller was to show the basic underlying control principle here. Instead of improving the controller performance by iteratively adjusting the network weights, the network was re-implemented using R-STDP synapses such that the weights could be automatically learned by the robot. In the previous section it was already shown that those learned weights are resembling the theoretically derived weights of the Braitenberg vehicle controller. Training the controller takes much less time than with the DQN algorithm. Moreover, off all 4 controllers the R-STDP controller shows the best performance in this task with comparatively very small deviations from the lane-center. This gets even clearer when looking at the performance histogram and the mean error that is almost an order of magnitude lower in comparison to the other controllers. First, one explanation for this behavior can be found in the very nature of SNNs that allow for high frequency decision making without the need of splitting time into discrete steps. Second, the R-STDP training algorithm and the related reward are to a great extent tailored to this specific problem. The great success of deep reinforcement learning methods such as DQN lie in their capability to learn value functions in high-dimensional state spaces. This property allows for a general algorithm that is capable of solving sequential decision making tasks formulated as MDP, even if rewards are sparse and delayed in time. The R-STDP controller, on the other hand, does not have this property. Basically, the R-STDP reward can be interpreted as a pre-defined value function with a global maximum that

the algorithm will seek. Furthermore, the reward signal incorporates prior knowledge, e.g. that in- or decreasing motor neuron activities will lead the robot back to the center. Therefore, the R-STDP training algorithm solves a mathematically much less complex problem leaving out the state evaluation step estimating future rewards that is crucial for every classical reinforcement learning algorithm solving MDPs with sparse, delayed rewards.

6. Conclusion & Outlook

Building energy-efficient mobile robots using event-based SNNs requires suitable training algorithms. In this work, two different approaches were presented both with the same goal of training SNNs for robot control in reinforcement learning tasks.
First, the problem has been formulated as a MDP and solved using the DQN algorithm. In the first two scenarios, the algorithm found a successful control strategy on different road patterns. In the third scenario, however, it did not find a stable policy, even after training for more than 1000 episodes. While learning usually takes a long time, the strength of the algorithm lies in its capability of solving general MDP problems. Although the mean error of the DQN controller in this work is comparatively high in contrast to the R-STDP controller, there is still a lot of room for improving its performance. First, a reward that is steeper at the lane-center could lead the robot to following a path closer to the center. Second, the action network architecture used in this work can be further extended, e.g. using multiple convolutional layers, leading to more accurate state evaluations and possibly a more capable controller that can learn a generalized policy in more complex scenarios such as scenario 3 as well. And third, instead of defining discrete actions, the algorithm by Lillicrap et al. [3] could be implemented using a continuous action space. For a car control task, this seems like a more suitable paradigm.
In order to build a controller that can handle the DVS events more naturally without any preprocessing, the pre-learned policy from the DQN algorithm was transferred to a SNN controller by training it on an artificial state-action dataset using supervised learning. In the first scenario, the DQN-SNN controller performed surprisingly well, even with a lower mean-error than the original DQN controller. If the information flow though the network is not high enough, however, output will be generated sparsely leading to high response latencies and performance losses. This was shown in the second scenario. A solution to this problem might be using higher spike-rates for the Poisson neurons that feed the SNN or lower thresholds for the membrane potentials. In general, this indirect training procedure is cumbersome, storing all the states might not always be possible and it introduces losses when transferring the policy. A better way of integrating classical reinforcement learning methods with SNNs could be given by training both network types in a single algorithm where both strengths and weaknesses can complement each other.

Luckily, reinforcement learning tasks can be divided into a separate state evaluation and a control part. Using such an actor-critic architecture, e.g. as shown in Lillicrap et al. [3], accurate state evaluations from conventional ANNs could be used for training an actor SNN.

Lastly, the problem was approached from the opposite direction. With the R-STDP learning rule that seeks to bring reinforcement learning capabilities to SNNs directly, a controller was implemented and tested in different scenarios with very promising results. First, in contrast to the DQN algorithm the controller learned a strategy in an order of magnitude less time. Second, in comparison with all other controllers it showed by far the best performance in all tested scenarios. Hereby, the algorithm is capable of learning to follow different road patterns, even if they are changing within a single scenario. In a single layer network, this will work as long as the evolving weight patterns won't overlap and each input neuron can be attributed to a distinct action. If in a more complex scenario, e.g. a road with junctions, the robot has to decide between going left or right based on more subtle hints and non-linear input relations, this basic network will fail. Having said this, comparing the R-STDP controller to the DQN algorithm is obviously flawed, because they do not solve exactly the same problem. While the R-STDP algorithm basically follows a minimization procedure on a pre-defined value function using prior knowledge, the DQN algorithm solves the reward prediction problem as well which allows for general MDP problem solving capabilities. Therefore, the R-STDP learning rule does not really qualify as reinforcement learning algorithm in that sense.

However, the R-STDP controller is build as first step towards more sophisticated algorithms with real reinforcement learning capabilities. It is scalable, such that the network could be implemented using deep architectures in the future. This will require capable hardware to be simulated on as well. In Section 2.2.3.3 evidence from neuro-scientific studies was discussed that points to some similarities between the dopaminergic neural implementation in the brain and TD learning. While the R-STDP algorithm is already able to handle delayed rewards using an eligibility trace that is very similar to standard reinforcement learning algorithms such as SARSA(λ) (Sec. 2.1.2), it does not incorporate reward prediction errors yet even though this was observed in the brain. Evans [94] have shown a robot controlled by a SNN that is trained using R-STDP synapses as well. Instead of directly setting the level of dopamine during learning, a second network was implemented evaluating states and setting the dopamine level accordingly. Chorley and Seth [139] showed how reward prediction errors could be implemented in SNNs using a model of short-term memory. In the future, combining the R-STDP algorithm with a reward prediction model could lead to an algorithm that is actually capable of solving sequential decision tasks such as MDPs as well.

A. Simulation Parameters

Table A.1.: DQN parameters

DQN	network architecture	512 - 200 - 200 - 3
	connections	feed-forward, fully connected
	batch size	32
	update frequency	4
	soft update	$\tau = 0.001$
	learning rate	$\alpha = 0.0001$
	buffer size	5000
ϵ-greedy policy	pre-training steps	1000
	annealing steps	49000
	random probability start	1.0
	random probability end	0.1
MDP	discount factor	$\gamma = 0.99$
	reset distance	0.5 m
	maximum episode steps	1000
	time step length	$0.5\,s$
Robot	motor speed straight	$v_s = 1.0\,m/s$
	motor speed turn	$v_t = 0.25\,m/s$

Table A.2.: DQN-SNN parameters

ANN training	network architecture	512 - 200 - 3 (bias only in first layer)
	connections	feed-forward, fully connected
	batch size	50
	training steps	10000
	Optimizer	ADAM
	learning rate	0.0001
SNN simulation	simulation time	10 ms
	max. firing rate	1000 Hz
	simulation step length	1 ms
	membrane potential threshold	1 mV

Table A.3.: Braitenberg parameters

Steering wheel model	max. speed	$v_{max} = 1.5\,m/s$
	min. speed	$v_{min} = 1.0\,m/s$
	turn constant	$c_{turn} = 0.5$
	max spikes during a simulation step	$n_{max} = 25$
Poisson neurons	max. firing rate	500 Hz
	number of DVS events for max. firing rate	$n = 10$
SNN simulation	simulation time	50 ms
	time resolution	0.1 ms
LIF neurons	NEST model	iaf_psc_alpha
	Resting membrane potential	$E_L = -70.0\,mV$
	Capacity of the membrane	$C_m = 250.0\,pF$
	Membrane time constant	$\tau_m = 10.0\,ms$
	Time constant of postsynaptic excitatory currents	$\tau_{syn,ex} = 2.0\,ms$
	Time constant of postsynaptic inhibitory currents	$\tau_{syn,in} = 2.0\,ms$
	Duration of refractory period	$t_{ref} = 2.0\,ms$
	Reset membrane potential	$V_{reset} = -70.0\,mV$
	Spike threshold	$V_{th} = -55.0\,mV$
	Constant input current	$I_e = 0.0\,pA$

Table A.4.: R-STDP parameters

Steering wheel model	max. speed	$v_{max} = 1.5\,m/s$
	min. speed	$v_{min} = 1.0\,m/s$
	turn constant	$c_{turn} = 0.5$
	max spikes during a simulation step	$n_{max} = 15$
Poisson neurons	max. firing rate	300 Hz
	number of DVS events for max. firing rate	$n = 15$
SNN simulation	simulation time	50 ms
	time resolution	0.1 ms
LIF neurons	NEST model	iaf_psc_alpha
	Resting membrane potential	$E_L = -70.0\,mV$
	Capacity of the membrane	$C_m = 250.0\,pF$
	Membrane time constant	$\tau_m = 10.0\,ms$
	Time constant of postsynaptic excitatory currents	$\tau_{syn,ex} = 2.0\,ms$
	Time constant of postsynaptic inhibitory currents	$\tau_{syn,in} = 2.0\,ms$
	Duration of refractory period	$t_{ref} = 2.0\,ms$
	Reset membrane potential	$V_{reset} = -70.0\,mV$
	Spike threshold	$V_{th} = -55.0\,mV$
	Constant input current	$I_e = 0.0\,pA$
R-STDP synapse	NEST model	stdp_dopamine_synapse
	Amplitude of weight change for facilitation	$A_+ = 1.0$
	Amplitude of weight change for depression	$A_- = 1.0$
	STDP time constant for facilitation	$\tau_+ = 20.0\,ms$
	Time constant of eligibility trace	$\tau_c = 1000.0\,ms$
	Time constant of dopaminergic trace	$\tau_n = 200.0\,ms$
	Minimal synaptic weight	0.0
	Maximal synaptic weight	3000.0
	Initial synaptic weight	200.0
	Reward constant	$c_r = 0.01$

B. Additional Training Details

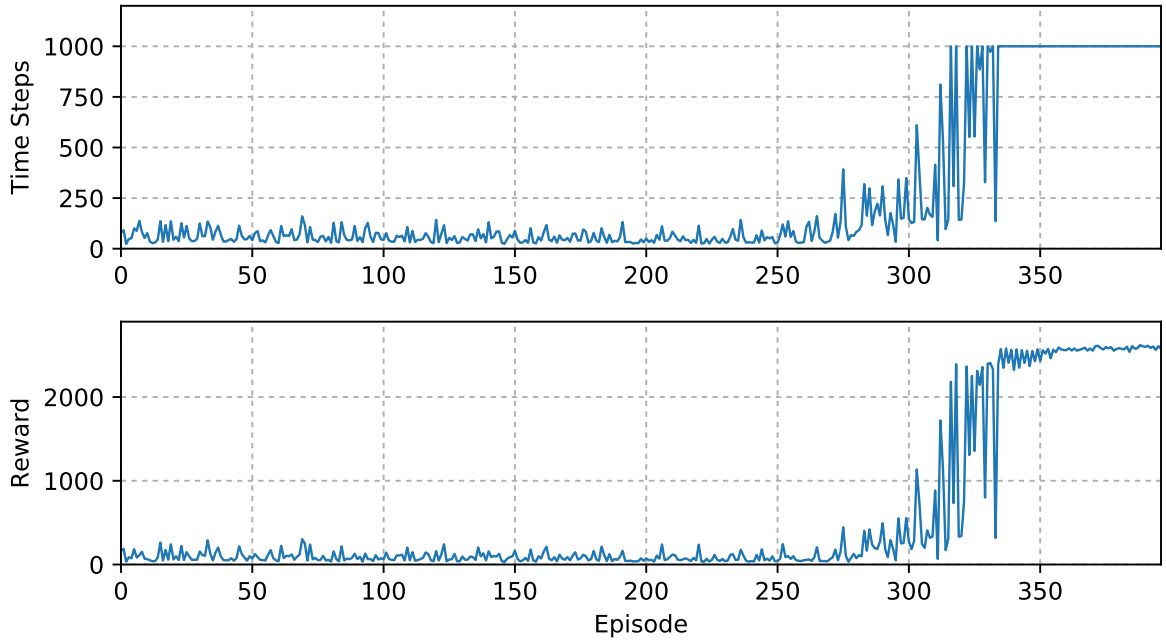


Figure B.1.: Scenario 2: Time steps and rewards for each training episode of the DQN controller.

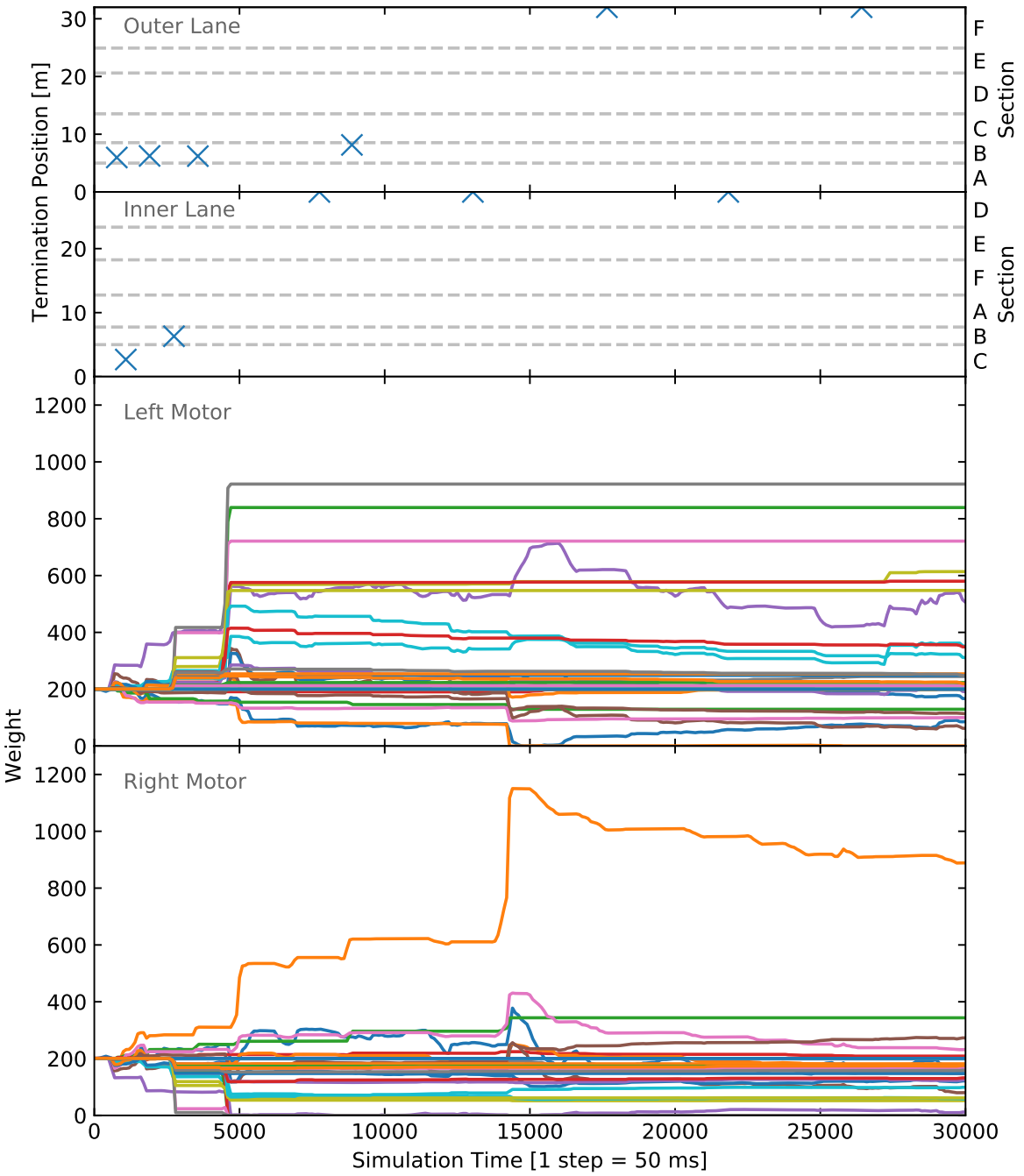


Figure B.2.: Scenario 2: Learning progress of the R-STDP controller. The termination position when the robot causes a reset and the network weights are shown over the number of simulation steps ($1\,step = 50\,ms$).

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[3] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[4] Daniel Drubach. *The brain explained*. Prentice Hall, 2000.

[5] Steve B Furber, Francesco Galluppi, Steve Temple, and Luis A Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, 2014.

[6] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128 $\times$ 128 120 db 15 μs latency asynchronous temporal contrast vision sensor. *IEEE journal of solid-state circuits*, 43(2):566–576, 2008.

[7] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement learning using a continuous time actor-critic framework with spiking neurons. *PLoS Comput Biol*, 9(4):e1003024, 2013.

[8] Sen Song, Kenneth D Miller, and Larry F Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9): 919–926, 2000.

[9] Peter U Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in computational neuroscience*, 9, 2015.

[10] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training Deep Spiking Neural Networks using Backpropagation. 10(November):1–10, 2016. ISSN 1662-453X. doi: 10.3389/fnins.2016.00508. URL `http://arxiv.org/abs/1608.08782`.

[11] Yoshua Bengio, Thomas Mesnard, Asja Fischer, Saizheng Zhang, and Yuhuai Wu. Stdp-compatible approximation of backpropagation in an energy-based model. *Neural computation*, 2017.

[12] Wolfram Schultz. Neuronal reward and decision signals: from theories to data. pages 853–951, 2015. doi: 10.1152/physrev.00023.2014.

[13] Robert Legenstein, Dejan Pecevski, and Wolfgang Maass. A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. *PLoS Computational Biology*, 4(10), 2008. ISSN 1553734X. doi: 10.1371/journal.pcbi.1000180.

[14] Răzvan V Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–1502, 2007.

[15] Jacques Kaiser, J Camilo Vasquez Tieck, Christian Hubschneider, Peter Wolf, Michael Weber, Michael Hoff, Alexander Friedrich, Konrad Wojtasik, Arne Roennau, Ralf Kohlhaas, et al. Towards a framework for end-to-end control of a simulated vehicle with spiking neural networks. In *Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), IEEE International Conference on*, pages 127–134. IEEE, 2016.

[16] Valentino Braitenberg. *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.

[17] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[18] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[19] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[20] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1321–1326. IEEE, 2013.

[21] Peter U Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through

weight and threshold balancing. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE, 2015.

[22] Wiebke Potjans, Abigail Morrison, and Markus Diesmann. Enabling functional neural circuit simulations with distributed computing of neuromodulated plasticity. *Spike-timing dependent plasticity*, page 357, 2010.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 1998. ISBN 0262193981.

[24] Filip Ponulak and Andrzej Kasinski. Introduction to spiking neural networks: Information processing, learning and applications. *Acta neurobiologiae experimentalis*, 71(4):409–33, 2011. ISSN 1689-0035. URL `http://www.ncbi.nlm.nih.gov/pubmed/22237491`.

[25] Jilles Vreeken. Spiking neural networks, an introduction. pages 1–5.

[26] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[28] Suzana Herculano-Houzel. The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. *Proceedings of the National Academy of Sciences*, 109(Supplement 1):10661–10668, 2012.

[29] Wolfgang Maass. Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8(1):32–36, 2002.

[30] Neuron - wikipedia. `http://en.wikipedia.org/wiki/Neuron`, . Accessed: 2017-01-30.

[31] Synapse - wikipedia. `http://en.wikipedia.org/wiki/Synapse`, . Accessed: 2017-01-30.

[32] David PM Northmore and John G Elias. Building silicon nervous systems with dendritic tree neuromorphs. *Pulsed neural networks*, pages 135–156, 1998.

[33] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[34] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8): 2554–2558, 1982.

[35] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[36] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

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

[38] Bhaskar DasGupta and Georg Schnitger. The power of approximating: a comparison of activation functions. In *NIPS*, pages 615–622, 1992.

[39] Marvin M Chun and Mary C Potter. A two-stage model for multiple target detection in rapid serial visual presentation. *Journal of Experimental psychology: Human perception and performance*, 21(1):109, 1995.

[40] Donald Olding Hebb. *The organization of behavior: A neuropsychological approach*. John Wiley & Sons, 1949.

[41] Geoffrey E Hinton and Terrence Joseph Sejnowski. *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

[42] Henry Markram, Joachim Lübke, Michael Frotscher, and Bert Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 275(5297): 213–215, 1997.

[43] Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of neuroscience*, 18(24):10464–10472, 1998.

[44] Claudia Clopath, Lars Büsing, Eleni Vasilaki, and Wulfram Gerstner. Connectivity reflects coding: a model of voltage-based stdp with homeostasis. *Nature neuroscience*, 13(3):344–352, 2010.

[45] Wulfram Gerstner and Werner M Kistler. Mathematical formulations of hebbian learning. *Biological cybernetics*, 87(5-6):404–415, 2002.

[46] Yann LeCun, Corinna Cortes, and Christopher JC Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[47] WT Thach. On the specific role of the cerebellum in motor learning and cognition: Clues from pet activation and lesion studies in man. *Behavioral and Brain Sciences*, 1996.

[48] John Montgomery, Guy Carton, and David Bodznick. Error-driven motor learning in fish. *The Biological Bulletin*, 203(2):238–239, 2002.

[49] R Christopher Miall and Daniel M Wolpert. Forward models for physiological motor control. *Neural networks*, 9(8):1265–1279, 1996.

[50] Yves Fregnac and Daniel E Shulz. Activity-dependent regulation of receptive field properties of cat area 17 by supervised hebbian learning. *Journal of neurobiology*, 41(1):69–82, 1999.

[51] Filip Ponulak. Resume-new supervised learning method for spiking neural networks. *Institute of Control and Information Engineering, Poznan University of Technology.(Available online at: http://d1. cie. put. poznan. pl/˜ fp/research. html)*, 2005.

[52] Mitsuo Kawato and Hiroaki Gomi. A computational model of four regions of the cerebellum based on feedback-error learning. *Biological cybernetics*, 68(2):95–103, 1992.

[53] Megan R Carey, Javier F Medina, and Stephen G Lisberger. Instructive signals for motor learning from visual cortical area mt. *Nature neuroscience*, 8(6):813–819, 2005.

[54] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015.

[55] Dopamine - wikipedia. `https://en.wikipedia.org/wiki/Dopamine`, . Accessed: 2017-07-05.

[56] Ikuko Miyazaki and Masato Asanuma. Dopaminergic neuron-specific oxidative stress caused by dopamine itself. *Acta medica Okayama*, 62(3):141–150, 2008.

[57] V Srinivasa Chakravarthy, Denny Joseph, and Raju S Bapi. What do the basal ganglia do? a modeling perspective. *Biological cybernetics*, 103(3):237–253, 2010.

[58] Hond van pavlov - wikipedia. `https://nl.wikipedia.org/wiki/Hond_van_Pavlov`, . Accessed: 2017-07-05.

[59] Ivan Petrovich Pavlov and Gleb Vasīlevīch Anrep. *Conditioned reflexes*. Courier Corporation, 2003.

[60] Edward Lee Thorndike. Review of animal intelligence: An experimental study of the associative processes in animals. 1898.

[61] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[62] Kazuki Enomoto, Naoyuki Matsumoto, Sadamu Nakai, Takemasa Satoh, Tatsuo K Sato, Yasumasa Ueda, Hitoshi Inokawa, Masahiko Haruno, and Minoru Kimura.

Dopamine neurons learn to encode the long-term value of multiple future rewards. *Proceedings of the National Academy of Sciences*, 108(37):15462–15467, 2011.

[63] Masayuki Matsumoto and Okihide Hikosaka. Two types of dopamine neuron distinctly convey positive and negative motivational signals. *Nature*, 459(7248): 837–841, 2009.

[64] Mathias Pessiglione, Ben Seymour, Guillaume Flandin, Raymond J Dolan, and Chris D Frith. Dopamine-dependent prediction errors underpin reward-seeking behaviour in humans. *Nature*, 442(7106):1042–1045, 2006.

[65] Verena Pawlak and Jason ND Kerr. Dopamine receptor activation is required for corticostriatal spike-timing-dependent plasticity. *Journal of Neuroscience*, 28(10): 2435–2446, 2008.

[66] Ji-Chuan Zhang, Pak-Ming Lau, and Guo-Qiang Bi. Gain in sensitivity and loss in temporal contrast of stdp by dopaminergic modulation at hippocampal synapses. *Proceedings of the National Academy of Sciences*, 106(31):13028–13033, 2009.

[67] Verena Pawlak, Jeffery R Wickens, Alfredo Kirkwood, and Jason ND Kerr. Timing is not everything: neuromodulation opens the stdp gate. *Spike-timing dependent plasticity*, page 138, 2010.

[68] Eugene M Izhikevich. Solving the distal reward problem through linkage of stdp and dopamine signaling. *Cerebral cortex*, 17(10):2443–2452, 2007.

[69] Wiebke Potjans, Abigail Morrison, and Markus Diesmann. A spiking neural network model of an actor-critic learning agent. *Neural computation*, 21(2):301–339, 2009.

[70] Apostolos P Georgopoulos, Ronald E Kettner, and Andrew B Schwartz. Primate motor cortex and free arm movements to visual targets in three-dimensional space. ii. coding of the direction of movement by a neuronal population. *Journal of Neuroscience*, 8(8):2928–2937, 1988.

[71] JM Valeton. Photoreceptor light adaptation models: an evaluation. *Vision research*, 23(12):1549–1554, 1983.

[72] Henrik Rehbinder and Bijoy K Ghosh. Pose estimation using line-based dynamic vision and inertial sensors. *IEEE Transactions on Automatic Control*, 48(2): 186–199, 2003.

[73] Jörg Conradt, Matthew Cook, Raphael Berner, Patrick Lichtsteiner, Rodney J Douglas, and T Delbruck. A pencil balancing robot using a pair of aer dynamic vision sensors. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 781–784. IEEE, 2009.

[74] Giacomo Indiveri. Neuromorphic analog vlsi sensor for visual tracking: Circuits and application examples. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(11):1337–1347, 1999.

[75] M Anthony Lewis, Ralph Etienne-Cummings, Avis H Cohen, and Mitra Hartmann. Toward biomorphic control using custom avlsi cpg chips. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 494–500. IEEE, 2000.

[76] Alessandro Ambrosano, Lorenzo Vannucci, Ugo Albanese, Murat Kirtay, Egidio Falotico, Georg Hinkel, Jacques Kaiser, Stefan Ulbrich, Paul Levi, Christian Morillas, et al. Retina color-opponency based pursuit implemented through spiking neural networks in the neurorobotics platform. In *Conference on Biomimetic and Biohybrid Systems*, pages 16–27. Springer, 2016.

[77] Xiuqing Wang, Zeng-Guang Hou, Min Tan, Yongji Wang, and Liwei Hu. The wall-following controller for the mobile robot using spiking neurons. In *Artificial Intelligence and Computational Intelligence, 2009. AICI'09. International Conference on*, volume 1, pages 194–199. IEEE, 2009.

[78] Richard R Carrillo, Eduardo Ros, Christian Boucheny, and J-MD Coenen Olivier. A real-time spiking cerebellum model for learning robot control. *Biosystems*, 94(1): 18–27, 2008.

[79] Alexandros Bouganis and Murray Shanahan. Training a spiking neural network to control a 4-dof robotic arm based on spike timing-dependent plasticity. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

[80] Paolo Arena, Luigi Fortuna, Mattia Frasca, and Luca Patané. Learning anticipation via spiking networks: application to navigation control. *IEEE transactions on neural networks*, 20(2):202–216, 2009.

[81] Paolo Arena, Sebastiano De Fiore, Luca Patané, Massimo Pollino, and Cristina Ventura. Insect inspired unsupervised learning for tactic and phobic behavior enhancement in a hybrid robot. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

[82] P Arena, S De Fiore, L Patané, M Pollino, and C Ventura. Stdp-based behavior learning on the tribot robot. In *SPIE Europe Microtechnologies for the New Millennium*, pages 736506–736506. International Society for Optics and Photonics, 2009.

[83] André Cyr and Mounir Boukadoum. Classical conditioning in different temporal

constraints: an stdp learning rule for robots controlled by spiking neural networks. *Adaptive Behavior*, 20(4):257–272, 2012.

[84] Xiuqing Wang, Zeng-Guang Hou, Anmin Zou, Min Tan, and Long Cheng. A behavior controller based on spiking neural networks for mobile robots. *Neurocomputing*, 71(4):655–666, 2008.

[85] Xiuqing Wang, Zeng-Guang Hou, Feng Lv, Min Tan, and Yongji Wang. Mobile robots modular navigation controller using spiking neural networks. *Neurocomputing*, 134:230–238, 2014.

[86] Kenji Iwadate, Ikuo Suzuki, Michiko Watanabe, Masahito Yamamoto, and Masashi Furukawa. An artificial neural network based on the architecture of the cerebellum for behavior learning. In *Soft Computing in Artificial Intelligence*, pages 143–151. Springer, 2014.

[87] Cristian Jimenez-Romero, David Sousa-Rodrigues, Jeffrey H Johnson, and Vitorino Ramos. A model for foraging ants, controlled by spiking neural networks and double pheromones. *arXiv preprint arXiv:1507.08467*, 2015.

[88] Cristian Jimenez-Romero. *A Heterosynaptic Spiking Neural System for the Development of Autonomous Agents*. PhD thesis, The Open University, 2017.

[89] Cristian Jimenez-Romero, David Sousa-Rodrigues, and Jeffrey Johnson. Designing behaviour in bio-inspired robots using associative topologies of spiking-neural-networks. In *proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS) on 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, pages 197–200. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[90] André Cyr, Mounir Boukadoum, and Frédéric Thériault. Operant conditioning: a minimal components requirement in artificial spiking neurons designed for bio-inspired robot's controller. *Frontiers in neurorobotics*, 8, 2014.

[91] André Cyr and Frédéric Thériault. Action selection and operant conditioning: a neurorobotic implementation. *Journal of Robotics*, 2015:6, 2015.

[92] Etienne Dumesnil, Philippe-Olivier Beaulieu, and Mounir Boukadoum. Robotic implementation of classical and operant conditioning as a single stdp learning process. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 5241–5247. IEEE, 2016.

[93] Etienne Dumesnil, Philippe-Olivier Beaulieu, and Mounir Boukadoum. Robotic

implementation of classical and operant conditioning within a single snn architecture. In *Cognitive Informatics & Cognitive Computing (ICCI* CC), 2016 IEEE 15th International Conference on*, pages 322–330. IEEE, 2016.

[94] Richard Evans. Reinforcement learning in a neurally controlled robot using dopamine modulated stdp. *arXiv preprint arXiv:1502.06096*, 2012.

[95] Lovisa Irpa Helgadottir, Joachim Haenicke, Tim Landgraf, Raul Rojas, and Martin P Nawrot. Conditioned behavior in a robot controlled by a spiking neural network. In *Neural Engineering (NER), 2013 6th International IEEE/EMBS Conference on*, pages 891–894. IEEE, 2013.

[96] Steven Skorheim, Peter Lonjers, and Maxim Bazhenov. A spiking network model of decision making employing rewarded stdp. *PloS one*, 9(3):e90821, 2014.

[97] Faramarz Faghihi, Ahmed A Moustafa, Ralf Heinrich, and Florentin Wörgötter. A computational model of conditioning inspired by drosophila olfactory system. *Neural Networks*, 87:96–108, 2017.

[98] Taylor S Clawson, Silvia Ferrari, Sawyer B Fuller, and Robert J Wood. Spiking neural network (snn) control of a flapping insect-scale robot. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pages 3381–3388. IEEE, 2016.

[99] Greg Foderaro, Craig Henriquez, and Silvia Ferrari. Indirect training of a spiking neural network for flight control via spike-timing-dependent synaptic plasticity. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 911–917. IEEE, 2010.

[100] Xu Zhang, Greg Foderaro, C Henriquez, Antonius MJ VanDongen, and Silvia Ferrari. A radial basis function spike model for indirect learning via integrate-and-fire sampling and reconstruction techniques. *Advances in Artificial Neural Systems*, 2012:10, 2012.

[101] Xu Zhang, Ziye Xu, Craig Henriquez, and Silvia Ferrari. Spike-based indirect training of a spiking neural network-controlled virtual insect. In *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*, pages 6798–6805. IEEE, 2013.

[102] Di Hu, Xu Zhang, Ziye Xu, Silvia Ferrari, and Pinaki Mazumder. Digital implementation of a spiking neural network (snn) capable of spike-timing-dependent plasticity (stdp) learning. In *Nanotechnology (IEEE-NANO), 2014 IEEE 14th International Conference on*, pages 873–876. IEEE, 2014.

[103] Pinaki Mazumder, D Hu, I Ebong, X Zhang, Z Xu, and Silvia Ferrari. Digital

implementation of a virtual insect trained by spike-timing dependent plasticity. *Integration, the VLSI Journal*, 54:109–117, 2016.

[104] George L Chadderdon, Samuel A Neymotin, Cliff C Kerr, and William W Lytton. Reinforcement learning of targeted movement in a spiking neuronal model of motor cortex. *PloS one*, 7(10):e47251, 2012.

[105] Samuel A Neymotin, George L Chadderdon, Cliff C Kerr, Joseph T Francis, and William W Lytton. Reinforcement learning of two-joint virtual arm reaching in a computer model of sensorimotor cortex. *Neural computation*, 25(12):3263–3293, 2013.

[106] Martin Spüler, Sebastian Nagel, and Wolfgang Rosenstiel. A spiking neuronal model learning a motor control task by reinforcement learning and structural synaptic plasticity. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE, 2015.

[107] Salvador Dura-Bernal, Xianlian Zhou, Samuel A Neymotin, Andrzej Przekwas, Joseph T Francis, and William W Lytton. Cortical spiking network interfaced with virtual musculoskeletal arm and robotic arm. *Frontiers in neurorobotics*, 9:13, 2015.

[108] Mehmet Kocaturk, Halil Ozcan Gulcur, and Resit Canbeyli. Toward building hybrid biological/in silico neural networks for motor neuroprosthetic control. *Frontiers in neurorobotics*, 9:8, 2015.

[109] Mohammad Sarim, Thomas Schultz, Rashmi Jha, and Manish Kumar. Ultra-low energy neuromorphic device based navigation approach for biomimetic robots. In *Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS), 2016 IEEE National*, pages 241–247. IEEE, 2016.

[110] Mohammad Sarim, Thomas Schultz, Manish Kumar, and Rashmi Jha. An artificial brain mechanism to develop a learning paradigm for robot navigation.

[111] Ting-Shuo Chou, Liam D Bucci, and Jeffrey L Krichmar. Learning touch preferences with a tactile robot using dopamine modulated stdp in a model of insular cortex. *Frontiers in neurorobotics*, 9:6, 2015.

[112] Eric Nichols, Liam J McDaid, and Nazmul Siddique. Biologically inspired snn for robot control. *IEEE transactions on cybernetics*, 43(1):115–128, 2013.

[113] Elmar Rueckert, David Kappel, Daniel Tanneberg, Dejan Pecevski, and Jan Peters. Recurrent spiking networks solve planning tasks. *Scientific reports*, 6, 2016.

[114] Johannes Friedrich and Máté Lengyel. Goal-directed decision making with spiking neurons. *Journal of Neuroscience*, 36(5):1529–1546, 2016.

[115] Zbigniew Michalewicz. Gas: What are they? In *Genetic algorithms+ data structures= evolution programs*, pages 13–30. Springer, 1994.

[116] Dario Floreano and Claudio Mattiussi. Evolution of spiking neural controllers for autonomous vision-based robots. In *International Symposium on Evolutionary Robotics*, pages 38–61. Springer, 2001.

[117] Dario Floreano, Yann Epars, Jean-Christophe Zufferey, and Claudio Mattiussi. Evolution of spiking neural circuits in autonomous mobile robots. *International Journal of Intelligent Systems*, 21(9):1005–1024, 2006.

[118] Hani Hagras, Anthony Pounds-Cornish, Martin Colley, Victor Callaghan, and Graham Clarke. Evolving spiking neural network controllers for autonomous robots. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4620–4626. IEEE, 2004.

[119] David Howard and Alberto Elfes. Evolving spiking networks for turbulence-tolerant quadrotor control. 2014.

[120] R Batllori, Craig B Laramee, W Land, and J David Schaffer. Evolving spiking neural networks for robot control. *Procedia Computer Science*, 6:329–334, 2011.

[121] Urszula Markowska-Kaczmar and Mateusz Koldowski. Spiking neural network vs multilayer perceptron: who is the winner in the racing car computer game. *Soft Computing*, 19(12):3465–3478, 2015.

[122] Fady Alnajjar and Kazuyuki Murase. Self-organization of spiking neural network generating autonomous behavior in a miniature mobile robot. In *Proceedings of the 3rd International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, pages 255–260. Springer, 2006.

[123] Naoyuki Kubota. A spiking neural network for behavior learning of a mobile robot in a dynamic environment. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 6, pages 5783–5788. IEEE, 2004.

[124] Harald Burgsteiner. Training networks of biological realistic spiking neurons for real-time robot control. In *Proceedings of the 9th international conference on engineering applications of neural networks, Lille, France*, pages 129–136, 2005.

[125] Dimitri Probst, Wolfgang Maass, Henry Markram, and Marc-Oliver Gewaltig. Liquid computing in a simplified model of cortical layer iv: Learning to balance a ball. *Artificial Neural Networks and Machine Learning–ICANN 2012*, pages 209–216, 2012.

[126] Eleonora Arena, Paolo Arena, Roland Strauss, and Luca Patané. Motor-skill

learning in an insect inspired neuro-computational control system. *Frontiers in Neurorobotics*, 11, 2017.

[127] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. Lua 5.1 reference manual, 2006.

[128] Adept MobileRobots. Pioneer p3-dx. *Website. http://www. mobilerobots. c om/ResearchRobots/PioneerP3DX. aspx*, 2012.

[129] Elias Mueggler, Basil Huber, and Davide Scaramuzza. Event-based, 6-dof pose tracking for high-speed maneuvers. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 2761–2768. IEEE, 2014.

[130] Garrick Orchard, Ajinkya Jayawant, Gregory Cohen, and Nitish Thakor. Converting static image datasets to spiking neuromorphic datasets using saccades. *arXiv preprint arXiv:1507.07629*, 2015.

[131] Teresa Serrano-Gotarredona and Bernabé Linares-Barranco. Poker-dvs and mnist-dvs. their history, how they were made, and other details. *Frontiers in neuroscience*, 9:481, 2015.

[132] inilabs. URL `https://inilabs.com/`. Accessed: 2017-06-20.

[133] Address-event representation (aer) protocol, Jan 2017. URL `https://inilabs.com/support/hardware/aer/`. Accessed: 2017-06-20.

[134] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[135] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[136] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[137] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[138] Charles F Stevens and Anthony M Zador. When is an integrate-and-fire neuron like a poisson neuron? In *Advances in neural information processing systems*, pages 103–109, 1996.

[139] Paul Chorley and Anil K Seth. Dopamine-signaled reward predictions generated by

competitive excitation and inhibition in a spiking neural network model. *Frontiers in computational neuroscience*, 5, 2011.