DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Deep Spiking Q Networks

Christoph Hahn

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Deep Spiking Q Networks

Author:	Christoph Hahn
Supervisor:	Prof. Dr.-Ing. Alois Christian Knoll
Advisor:	Mahmoud Akl
Submission Date:	15.01.2020

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

Place, Date	Christoph Hahn

Acknowledgments

I sincerely thank my advisor Mahmoud Akl for overseeing this thesis. Thanks especially for the many intriguing discussions, I've definitely learned a lot during research and writing, and I hope that you could learn something from it as well. Further, I want to thank Prof. Dr.-Ing. Alois Christian Knoll for officially supervising the thesis. I'm also much obliged to Fabian Batsch and Sebastian Schmidt for proof-reading and to all of my family and friends for always supporting me. Last and definitely least, I want to appreciate the coffee machines throughout TUM for getting me through the chewy parts. So long, and thanks for all the coffee.

Abstract

Spiking neural networks are the upcoming third generation of neural networks. Due to the binary and sparse nature of their communication, they promise energy efficiency, inference in real-time, and a deeper understanding of the human brain.
Recent literature shows that competitive spiking networks can be trained on supervised and unsupervised learning tasks, while only little research exists in the reinforcement learning domain.
Deep reinforcement learning, heralded by the emergence of *Deep Q Networks (DQNs)*, has gained great attention in recent years by solving unprecedented challenges including beating the Go world champion and solving complex motion control tasks.
In this thesis, we attempt to bridge the gap between various existing training methods for deep spiking networks in the supervised learning domain and using spiking neurons in deep reinforcement learning.
In order to achieve this, we theoretically analyze the existing training methods and practically implement different conversion techniques of conventional neural networks and Backpropagation using surrogate gradients on diverse *Open AI Gym* environments. The reinforcement learning algorithms we employ are mainly Deep Q Learning and additionally, *Policy Gradient* methods. For conversion we compare encoding, decoding, normalization, and resetting models as well as direct conversion approaches and indirect approaches, where a classification network learns the policy of a DQN prior to conversion.
Our results show that all tested training methods for spiking neural networks are able to successfully solve the *CartPole* and *MountainCar* problems. Further, we apply them to the more complex *Breakout* environment. In brief, conversion based methods are faster to train, while directly trained methods provide shorter inference time and generalize better to unseen data. Training a spiking network on the policy of a conventional DQN in a supervised manner provides a trade-off between the two methods. We find that direct DQN conversion can require long simulation times on certain problems and that indirect conversion can improve on this, but comes with its own flaws.
Additionally, we transfer our methods to a complex neuron simulator (*NEST*) and to the neuromorphic hardware chip *SpiNNaker*, and outline the challenges of adapting the algorithms to them.
Although we successfully demonstrate the combination of deep reinforcement learning with spiking neural networks, their intersection is a field which only recently emerged. Therefore, many intriguing research questions remain that are not addressed by this thesis. We conclude with an outlook over this unexplored research.

Contents

Abbreviations

ACC (Conversion) Accuracy.

ANN Artificial (Conventional) Neural Network.

AVG Average (Performance).

BP Backpropagation.

CPU Central Processing Unit.

DQN Deep Q Network.

DSQN Deep Spiking Q Network.

DVS Dynamic Vision Sensor.

eRBP event-based Random Backpropagation.

GPU Graphics Processing Unit.

LIF Leaky-Integrate-and-Fire.

MDP Markov Decision Process.

ms Milliseconds.

MSE Mean Squared Error.

NN Neural Network.

PG Policy Gradient.

R-STDP Reward-modulated Spike Timing Dependent Plasticity.

RBP Random Backpropagation.

RL Reinforcement Learning.

RNN Recurrent Neural Network.

SG Surrogate Gradients.

SNN Spiking Neural Network.

STD Standard Deviation (of Performance).

STDP Spike Timing Dependent Plasticity.

List of Figures

List of Tables

Chapter 1
Introduction

Mobile devices, including robots, become more and more common, not only in businesses but also in every day life. All these devices share two common problems, battery power and real-time computation. Especially robots need to solve complex computational problems, e.g. inverse kinematics to provide just one example, which require a lot of computational power and therefore time and energy. In this thesis we combine two methods, *Spiking Neural Networks (SNNs)* and *Reinforcement Learning (RL)*, which both promise to decrease energy usage and to deliver real-time computation, while at the same time offering to explore the inner workings of the human brain and to solve unprecedented problems.

SNNs are a type of neural network, but unlike in their traditional non-spiking counterpart, all communication happens through binary signals, called *spikes*, which are transmitted between neurons.
This type of computation is extremely energy-efficient, because it is *local*. Local means that neurons do not share any information beyond what is transmitted in the form of spikes. Moreover, neurons perform only basic mathematical operations, multiplication and addition. In fact, this efficiency should not come as a surprise, as SNNs are inspired by the mammalian brain which can solve problems still out of reach for modern computers, while barely requiring energy. The human brain operates on only around 20 Watts [56]. However, the energy efficiency can only be exploited, when using dedicated hardware, so called *neuromorphic hardware*.
Moreover, some authors [68] postulate that SNNs are more suited to solve certain problem classes than traditional networks. These include tasks, where the inputs are discrete events, and tasks, where the network has to learn a temporal pattern. SNNs have an intrinsic notion of time that allows them to discover temporal structures. In the extreme case, the network needs only one spike per neuron to do meaningful computations. If this can be achieved, the inference time and the energy consumption is minimal.
The challenge of SNNs lies in their training. Approaches include *conversion* from conventional neural networks, derivatives of the *Backpropagation* algorithm on conventional hardware and *local learning* rules to train them directly on neuromorphic hardware. If direct training on the hardware can be achieved, SNNs have the potential to decrease the ever rising amount of electricity use in computation by replacing energy hungry conventional networks.
Networks of spiking neurons are also of interest for theoretical neuroscience, where the most biologically realistic (and also most complex) models are used to simulate how neuron populations behave.

Reinforcement Learning (RL), alongside supervised and unsupervised learning, is one of the three main machine learning paradigms and can simply be described as an *agent* which interacts with an *environment*, from which it receives *inputs* and *rewards*, by taking *actions*. In the last years interest in RL has skyrocketed with the introduction of stable deep networks, most notably the *Deep Q Network (DQN)* [53]. Since then, various other deep learning algorithms have been designed to

perform stunning feats. Beating the Go world champion, playing computer games on the same level as professional human players, and solving complex robotics problems are just a few of them [45]. In robots, RL offers real-time, energy-efficient problem solving as traditional analytical or numerical algorithms like they are used in inverse kinematics are computationally difficult and therefore time and energy consuming if they can be solved at all. On the other hand, RL algorithms are only expensive to train, but then cheap in performing inference. [69]

In this thesis we investigate Reinforcement Learning using Deep Spiking Neural Networks. This field not only promises energy-efficient, real-time computation for robots and mobile devices, but also to deepen our understanding of the human brain which itself uses complex spiking neurons and trains them using reinforcement learning in the form of neuromodulators like dopamine [22]. Up to now, little research exists in this area, especially for deep networks. The most important approaches are transferring deep RL algorithms from conventional networks and designing completely new algorithms based on local learning rules. In the first approach, Patel et al. [66] recently converted a DQN, while in the second, one strategy is to train pools of neurons using *Spike Timing Dependent Plasticity (STDP)* to obtain an *actor-critic* architecture (see section 2.2.3) [22]. We follow the first approach and explore different training techniques to modify existing deep RL algorithms, mostly Deep Q Learning, but also *Policy Gradient* methods, such that they become applicable to spiking neural networks. The algorithms so far suggested in the second approach are not as general as traditional reinforcement learning algorithms and are analyzed in section 3.1.3.

To investigate the topic, this work is split into a theoretical and an experimental part. First, we explain the basics of deep and reinforcement learning. Then, we explain Spiking Neural Networks in detail, proposing a classification of training methods and elucidating some of them in detail. The third chapter reviews the papers most important to this thesis, focusing on the recent advancements in training SNNs. The experimental part describes experiments to test the training methods *Conversion* and *Backpropagation with Surrogate Gradients* on different environments from *Open AI Gym* [11]. Finally, we discuss our results and elaborate on possible future work. The thesis is supplemented with our code repository, available on Github [1]. Its structure is explained in Appendix A.

[1] https://github.com/vhris/Deep-Spiking-Q-Networks.git

Chapter 2
Theoretical Background

The first part of this chapter summarizes the basics of deep and reinforcement learning, including the algorithms deep Q learning, policy gradient and actor-critic, and introduces the Open AI Gym [11] framework. Then, spiking neural networks are investigated in depth. After explaining their basic concept, various neuron models, neuromorphic hardware as well as en- and decoding methods, our main focus lies on training techniques. We first suggest a classification for SNN training methods and then elaborate on conversion, Backpropagation with surrogate gradients, STDP and event-based random Backpropagation.

2.1 Deep Learning

First, we provide a short overview over deep learning. We assume that the reader has previous knowledge on the topic, so we limit this section to the concepts crucial to this thesis. To learn more about neural networks we refer to Goodfellow et al. [24].
Deep Learning is about training neural networks (in this thesis referred to as *Artificial (Conventional) Neural Networks (ANNs)*). These networks are inspired from biology in so far that they use *neurons* which perform simple computations and are organized in several *layers*, an input layer, multiple hidden layers (hence deep learning), and an output layer. A neuron j is connected to some (e.g. convolutional layers) or all (fully connected layer) neurons i in the previous layer and each connection is associated with a weight w_{ij}. Additionally, each neuron can have a bias b_j. Figure 2.1 illustrates a fully-connected network. In a convolutional layer, one or more filters are applied to each input pixel and the weights for the filters are learned. Each neuron computes the sum of all previous neurons multiplied by their weights and the bias and applies a non linear activation function ϕ on the result. As an activation function, most commonly the *ReLu* function [58]

$$ReLu(x) = \max(x, 0) \tag{2.1}$$

is used. Neural networks can be used in many different settings, including supervised, reinforcement, and unsupervised learning. In particular, they can be used as *universal function approximators* [14] which is used in the reinforcement learning setting (see section 2.2.2). The weights and biases of the network are the parameters that can be optimized. On the other hand, the architecture (number of layers, number of neurons), the activation functions, and the parameters of the optimizer (most notably the learning rate) are *hyperparameters* and have to be specified before the training process. The standard way to train a neural network is *Backpropagation*.

2.1.1 Backpropagation

As we introduce several variants of the *Backpropagation (BP)* algorithm, it is important to understand its central concepts. In BP an external error signal E is generated on the output of the neural network and is then propagated backwards through the neural network. The error signal depends

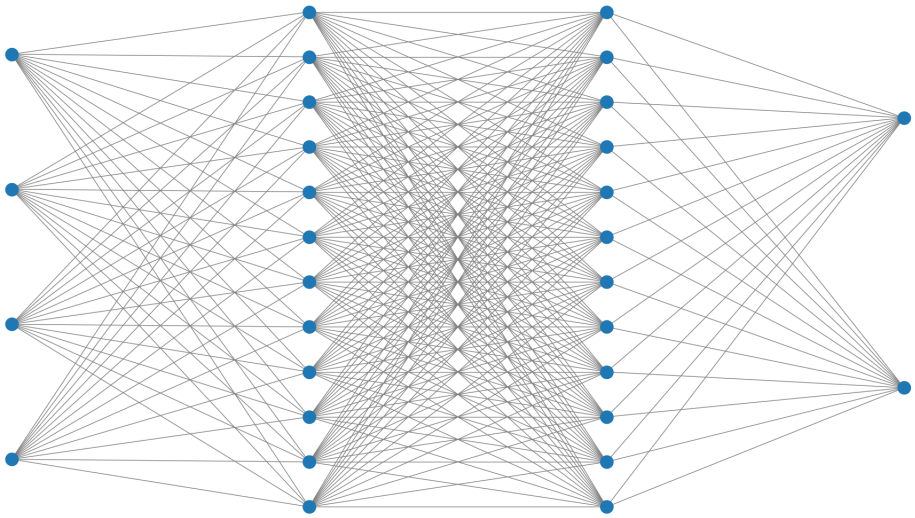

Figure 2.1: Example of a fully connected neural network with 4 inputs, two hidden layers with 12 neurons each, two output neurons, and no biases. The forward computation happens from left to right using the values of the weights (omitted in the graphic). The backward pass moves from right to left using the derivatives with respect to the weights.

on the given task and can for example be the difference of the computed and the expected target outputs. In the forward computation of the network, the output o_j of each neuron j is computed as:

$$o_j = \phi(net_j) \tag{2.2}$$

$$net_j = \sum_{i=1}^{n} o_i w_{ij} + b_j \tag{2.3}$$

For one neuron, the weight update is computed with the help of the partial derivative of the error term

$$\frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta o_j} \frac{\delta o_j}{\delta net_j} \frac{\delta net_j}{\delta w_{ij}} \tag{2.4}$$

which leads to the weight update

$$\Delta w_{ij} = -\eta \frac{\delta E}{\delta w_{ij}} = -\eta \delta_j o_i \tag{2.5}$$

with learning rate η and

$$\delta_j = \phi'(net_j)(o_j - t_j) \tag{2.6}$$

for output neurons with target t_j and

$$\delta_j = \phi'(net_j) \sum_k \delta_k w_{jk} \tag{2.7}$$

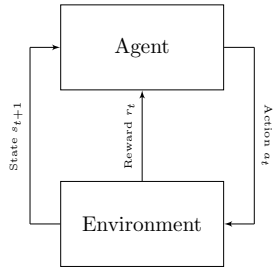

Figure 2.2: The general reinforcement learning framework. The agent, given a state s_t, computes an action a_t. Based on this action, the environment returns a new observation s_{t+1} an a reward r_t back to the agent. In the Open AI Gym setting, the agent additionally receives the information whether the environment is terminated (final state).

for hidden neurons. Each neuron is now updated using this equation, starting with the output neuron and moving through the hidden layers to the input layer. The last equation (2.7) illustrates the name Backpropagation, because the backpropagated error δ of a neuron in layer l depends on the δs of the neurons in the following layer $l + 1$. [24]

2.2 Reinforcement Learning

Reinforcement Learning, alongside supervised and unsupervised learning, is one of three main machine learning paradigms. In reinforcement learning an *agent* has to find a good strategy in an *environment*, only given a reward function that specifies a *reward* for the current state or transition between states. To discover a good strategy the agent has to find a trade off between *exploration* (of new states) and *exploitation* (of already known strategies). This makes reinforcement learning a very general framework (illustrated in figure 2.2) that can be used for a huge range of different problems which moreover is rooted in psychological models of learning and intelligence as the agent learns only from interacting with the environment, without getting any further input beyond the reward function. Sutton and Barto [86] provide a broad overview over reinforcement learning. This thesis will mostly deal with a specific variant of *Temporal Difference Learning*, termed *Q-learning*, and its variants which are introduced in the following sections.

In general, algorithms can be grouped using several criteria. *Model-free* methods, in contrast to *model-based* methods, learn exclusively from their experience in the environment, while model-based methods try to build a model from the environment which they can then use for further learning without actually interacting with the environment anymore.
Further, we distinguish between *on-policy* and *off-policy* algorithms. In the latter method, the algorithm optimizes a policy using actions which can be chosen by a different or outdated policy. In the first method, the policy which is optimized is the one that also determines the actions.
Next, we differentiate between *value-based* and *policy-based* methods. The former try to predict the reward which the agent is going to obtain in the future when taking action a in state s, while the latter only predicts which action to take with which probability without bothering with the exact reward.

Finally, algorithms can be grouped by the nature of their action-space and state-space, discrete or continuous. [86]

In the following, we describe Markov Decision processes which provide a theoretical framework for reinforcement learning environments, then continue with illustrating Q-learning by example and investigating deep Q-learning and techniques used for its stabilization. Next, we explain policy gradient and actor-critic methods, and finally the Open AI Gym framework is introduced.

2.2.1 Markov Decision Process

A *Markov Decision Process (MDP)* can be used to model an agent in an environment, where the agents actions stochastically influence the environment. For simplicity we first consider an example with a discrete environment and discrete actions. In this example an agent has to navigate its way through a house with four rooms, where it needs to find the toilet in room three (we assume a biologically extremely plausible robot). If it reaches the goal state in room three it gets a reward of 10. For every transition between rooms, where it does not reach the goal, the reward is -1. This example and its corresponding MDP are illustrated in figure 2.3.

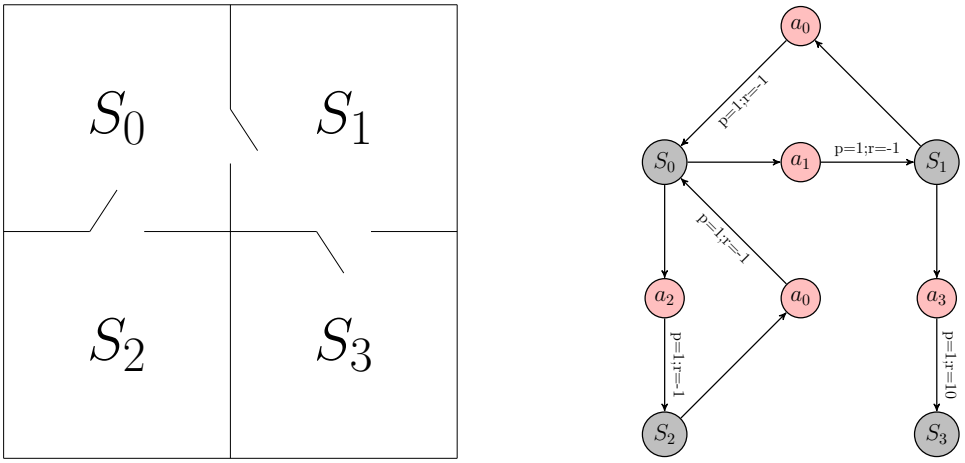

Figure 2.3: 4 room navigation problem. An agent has to find its way through a house with four rooms (left) to the goal state S_3. The possible actions a_i describe switching from the current state to state i. It is not always possible to take every action, but for simplicity we assume that the agent can only take valid actions. Alternatively, one could assign a negative reward to impossible actions, such that the agent has to learn which actions to avoid. This is modeled as a Markov decision process (right). In each state s a number of actions a can be taken. After taking an action the agent reaches a new state with probability p (here and generally in case of a deterministic environment $p = 1$) and obtains a reward r. S_3 is a final state and therefore the environment terminates after reaching it.

The environment is represented by a set of states S. In each state s the agent can choose an action a from a set A. After taking the action the agent will go to a state s' with probability $P_a(s, s')$. All the transitions are assigned a reward (possibly zero) of $R_a(s, s')$. The goal of the agent is to find a *policy* π which picks the action in each state that maximizes the long-term reward.

It often makes sense to value short term rewards somewhat higher than long term rewards (consider having chocolate now or having it in 10 years time). This can be modeled by a *discount factor* γ. A future reward is multiplied by γ for every step the agent needs to reach it. E.g. if γ is 0.99 and a reward of 10 can be reached in 10 steps, the reward is discounted to $10 * 0.99^{10} \approx 9.04$.
The future reward of any state for a policy and an action pair (s, a) is described by the *Q-value* $Q_\pi(s, a)$ and can be represented as the expectation of the sum of the immediate reward r and the future reward for the policy discounted by γ which is itself given by the Q-value when taking the optimal action in the next state. This equation is called the *Bellman equation*:

$$Q_\pi(s, a) = \mathbb{E}_{s'}[r + \gamma max_{a'} Q_\pi(s', a')] \tag{2.8}$$

We can now reformulate the problem of Reinforcement Learning as solving a MDP, where the transition probabilities are not known.
The idea of MDPs can be extended to both continuous state and action spaces. [86]

2.2.2 Q-Learning

Q-Learning is a model-free, off-policy, and value-based reinforcement learning algorithm which can be modeled by a MDP. We illustrate how it works using the example, given in figure 2.3, already used in the previous section.
As both the state space and the action space are discrete and finite we can represent the Q-values $Q(s, a)$ in a table.

States \ Actions	a_0	a_1	a_2	a_3
S_0	–	9	7	–
S_1	8	–	–	10
S_2	8	–	–	–
S_3	–	–	–	–

Table 2.1: The *Q-table* contains (immediate and expected future) rewards the agent expects to get if action a_i is taken in state S_j. It is iteratively updated whenever the agent takes a step. In this example the algorithm has already converged for $\gamma = 1$. If for example the agent is in state S_1 and takes the action a_0, it expects a future reward of 8, because the agent will reach the goal state (reward 10) in two steps (reward -2) given the agent follows the optimal strategy. Impossible actions are represented with a dash.

We now try to learn the environment by picking actions until we reach a terminal state. When a terminal state is reached, we have finished one learning *episode*. We then execute many such episodes until the Q-values in the table converge. With each step we iteratively update the value $Q(s_t, a_t)$ in the table for the current state-action pair by applying the Bellman equation:

$$Q_{new}(s_t, a_t) = (1 - \alpha)Q_{old}(s_t, a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1}, a)) \tag{2.9}$$

The *learning rate* α determines how important the newly acquired knowledge is. If α is zero, the agent considers only prior knowledge, while if α is one, the agent only considers newly found

knowledge. In our case the environment is deterministic so we can just choose α equal to 1.
Another aspect which we need to consider is whether to follow the best known policy so far or whether to explore new states. This is usually done by choosing a random action with probability ϵ in each step. This is called an *epsilon-greedy* policy. When ϵ is low we can exploit the so far acquired knowledge, but hardly explore the unknown states. If ϵ is high on the other hand, we might not get good results or do not explore states which can only be reached by playing well for a certain amount of time (imagine a video game where the speed increases after not going game over for a certain time).
As a trade off, ϵ is often chosen high in the beginning and is reduced over time. [86]
However, the environment often has an unknown number of states, a continuous state space, or the state space is simply too large. In this case, keeping all values in a table becomes impracticable because it will take too much time to explore all the possible states which brings us to the next section.

Deep Q-Learning

To avoid the problem of having to visit every state in order to learn their Q-value, we assume that the Q-value for the optimal policy Q^* is some function of the current state and action. If this assumption holds, the agent can generalize from the states it already observed to make good decisions. We will use a neural network to approximate Q^* as neural networks are universal function approximators [14]. This network is called a *Deep Q Network (DQN)* [53]. To train the DQN we need to reformulate the reinforcement learning problem as a supervised learning problem. This can be achieved by viewing the problem as an alternating series of optimization problems, where first the DQN parameters are initialized, then the agent does one step and the Q-value of the current state-action pair is calculated using these parameters. The network now calculates the best possible Q-value in the next state after taking action a and receives the reward associated with a. The sum of the best next state Q-value $\max_a Q(s_{t+1}, a)$ and the reward r_t are used as the ground truth to update the predicted Q-value of the current state $Q(s_t, a_t)$. As this leads to Q-values being propagated from later to earlier states this is also called a *temporal difference update*. The new parameters of the DQN are then used to calculate a new Q-value and so on. When using the Mean Squared Error (MSE) to calculate the loss L, we obtain the equation:

$$L = (r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))^2 \tag{2.10}$$

This loss is then backpropagated through the network.
As convergence in DQNs is generally poor (it often diverges or oscillates), several techniques to improve on it are explained in the following. Table 2.2 shows an overview over the hyperparameters that need to be chosen before training a DQN. [53]

Experience Replay One of the main obstacles to conversion is that the Q-value of two nearby states is strongly correlated when updating the Q-value every step. Using several correlated data points in a row leads to high variance in the network updates. To reduce this effect, a step (s, s', a, r) is pushed into a so called *replay memory*. To update the network parameters, we now draw random samples from the replay memory. Note that this means updating the network parameters and calculating new Q-values is now uncorrelated, so it would be possible to do several steps and then update the network parameters. In practice, we usually do one step in the environment and then use several random samples from the replay memory to update the network. This leads to a higher

Parameter	Domain	Function
Discount Factor γ	$\gamma \in \mathbb{R}, 0 \leq \gamma \leq 1$	Trade off between low rewards soon and high rewards in the far future
Learning Rate α	$\alpha \in \mathbb{R}, 0 \leq \alpha \leq 1$	Trade off between using prior and new knowledge
Epsilon Start ϵ_{start}	$\epsilon_{start} \in \mathbb{R}, 0 \leq \epsilon_{start} \leq 1$	Probability of choosing a random action in the first timestep
Epsilon End ϵ_{end}	$\epsilon_{end} \in \mathbb{R}, 0 \leq \epsilon_{end} \leq 1$	Minimum probability of choosing a random action in any timestep
Epsilon Decay	$f : \epsilon_{start}, \epsilon_{end}, t- > \mathbb{R}$	Function that maps a probability of choosing a random action between ϵ_{start} and ϵ_{end} to every timestep t
Replay Memory size s_R	$s_R \in \mathbb{N}$	Number of transitions that can be stored in the replay memory
Memory Sample size s_B	$s_B \in \mathbb{N}$	Training batch size
Target Update Frequency f_T	$f_T \in \mathbb{N}$	Every f_T episodes the target network is updated

Table 2.2: Overview of the hyperparameters of a DQN. Additionally to the listed parameters one needs to choose the network architecture and the optimizer which can come with its own set of hyperparameters.

data efficiency as each transition is used several times. [53]
The replay memory can further be improved by sampling it in a prioritized instead of random manner. In this technique, introduced in [75], samples that have a higher temporal difference update and therefore carry more new information are favored. Always choosing the sample with the highest priority in a greedy manner would lead to always seeing the same samples. To alleviate this, each memory is assigned a sampling probability depending on its priority instead. This still introduces a bias towards the samples with higher priority and thus a risk of overfitting to these samples. To avoid overfitting, the weight update induced by each sample is multiplied with the inverse of the probability of choosing this sample.

Separation of Target Network and Policy Network A *target network* is maintained which is a copy of the *policy network* (the network that determines the actions to take), but is only updated every f_T iterations. This target network is used to compute the next state Q-value $Q(s_{t+1}, a)$. Without doing this, increasing of $Q(s_t, a)$ tends to increase the value of $Q(s_{t+1}, a)$ as well which in turn increases the target value. This can lead to oscillation or divergence. Using the target network to calculate the next state value makes the learning more stable because it adds a delay between updating the current action values and updating the targets. [53]

Error Clipping The error is clipped to be between -1 and 1. This is equivalent to using the absolute value loss if the error is greater than 1 or smaller than -1 which is similar to the Huber-loss function. The advantage of this technique is that it combines the sensitivity of the MSE-loss, while at the same time it is not as prone to outliers. [53] [27]

Double Q Learning It has been shown that the Q-learning algorithm introduced so far is prone to overestimation of the Q-values when any kind of estimation error, including using a function approximator, is present in the set-up. This error can be alleviated by decoupling action selection and action evaluation. As a target net which is more or less decoupled from the policy net was already introduced, we can use this target net to estimate the function value for $Q(s_{t+1}, a)$, while the policy net is used to select the corresponding action a. This works reasonably well without introducing a new network to estimate the action values. [92]

Dueling DQNs Intuitively it makes sense to calculate the Q-value of the state and the Q-value of the action taken at the state separately because some states are good or bad regardless of the action being taken. This idea is utilized in Dueling DQNs [95]. The Q-value thus is expressed with the following equation:

$$Q(s, a) = V(s) + A(s, a) \tag{2.11}$$

The *value function* $V(s)$ and the *advantage function* $A(s, a)$ are both calculated by a separate neural network, possibly directly on the inputs or possibly on the output of a network that preprocesses the input state. The intuitive equation above does not allow for Backpropagation of the gradient as it is unclear which network causes the error. We can circumvent this problem by subtracting the average of the advantage function over all actions.

$$Q(s, a) = V(s) + (A(s, a) - \frac{1}{\mathbf{A}} \sum_{a'} A(s, a')) \tag{2.12}$$

where $\mathbf{A}$ is the number of actions.

2.2.3 Policy-Gradient and Actor-Critic Methods

While the Q learning algorithm aspires to calculate the expected future reward value for any state-action pair, *Policy Gradient (PG)* based methods, originally called REINFORCE algorithms [99], try to directly estimate the optimal action probability distribution in each state without bothering about the exact future reward. Calculating the probability of a state-action pair $P(s, a)$ is described with the following equation:

$$P(s, a) = Softmax(\pi'(s, a, \theta)) \tag{2.13}$$

In this equation, π' is the result of the output layer of the network, θ are the network parameters, and *Softmax* stands for the softmax function [98]. The *policy* π of the network is the result after applying the softmax function. We optimize this policy using gradient ascent. The parameter update in every step can be expressed with the equation [82]:

$$\Delta\theta = \alpha \nabla_\theta (\log \pi(s, a, \theta)) R(\tau) \tag{2.14}$$

Here, θ describes the current set of parameters, α is the learning rate, and $R(\tau)$ is the expected future reward in each state as observed in the current episode. To avoid $R(\tau)$ being always positive (in an environment with exclusively positive rewards), which would result in the algorithm believing every state to be good, the average reward of the states is subtracted from $R(\tau)$ and it is divided by its standard deviation. Policy Gradient methods have the advantage of more stable convergence to a local maximum and of being better able to handle high dimensional and continuous action spaces as well as stochastic policies [82].

Actor-Critic methods [52] combine the two approaches of value based and policy based methods. The *critic* estimates the Q-function of state-action pairs, similar to a DQN, while the *actor* determines the action to take in the current state. This action and the next state resulting from it are then used to update the critic by using the Bellmann equation equivalent to equation 2.9 in DQNs.

$$\Delta w = \beta(R(s,a) + \gamma q_w(s_{t+1}, a_{t+1}) - q_w(s_t, a_t))\nabla_w q_w(s_t, a_t) \tag{2.15}$$

where β is a hyperparameter which expresses a ratio between the learning rate α of the actor and the critic's learning rate. As described in section 2.2.2, learning can be made more stable using the advantage function (reorganized equation 2.11):

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{2.16}$$

which can further be approximated by the temporal difference error:

$$A(s_t, a_t) = R(a_t, s_t) + \gamma V(s_{t+1}) - V(s_t) \tag{2.17}$$

This means the critic only needs to calculate values depending exclusively on the state as the action is always determined by the actor. Using this we can reformulate our update equation 2.15 for the critic parameters as

$$\Delta w = \beta(A(s_t, a_t))\nabla_w A(s_t, a_t) \tag{2.18}$$

After updating the critic, we can now update the actor by using the value function of the state as estimated by the critic as an approximation of the future reward, which results in the following update equation for the actor:

$$\Delta\theta = \alpha\nabla_\theta(\log\pi(s_t, a_t, \theta))A(s_t, a_t) \tag{2.19}$$

Note that compared to equation 2.14 we replaced $R(\tau)$, which relies on observations made until the end of the episode, with $A(s_t, a_t)$ which is readily available from the critic.
Alternatively, we can express these equations in a single loss function (equations 2.20 to 2.23) where we additionally include an entropy loss which encourages the actor to learn the best policy that is at the same time as close to a random policy as possible. This improves exploration and robustness [26].

$$L_{policy} = -\log\pi(s_t, a_t, \theta)A(s_t, a_t) \tag{2.20}$$
$$L_{value} = 0.5A(s_t, a_t)^2 \tag{2.21}$$
$$L_{entropy} = \log(\pi(s_t, a_t, \theta))\pi(s_t, a_t, \theta) \tag{2.22}$$
$$L_{total} = L_{policy} + \beta_1 L_{value} + \beta_2 L_{entropy} \tag{2.23}$$

We make some small adjustments so that we can then update the network in one single step using the total loss function. First, we take the negative logarithm to calculate the policy loss, because we want to do gradient ascent on the policy, while we do gradient descent on the value function. Second, we use the mean squared error to calculate the loss for the critic, analog to the DQN algorithm. Lastly, we now have two factors β_1 and β_2 that determine the learning rate ratios between the actor and the critic, and between the actor and the entropy respectively. [81]
Both policy gradient and actor-critic are on-policy, model-free, policy-based methods. As shown

in [53], learning with the value function in an on-policy way is highly unstable because of the correlation of values between nearby states. Instead of using an experience replay memory which is only possible for off-policy algorithms, more uncorrelated updates of the value function in actor-critics can be achieved by running several actors in parallel using the same parameters. These actors can either update the policy asynchronously (AC3) or they can wait for all parallel agents to perform one action and then perform a synchronous batch update (AC2) [52].

2.2.4 Open AI Gym

Open AI Gym [11] is a collection of reinforcement learning problems which act as benchmarks. The environments include classic control problems, robot control tasks, Atari games and others. The environments used in this thesis are introduced in detail in section 4.1. The performance of the agents can be measured in two ways, the final performance of the agent (how much reward can it collect in each episode) and how long it needs to be trained to get there. Additionally, for some environments, Open AI Gym has a notion of solving the environment. In this case, whether solving is achieved and how long it took are auxiliary metrics.
This offers a straightforward way to compare different algorithms, but many reinforcement learning algorithms are highly dependent on hyperparameters and initial weights as well as on the randomness that is inherent to the environment which makes comparison difficult. This problem is further discussed in section 5.1.

2.3 Spiking Neural Networks

Spiking Neural Networks (SNNs) are often called the third generation of neural networks (Maass [48]) as they are theoretically more powerful than the previous generations, perceptrons (first generation) and networks based on activation functions, which we explain in section 2.1 and refer to as conventional or artificial neural networks (ANNs). SNNs consist of simple independent computation units, called *neurons*, which interact with each other via binary signals, called *spikes*. In most models the neurons consist of a *synapse* which accumulates the arriving spikes and then excites the *membrane* which itself emits a spike once it crosses its *firing threshold*. Different neuron models are discussed in section 2.3.1.

A SNN differs from an ANN in several characteristics. First, signals are binary rather than real-valued. Second, they rely on a notion of time which can be analog or discrete. The neurons are simulated for a time period and each neuron can emit several spikes. In contrast, ANNs compute only a single forward pass which yields the result. Moreover, SNN neurons can act completely independent and are thus not bound to the same clock cycle. This is sometimes referred to as asynchronous computation. [68]

The main advantages which are attributed to SNNs are energy efficiency, low latency and using them as research models in theoretical neuroscience [68] [35].
SNNs are suited for investigating neuroscientific questions, as they are implementing basic functionalities which have actually been observed in neurons from the mammalian brain. This makes them, although not an accurate description, a simplified model of biological nervous systems.
The energy efficiency results from the use of neuromorphic hardware (see section 2.3.2), dedicated hardware which specializes in performing computations which resemble neuron functionality in the

brain. This hardware uses little energy as every neuron only performs simple computations (e.g. only addition and multiplication). Moreover, it promises fast inference as neurons are independent and can thus execute their computations in parallel. Further, there is no notion of a central memory which avoids the von Neumann bottleneck which limits computational speed in conventional machines [78]. This bottleneck comes from the low bandwidth between CPU and memory. Assuming it is possible to formulate an algorithm which only needs few spikes to achieve good results, the computation of its result is quick. Further, when using so called *local learning* rules, training can also be made energy efficient on neuromorphic hardware. Local learning has to obey the constraints induced by the special hardware. These are that neurons can only communicate with their neighboring units and no central optimization algorithm can be applied. [68]
On the contrary, when simulating SNNs on conventional hardware (CPU or GPU), none of the mentioned advantages apply [68]. Furthermore, we observed that, if using simulators such as NEST [46] or SpyTorch [61], simulating SNNs is more computationally demanding than running conventional ANNs. In the case of SpyTorch this is due to the network computing one forward pass (analog to a forward pass of an ANN in PyTorch) for every simulated time step. Different simulators for SNNs are presented in section 3.2.

Although, up to now, conventional neural networks outperform their spiking counterparts on all tasks [68] [88], some authors postulate that SNNs can surpass ANNs on certain problems which are especially suited for spiking neurons. These include tasks which naturally use spiking inputs like videos recorded with a Dynamic Vision Sensor (DVS) [3]. For most tasks, however, the input is not given as spikes, which makes it necessary to encode the input into a form the neural network can process. Likewise, it is often necessary to decode the output into the desired form. En- and Decoding is elaborated in section 2.3.3. Coming back to the advantages of SNNs, it has been shown that SNNs using Leaky-Integrate-and-Fire neurons are mathematically equivalent to recurrent neural networks [61] and therefore potentially perform well on the same tasks which especially includes problems that involve sequential data [12] (e.g. speech recognition [85]).

The main disadvantage of spiking neural networks is that they are harder to train. Firstly, their gradients are zero or not well defined due to the binary nature of the signals which makes it more difficult to apply Backpropagation. Secondly, when training directly on neuromorphic hardware, the independent computation of the neurons implies that the learning rules must also be local. This, for example, does not apply to BP. [68] The different training methods are discussed starting from section 2.3.4.

2.3.1 Neuron Models

There exists a wide range of different neuron models which vary in their complexity and biological realism. Less realistic neurons are easier to use for implementation and conception of algorithms, while complicated models are better suited to tackle research questions aimed at understanding how the brain works. As the goal of this thesis is to implement reinforcement learning algorithms, we restrict ourselves to relatively simple models.

Leaky Integrate and Fire Neurons

One often applied method to model neurons is Leaky-Integrate-and-Fire (LIF) [61]. A LIF neuron i can be represented with the equations for the synapse and membrane dynamics.

$$I_i[n+1] = \alpha I_i[n] + \sum_j W_{ji}[n] S_j[n] \tag{2.24}$$

describes the synapse dynamics where $I_i[n]$ is the synaptic current at time step n, $\alpha = \exp(-\frac{\Delta t}{\tau_{syn}})$ is the synaptic decay, determined by the synapse time constant τ_{syn}, $S_j[n]$ are the incoming spike trains and W_{ji} their respective weights. The equation

$$U_i[n+1] = \beta U_i[n] + I_i[n] - S_i[n] \tag{2.25}$$

describes the membrane dynamics, where $U_i[n]$ is the membrane potential at time step n, $\beta = \exp(-\frac{\Delta t}{\tau_{mem}})$ is the decay of the membrane potential determined by the membrane time constant τ_{mem}, and $S_i[n] = 1$ if the neuron emitted a spike in the previous step. [61] This method is also known as *reset by subtraction*. A different method would be to reset the neuron potential to the base potential (e.g. 0) whenever the neuron has spiked in the previous time step. We refer to this second method as *reset to zero* [72].
Neftci et al. [61] show that the equations that describe a SNN with LIF neurons are equivalent to the definition of a Recurrent Neural Network (RNN).
Table 2.3 gives an overview over the hyperparameters that can be set for LIF neurons.

Parameter	Domain	Function
Simulation Length	$\mathbb{N}$	Number of simulated time steps
Membrane Time Constant τ_{mem}	$0 < \tau_{mem}$	Decay of the membrane potential
Synapse Time Constant τ_{syn}	$0 < \tau_{syn}$	Decay of the synapse potential

Table 2.3: Overview of the hyperparameter of a SNN with Leaky-Integrate-and-Fire neurons. Additionally, when using the SNN as a DQN, all hyperparameters of the DQN have to be specified (see Table 2.2).

(Non-Leaky) Integrate and Fire Neurons

This model is very similar to the first one but more simple. In this model, the membrane potential does not decay ($\beta = 1$) with time, therefore:

$$U_i[n+1] = U_i[n] + I_i[n] - S_i[n] \tag{2.26}$$

On the other hand, the synapse potential decays immediately ($\alpha = 0$), such that the incoming spikes only contribute to the membrane potential once. This is also referred to as a delta shaped synaptic current:

$$I_i[n+1] = \sum_j W_{ji}[n] S_j[n] \tag{2.27}$$

Again it is possible to use reset to zero instead of reset by subtraction. In terms of hyperparameters one only needs to choose the simulation length. [66]

More complicated neuron models

A variety of other neuron models exist which increase the complexity. Starting from the Integrate-and-Fire-Neuron we can differentiate between synapse shapes. Possible are delta shaped (current is added in one spike to the membrane potential), or exponential or alpha shaped currents (the current is added to the synapse potential and then leaks to the membrane potential while decaying according to an exponential or alpha function). Further, the dynamics of the membrane can be made non-linear. A simple example for this is the Izhikevich neuron model [36]. A more complicated example is the Hodgin-Huxley [33] model which is based on observations of neurons from the giant squid and was awarded the Nobel prize in Medicine or Physiology in 1963 [97]. This model uses several ionic potentials which are derived from the difference in extra- and intracellular concentrations of ions which then determine the total membrane potential of the cell. Although this model is biologically very accurate, it is difficult to use in simulations, because even simulating a single neuron becomes computationally expensive. The models mentioned in this section are only a small subset of the existing neuron types and should give an idea of the different modeling techniques. An overview over many of the existing models can be found in the NEST documentation [46]. However, none of them are relevant to this thesis, as we restrict ourselves to the use of LIF neurons.

2.3.2 Neuromorphic Hardware

Neuromorphic hardware actually describes a broad spectrum of hardware with the goal to achieve neuron like computation. In contrast to a traditional von Neumann computer, computation is distributed among many individual neurons which communicate with each other via spikes. Many different approaches to achieve this exist, for an overview see Schuman et al. [78]. The main motivations for using neuromorphic hardware are on the one hand its similarity to the brain, which helps neuroscientists to better understand how the brain works. On the other hand it has several advantages over von Neumann architectures including (potentially) better energy-efficiency, higher speed due to avoiding the von Neumann bottleneck, inherent parallelism, and the capability to perform online learning. Perhaps most importantly, neuromorphic hardware offers the chance of brain-like computation, which in many areas still performs better than state-of-the-art artificial neural networks and especially is significantly less energy demanding. [78]

2.3.3 Encoding and Decoding

In most settings for spiking neural networks, the input is not received in the form of spikes. In many papers (e.g. [17, 66, 72] among others), SNNs are employed on the same input as their non spiking counterparts for comparison. These inputs often come in the form of images or other real-valued input streams and first need to be encoded into a form the SNN can process.

There are some exceptions to this, most notably inputs obtained with a Dynamic Vision Sensor (DVS) [3], a kind of camera which accumulates changes for each pixel in a continuous input stream. Each pixel is connected to two neurons which spike when the value of the changes crosses a certain threshold. One neuron is for positive changes, the other for negative changes (the firing of the neurons are denoted as on- and off-events respectively). After a neuron fires, the change value is reset to zero or by subtracting the firing threshold.
It is believed that spiking neural networks are favorable to conventional neural networks for event based inputs [68], but there are few benchmark datasets as of now (for example Spiking MNIST [19] or DVS gesture [3]).
Similarly, tasks often require non-spiking output, like classifications or Q-values. Therefore, the SNN output needs to be decoded to a suitable form.

Input Encoding

The most widely used encoding method is *Poisson Spike Trains* [68]. In this method, values are converted by normalizing them to a probability r between zero and one. Alternatively, two input neurons may be used for encoding one input value, where one neuron fires if the input is positive and the other fires when the input is negative. The weight of the negative neuron is then minus one times the original weight (of the positive neuron). Input neurons fire with probability r in every time step, this parameter is referred to as rate (note that different definitions of the rate exist) [30]. Poisson spike trains have the issue of not always producing the same number of spikes (only in expectation). To avoid this, one can use the rate to calculate the expected number of spikes beforehand and then place this number of spikes equidistantly among the time steps. We refer to this method as *(Fixed number of) Equidistant spikes*.
Another possibility is to not convert the inputs at all, but rather use the value of the inputs as a *constant input current* to the first layer neurons, as suggested in [72].
These three conversion mechanisms all imply a *rate-based encoding* scheme which means that only the firing rates will matter for the computation throughout the network, while precise timing information is ignored. This is a suboptimal use of the SNN, as it ignores some of the information inherent to SNNs and does not produce a sparse encoding. Sparse encodings, however, are preferable as they reduce the amount of energy needed to run the network on neuromorphic hardware. [68]
Other encoding methods exist to enforce sparse spikes, but these methods have not produced as strong results as rate-coding schemes [68]. These methods include *Rank-Order-Coding* where the input neurons fire in order of their values, but precise timings are neglected, [91] and *Time-to-First-Spike-Coding* where the input value is translated to a latency time before the first and only firing of the input neuron [56].
Beyond this, *population encoding* methods exist where input values, rather than being represented by a single input neuron, are encoded with a population of several neurons. This method makes the network more resistant to noise or disturbing factors and has been observed in certain areas of the brain [101].

Output Decoding

In rate-based classification tasks, the easiest decoding method is to have one output neuron for each class and to declare the neuron with the highest firing rate as the winner. In temporal or rank order schemes where any neuron will fire at most once, the output neuron that fires first will determine the class. This method can also be applied to rate-based schemes to speed up the computation,

but it results in a loss of accuracy. [68] Further, the classification in rate-based networks can be improved by prohibiting the neurons in the output layer from firing and instead using the maximum of the potentials to determine the class [72]. Like in encoding, it is possible to use population codes to reduce the variance of the output [68].

2.3.4 Training Methods of SNNs

We propose the following classification of training methods, which is inspired by the classification suggested by Pfeiffer and Pfeil [68] and the classification suggested by Shrestha and Orchard [80]. Further, it includes a method from the recent paper by Wu et al. [100] and splits local learning into two sub categories. We differentiate between three main groups, each consisting of several subgroups:

1. *Conversion-based methods*: These include all methods which first train a conventional neural network and then convert it to a spiking one.
   (a) *Generic conversion*: An ANN is trained which is restricted by few constraints. Examples for this method include Diehl et al. [17] and the extension on this paper from Rueckauer et al. [72]. The only constraint on the training is that hidden layers have to use ReLu activation functions. The network is then converted by scaling the weights. This method is explained in more detail in section 2.3.5.
   (b) Training of a *constraint network*: Pfeiffer and Pfeil [68] differentiate training of constraint networks from the first method. Here, the hyperparameters of the SNN are chosen beforehand (e.g. neuron model) and the ANN is subject to constraints imposed by these parameters. This implies a trade-off between more complicated training and straightforward conversion as the weights of the ANN are then used one to one in the spiking network.
2. *Backpropagation based methods*: This part illustrates methods which are adapted from classic Backpropagation such that a SNN can be trained directly on conventional hardware. They do not, however, have to obey the local learning constraints which would allow them to be employed directly on neuromorphic hardware.
   (a) *Backpropagation with surrogate gradients*: As the gradients of binary activation functions are always zero or not well defined, standard Backpropagation can not be applied to training spiking neural networks. A simple method to circumvent this problem is to use the gradients of a shallow function as a replacement in the backward pass [61]. See section 2.3.6. As the paper "SLAYER: Spike Layer Error Reassignment in Time" [80] points out, this method ignores the temporal dependencies of the current neuron state. They propose a more complex learning framework which is also based on surrogate gradients.
   (b) *Tandem Learning*: Wu et al. [100] propose a method where an ANN and a SNN which share the same weights are trained simultaneously. In the forward pass, each layer, both in the ANN and SNN, takes the output of the previous layer of the SNN as input for their computation. The backward pass is then exclusively computed with the ANN.
   (c) *Binary NNs*: Binary neural networks can be viewed as a hybrid of conventional and spiking neural networks. They behave like SNNs in so far that they use binary signals, but lack a notion of time and instead compute their results with a single forward pass

as ANNs. Similar to SNNs, they can be trained using surrogate gradient methods, conversion, or in tandem. They can be run efficiently on neuromorphic hardware and also require less energy when running on CPU or GPU as the computations become much simpler. Unlike SNNs though, they cannot exploit temporal patterns as they lack a notion of time. [68]

3. *Local Learning*: When training a network directly on neuromorphic hardware, one has to take into account that each neuron does its computations independently and thus cannot access globally stored variables. To be more precise, no global variables exist. As Backpropagation relies on several principles that violate this locality rule, most notably keeping symmetric weights for the forward and backward passes [59], new learning rules have to be devised.

   (a) *Spike-Timing-Dependent-Plasticity (STDP)*: First introduced in [49], STDP changes weights based on the relative spike timing between the pre- and postsynaptic neurons. It is explained in more detail in section 2.3.7.

   (b) *event-based Random Backpropagation (eRBP)*: This method, introduced by Neftci et al. [59], circumvents all the locality issues given in standard Backpropagation and is explained in detail in section 2.3.8.

In our experiments we focus on the methods generic conversion (1a), and surrogate gradients (2a). For the theoretical part we also investigate STDP (3a) and eRBP (3b) as it would be desirable to train SNNs directly on neuromorphic hardware.

2.3.5 Conversion from Conventional Neural Networks

The conversion method used throughout this thesis was first suggested by Diehl et al. [17] and was further improved by Rueckauer et al. [72]. In this method, a fully-connected or convolutional neural network is converted into a spiking counter-part. Four things need to be considered when converting a NN. First, the encoding of the input (see section 2.3.3). Second, the conversion of the weights. Third, the decoding of the output (see section 2.3.3). And last, the hyperparameters of the spiking neural network, including simulation length and neuron model. The mentioned papers use Integrate and Fire neurons (see section 2.3.1). The simulation length implies a trade off: When making it too short, the results are bad, while when choosing it very high, results hardly improve, while the run time dramatically increases. The different conversion methods are summarized in table 2.4.

The method from Diehl et al. [17] directly transfers the weights from the original NN to the created SNN, changing them only by a layer-wise normalization. Diehl et al. consider two different methods, model-based normalization and data-based normalization. Before going into these methods, it is important to understand how normalization can reduce the conversion error. Errors in the converted network come from two sources. Neurons are only allowed to emit one spike per timestep which means if a neuron is *overactivated* (e.g. the membrane potential is 2, while the threshold is 1) it will still emit only one spike. The second source of error is *underactivation*. The problem here is that a neuron might fire only late in the simulation or not at all if the value in the original NN was small. Yet, this can lead to an error in the next layer if this neuron has a large outgoing weight.

Model-based normalization avoids overactivations completely by rescaling the weights such that the highest possible activation is one. This method can lead to problems through underactivation, so *data-based normalization* only rescales the weights using the highest activation that has occurred in practice which is usually much lower than the highest theoretical activation. Rueckauer et al. [72] observed that this method still is prone to error from underactivations, so they suggest *robust normalization* which rescales the weights using a p-percentile of the observed activations to rescale them. If for example $p = 99\%$, weights are rescaled using a value where only 1% of the values observed in practice are higher.
To get even better performance, Patel et al. [66] suggest doing a parameter search for the rescale factors, which allows to choose the best rescale factor for each layer. However, this approach is computationally much more expensive than using the normalization methods mentioned before.

When converting Deep Q Networks, Meschede [51] reports that directly converting the network using the method from [72] does not perform well. Instead, he proposes to first train a classifier which learns the policy of the DQN from its saved experience replay memory and then is converted to a SNN. Specifically, the classifier is trained by assigning each input in the replay memory the action with the highest Q-value as a class. This approach is motivated by the many papers that show effective conversion of classifiers, while only little research exists on direct conversion of DQNs. However, Patel et al. [66] do not report any problems with direct conversion, so we test both methods in the experiments. Furthermore, we propose yet another conversion technique, where a spiking classifier is trained from the original DQN by learning its policy via direct training with surrogate gradients. This approach allows the SNN to use less spikes by using a small simulation time in the training or even adding a regularization term depending on the number of spikes. Additionally, it is different from the conversion techniques introduced above such that it does not exhibit the same weaknesses. This technique can be viewed as a hybrid of conversion and direct training using surrogate gradients. Like the indirect conversion method, it is motivated by existing papers which show direct training of classification networks, while to our knowledge no papers for direct training exist for DQNs. The three described conversion methods (direct, via classifier, SNN classifier) are illustrated in figure 2.4.

Parameters	Conversion Methods
Weights and Bias	Model-based, Data-based, or Robust Normalization, Parameter Search
Input	Poisson Spike Trains, Equidistant Spikes, Constant Input Currents
Output	Spike Output, Potential Output
Neuron Model	Non-Leaky-Integrate-and-Fire
Reset Method	Reset by Subtraction, Reset to Zero
Simulation Time	Parameter Search

Table 2.4: Overview of the different conversion methods and hyperparameters one needs to choose in order to convert an ANN to a SNN. The table only shows the methods we considered in the thesis.

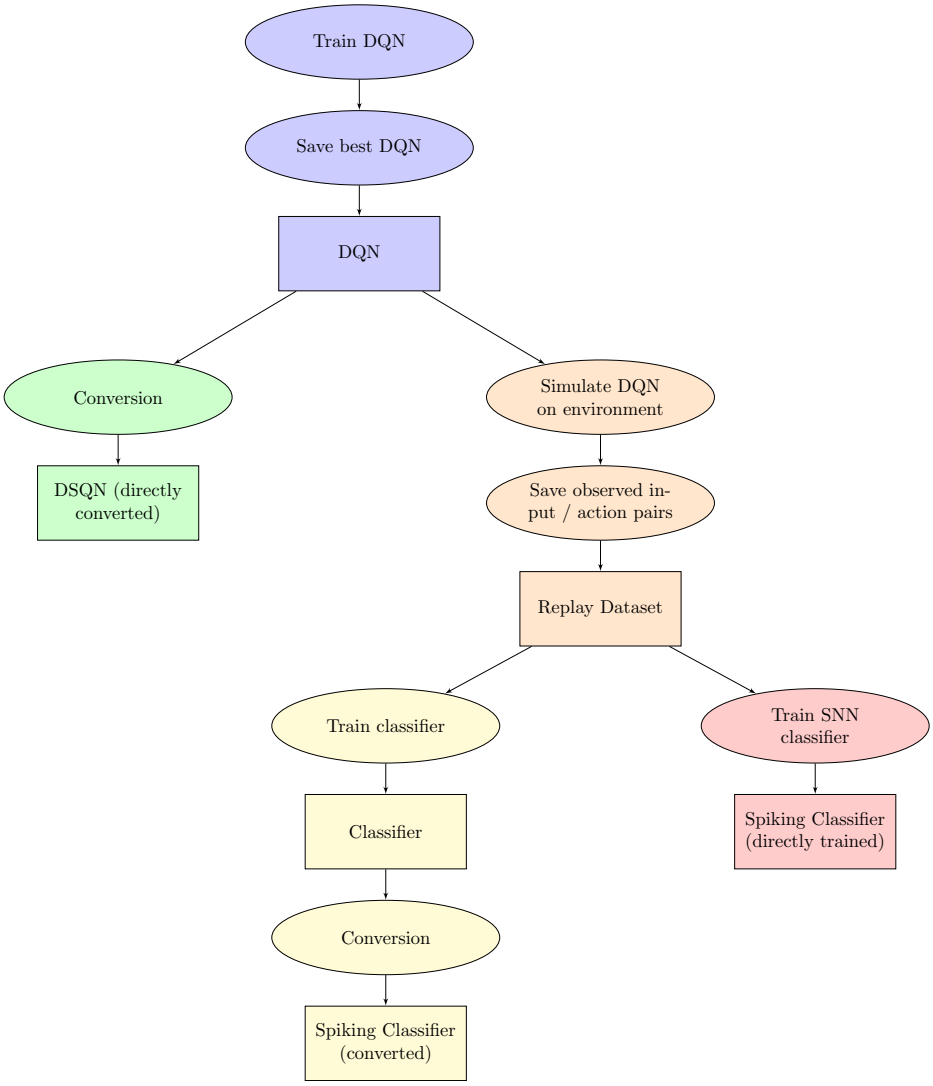

Figure 2.4: This graphic illustrates the different conversion methods. In all cases, first a DQN is trained and the model is saved (blue). In the case of direct conversion (green), the conversion methods [17, 72] are applied directly. For the indirect conversion methods, the DQN is simulated on the environment and a replay dataset is created by saving the observed inputs and the predicted actions (orange). Then, either an ANN classifier is trained to learn the policy from the dataset and then converted to a SNN (yellow) or a classifier is trained using a direct training method (red). In this last case no actual conversion in the sense of weight transfer is happening.

2.3.6 Backpropagation with Surrogate Gradients

One problem when directly training SNNs is that the activations are binary and thus their gradients are zero or not well defined. To overcome this problem, the gradients are replaced by surrogate functions in the backward pass which allow training without changing the forward pass of the network. As a surrogate gradient we use the derivative of the normalized negative part of a fast sigmoid function as suggested in Zenke et al. [102]. Zenke et al. then avoid Backpropagation through time and instead distribute error signals directly to the hidden units using a feedback matrix for which all quantities are calculated in an online manner before backpropagating the error. This technique is implemented in the SpyTorch framework [61]. Here, the hyperparameters one can choose are the simulation time as well as the neuron model, possible are leaky or non-leaky Integrate-and-Fire neurons. Additionally, SpyTorch allows to add a regularization term based on the number of spikes such that sparse spiking can be encouraged.

2.3.7 Spike-Timing-Dependent-Plasticity

Spike Timing Dependent Plasticity (STDP) is a process which was observed in several areas of the brain and thus is believed to contribute to learning there. It works by strengthening neuron connections which appear to be causal, or more specifically, if a pre-synaptic neuron spikes and the post-synaptic neuron spikes soon after, these events are assumed to be connected and the connection is strengthened. On the other hand, if the post-synaptic neuron spikes soon before the pre-synaptic neuron, the connection is assumed to be acausal and it is weakened. [49]
The concept of STDP can be extended to *reward-modulated STDP* or *R-STDP*. In this method, the network produces a result and then receives a reward (or punishing) signal which then leads to strengthening or weakening of the weights in the R-STDP layer.
The problem when training a deep neural network using STDP is that it is unclear how to propagate errors to the deeper layers of the network. State-of-the-art implementations [57] [39] use unsupervised STDP layers for the lower layers and train the top (classification) layer using R-STDP. The problem with that approach is that it only works if the unsupervised layers extract meaningful representations.

Thus, to develop deep reinforcement learning methods, one either needs to design new algorithms which are local in nature or STDP needs to be adapted such that it approximates Backpropagation. Regarding new learning strategies, methods inspired from actor-critic algorithms exist, see section 3.1.3. The same section introduces recent work on approximating Backpropagation with STDP.

2.3.8 Random Backpropagation

Another training approach using only local learning rules is *Random Backpropagation (RBP)*. Standard Backpropagation relies on network-wide information being available [59]. Explicitly, the information is used to access computations from the forward pass, most importantly the weights of the connections, to calculate the gradients in the backward pass. As this violates the assumption of local learning, RBP circumvents the problem by using random feedback weights instead of the actual weights of the network to backpropagate the error. Mathematically, this can be described in the following way: First, we recall the update equations for the hidden layers in standard BP (see

section 2.1.1, equations 2.5 and 2.7):

$$\Delta w_{ij} = -\eta \frac{\delta E}{\delta w_{ij}} = -\eta \delta_j o_i \tag{2.28}$$

$$\delta_j = \phi'(net_j) \sum_k \delta_k w_{jk} \tag{2.29}$$

These equations change to

$$\Delta w_{ij} = -\eta \frac{\delta E}{\delta w_{ij}} = -\eta r_j o_i \tag{2.30}$$

$$r_j = \phi'(net_j) \sum_k r_k c_{jk} \tag{2.31}$$

where c_{jk} replace the weights w_{jk} and are random and fixed. The update of the output layer remains unchanged. [5]
Although it is not quite clear why RBP works theoretically [59], it has been shown to work similarly well as standard Backpropagation in practice [5].
Neftci et al. [59] suggest an extension of the algorithm called event-based Random Backpropagation (eRBP). Using so called two-compartment neurons (illustrated in figure 2.5), this algorithm can backpropagate error signals relying only on spikes. This is achieved by connecting the output neurons to error neurons which emit spikes back to the output neurons when the error signal they receive crosses their firing threshold. The output neurons (and all other neurons throughout the network) then accumulate an error term in their second compartment and likewise send an error spike backwards once the compartment crosses its firing threshold.

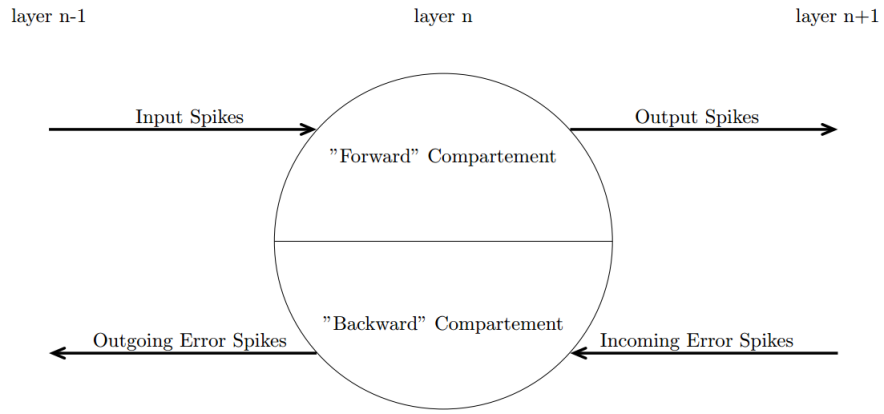

Figure 2.5: Two compartment neuron model as it is used in eRBP. The upper compartment performs the forward calculations, while the lower accumulates the error.

Chapter 3
State-of-the-Art

In this chapter, we analyze the state-of-the-art of the methods described in the previous chapter. As the scope of this thesis is quite broad and there is a myriad of recent papers on both deep reinforcement learning and spiking neural networks, we neither attempt to provide an exhaustive literature review nor a historical account of the topics. Instead, we analyze the papers most relevant to this thesis and refer the reader to recent reviews.

Our focus lies on training deep spiking neural networks in the reinforcement learning domain rather than on the latest advances in reinforcement learning itself. Most of this recent advancement comes from deep value based methods and deep policy based methods as described in sections 2.2.2 and 2.2.3. State-of-the-art methods include Trust Region Methods [76], Soft Q-Learning [25], Soft Actor-Critic [26], Proximal Policy Optimization [77], and Off-Policy Policy Gradient methods [96]. Li [45] provides an overview of recent algorithms and Sutton and Barto [86] recently published a book on the underlying theory of reinforcement learning. To dive deeper into reinforcement learning they provide good starting points.

3.1 Training of Spiking Neural Networks

Based on the classification of training methods in section 2.3.4, this section provides the state-of-the-art papers for each of the methods. An overview over recent advances in training of deep spiking neural networks can be found in Pfeiffer and Pfeil [68] and Tavanaei et al. [88].

3.1.1 Conversion

Patel et al. [66] recently converted a DQN trained on the Atari game Breakout to a spiking network, reporting near loss-less conversion and higher robustness after conversion. They tested the robustness by obscuring parts of the image with a black bar. The SNN then suffers a smaller loss in performance compared to the DQN. However, they have shown these results only for a smaller network which does not perform anywhere close to the results reported by Mnih et al. [53]. They aditionally convert a competitive DQN on the task which works with a very small loss of performance, but they do not repeat the robustness experiments for the larger network.
Meschede [51] converts a DQN for a simple lane following task indirectly by training a classifier which learns to simulate the policy of the DQN. It is trained on a dataset which is produced by the replay memory of the DQN where each state in the memory is labeled with the action the DQN predicted. They then convert the DQN to a SNN and report good results of the converted network. To our knowledge, these are the only works so far that attempt to use a converted SNN for deep reinforcement learning.
However, conversion methods have been widely deployed for classification tasks. They perform

better than SNNs trained with different methods [68] but not as well as the original ANNs [68] [88]. Diehl et al. [17] present a straight forward conversion method which only imposes two constraints on ANN training, ReLu activations for hidden layers and no biases. Additionally, they cannot convert all types of possible layers, for example Max Pooling and Batch normalization layers. The paper reports strong classification accuracy on the MNIST [8] dataset (99.1%). Rueckauer et al. [72] remove all of the constraints on the original ANN except the ReLu activations in the hidden layers. Moreover, they provide a theoretical framework which allows to capture the loss introduced by the conversion. Hu et al. [34] show that the method from Rueckauer can convert state-of-the-art networks (a Residual network [29]) with high accuracy. Sengupta et al. [79] propose a slightly different conversion method, also based on Diehl, which does not allow biases or batch normalization and thus introduces more constraints on the training of the ANN, but they report less loss introduced by the conversion itself. They demonstrate their algorithm on state-of-the-art residual [29] and VGG [84] networks.
A slightly different approach, coined *constrain-then-train* by Esser et al. [18], is to introduce additional constraints before training the ANN which should make the conversion simpler. Unlike generic conversion methods (*constrain-while-train*), this does not allow to convert networks trained without these constraints. This differentiation is somewhat arbitrary as generic conversion approaches also rely on some constraints in the ANN training (ReLu activations and sometimes no biases). Esser uses various such constraints to map an ANN to a TrueNorth [74] neuromorphic hardware chip and reports state-of-the-art results for classification tasks on various image datasets.
Hunsberger and Eliasmith [35] introduce noise into the training of the ANN. The idea here is that the resulting ANN will be resistant against noise and is therefore also robust against the error introduced in the SNN conversion process.
Krishnan et al. [42] propose a completely different strategy where an ANN is repeatedly converted into a SNN and back. The ANN is trained using Backpropagation, then after conversion, the SNN is trained using STDP. This process is inspired by sleep and prevents the network from forgetting old samples and leads to better generalization and improved transfer learning.

3.1.2 Backpropagation Based Methods

In this section, we discuss training methods which adapt Backpropagation in some sort but do not achieve local learning. Additionally, we focus on deep learning methods and ignore learning rules which can only be applied to one layer networks. To our knowledge, there are no papers that investigate reinforcement learning problems in SNNs using Backpropagation based methods. Instead we provide a short overview of papers dealing with classification tasks.
The main challenge in training SNNs using Backpropagation lies in overcoming the problem of zero and not well defined gradients caused by the binary nature of the spikes. One method is to define a surrogate gradient function that is used instead of the actual gradient. Zenke et al. [102] simply propose to replace the gradient by the gradient of a function that is shallow for low membrane potentials and gets increasingly steep when nearing the firing threshold. Any steep exponential function fulfills these criteria, while they use the negative part of a steep sigmoid for performance reasons. Zenke reports good results on the MNIST [8] dataset. Other surrogate gradient methods include SpikeProp [10] where the error is only backpropagated when spikes occur as well as others [44] [64]. Mostafa [55] proposes a gradient based on the spike times in a temporal coded network. This enables the network to learn sparse temporal patterns. Shrestha and Orchard [80] introduce SLAYER, a method which combines the concepts of different Backpropagation based methods. In

addition to the weights, it is also able to learn temporal patterns and axonal delay parameters.
Apart from the methods we summarized as surrogate gradient based methods, Wu et al. [100] have proposed training a SNN and an ANN that share their weights in tandem where the SNN is used to compute the forward pass and the ANN to compute the backward pass. They report good results on the CIFAR-10 [43] and ImageNet [73] datasets.
In section 2.3.4, we have also grouped binary ANNs into methods based on Backpropagation. Wang [93] binarizes the weights of a DQN using Open AI Gym as benchmark. Unfortunately, we could not get access to this paper, so we did not analyze its methods.
Apart from that, Wang et al. [94] use reinforcement learning to train a binary network more efficiently, but apply the resulting network to classification tasks. Other papers focus on lower memory requirements and faster inference [13, 40, 70, 87] or on improved robustness [23] of binary classification networks. For a recent review of binary networks, we refer to Simons et al. [83].

3.1.3 Local Learning

We group local learning rules into two categories (see section 2.3.4). While STDP based rules have been successfully applied to deep unsupervised and supervised learning, in reinforcement learning they have so far only be employed using shallow networks or on simple problems. The eRBP algorithm is a very recent development and so far has been applied only to classification tasks.
Florian [21] derives a reinforcement learning algorithm based on STDP and a global reward signal to solve the XOR problem with a small network with one hidden layer. Potjans et al. [71] propose an actor-critic model which is also based on STDP and global reward signals that can solve a simple Gridworld navigation task.
Fremaux et al. [22] propose a similar actor-critic SNN framework which can solve the Morris Watermaze Task, as well as a variation of the acrobot and the cartpole problem. Although their results seem very promising, it is unclear how to compare them to the gym benchmarks. Their model relies on continuous state and action spaces as well as complex preprocessing of the input states. Additionally, in the case of cartpole, they use a continuous reward function which produces higher rewards when the pole is closer to the center. Due to these difficulties, adapting and reimplementing this paper for Open AI Gym is out of scope for this thesis. Additionally, the approach is limited to low dimensional input spaces because the actor neurons are manually chosen such that they map the input space to the action space in a specific way. This makes the framework less generic than traditional reinforcement learning approaches. Fremaux et al. suggest to preprocess the input state in a smart way for higher dimensional problems such that it becomes possible to manually decide on a mapping.
Aenugu et al. [2] propose a different actor-critic method where several parallel actors are trained and then combined to achieve good performance. However, the training relies on a global critic which predicts the temporal difference error. How this critic is obtained is not clear from the paper. They report meeting the Open AI Gym benchmark for the CartPole problem.
Outside of the reinforcement learning domain, several papers [16, 38, 39, 90] focus on mimicking the visual information processing of mammals. This information processing is based on extracting features in an unsupervised manner. All these papers show that STDP is capable of extracting features in a hierarchical way and then train an external classifier on top of the learned features. Mozafari et al. [57] extend on this method by training the classifier using the biologically inspired R-STDP rule and thus present an end-to-end biologically realistic training framework. They report competitive results on the MNIST data set, and moreover use a temporal coding scheme which

allows at most one spike per neuron.
Another line of research tries to establish a Backpropagation framework for SNNs using only local learning rules. Bengio et al. [9] establish a theoretical framework which approximates BP using STDP. Tavanaei and Maida [89] extend on this framework and show promising results on the XOR, Iris [20], and MNIST datasets. Neftci et al. [59] propose event-based Random Backpropagation (see section 2.3.8) which approximates BP using two compartment models and fixed feedback weights. Recently Kaiser et al. [37] have successfully applied the method on the DVS gesture dataset [3]. O Connor and Welling [62] propose an extension on the RBP algorithm by using adaptive rather than fixed feedback weights.

3.2 Spiking Neural Network Simulators

Recently, quite a few different frameworks for spiking neural network simulators have been proposed. We differentiate two types of simulators. The first can simulate neurons on neuromorphic hardware, while the other allows for simulation on conventional CPUs or GPUs. Further, they differ in their support of different neuron models, algorithms, and richness of features. Some frameworks are more general, while others are optimized for a specific method.
The first set of frameworks we analyze in this thesis focus on implementing different methods in an efficient way on conventional hardware. All these frameworks are based on PyTorch [65]. SpyTorch [61] implements Backpropagation with surrogate gradients with the additional feature that a regularizer can be added which encourages sparse spikes. SpykeTorch [56] takes this one step further by only allowing neurons to emit at most one spike per simulation. SpykeTorch exclusively uses STDP and R-STDP based learning rules for training. Similarly, BindsNet [28] also offers learning based on STDP and R-STDP but is not limited to at most one spike per neuron. The more complicated surrogate gradient training framework SLAYER [80] is also available for PyTorch.
The next set of frameworks focuses more on offering diverse neuron models which makes them more difficult to use but allows for more biologically realistic modeling. Many of these frameworks exist. In this thesis, we use NEST [46] which provides many different neuron models as well as STDP based learning rules and visualization methods. Additionally we use PyNN [4] which is a more generic framework that works with different neuron simulators, including NEST and SpiNNaker [63] neuromorphic hardware as backend engines.
Of the simulators described above, we use SpyTorch, NEST, and PyNN. Using PyNN we also test our methods on a SpiNNaker chip.

Chapter 4
Experiments

After exploring the theoretical background and the current state-of-the-art, we now test different methods to train a Deep Spiking Q Network (DSQN) using conversion from a DQN and Backpropagation with surrogate gradients. We test these methods on three different environments: CartPole, MountainCar, and Breakout. Additionally, we convert a policy gradient network for the CartPole environment and transfer some of the trained networks to the neural simulators NEST and PyNN. Using PyNN we moreover transfer networks to neuromorphic SpiNNaker hardware. In this chapter the results are reported, while their implications are discussed in chapter 5.

4.1 Environments

We begin with describing the three problem environments CartPole, MountainCar, and Breakout.

4.1.1 CartPole

The first environment we consider is the Open AI Gym [11] environment "CartPole-v0" (for a screenshot see figure 4.1). This problem was first described by Barto, Sutton, and Anderson [6]. The task is to balance a 2D inverted pendulum standing on a cart by moving left and right. Possible actions are applying a force of plus or minus one to the cart. The environment is fully described with the four physical parameters angle of the pole, angle speed of the pole, position of the cart, and speed of the cart. An episode ends when the pole falls more than 15 degrees to one side or the cart moves more than 2.4 units from the center or when the maximum episode length is reached which is set to 200. For every step where the episode does not end, the agent receives a reward of +1. After one episode, the pole is reset to a random position close to the center which is done to avoid overfitting of the policy to a fixed initial state. The environment is considered to be solved when the agent obtains an average reward over the last 100 episodes of at least 195 for 100 consecutive episodes.

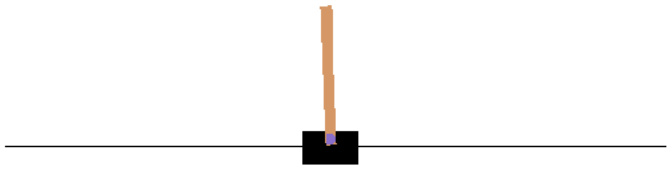

Figure 4.1: Open AI Gym [11] CartPole environment: Force has to be applied to the cart such that the Pole remains upright and the car does not move too far away from the center.

4.1.2 MountainCar

For the MountainCar [54] problem, we use the gym environment "MountainCar-v0" (screenshot in figure 4.2). Here, a car starts in a 2D valley and has to climb up a mountain on the right side. As it lacks enough motor power to do so in a single leap, it needs to use the mountain on the left side to gather momentum. The environment is fully described by its x-coordinate and the speed in x direction. Three discrete actions can be applied to the cart: Applying a fixed amount of force to either of the sides or to do nothing. For every step in which the car fails to reach the top, it receives a reward of minus one. When it reaches the top or when it does not succeed within the maximum episode length of 200 steps, the episode ends. After each reset, the car is randomly positioned close to the valley bottom to introduce stochasticity. The environment is considered to be solved when the agent obtains an average reward over the last 100 episodes of at least minus 110 for 100 consecutive episodes.

This is much harder to achieve than solving the CartPole environment because the agent never obtains positive rewards and instead needs to figure out that reaching the goal terminates the episode which leads to not collecting anymore negative rewards. This is similar to sparse rewards and thus the agent needs to have a good trade off between exploration and exploitation.
Several approaches exist to make the environment easier. We adjust the reward function by adding the distance to the center to the reward, but limiting the reward to be at most -0.1, so that the optimal strategy will still be to reach the goal as quickly as possible.

$$r' = \min(r + |x + 0.5|, -0.1) \tag{4.1}$$

Additionally, we lower the Open AI Gym standard to reaching a 100 episode average of -130 for 50 consecutive episodes.

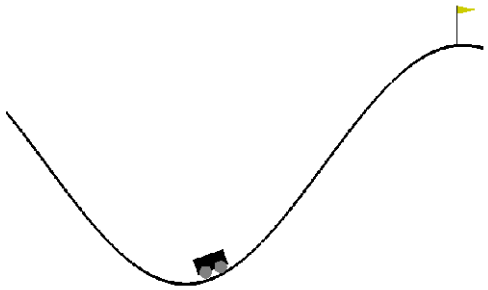

Figure 4.2: Open AI Gym [11] MountainCar environment: The agent should learn to reach the goal on the right by building up momentum by driving back and forth the mountains.

4.1.3 Breakout

In the Atari game Breakout [7] the player has to move a bar right and left to destroy bricks at the top of the screen by hitting them with a ball (screenshot shown in figure 4.3). When the ball falls to the bottom of the screen, the player looses a life. Once all 5 lifes are lost or the time limit is exceeded, the episode ends. The player or agent takes the red, green, and blue values of the pixels as input and has to choose between 4 possible actions: "Do nothing", "Fire" (Brings a new ball into the game after the last one was lost. If there is a ball in play, nothing happens.), "Move Right", and "Move Left". We use the deterministic version of the game (BreakoutDeterministic-v4) where the only uncertainty comes from where the ball appears. To reduce computational demand, the agent sees only every 4th frame and the action it then chooses is repeated until the next frame it sees (analog to [53]). We refer to this as frame skipping.
Since Open AI Gym does not define a notion of solving Breakout, we compare to the results from other papers. Mnih et al. [53] report an average score of 401.2 for a large convolutional DQN. Patel et al. [66] report a score of 307.06 for a similar network and a score of 9.32 for a much smaller fully connected DQN using a greedy policy on the trained network.

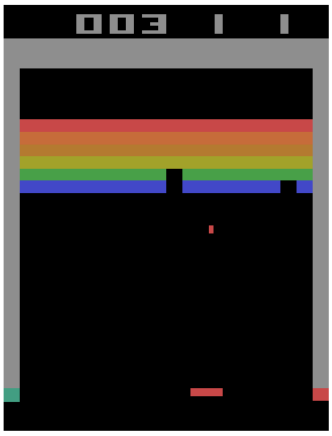

Figure 4.3: Open AI Gym [11] Breakout environment: The player has to hit the bricks with the ball by moving the bar at the bottom left and right.

The environment needs to be preprocessed in a way that the agent can infer information about the speed and direction of the ball and the bar. Additionally, it is beneficial to reduce the image size to lower the number of weights in the network. Mnih simply resizes the image to 84x84 pixels and then stacks the last four screens seen by the agent on top of each other to obtain an input image of size 4x84x84. Patel suggests a more involved *grayscale preprocessing*, where one image is cropped so that it does not include the score and number of lifes left and is then resized to 80x80. To visualize temporal information, the agent then receives an input I where the last four screens

are weighted and added:

$$I = F_t + 0.75 * F_{t-1} + 0.5 * F_{t-2} + 0.25 * F_{t-3} \tag{4.2}$$

Parameter F_t represents the current frame, F_{t-1} the one before and so on. This leaves the agent with an input of size 80x80. Figure 4.4 shows an example where the bar moves rapidly to the right and the ball moves up and right.

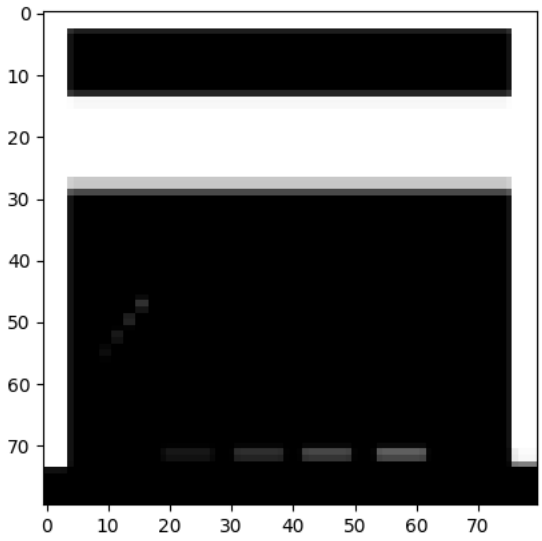

Figure 4.4: Example of grayscale preprocessing used in Patel et al. [66].

4.2 Evaluation Metrics

To evaluate and compare the different algorithms, we report the *average performance* and *standard deviation* on the environment of the trained algorithms. For the conversion we propose a further comparison metric, the *conversion accuracy*, which shows how similar the original and the converted network are. To compute it, we run the converted SNN on the environment and compare its prediction with the predictions made by the DQN it was converted from. We then calculate the accuracy by dividing the number of equal predictions (DQN and SNN predict the same action) by the total number of predictions. Although this metric gives an idea on how similar the two networks are, note that the conversion accuracy will not only be low if the SNN performs worse than the DQN but also if the SNN performs better.
To compare the training process of a DQN and of a directly trained DSQN, we run each training 5 times and report whether the Open AI Gym standard was reached and what the best 100 episode average was as well as after how many episodes those were reached. We then compare the average performances of the networks.

4.3 Deep Spiking Q Networks

The first part of the experiments investigates different methods to train a deep spiking Q network. Initially, we train a conventional DQN. Next, we investigate different conversion methods and finally direct training using Backpropagation with surrogate gradients. Each of the experiments is repeated with different environments.

4.3.1 CartPole

For our first problem, CartPole, we first train a DQN and convert it directly. Next, we train a conventional and a spiking classifier on the replay memory of the DQN and further convert the conventional one. Finally, we compare training an ANN and a SNN using Backpropagation with surrogate gradients.

ANN-Training

A DQN with two hidden layers with 16 neurons each and one output layer with 2 neurons is trained. The two output neurons predict the Q-values for a given state and the two possible actions. Figure 4.5 shows the episode durations (blue) and the 100-episode averages (orange) during training. It can be seen that the agent converges to the maximum episode duration of two hundred. Note that the episode duration is equivalent to the agents reward in the CartPole environment. Table B.8 shows the hyperparameters that were used (CartPole A).

When continuing the training, the DQN does not converge which can be seen when doing a longer training run (see figure 4.6). The performance will then often get worse some time after the Open AI Gym standard has been reached. This can be explained by the fact that the agent sees only similar states once it learns to balance the pole. It thus forgets about the crucial states where the pole is threatening to fall. This problem is known in the literature as *Catastrophic Forgetting* [50].

In order to obtain a well performing model, we save the model once the Open AI Gym standard is reached. The model obtained with this method usually generalizes well but has not yet reached the stage of catastrophic forgetting. To evaluate the performance of the saved agent, it is simulated on the environment without further training for 500 episodes. The average performance and standard deviation are reported in table 4.1. Even though the Open AI Gym standard is reached, we can see that the model has not really generalized well as the number of iterations is heavily dependent on the initial position of the pole. This can be seen from the high standard deviation. The same table also reports the performance of all other training methods and a random agent which can be seen as a baseline to compare against.

DQN-SNN Direct Conversion

We now convert the saved DQN to a spiking network. For this we test different normalization, en-, and decoding methods (see section 2.3.5). We compare the normalization methods model-based, data-based and robust with a p-percentile of 0.99 and 0.95 in table 4.2. Note that data-based normalization is equivalent to robust normalization with a p-percentile of 1.0. To obtain the data

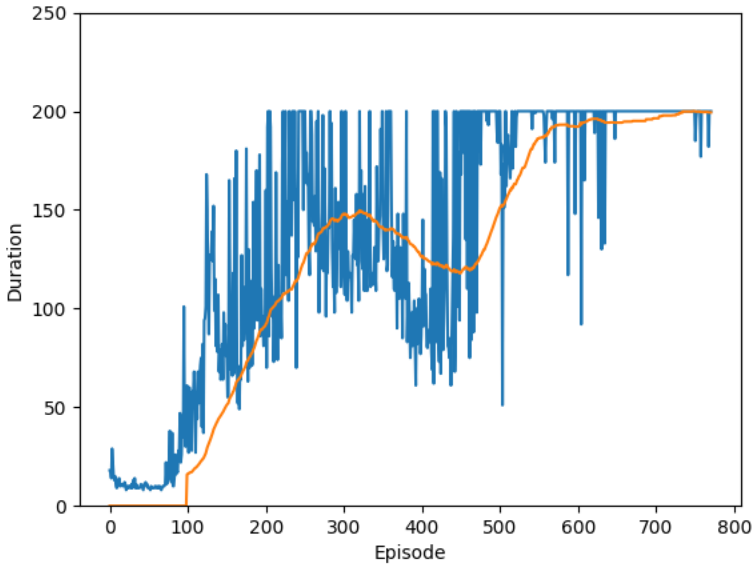

Figure 4.5: Successful DQN training run on CartPole. Blue shows the episode duration in each individual episode, while orange indicates the average duration over the last 100 episodes.

used in the latter three methods, we simulate the DQN agent for twenty thousand iterations and save the observed input-action pairs.
En- and Decoding methods are compared in table 4.3. To use Poisson or equidistant spikes for encoding, usually the values need to be normalized between 0 and 1 (the rate describes a spiking probability). This is a posteriori not possible as the DQN was trained with non normalized values. Instead we use two input neurons, one for positive inputs and one for negative inputs. The latter ones are connected to the next layer with the same weights as the former ones multiplied by minus one. Values that are higher than 1 or lower than -1 are simply cut off which implies that from a certain point on the agent can no longer distinguish how far off the center it is or how fast it is (e.g. a constantly firing input is then only interpreted as "very far to the right" or "moving left very fast"). The alternative would be to train the DQN with normalized data, but that changes the environment and is less generic as how inputs are normalized is only clear with prior knowledge of the environment or by keeping a dynamic list of minimum and maximum input values. We did not investigate this alternative as it does not seem promising (see section 5.2.1).
Additionally, we investigate the performance of the SNN depending on the number of simulated time steps (see table 4.4).
In section 5.2.4, we further compare the two different resetting methods reset-by-subtraction and reset-to-zero, where all experiments in this section use the former and the results from NEST in section 4.5 use the latter reset mechanism.

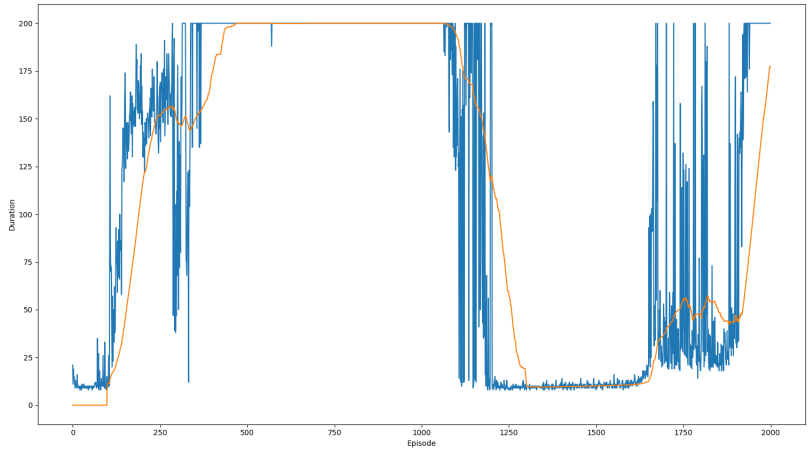

Figure 4.6: Catastrophic Forgetting of a DQN on CartPole. Blue indicates single iteration durations, orange shows the duration average over the last 100 episodes.

DQN-SNN Indirect Conversion

The indirect conversion method as illustrated in figure 2.4 is split into two different sub methods and each method is split into several sub steps. We start with training a conventional and a spiking classifier and then move to the conversion of the non-spiking classifier. From table 4.4 it can be seen that direct conversion does not work well for the CartPole problem. The performance, even for a long simulation time of 1000 steps, falls short of the original performance and the conversion accuracy reaches only around 90%. Meschede [51] suggests to train a classifier which learns the policy of the DQN and then to convert this classifier (see section Conversion 2.3.5).
To obtain the training set we run the saved DQN for two hundred thousand iterations. The classifier is then trained using the same architecture as the DQN and the NLL-loss function. Hyperparameters of the training process are detailed in table B.1.
Figure 4.7 shows the convergence of the training. We observe that the accuracy does not reach 100%. This, however, is expected because the loss function forces more smoothness on the predictions compared to the original DQN which uses the temporal difference loss and picks actions according to the argmax function. The softmax function used by the classifier will naturally be smoother than the argmax function. This effect can be seen especially well when initializing the classifier with the same weights as the original DQN. Doing so yields an accuracy of > 0.999 and a loss of 0.5124. Compared to the result obtained from training (accuracy: 0.938, loss: 0.1383) the accuracy is higher yet the loss is larger.
For the Poisson and Equidistant encoding we use the same method as described in the previous section. Alternatively the inputs could be normalized while training the classifier by using the entries of the replay memory to perform rescaling. A quick test (results not presented) suggests that this alternative works well if the replay memory is sufficiently large.

The conventional classifier is then tested on the environment. It achieves an average performance of 230.39 with a standard deviation of 66.27 (compare overview table 4.1). This is very similar to

Network Training Method	Average Performance	Standard Deviation
Random Agent	20.96	10.99
DQN (Conventional)	226.31	32.74
SNN (Direct Conversion)	160.12	9.17
Classifier	230.39	66.27
SNN (Indirect Conversion)	218.71	32.48
SNN (SpyTorch Classifier)	232.26	60.81
SNN (SpyTorch DSQN)	454.84	81.29

Table 4.1: Overview of DQN training methods for CartPole and their performance. The reported DQNs are saved after reaching the Open AI gym standard, and then simulated on the environment for 100 episodes with a time step limit of 500 each. Average and standard deviation are reported. Converted SNNs are simulated for 500 time steps using the robust conversion method ($p = 0.99$) combined with constant input currents and potential outputs, directly trained SNNs (classifier and DQN) are simulated for 20 time steps.

Normalization Method	DQN Conversion AVG (STD), ACC	Classifier Conversion AVG (STD), ACC
Model-based	120.85 (4.08), 59.97%	203.23 (24.55), 96.63%
Data-based (p-percentile = 1)	125.47 (3.70), 61.35%	190.28 (22.68), 94.88%
Robust (p-percentile = 0.99)	121.99 (3.60), 60.82%	195.18 (18.36), 95.58%
Robust (p-percentile = 0.95)	127.59 (3.89), 62.89%	196.27 (23.53), 96.06%

Table 4.2: Comparison of Normalization Methods for CartPole Conversion. For each method, we report the average performance over 100 episodes, the standard deviation and the conversion accuracy. As en- and decoding methods for the normalization experiment, we use constant input currents, potential outputs, and a simulation time of 100 time steps. The performance of the original DQN is 226.31 (std: 32.74) and of the original classifier it is 230.39 (std: 66.27).

the original DQN's performance.

The classifier is next converted analog to the DQN conversion. The results from the conversion are compared to the other methods in table 4.1 and normalization (table 4.2), en- and decoding methods (table 4.3), and the influence of simulation length (table 4.4) are reported in the same tables as the original DQN.

As the approach of indirect conversion is quite cumbersome, we also "converted" the network by training a classifier which learns the policy of a DQN by using Backpropagation with surrogate gradients. In this case the classifier is already a spiking neural network and does not need to be further converted. Moreover, using Backpropagation with surrogate gradients allows the network to be trained with much shorter simulation times (we choose 20 time steps) as it can adjust to the given time during training. The loss and accuracy of the training process are displayed in figure 4.8. It can be seen that training the classifier using Backpropagation with surrogate gradients results in a slightly less smooth training process compared to conventional Backpropagation (figure 4.7), but the final loss and accuracy are similar. The performance of both agents is reported in the overview

Conversion Method	**DQN Conversion** AVG (STD), ACC	**Classifier Conversion** AVG (STD), ACC
Poisson input & spike output	423.89 (76.96), 63.98%	374.70 (87.92), 83.39%
Poisson input & potential output	128.77 (6.18), 63.71%	384.47 (96.77), 83.27%
Fixed number of equidistant spikes & spike output	105.51 (61.76), 65.28%	181.08 (49.03), 82.41%
Fixed number of equidistant spikes & potential output	88.45 (21.87), 61.20%	181.62 (45.08), 81.20%
Constant input current & spike output	390.94 (102.37), 64.81%	190.03 (16.58), 95.23%
Constant input current & potential output	121.99 (3.60), 60.82%	195.18 (18.36), 95.58%

Table 4.3: Comparison of different En- and Decoding methods for both direct and indirect conversion. The SNN is converted using robust normalization ($p = 0.99$) and is simulated for 100 time steps for each of the experiments. Reported are the 100 episode average performance, standard deviation and conversion accuracy. The performance of the original DQN is 226.31 (std: 32.74) and of the original classifier it is 230.39 (std: 66.27).

table 4.1 and one sees that the performance of both the SNN and conventional classifier are similar to the original DQN, while the standard deviation for the classifiers is slightly higher.
The results of the different conversion methods are investigated in detail in the discussion, see section 5.2. To summarize, both direct and indirect conversion work in principal. However, direct conversion relies on long simulation times to achieve comparable results to the DQN, while the classifiers perform equally well with much shorter simulation times.

Simulation Time	**DQN Conversion** AVG (STD), ACC	**Classifier Conversion** AVG (STD), ACC
10	18.61 (14.50), 56.00%	14.12 (1.57), 44.55%
50	122.4 (3.93), 62.08%	169.61 (17.24), 88.97%
100	122.26 (3.60), 60.72%	195.18 (18.36), 95.58%
500	160.12 (9.17), 81.61%	217.56 (32.99), 99.10%
1000	175.57 (12.14), 89.39%	229.72 (58.33), 99.43%
5000	– (–),97.6%	– (–) ,>99.9%
10000	– (–),99.0%	– (–),>99.9%

Table 4.4: Influence of Simulation Time on Conversion. All simulations use robust normalization ($p = 0.99$) combined with constant input currents and potential outputs. Reported are the average performance over 100 episodes, its standard deviation, and the conversion accuracy. The performance of the original DQN is 226.31 (std: 32.74) and of the original classifier it is 230.39 (std: 66.27). For 5000 and 10000 simulation time steps, simulating for 100 episodes was infeasible with the available computation power, instead only conversion accuracy over a smaller number of episodes is reported.

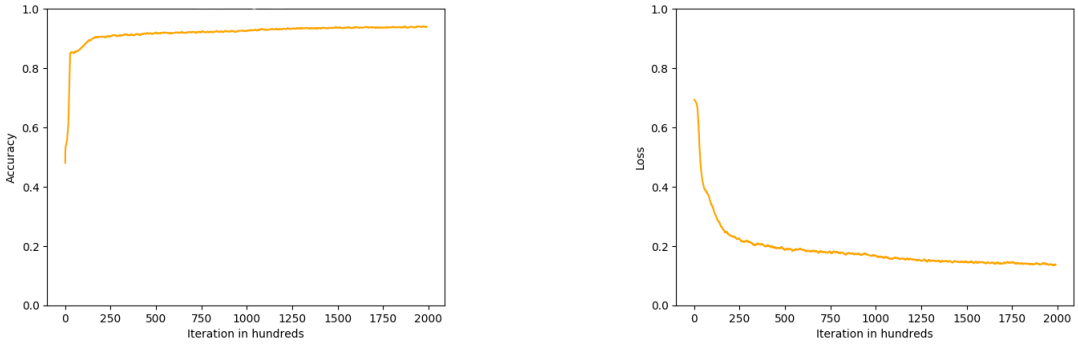

Figure 4.7: Accuracy (left) and Loss (right) over the number of training iterations for the CartPole classifier.

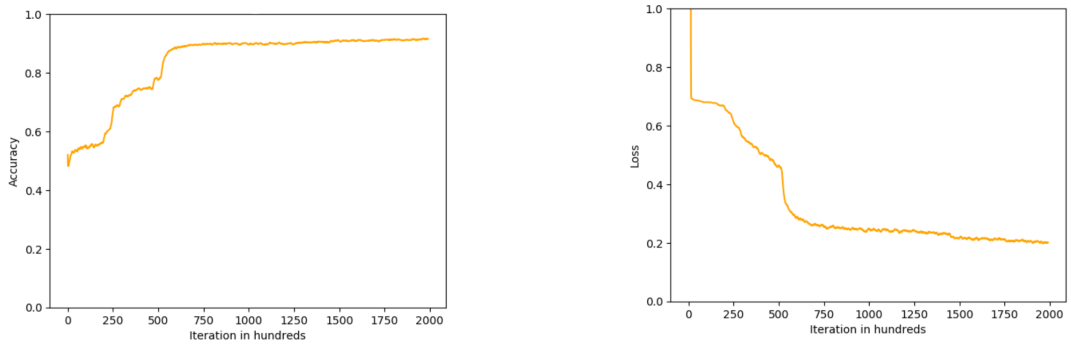

Figure 4.8: Accuracy (left) and Loss (right) over the number of training iterations for the CartPole SNN classifier.

Backpropagation with Surrogate Gradients

We now train a DSQN directly using the method Backpropagation with surrogate gradients (see section 2.3.6). In order to compare this method with the conventional DQN training, we train a network with each algorithm 5 times using the same hyperparameters reported in table B.8 and repeat the experiment with two different learning rates (CartPole B/C describe the non spiking networks and CartPole D/E the spiking networks). The results are reported in table 4.5. Compared to section 4.3.1 we changed the hyperparameters which is motivated by two reasons. First, when running multiple trainings, the hyperparameters used in this experiment reach the Open AI Gym standard more often and thus are superior. Second, in the course of the thesis we changed an implementation detail in the training process which changes the best choice of hyperparameters. This detail is that in the old training we pushed every state seen by the agent to the replay memory, while in the experiments in this chapter we do not push the last state before reaching the maximum number of steps for one episode into the replay memory. This makes sense because the last state is not getting any reward due to the time out, even if it is a very good state. We think that this

behavior can negatively affect the training process.
We fix the simulation length of the DSQN to 20 as this leads to reasonable fast computations and good results. Compared to the DQN, it has no hidden layer biases as SpyTorch does not support this. Instead we add a constant input (equivalent to bias for the first hidden layer) and one additional neuron for each hidden layer to compensate. The spiking network uses Non-Leaky Integrate-And-Fire neurons.
Using the same set of hyperparameters for both training methods has the advantage that no additional hyperparameter search is required for the SNN. On the other hand, there is no guarantee that parameters which work well on the DQN are good for the DSQN. Although this makes comparing the peak performance of the training methods infeasible, achieving peak performance is out of scope for this thesis as it requires an extensive hyperparameter search for all of the problems.
Figure 4.9 shows a successful training run using SpyTorch where the Open AI Gym standard is reached after 274 episodes (using Network E). After that, similarly to DQN training, the performance degrades due to catastrophic forgetting. The performance of one surrogate gradient trained network in comparison to the conventional DQNs and converted networks is reported in the overview table 4.1.

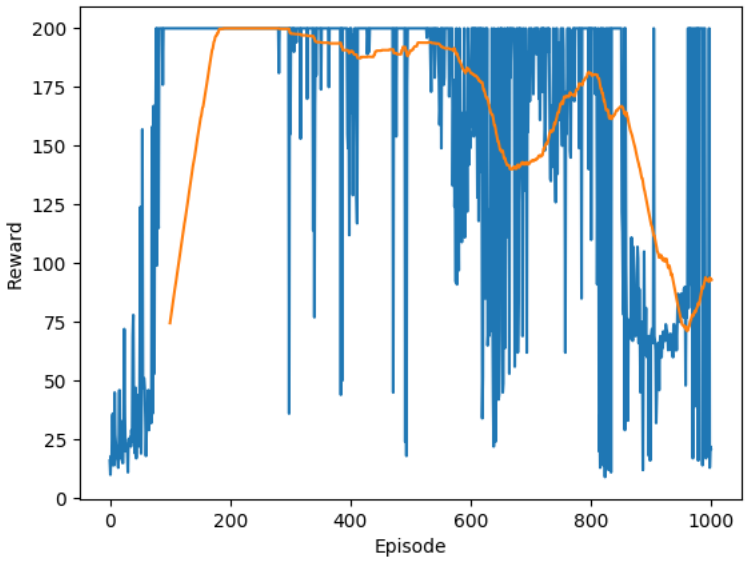

Figure 4.9: Successful DSQN training using Backpropagation with surrogate gradients and SpyTorch on CartPole environment. Blue indicates single iteration durations, orange shows the duration average over the last 100 episodes. The Open AI Gym standard is reached after 274 episodes and after that the performance starts to degrade which can be explained by catastrophic forgetting (see also figure 4.6).

Network	CartPole B	CartPole D		CartPole C	CartPole E
Spiking	No	Yes		No	Yes
Learning Rate	0.001	0.001		0.0005	0.0005
Gym Standard reached?	3 out of 5	1 out of 5		1 out of 5	1 out of 5
At Episode	646.33	749.00		823.00	274.00
(on average if gym standard reached)					
Best 100-episode average	197.79	190.94		198.00	195.54
(average over five runs)					
At Episode	784.20	468.60		731.40	506.80
(average over five runs)					

Table 4.5: Comparison of 5 training runs each for a DQN and a DSQN trained using Backpropagation with surrogate gradients for CartPole and two different learning rates. All trainings are run for 1000 episodes.

4.3.2 MountainCar

In this section, we repeat the same set of experiments for the MountainCar environment. Note the changes we made to the original gym environment as described in section 4.1.2.

ANN-Training

As in CartPole, we first train a DQN and report the training process in figure 4.10 and the performance after saving the agent (along with the performance of all other training methods) in table 4.6. Hyperparameters are reported in table B.8 under MountainCar A. In figure 4.10 it can be seen that the network steadily achieves lower episode durations on average (orange line) until it reaches the relaxed Open AI Gym standard (see section 4.1.2). At this point we interrupt the training and save the agent for performance evaluation and conversion. When continuing the training, the agent would next reach a phase of catastrophic forgetting before reaching the actual Open AI Gym standard which motivated the relaxation.

Network Training Method	**Average Performance**	**Standard Deviation**
Random Agent	-500.0	0.0
DQN (Conventional)	-175.4	100.92
SNN (Direct Conversion)	-147.43	57.51
Classifier	-186.51	107.80
SNN (Indirect Conversion)	-199.75	135.57
SNN (SpyTorch Classifier)	-176.72	110.38
SNN (SpyTorch DSQN)	-130.92	27.90

Table 4.6: Overview of DQN training methods for MountainCar and their performance. The DQN models are saved after reaching the Open AI Gym standard, and then simulated on the environment for 100 episodes with a time step limit of 500 each. Converted SNNs are simulated for 500 time steps and use robust normalization ($p = 0.99$), constant input currents, and potential outputs in the conversion process. Directly trained SNNs are simulated for 20 time steps.

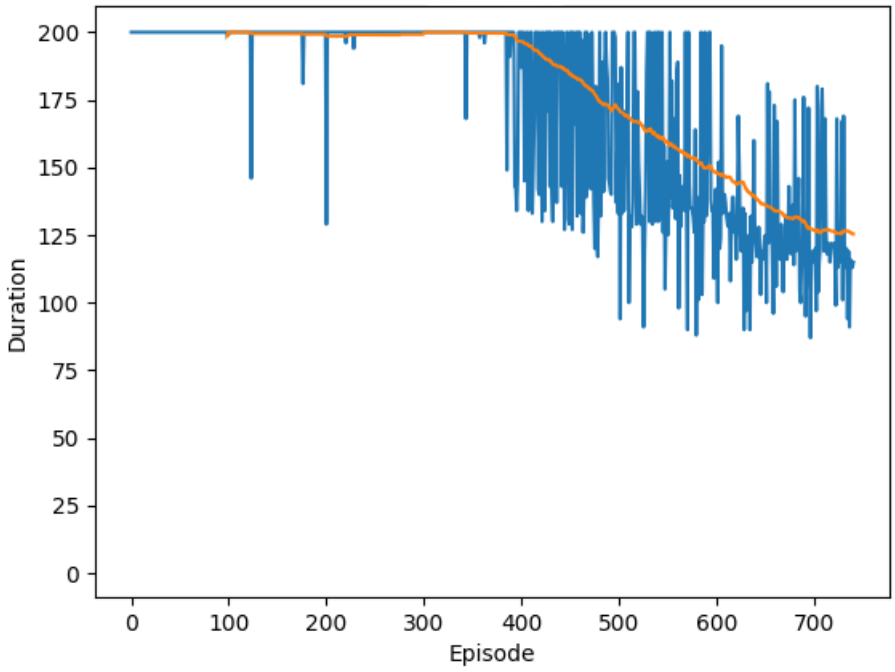

Figure 4.10: Successful DQN training run on MountainCar. Blue indicates single iteration durations, orange shows the duration average over the last 100 episodes. Note that we plot episode durations which is equivalent to the negative reward value. Lower episode lengths are desirable.

Conversion

As for the CartPole environment, the original DQN is converted directly and indirectly. In MountainCar all the predicted Q-values are negative because there only exist negative rewards. When directly converting, this provides no challenge for the potential output method, but when using spiking output neurons, the neurons will never spike. This problem can be circumvented by using two output neurons, one that produces spikes when the potential crosses +1 and the other using the inverse weights of the first neuron, such that it produces spikes when the potential of the first neuron crosses -1. However, we do not test this as our experiments suggest no benefit of spiking outputs (see section 5.2.2). Table 4.6 reports the performance of the different methods, tables 4.7, 4.8, and 4.9 respectively compare normalization methods, en- and decoding methods, and various simulation lengths. Additionally, the training process and hyperparameters of the classifiers (SNN and conventional) are reported in the appendix (figures C.1,C.2, and table B.1). Both classifiers achieve an accuracy of close to 100% when compared to the original DQN.
The implication of the results are investigated in depth in the discussion, see section 5.2. In brief, both direct and indirect conversion methods again principally work. Contrary to CartPole, in

MountainCar, direct conversion improves the results of the DQN, while converting a classifier performs slightly worse than the DQN, and training a SNN classifier results in comparable results (to the DQN).

Normalization Method	**DQN Conversion** AVG (STD), ACC	**Classifier Conversion** AVG (STD), ACC
Model-based	-330.57 (164.63), 36.93%	-256.74 (162.37), 92.61%
Data-based (p-percentile = 1)	-184.18 (71.34),67.14%	-256.23 (158.52), 94.75%
Robust (p-percentile = 0.99)	-180.32 (57.51), 64.38%	-278.92 (168.65),93.89%
Robust (p-percentile = 0.95)	-271.05 (155.56), 36.85%	-283.45 (172.68), 95.20%

Table 4.7: Comparison of Normalization Methods for MountainCar Conversion. For each method, we report the average performance over 100 episodes, the standard deviation and the conversion accuracy. As en- and decoding methods for the normalization experiment, we use constant input currents, potential outputs, and a simulation time of 100 time steps. The performance of the original DQN is -175.4 (std: 100.92) and of the original classifier it is -186.51 (std: 107.80).

Conversion Method	**DQN Conversion** AVG (STD), ACC	**Classifier Conversion** AVG (STD), ACC
Poisson input & spike output	–	-169.16 (20.63),84.59%
Poisson input & potential output	-225.7 (49.18), 57.09%	-166.33 (12.23),86.30%
Fixed number of equidistant spikes & spike output	–	-334.14 (167.88), 84.67%
Fixed number of equidistant spikes & potential output	-430.65 (121.75), 12.44%	-314.6 (166.06), 84.83%
Constant input current & spike output	–	-316.83 (171.39), 94.38%
Constant input current & potential output	-180.32 (57.51), 64.38%	-278.92 (168.65),93.89%

Table 4.8: Comparison of En- and Decoding Methods for MountainCar. The SNN is converted using robust normalization ($p = 0.99$) and simulated for 100 time steps for each of the experiments. Reported are the 100 episode average performance, the standard deviation, and conversion accuracy. The performance of the original DQN is -175.4 (std: 100.92) and of the original classifier it is -186.51 (std: 107.80).

Backpropagation with Surrogate Gradients

Analog to CartPole (see section 4.3.1), we train a DQN and a DSQN 5 times with the same set of hyperparameters except for a slightly different architecture. Compared to the previous sections we again use a different set of hyperparameters for the same reasons as in CartPole. Hyperparameters are reported in table B.8, MountainCar B/C describe the non-spiking networks and MountainCar D/E describe the corresponding spiking versions. From table 4.10, one can see that training the

Simulation Time	DQN Conversion AVG (STD), ACC	Classifier Conversion AVG (STD), ACC
10	-500.0 (0.0), 0.0%	-500.0 (0.0),0.0%
50	-463.51 (72.71),5.68%	-417.34 (145.63), 81.21%
100	-206.27 (102.24), 51.65%	-278.92 (168.65),93.89%
500	-147.34 (31.61), 96.56%	-199.75 (135.57), 99.49%
1000	-169.5 (89.29), 98.36%	-188.33 (107.70), 99.71%
5000	– (–), 99.56%	– (–), >99.9%
10000	– (–), 99.79%	– (–), >99.9%

Table 4.9: Influence of Simulation Time on Conversion for MountainCar. All simulations use robust normalization ($p = 0.99$), constant input currents and potential outputs. Reported are the average performance over 100 episodes, standard deviation, and conversion accuracy. The performance of the original DQN is -175.4 (std: 100.92) and of the original classifier it is -186.51 (std: 107.80). For 5000 and 10000 simulation time steps, simulating for 100 episodes was infeasible with the available computation power, instead only conversion accuracy over a smaller number of episodes is reported.

DSQN using a higher learning rate of 0.001 works much worse than conventional training. However, when lowering the training rate to 0.0005, our DSQN successfully reaches the relaxed Open AI Gym standard several times.

Network	MC B	MC D		MC C	MC E
Spiking	No	Yes		No	Yes
Learning Rate	0.001	0.001		0.0005	0.0005
Gym Standard reached?	3 out of 5	0 out of 5		3 out of 5	3 out of 5
At Episode (on average if gym standard reached)	713.67	–		802.33	864.00
Best 100-episode average (average over five runs)	-120.13	-137.52		-123.85	-130.05
At Episode (average over five runs)	820.20	710.80		709.00	891.40

Table 4.10: Comparison of 5 training runs each for a DQN and a DSQN trained using Backpropagation with surrogate gradients for MountainCar (MC) and two different learning rates. All trainings are run for 1000 episodes.

4.3.3 Breakout

With CartPole and MountainCar we have shown that all three conversion methods as well as direct training with surrogate gradients principally work. We now test some of the methods on the more complex Breakout problem.

Small Network

Following the paper Patel et al. [66], we start with training a small fully connected network which consists of an input layer of 6400 neurons, one hidden layer with 1000 neurons, and an output layer

with 4 neurons. We use the grayscale preprocessing described by Patel et al. (see section 4.1.3) and the same hyperparameters as Patel et al. and Mnih et al. [53] reported in B.3. We train a DQN and a DSQN (simulation length 20) one time each and report the results in figure 4.11.

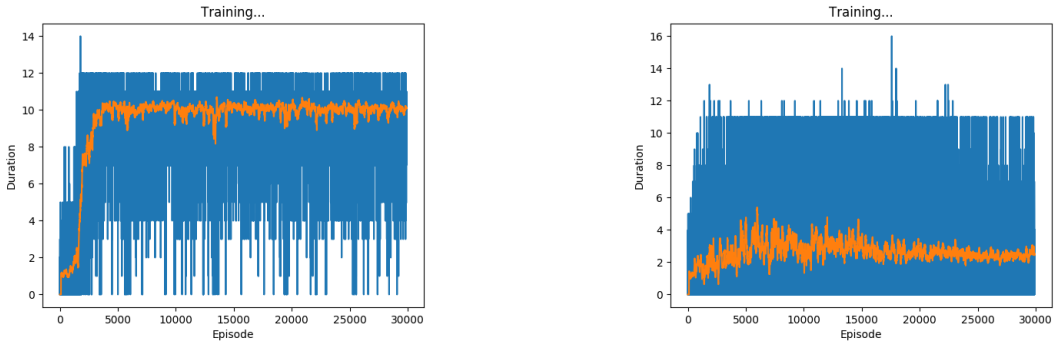

Figure 4.11: Training process of the small fully-connected DQN (left) trained according to Patel et al. [66] and an analog DSQN (right) trained using surrogate gradients. Blue shows individual episode performance and orange shows the average over the last 100 episodes.

The DQN reaches a peak 100-episode average performance of 10.7 during training and stays around that value starting approximately from episode 4000. This is similar to the results reported by Patel who reports a 100-episode average of 9.32. A random agent achieves only an average of 0.99 (std: 1.02). When loading the agent with the best performance during training using an epsilon-greedy policy with $\epsilon = 0.1$, it reaches an average score of 10.12 (std: 1.67). Unfortunately, we found that our agent learns the policy of hitting the left border and then staying there. At the border, it is likely to hit the ball a lot because it is often spawned such that it first arrives near one of the bottom corners of the screen. This is not really an intelligent behavior. Patel et al. do not report how their agent behaves so we cannot compare against their solution. However, they report a big drop of performance when obscuring the bottom left part of the screen, while obscuring other parts of the bottom screen does not lead to such big performance drops. This might indicate that the bar often is at the bottom left corner and obscuring it confuses the agent.
Nevertheless, we convert the network using direct conversion with robust normalization ($p = 0.99$), reset-by-subtraction, constant input currents, potential outputs, and 100 simulation time steps. After conversion, the performance is 10.50 (std: 1.16) with a conversion accuracy of >99.9%. The high accuracy can be attributed to the network strongly preferring to move left in any situation. This results in a relatively large difference of the Q-values where "Move Left" has the highest value. Thus, the network is easy to convert.
The DSQN training looks much worse on first view. The best 100-episode average it reaches is only 5.34. However, the agent actually first learns the same strategy of always moving left at around 6000 episodes where the best average is reached. This becomes apparent when we load the agent with the highest performance which analog to the DQN always moves left, thereby achieving an average score of 10.16 (std: 1.73). This is the same score as the best DQN achieves (small deviation due to stochasticity from the environment). After learning this strategy, the DSQN agent discards it but fails to learn a superior strategy.
The results from our Breakout experiments carry little significance as no intelligent behavior is

learned and the score falls short of the results achieved by deep convolutional networks as trained by Mnih et al. [53]. In our code repository A we provide first steps towards training and converting a network using the same hyperparameters as Mnih and encourage to complete these experiments as future research.

4.4 Policy Gradient Methods

From our observations on DQN conversion we think that networks with a softmax output layer, in this case a classifier, are easier to convert than DQNs, see section 5.2.3. Easier in this case means that the SNN needs to be simulated for fewer time steps to obtain good results and is more robust to different encoding, decoding and normalization methods. To test this assumption we convert a deep RL network trained with the policy gradient method which also uses a softmax output layer.

4.4.1 CartPole

We test the vanilla policy gradient algorithm (see section 2.2.3) on CartPole. For fair comparison, the architecture of the network is the same as for the DQN. Training hyperparameters are reported in table B.2. The network successfully reaches the Open AI Gym standard after around 600 episodes and upon loading, the agent achieves an average performance of 434.00 (std: 49.22). The training process is shown in figure 4.12. To check our assumption that policy gradient methods are easier to convert, we perform a direct conversion of the network and report its performance for different simulation times, see table 4.11. The conversion uses robust normalization ($p = 0.99$), reset-by-subtraction, constant input currents (encoding) and potential outputs (decoding). It can be seen that the policy gradient agent indeed converts well. This is further discussed in section 5.2.3. Unfortunately, in our experiments, the vanilla policy gradient method was not able to solve any of the two more complex environments, MountainCar and Breakout.

Simulation Time	**Conversion Performance (Direct Conversion)** AVG (STD), ACC
10	500.00 (0.00), 79.27%
50	467.78 (43.53), 83.28%
100	462.15 (39.27), 89.37%
500	438.04 (47.01),97.76%
1000	– (–), 98.98%
5000	– (–), 99.60%
10000	– (–), 99.86%

Table 4.11: Conversion of a policy gradient network for the CartPole environment. The performance of the original DQN was 434.00 (std: 49.22) and the performance of a random agent is 20.96 (std: 10.99). Conversion uses robust normalization ($p = 0.99$), constant input currents, potential outputs and reset-by-subtraction. For longer time periods, simulating 100 episodes took too long, so that we only report the conversion accuracy for a smaller number of episodes.

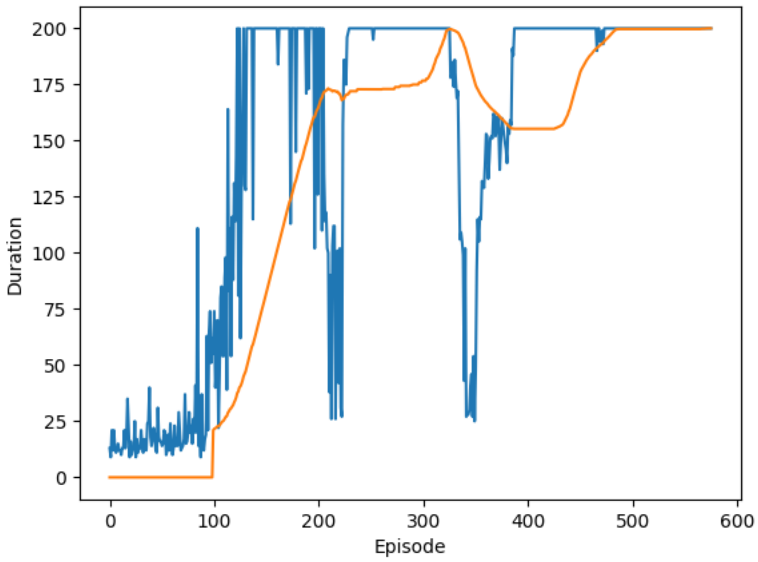

Figure 4.12: Training process of the vanilla Policy Gradient method on CartPole. Blue shows individual episode performances, orange shows the average performance over the last 100 episodes.

4.5 NEST, PyNN and SpiNNaker

In this section, we first investigate how to run some of the networks with the neural network simulator NEST [46]. NEST does not offer using the exact same neuron models as specified in the papers for conversion and surrogate gradients. Instead, we use the models "*iaf_psc_delta*" and "*pp_psc_delta*". The "iaf_psc_delta" model is like the LIF neuron using the reset to zero mechanism (see section 2.3.1). We also use a more complicated model, "pp_psc_delta", that uses *adaptive thresholds* which we use to implement the reset-by-subtraction mechanism. Additionally, "pp_psc_delta" neurons fire *stochastically*, that means instead of firing when the threshold is crossed, they fire at any time with a probability that is dependent on the current potential. We use the more complicated model only for the hidden layers as we use no threshold for the output layer neurons.
We set the hyperparameters reported in table B.4 and B.5 to model the dynamics of a non-leaky IF neuron as close as possible. Most importantly, we set the membrane decay potential tau_m very high, such that the membrane shows almost no decay and thus behaves like a non-leaky IF neuron. In case of the iaf_psc_delta neuron, all other parameters (except for the reset mechanism) can be set equivalently to the non-leaky IF neuron. For the pp_psc_delta model, we need to set additional parameters, including the time constant of the threshold, the dead time, and the stochastic firing parameters to approximate a non-leaky IF neuron. The threshold time constant q_sfa is set high, such that the threshold almost instantly adapts after it is crossed. The dead time is set as low as

possible. The parameters c_1,c_2, and c_3 describe the firing probability P_{fire} in the following way [47]:

$$P_{fire} = 1 - \exp(-r * h) \tag{4.3}$$

$$r = ReLu(c_1 * V' + c_2 * exp(c_3 * V')) \tag{4.4}$$

In the equation, h describes the constant simulation time step (a hyperparameter in NEST), r is called the rate, $ReLu$ is the ReLu function as described in equation 2.1, and V' describes the effective membrane potential. For simplicity we abstain from explaining the effective membrane potential. Important to know is only that crossing the threshold in a non-leaky IF neuron is equivalent to $V' \geq 0$. Therefore, our goal of approximating a non-leaky IF neuron is to have a high firing probability for $V' \geq 0$ and a low firing probability for $V' < 0$. We achieve this by setting c_2 and c_3 high (10.0 and 5.0 respectively) which leads to firing probabilities close to one for V' greater or equal to zero. Unfortunately, firing probabilities of values smaller than 0 now also have high firing probabilities. To correct this, we set c_1 much larger than c_2 (10^{10}). This brings down the firing probability drastically for all values smaller 0, except the ones extremely close to 0.

We load both the converted networks (using robust normalization, constant input currents, potential outputs) and the networks trained using surrogate gradients into NEST and report the results in tables 4.12 and 4.13 respectively. The results for conversion using iaf_psc_delta neurons are discussed in detail in section 5.2.4, while the pp_psc_delta model and loading surrogate gradient trained networks in NEST are discussed in section 5.5.

Another neural simulator, PyNN [4], provides an interface to interact with several neural simulators as backend, including NEST and the neuromorphic hardware SpiNNaker [63]. In our code repository (see Appendix A), we provide a PyNN script using the native NEST neuron type iaf_psc_delta. As it uses the NEST backend, results are similar to table 4.12 and are not reported. Unfortunately, as of now, PyNN does not provide any native PyNN neuron types with delta shaped synaptic currents and simple membrane dynamics. The advantage of native PyNN neuron types is that they are compatible with all the different backends. We experimented, unsuccessfully, to simulate the LIF neuron with delta synapses (or iaf_psc_delta) using exponentially decaying synapse currents and using the Izhikevich [36] model which uses delta shaped synapses but more complicated neuron dynamics. In the first case, the problem is that decreasing the synapse decay which approximates a delta shape also leads to the membrane potential increasing more slowly due to the inherent dynamics of the model. For the Izhikevich model, it is not possible to set the parameters in such a way that it approximates the more simple LIF dynamics. To resolve this, it is probably necessary to implement a new native PyNN model with delta shaped synaptic currents.
We also use PyNN to run some of our experiments using SpiNNaker neuromorphic hardware. A script to load converted networks on SpiNNaker is available from our code repository (see appendix A). We use the SpiNNaker model "IFCurDelta" which is a LIF neuron with delta shaped synapses, similar to the NEST model "iaf_psc_delta". Neuron hyperparameters are reported in table B.6. Unfortunately, simulation on the hardware takes a lot of time, either due to slow communication between our main machine and the SpiNNaker board, or due to the hardware being slow itself. Nevertheless, we repeat some of our NEST experiments on the hardware for a shorter duration of 5 episodes and display the results in table 4.14. Because of the short simulation times, the SpiNNaker experiments should be seen as a proof of concept rather than as meaningful results to compare the other experiments against. Yet, we discuss the results in section 5.5.

Environment	**Neuron Type**	**Algorithm**	**Simulation Time**	**Performance** AVG (STD), ACC
CartPole	Non-spiking	DQN-original	–	226.31 (32.74)
		Classifier-original	–	230.39 (66.27)
		PG-original	–	434.00 (49.22)
CartPole	iaf_psc_delta	DQN Direct	100ms	102.75 (3.10), 58.88%
			500ms	109.58 (3.34),60.53%
		DQN Indirect	100ms	201.97 (21.50), 94.73%
			500ms	224.34 (39.02), 98.69%
		PG Direct	100ms	462.59 (42.30), 88.70%
			500ms	440.63 (45.57), 92.76%
CartPole	pp_psc_delta	DQN Direct	100ms	318.18 (28.44), 78.10%
			500ms	453.20 (80.72), 78.76%
			5000ms	230.79 (42.79), 98.90%
		DQN Indirect	100ms	273.35 (79.49), 93.16%
			500ms	232.84 (56.26), 98.99%
		PG Direct	100ms	414.32 (42.60), 84.37%
			500ms	436.82 (47.10), 96.85%
MountainCar	Non-spiking	DQN-original	–	-175.4 (100.92)
		Classifier-original	–	-186.51 (107.80)
MountainCar	iaf_psc_delta	DQN Direct	100ms	-150.08 (31.18), 89.46%
			500ms	-137.90 (33.76), 92.86%
		DQN Indirect	100ms	-254.88 (163.28), 95.70%
			500ms	-203.78 (127.52), 99.28%
MountainCar	pp_psc_delta	DQN Direct	100ms	-500.00 (0.00),7.86%
			500ms	-157.5 (92.27), 95.31%
		DQN Indirect	100ms	-450.48 (117.94), 80.92%
			500ms	-221.84 (151.19), 97.84%

Table 4.12: Different networks converted in NEST. Reported is the average 100 episode performance, its standard deviation, and the conversion accuracy. Direct and indirect stand for the two conversion techniques (direct and via a classifier).

Environment	**Neuron Type**	**Simulation Time**	**Performance** AVG (STD), ACC
CartPole	SpyTorch IF	20 time steps	454.84 (81.29)
	iaf_psc_delta	20ms	500.00 (0.00),67.59%
	pp_psc_delta	20ms	500.00 (0.00), 66.25%
MountainCar	SpyTorch IF	20 time steps	-130.92 (27.90)
	iaf_psc_delta	20ms	-159.83 (69.07), 61.83%
	pp_psc_delta	20ms	-165.71 (51.89), 60.44%

Table 4.13: Performance of directly trained DSQNs in SpyTorch and NEST using two different neuron models. Further, the standard deviation (in brackets) and the similarity (conversion accuracy) to the SpyTorch network is given.

Environment	**Algorithm**	**Simulation Time**	**Performance** AVG (STD), ACC
CartPole	DQN Direct	100ms	42.20 (54.43), 61.61%
		500ms	33.80 (37.38), 58.58%
		5000ms	18.40 (10.16), 56.52%
CartPole	DQN Indirect	100ms	174.40 (5.13), 90.71%
		500ms	188.20 (9.63), 93.41%
CartPole	PG Direct	100ms	500.00 (0.00), 87.24%
		500ms	489.00 (22.43), 83.80%
CartPole	DSQN (SG)	20ms	288.2 (152.77), 74.25%
MountainCar	DQN Direct	100ms	500.00 (0.00), 19.36%
		500ms	227.8 (154.65), 16.42%
MountainCar	DQN Indirect	100ms	399.00 (148.50), 78.90%
		500ms	230.80 (150.54), 96.53%
MountainCar	DSQN (SG)	20ms	220.4 (157.29), 36.66%

Table 4.14: Simulation Results on SpiNNaker. We load some of the networks onto a SpiNNaker chip and simulate them using the model IFCurDelta (hyperparameters see table B.6). Direct and indirect stand for the two conversion techniques (direct and via a classifier). Because of the long runtime, average, standard deviation, and conversion accuracy are reported over only five episodes. For the original results compare with the NEST tables 4.12,4.13.

Chapter 5
Discussion

Before analyzing the results from the experiments, we first discuss comparability of algorithms and experiments in reinforcement learning in general. We then investigate how the different conversion methods compare, what their benefits are, and where their weaknesses lie. Next, we delve into the results from the experiments that deal with Backpropagation with surrogate gradients followed by comparing the different training methods, including STDP and eRBP which we analyzed only in the theoretical part. After this, we discuss the experiments using NEST, PyNN, and SpiNNaker. Finally, we discuss possible directions for future work.

5.1 Reproducibility and Comparability of Results

In general, reported reinforcement learning algorithms can be difficult to reproduce and even results across several trainings of the same algorithm on the same environment vary strongly. In this section, we outline key challenges in comparing algorithms and explain how we address them in our experiments. Henderson et al. [31] identify several sources of stochasticity and reproducibility issues:

1. Hyperparameters: The choice of hyperparameters including the network architecture and the scale of the rewards is crucial for the performance. Additionally, the correct choice of parameters is environment and algorithm specific such that no general solutions can be given. This impedes comparison of different algorithms as one algorithm can only be declared superior to the other if a thorough hyperparameter search has been conducted for both. On the other hand, it is desirable to find algorithms which perform well for a wide set of hyperparameters such that they become more robust to different initializations.

2. Random seeds: Usually, both environment and network weights are initialized randomly. The training process can be highly dependent on the initial parameters. Even when reporting the initial weights, randomness will always come from other sources because for example an environment with a deterministic starting position becomes easier to solve.

3. Stochasticity in the environment: Likewise, some environments are intrinsically stochastic. For example, the Open AI Gym version "Breakout-v0" of the Breakout problem repeats any action with a probability of 0.25 regardless of what the agent predicts.

4. Code bases: Henderson et al. show that performances of different code bases that are available online vary strongly. Our experiments confirm this. We found that implementations based on different machine learning packages (e.g. tensorflow [1] and PyTorch [65]) perform differently, even if the implementations superficially seem to be equivalent. This could be due to slightly different implementations or different standard parameters of background processes like the optimizer.

5. Reported evaluation metrics: Finally, reporting methods of the results can differ between authors. Some papers report the maximum or average performances of several runs, where one run can either be an online evaluation (of the training) or an offline evaluation (of the saved network). Both methods do not fully express the performance of the algorithm, Henderson et al. propose to use confidence intervals instead.

The described issues make it difficult to compare different algorithms without applying them across a set of several diverse environments and repeating both training as well as evaluation runs multiple times. Our thesis should be seen as an explorative work which investigates several algorithms and shows proofs of concept rather than an in depth comparison of the different algorithms. Nevertheless, we encourage reproducibility by reporting all hyperparameters including initial network weights, seeds, and software versions in our code repository (see Appendix A). Further, we report averages and standard deviations for our conversion experiments instead of only maximum performances and repeat experiments several times to compare Backpropagation using surrogate gradients and standard Backpropagation.

5.1.1 Reproducibility across Frameworks

Another problem we frequently encountered during our work is transferring algorithms across frameworks. Although most reinforcement learning papers use Python based implementations, different machine learning frameworks are popular, including, but not limited to, PyTorch [65] and tensorflow [1]. As one of our central experiments, Backpropagation with surrogate gradients, is based on PyTorch, we implemented all our algorithms, as far as possible, in the same framework to ensure good comparability. However, this approach limits progress as many implementations are available exclusively in tensorflow. These algorithms often do not behave in the same way when ported to PyTorch, even when setting the hyperparameters identically. This is probably due to fine implementation differences of the two frameworks which can influence the agents behavior. Retraining algorithms on PyTorch therefore becomes an unnecessary high amount of work if good tensorflow frameworks would be readily available. This especially is the case for Open AI Gym where a strong set of baseline algorithms is included in the package stable baselines [15]. For future work, therefore, it would be beneficial either to have an algorithm which can convert models between PyTorch and tensorflow, or to start a similar baseline package for Open AI Gym using PyTorch, or to write a tensorflow version of the algorithm used in SpyTorch [61]. Alternatively, algorithms trained with surrogate gradients using SpyTorch could simply be compared against models trained with tensorflow and converted in the following. Although this reduces comparability, comparison would still be meaningful as Backpropagation using surrogate gradients behaves differently from standard Backpropagation anyway. Furthermore, this would allow to use a maximum set of algorithms as many SNN frameworks build on PyTorch, while the baseline gym algorithms are implemented in tensorflow.
Another source of discrepancies are the different SNN frameworks. Often, they do not support the same neuron models. E.g. in SpyTorch we use non-leaky IF neurons, while in NEST only leaky IF neurons are available. In PyNN there are also LIF neurons but not with delta shaped synapses and so on. Another problem is that when using a native NEST model in PyNN, the results are not automatically equivalent to NEST because internal hyperparameters are set differently. These discrepancies, combined with poor documentation of some of the simulators make it difficult to implement algorithms which are exactly equivalent. For this reason, we suggest to focus on one particular simulator or, ideally, directly on neuromorphic hardware implementations as the goal

should be to run spiking neural network algorithms on hardware to exploit their advantages. The approach we followed in this thesis, using our adaptation of SpyTorch, then porting it to NEST, PyNN and finally SpiNNaker is not productive if the goal is to obtain energy efficient neuromorphic hardware implementations as it requires a lot of unnecessary work.

5.2 Conversion

In the experiments chapter, we compare the three different conversion methods (direct, indirect, indirect using a SNN classifier to mimic the policy), suggested in section 2.3.5. Additionally, we analyze, how different normalization, en-, and, decoding methods compare against each other and how the performance changes with different simulation lengths for the first two methods (direct and conventional classifier conversion).

5.2.1 Encoding

We start with comparing the different encoding strategies. Poisson and Equidistant encoding schemes rely on normalized input data for which different ways of normalizing exist. As a first method, one could normalize the inputs already when training the DQN using prior knowledge of the environment. Secondly, one could keep a dynamic list of minimum and maximum values during DQN training and normalize the values according to the so far seen states. Thirdly, one could use two input neurons, one for positive and one for negative values and cut of values higher than one or lower than minus one. Fourthly, but only for indirect conversion, one could normalize the input data before training the classifier using knowledge obtained from the DQN agent's replay memory. The disadvantages of the described methods are, respectively, relying on prior knowledge and potentially changing the training process (first method), changing input values during the training which probably makes effective learning infeasible (second method), being limited to problems where most inputs are between -1 and 1 (third method), and relying on the majority of possible inputs being saved in the replay memory (fourth method). In our experiments for CartPole and MountainCar, we only report the third method. It seems reasonable to apply it here as all good states for CartPole are between -1 and 1, while for MountainCar almost all the states are within the limits. A quick test of the fourth method (not presented) suggests that it also works well, but obviously it is limited to conversion of the classifier.
The constant input current encoding has a huge advantage over Poisson and Equidistant encoding because it has no constraints on how the inputs look like. That alone makes it superior to the other methods and because of that in all other experiments except en- and decoding we only report results using this method.
However, in CartPole Poisson encoding achieves far superior results to constant input currents (both for direct and indirect conversion, see table 4.3). This suggests either that the randomness introduced by the Poisson process can improve the performance or that cutting off the values makes the agent react better to positions and speeds far off the center. However, as equidistant spikes do not perform equally well, cutting off probably does not improve the performance. On the contrary, it seems to weaken the performance as constant input currents perform better than equidistant spikes.
In terms of conversion accuracy, when converting the classifier, constant input currents produce the network most similar to the original classifier and equidistant spikes the least similar. For the

DQN conversion all methods perform similarly poor for a simulation length of 100 time steps.
In the MountainCar problem (results presented in table 4.8), Poisson encoding is the best performing method for the classifier conversion as well, but for the DQN constant input currents clearly outperform it. In terms of accuracy, constant input currents are more similar to the original network than Poisson spike trains which are more similar than equidistant spikes. When converting directly, the performance (and the conversion accuracy) of equidistant spikes is an anomaly, as it performs poor compared to the other methods. We think this is due to the fact that values close to 0 (especially for the speed) deterministically lead to 0 spikes in the input neurons, regardless whether the speed is positive or negative. However, whether the speed is positive or negative can make a major difference for the next action to be taken. Poisson encoding does not suffer from this problem as strongly as it can still stochastically emit spikes for those values.
To summarize, the input method constant input currents is the most generic method and produces the most similar network to the one which it is converted from. This is also in line with the results from Rueckauer et al. [72] for classification tasks. However, Poisson processes can improve the performance of the converted network due to the introduced noise. This noise can potentially compensate weaknesses from the original network (CartPole DQN, CartPole classifier, and Mountain Car classifier), but it can also lead to worse performance (MountainCar DQN). Equidistant spikes is the worst performing method in our experiments.

5.2.2 Decoding

We compare two different decoding methods: Taking the argmax function of the spikes in the output layer to determine the action and using the argmax of the potentials where the output layer does not produce spikes. Results are reported in tables 4.3 and 4.8. When performing indirect conversion (of a classifier), both methods lead to similar performance and conversion accuracy.
When converting the DQN trained on CartPole (see table 4.3), spikes strongly outperform potentials for all encoding methods. On further investigation, this turned out to be an implementation quirk which is specific on the network that is converted. What happens is that the converted DQN using potential outputs usually drives out of the screen to the right. The spike method often produces the same number of spikes, because the Q-values calculated by the original DQN are very close to each other and the spike output method does not have enough resolution to capture this difference (at least for a short simulation time). Now, the argmax function from the torch library deterministically picks the first element if several elements have the same value. This biases the network to move to the left (action 0) and therefore compensates the behavior of the spiking DQN which tends to drive out of the screen to the right.
For the MountainCar problem we did not use the spike method because it would need to be adapted to produce spikes for negative neurons (compare section 4.3.2) and the results from CartPole do not look promising.
Additionally, we tested the output method "time-to-first-spike" where the first spiking output neuron determines the action. This method does not produce useful results (not presented). This could be due to the first spike being strongly influenced by the bias of the output neuron which makes it independent of the actual input.
To summarize, the choice of the decoding method, spikes or potentials, does not make a difference when converting a classifier. However, when converting a DQN, output spikes do not offer enough resolution to distinguish between values which are very similar. Additionally, using the potentials is a more generic method as it can represent negative Q-values. This makes potential outputs

superior and we stick to this method for the remaining experiments. Again, our results for DQNs are equivalent to the ones from Rueckauer et al. [72] for classification networks.

5.2.3 Simulation Length

In tables 4.4, 4.9, and 4.11, we compare the influence of simulation time on the performance and conversion accuracy. For all experiments the conversion accuracy increases with longer simulation times and gets close to 100% for long enough simulations. This experimentally shows the results from Rueckauer et al. [72] that a spiking network using their conversion method yields an arbitrarily close approximation of a non-spiking network for sufficiently long simulation times. However, how fast the conversion accuracy rises depends on the network being converted. The classifiers (tables 4.4, 4.9) reach a conversion accuracy well over 90% for 100 time steps already and a conversion accuracy of around 99% for 500 steps.
Converting a Policy Gradient method (see table 4.11) is slightly less effective ($\approx 90\%$ and $\approx 98\%$ accuracy for 100 and 500 time steps respectively).
In contrast, the DQNs (tables 4.4, 4.9) need a considerable longer simulation time (51.56% and 96.56% accuracy on MountainCar and 60.72% and 81.61% on CartPole for 100 and 500 time steps respectively). We assume this is due to the loss function which is a cross entropy loss for both the classifier and policy gradient network whereas it is a temporal difference loss for the Q network. We suggest the following explanation for the different behaviors: The cross entropy loss of a classifier or a policy gradient network in a deterministic environment (one set of input values corresponds to exactly one state) encourages the network to calculate a high value for the correct output neuron and a low value for every other neuron for each input. The DQN, on the other hand, attempts to calculate the correct Q-value for each output (action) neuron. These values can be very similar. Now, when converting a network it is easier (only needs short simulation time) to distinguish values correctly which are far apart than values that are close together.
Contrariwise, one could argue that if the DQN learns the true distribution of the environment similar Q-values would mean that it does not matter which action to take. Although this is true, in our experiments it does matter which action is taken when the Q-values are similar. This suggests that the DQN was not able to learn the underlying distribution but is still able to produce good results.
As a side note, when using policy gradient on a stochastic environment (the same input values can encode several different states) converting becomes much more difficult as after conversion the network is able to produce the action with the highest probability but not the correct probability distribution. This is due to the fact, that state-of-the-art conversion methods focus on classification problems where it is sufficient to produce the correct classification which is possible without reproducing the same values for the output neurons computed by the original network. Stochastic policies, however, rely on computing exact probabilities. Likewise, the converted DQN does not produce the same Q-values as the original network anymore but keeps only the relative order of the Q-values. The conversion methods we investigated cannot maintain exact values.
In terms of performance, the CartPole DQN and the classifiers perform better with longer simulation times, while the MountainCar DQN reaches an optimum at around 500 time steps and the Policy Gradient network has its optimum around 10 time steps. This shows that a converted network which does not exactly equal the network it is converted from can potentially both improve or degrade the performance of the agent, specific on the environment and the network that is converted.

5.2.4 Resetting Method

To compare the resetting methods, reset-by-subtraction and reset-to-zero, we compare the results from the simulation time experiments (see tables 4.4,4.9,4.11 and previous section) with the results from the conversions in NEST (see table 4.12). The first three tables use the reset-by-subtraction method, while the iaf_psc_delta neurons in NEST use reset-to-zero. Note that the neuron models are not exactly equivalent as NEST uses LIF neurons, but the NEST model approximates the non-leaky neurons in the other experiment because the time constant is extremely large. Otherwise, the models are equivalent such that the comparison is meaningful.
The conversion accuracies for converting a classifier are similar for both resetting methods (94.73% and 98.69% (reset-to-zero) against 95.37% and 98.95% (reset-by-subtraction) for 100 and 500 time steps for Cartpole, and 95.70% and 99.28% (reset-to-zero) against 93.89% and 99.49% (reset-by-subtraction) for MountainCar).
For direct conversion of the CartPole DQN however, the values for 100 time steps are similar (58.88% (reset-to-zero) against 60.72% (reset-by-subtraction)), while for 500 time steps the accuracy (and performance) for reset-by-subtraction increases (81.61%), but for reset-to-zero it stays low (60.53%). Interestingly, for MountainCar the values for reset-to-zero for 100 time steps are much higher compared to reset-by-subtraction (89.46% against 51.65%), but then do not increase much for reset-to-zero when going up to 500 time steps, while shooting up for reset-by-subtraction (92.86% against 96.56%). The fact that increasing the simulation time does not increase the conversion accuracy for reset-to-zero becomes especially clear when simulating the CartPole DQN for long simulation times (e.g. 10000 time steps). Even then, the conversion accuracy does not increase. This happens because reset-to-zero introduces a systematic error [72] into the simulation which does not disappear with longer simulation times. On the contrary, reset-by-subtraction converges to the original network behavior when given sufficient time.
For Policy Gradient reset-to-zero works slightly worse but still comparable to reset-by-subtraction. To summarize, reset-to-zero introduces a systematic error into the conversion process which cannot be avoided by longer simulation time (according to Rueckauer et al. [72] it could be avoided by splitting inputs into smaller packages distributed over several time steps equivalent to increasing the resolution, but we did not implement this). Reset-by-subtraction does not suffer from this problem. The conversion accuracy of the classifiers and the policy gradient network is hardly affected by switching reset mechanisms, while the accuracy for the DQN for CartPole degrades strongly. This further strengthens the results from the previous section that converting a network trained with cross entropy loss is more robust than converting a DQN.

5.2.5 Normalization Method

We tested the normalization methods model-based, data-based and robust with results being presented in table 4.2 for CartPole and in 4.7 for MountainCar. Rueckauer et al. [72] report that robust normalization works best for a p-percentile in the range [99.0, 99.999]. We test robust normalization with 99% and 95%. In CartPole, model-based normalization works non significantly worse for direct conversion than the other three methods, while when converting the classifier, it works non significantly better. However, all methods perform very similar and no clear pattern emerges. In DQN conversion of MountainCar, model-based and robust normalization with a 95% p-percentile work considerably worse than the remaining methods. Data-based normalization non significantly outperforms robust normalization with 99% p-percentile in terms of conversion accuracy but is slightly worse in terms of performance. When converting the classifier, no significant differences

between the normalization methods emerge.
All-in-all, our experiments are not significant. However, we assume that if the replay memory is large enough, so that a representative subset of input states is captured, robust normalization with a high p-percentile works best as reported by Rueckauer. For the remaining experiments we therefore only use robust normalization with a 99% p-percentile.

5.2.6 Direct vs Indirect Conversion

In section 2.3.5, we suggest three different methods for conversion:

1. Direct conversion
2. Indirect conversion: Training a classifier to learn the policy from replay memory, then convert the classifier.
3. SNN classifier: Train a SNN using Backpropagation with surrogate gradients to learn the policy from replay memory.

We start with comparing the first two methods. From the previous sections we have already seen that converting classification networks is more robust compared to converting DQNs. Explicitly, much shorter simulation times are needed to achieve good conversion accuracy and the conversion is robust against changing the resetting method.
However, the classifier network behaves differently from the DQN. While in CartPole, the classifier performs as good as the DQN it was trained from (see table 4.1), in the MountainCar problem the DQN considerably outperforms the classifier both before and after conversion (see table 4.6) even though the conversion accuracy is lower (see table 4.9). This could either be because the replay memory from which the classifier learns is not representative for all states or because the cross entropy loss leads to a slightly different distribution where the best accuracy is not the same as the smallest loss (compare section 4.3.1, paragraph DQN-SNN Indirect Conversion).
Additionally, training the classifier incurs a small runtime overhead because the DQN agent needs to be simulated to save the replay memory and the classifier needs to be trained. Compared to the DQN training time, this overhead is relatively small.
Training a SNN classifier on the replay memory has the advantage that the simulation time can be set much shorter as the network adapts to it during training. However, it has the same potential weaknesses as training a conventional classifier and the time needed for training is considerable larger.

5.2.7 Summary

When converting DQNs, we have seen that direct conversion relies on long simulation times during inference and does not work well when using reset-to-zero.
First training a classifier to learn the policy of the DQN and then converting the classifier has the advantage of needing shorter simulation times and being robust against changing the resetting method. However, the classifier does not necessarily learn a policy which is equivalent to or as good as the DQNs and training incurs a small runtime overhead.
Training a spiking classifier can bring down the inference time even further, but suffers from the same weaknesses as the conventional classifier and incurs a larger overhead in training time.
Our experiments suggest that converting a network trained by a Policy Gradient method directly

is as easy as converting a classifier and therefore spiking networks converted from Policy Gradient methods could turn out to be superior to networks converted from DQNs. However, the converted Policy Gradient network does no longer represent a probability distribution over the actions but only determines the best action (with the highest probability). This cancels out the advantage PG methods have over DQNs in stochastic environments. Similarly, the converted DQN does no longer predict the correct Q-values but only the best actions according to its policy. Note that this observation holds independent of the used conversion method.
We did not find significant differences between normalization methods, except that model-based and robust normalization with a small p-percentile perform worse for direct conversion of the MountainCar DQN. The results from Rueckauer et al. [72] imply that robust normalization using a p-percentile greater 99% but smaller than one works best.
Concerning the different encoding strategies, constant input currents are the most generic method. However, Poisson spike trains can outperform it if it is possible to apply them and the agent benefits from the introduced noise. The choice of the decoding method does not matter when converting a classifier, while when converting a DQN potential outputs are strongly preferred. Finally, reset-by-subtraction is theoretically superior to reset-to-zero [72] which is confirmed in our experiments. Conversion of a classifier or Policy Gradient network, however, is robust against changing the resetting method, while converting a DQN only works well with reset-by-subtraction.

5.3 Backpropagation with Surrogate Gradients

To compare direct training of a SNN using surrogate gradients with conventional Backpropagation, we train a DSQN and a DQN five times each on the CartPole (see table 4.5) and MountainCar (see table 4.10) environments. For the Breakout problem we run the training only once (see figure 4.11). For one environment we always use the same set of hyperparameters.
Using the same hyperparameters has the advantage that if suitable hyperparameters are known for a DQN, they can be used without adaptation on the DSQN. At the same time, performance under the same hyperparameters can be observed. On the other hand, hyperparameters that are optimal for non-spiking networks are not necessarily the best choice for spiking networks. It would be better to compare their peak performance which means comparing each method using the best possible set of hyperparameters. In order to obtain the optimal parameters one would need to run an extensive hyperparameter search which is outside the scope of this thesis.
In CartPole, the DQN compared to the DSQN reached the Open AI gym standard more often (three times compared to only once) and the DQN reached a higher 100-episode average of 197.79 compared to 190.94. However, the DSQN reached its best average after viewer iterations (468.6 on average compared to 784.2) which suggests that DSQNs learn faster and might profit from stabilizing the training. This led us to re-run the experiments using a lower learning rate. However, when using a lower learning rate, the gym standard was reached as often as before for the DSQN, while the best 100-episode average improved slightly. The DQN only reached the Open AI Gym standard once with the lower learning rate, while the 100-episode average remained unchanged. Another way to improve stability is to increase the target update frequency or generally experiment with different hyperparameters.
Analog to CartPole, using the higher learning rate (0.001) in MountainCar leads to a higher success rate in DQN training (3 out of 5, best 100-episode average of -120.13) compared to DSQN training

(never reached gym standard, best 100-episode average of -137.52). When lowering the learning rate to 0.0005, this time DSQN training improved significantly (3 out of 5, best 100-episode average of -130.05), while DQN training got marginally worse (3 out of 5, best 100-episode average of -123.85).
In Breakout, the DQN appears to perform much better than the DSQN on first sight (see figure 4.11). However, when examining closer by loading the best performing agent of each training, those agents follow the exact same strategy. It turns out that the DQN learns a strategy with an average performance of around 10 and then sticks with it, while the DSQN learns the same strategy, but then discards it and fails to learn a superior policy. Unfortunately, none of the agents learn an intelligent strategy such that our results from Breakout are not meaningful (compare section 4.3.3).
When loading the trained DSQN in CartPole and MountainCar (see tables 4.1 and 4.6), the average performance is much higher compared to the DQN: 454.84 against 226.31 for CartPole and -130.92 against -175.40 for MountainCar. This suggests that although the DSQN is more difficult to train it generalizes better.
A huge disadvantage of training a DSQN with surrogate gradients is the long runtime. Even using a relatively short simulation time of 20 time steps increases the run time considerably.

5.4 Comparison of Training Methods

After analyzing our experiments, we now compare the different training methods against each other, as well as against the methods discussed in the theoretical background part. In terms of performance, directly trained networks using SpyTorch achieved higher scores and therefore better generalization than conventionally trained DQNs. However, the training itself becomes more unstable and thus difficult.
A further disadvantage of SpyTorch is that it increases the training time as in each prediction the network has to be simulated for a fixed number of time steps, where each time step takes roughly as much time as a forward pass through a conventional neural network. This is because in every time step spikes of each layer are multiplied with their corresponding weights (analog to a forward pass in the DQN) and additionally, the membrane and synapse potentials are updated.
On the other hand, SpyTorch allows to successfully train the network with a small simulation time (we used 20 time steps), while converted networks need to be simulated for a longer period. Further, in SpyTorch a regularization term can be used which encourages sparse spikes [61].
Even though we did not measure energy consumption, it is clear that SpyTorch will use more energy than converted SNNs in training (longer training time) but less energy during inference (shorter simulation times).
Conversion methods have the additional advantage that they might improve robustness [66]. Additionally, in some of our experiments they improve the performance of the original network when using certain en- and decoding methods and a certain number of simulation time steps. For example, the CartPole classifier could be improved by using Poisson encoding (see table 4.3), while the MountainCar DQN performs better when simulating it for 500 time steps, but it does not perform better when simulating it for 100 or 1000 time steps (see table 4.8). Unfortunately, from our work no clear pattern emerges which encoding and decoding methods or simulation lengths are favorable, so we assume it is problem dependent and even dependent on the specific network which is converted. However, even when conversion improves the performance of the DQN, it still falls short of the performance of a directly trained DSQN.
The two methods we analyzed in the theoretical part (eRBP and STDP) have the advantage that

they can be used to train a network directly on neuromorphic hardware and could therefore make the training energy-efficient. The framework suggested in [56] seems especially promising as they use only a single spike per neuron. This makes these methods far superior in theory. However, whether they can be used to efficiently train a deep neural network for reinforcement learning is yet an open question.

5.5 NEST, PyNN and SpiNNaker

We tested two different NEST neuron models to load and simulate the DSQNs obtained through conversion and direct training. In the discussion on resetting methods for conversion (section 5.2.4), we already analyzed the iaf_psc_delta neuron when loading the converted network. When directly converting the DQN for CartPole in NEST, this model exhibits a systematic error which cannot be reduced by increasing the simulation time. The pp_psc_delta model in NEST can imitate reset-by-subtraction by increasing the firing threshold after a spike occurs instead of resetting to zero. This makes it a promising candidate to mimic our Non-Leaky IF neuron with reset-by-subtraction. However, pp_psc_delta is a stochastic firing model (it becomes more likely to fire when the potential approaches and eventually exceeds the threshold) which makes it much more complex than the simple IF model. We set the hyperparameters in such a way that the model approximates the IF model as closely as possible (see section 4.5) .
The results for the pp_psc_delta neuron are reported in table 4.12. Direct DQN conversion accuracy using CartPole improves to just below 80% for 100 and 500ms simulation time and likely converges to 100% when increasing the simulation time indefinitely (5000 ms already leads to 98.9%). For the other models the pp_psc_delta model works similarly well to the iaf_psc_delta model with two interesting artifacts. When running indirect conversion on CartPole for 100ms the performance improves significantly (to 273.35). This might be explained by the stochastic firing and could be a similar effect to Poisson spike trains (see section 5.2.1) where introducing noise improves the agents performance. In conversion of the MountainCar DQN and classifier, pp_psc_delta neurons perform worse than iaf_psc_delta neurons for the same runtime, but the issue should disappear when further increasing the simulation time.
Results of loading the directly trained network using surrogate gradients are presented in table 4.13. Both neuron types achieve only a relatively low similarity to the original network run in SpyTorch, between 60% and 70%. The performance on CartPole increases when simulating in NEST, while the performance on MountainCar decreases. The low conformity between the SpyTorch network and the version loaded in NEST suggests that the learned model in SpyTorch is very sensitive to changes made to the neuron model and the simulation. To get a higher similarity, the SpyTorch and NEST models as well as the simulation parameters would need to be compared thoroughly in order to set the NEST hyperparameters more carefully or to adapt the SpyTorch model already before the training process.
Apart from NEST, we use the neuron simulator PyNN [4] which provides a common modeling language for various backends. In our code repository (see appendix A), we provide a PyNN script which implements the iaf_psc_delta neuron native to NEST. This works well, although PyNN sets some of the simulation parameters different to NEST which results in slightly different results.
We also use PyNN to utilize a neuromorphic SpiNNaker [63] chip as backend using the SpiNNaker neuron model "IFCurDelta". As the runtime of our simulations is very long, either due to the communication between the two hardware components or due to the neuromorphic hardware itself

being slow, we run the conversion experiments only for five episodes each. Results are reported in table 4.14. Similar to our previous results, it can be seen that conversion of a classifier works relatively well ($> 90\%$ conversion accuracy for 500 ms), although worse compared to NEST and SpyTorch, and that increasing the simulation time increases the conversion accuracy. Conversion of the DQN does not yield good results and increasing the simulation length does not improve this. Like in NEST, this is probably due to the reset-to-zero mechanism. The policy gradient network converts well for 100 ms, but when increasing the simulation time, conversion accuracy degrades. This contrasts our previous results, that policy gradient and classifier networks behave similarly. Finally, loading directly trained DSQNs does not work very well and even worse than loading them in NEST.
However, all results in this section are not representative as simulating for only five episodes does not yield meaningful results considering the high standard deviations across all experiments. Our SpiNNaker tests should therefore be seen as a proof-of-concept rather than be compared on the same level with our remaining experiments. Further investigation of the simulation hyerparameters and neuron model in SpiNNaker is needed to improve the results.

In conclusion, simple neuron models as used by Rueckauer et al. [72] for conversion or as used in SpyTorch for direct training are not generally available in the much more sophisticated neural simulators NEST and PyNN or on SpiNNaker hardware. These offer many different models and various tunable simulation hyperparameters. We tested all the different simulators which prohibited us making in depth comparison of each. Likely, our results can be improved by more carefully analyzing the models and hyperparameters that need to be set. However, we show that all our converted networks can achieve good performance using NEST and the converted classifiers also yield good performance using SpiNNaker. Solely loading directly trained networks in SpyTorch does not result in high similarity although no conversion is taking place. This is probably due to high sensitivity of the networks trained with SpyTorch which makes a thorough comparison between the SpyTorch and NEST models and hyperparameters necessary.

5.6 Future Work

In this thesis, we investigated a variety of different techniques to train spiking neural networks, where for most of them little research in the reinforcement learning domain exists. This offers a wide range of possible future directions:

1. Further analysis of the methods investigated in the experiments section: We compared different encoding, decoding, resetting, and normalization methods for conversion and found a set of preferred candidates: Constant input currents, potential outputs, reset-by-subtraction, and robust normalization with a p-percentile greater 99%. These can be applied in every situation we encountered without adaptation, lead to the highest similarity of the converted network and usually to comparable results. However, our results suggest that Poisson inputs can improve the network due to noise being introduced. Which agents and environments benefit from noise would be an interesting topic for further analysis.
   Most neuron simulators use reset-to-zero instead of reset-by-subtraction so it could be further investigated for which cases reset-to-zero works well and for which cases it does not or how a DQN could be made robust against switching the conversion mechanism. A good starting point would be Hunsberger and Eliasmith [35] who suggest to introduce noise into SNN train-

ing (for classification) to make the network robust against errors introduced by conversion.
For the surrogate gradient experiments we conducted it would be of interest to compare the peak performance using the best hyperparameters for DQN and DSQN training. To find them, one would need to conduct a systematic hyperparameter search.

2. More advanced algorithms: We analyzed the DQN algorithm for spiking neural networks and conducted first experiments on conversion of policy gradient trained networks.
For DQNs we analyzed the vanilla DQN algorithm as well as double Q learning. In future work, further techniques of Q learning, like prioritized experience replay or dueling DQNs can be implemented. Conversion of these network should still work straightforwardly and direct training could be improved considerably. Apart from that, conversion and direct training methods could be investigated for policy gradient and actor-critic methods as well as more advanced reinforcement learning algorithms.
A second approach which we did not follow is to design new RL algorithms that are not one-to-one derived from the most successful algorithms using conventional, Backpropagation based neural networks. In this line of research Fremaux et al. [22], Aenugu et al. [2], and Potjans et al. [71] suggest actor-critic architectures based on STDP which they apply to relatively simple problems. In future research, these methods could be compared to the methods based on Backpropagation as well as extended to more complicated problems.

3. More complicated environments: We presented three different Open AI Gym environments on which we tested the algorithms. Two of them, CartPole and MountainCar, have a very low dimensional input and action space and are therefore very simple. The third, Breakout, is more complicated as it takes the pixels of an image as input channels, but the action space is still low dimensional. In a next step, one would need to verify the results on still more complex environments like complicated games from the Atari environment or real-world robotics task using much larger networks.

4. More complicated neuron models: We mostly analyzed non-leaky Integrate-and-Fire neurons as well as leaky IF neurons but with the parameters set in a way that they approximate the non-leaky version. As a first step towards more biological realistic models, it would be interesting to test true LIF neurons (SpyTorch supports this) and different synapse shapes to analyze how the performance changes. Later, more realistic models like Izhikevich or Hodgkin-Huxley (see section 2.3.1) could be investigated.

5. Local Learning: In chapter 2 and 3 we introduced algorithms that are capable of local learning and can therefore potentially learn on neuromorphic hardware. These algorithms are STDP and eRBP (see sections 2.3.7 and 2.3.8). In section 3.1.3, we outlined research that shows proof-of-concepts for supervised learning for both methods. It would be interesting to adapt these approaches to reinforcement learning algorithms like Q-learning.

6. Neuromorphic Hardware: We provided a proof-of-concept that our algorithms can be run on SpiNNaker chips. However, we encountered very slow performance, so it would be interesting to speed up runtime which would allow usage of the hardware for larger experiments. Furthermore, energy usage of the chip could be measured in order to compare against conventional hardware. Also, our algorithms only successfully converted the classifiers. In depth analysis of the SpiNNaker neuron models and simulation parameters is necessary to improve our results.

7. Robustness: Patel et al. [66] suggest that converting DQNs to spiking neural networks makes the network more robust. Their results on Breakout, where part of the screen is obscured by a black block, suggest that this holds. However, they test this only using a small non-competitive network and on only one problem. Further studies on a variety of problems are necessary to verify this claim. Galloway et al. [23] suggest that binary networks are more robust against attacks compared to conventional neural networks. This suggests that SNNs are more robust as well, due to their binary communication.

8. Dynamic Vision Sensor: Another promising research direction would be to use input given by a dynamic vision sensor. Using spiking neural networks on that would give an end-to-end event-based framework which would be very energy-efficient. Problems could be identified where the input can be recorded with a DVS without significant loss of information. Then, the presented algorithms can be applied straightforwardly. One method to achieve this is to simply set up a DVS in front of the screen to record it, although this can make simple problems much more complicated. Another method would be to simulate the DVS on hardware by keeping track of pixel changes and emitting signals once the changes cross a certain threshold. Ideally, one could identify tasks where it actually makes sense to use a DVS input. One example for this are robots using a DVS as sensor.

9. Recurrent Networks: One of the main advantages of SNNs is that they have an intrinsic notion of time. Likewise, the environments we analyzed are simulated for a number of time steps. In our implementation, for every time step in the environment, the SNN is reset and simulated for a number of time steps (between 20 for direct training and for at least 500 for direct conversion). A major improvement would be to not reset the neural network but to establish a one to one correspondence between the environment and the network (one time step in the environment is one time step in the neural network). The network then makes predictions based on the new input and the current state of the network. This would make the agent extremely fast and energy efficient, while it could potentially also make predictions based on states further back in time. The problem is to adapt the reinforcement learning algorithms to recurrent networks. If combined with the idea of using a DVS (or other event-based input) this would result in an end-to-end event-based, extremely energy efficient framework for reinforcement learning.

To make some more practical and immediate proposals for future research, we think it would be promising to train an actor-critic agent on a difficult environment and convert that agent. Conversion should work straightforwardly if the actor-critic does not use a long-term short-term memory layer which is the state-of-the-art [52] and would test our assumption that actor-critic methods, like policy gradient methods, are easy to convert. In order to convert an actor-critic, it is sufficient to convert the actor as only the policy is retained after conversion anyway (see section 5.2.3).
Another promising experiment would be to directly train a set of more complex environments like the Atari games using Backpropagation with surrogate gradients and compare the training to conventional DQNs. Our code repository already contains first steps towards training the large DQN from Mnih et al. [53] on Breakout (see Appendix A).
Finally, it would be interesting to implement local learning using eRBP. Auryn [60], a simulator for eRBP, is available online. Alternatively, the method could be implemented in NEST or PyNN.

Chapter 6
Conclusion

In this thesis, we analyzed deep spiking networks for reinforcement learning applications, particularly Q learning.
We analyzed the theoretical background of deep reinforcement learning and spiking neural networks focusing particularly on state-of-the art training methods for deep spiking networks. We proposed a classification of these training methods, presented corresponding state-of-the-art research, and analyzed which are suited to be applied to deep reinforcement learning. In our experiments, we tested three conversion methods (direct, indirect via classifier, SNN classifier that learns the DQNs policy) and Backpropagation with surrogate gradients on three Open AI Gym [11] environments, CartPole, MountainCar, and Breakout. We showed that it is possible to train DSQNs using all methods and compared them against each other.
We found that each method has its strengths and weaknesses and that successful training is depending very strongly on the specific problem and hyperparameters as well as on randomness induced by the training process and the environment. This makes clear-cut propositions and reproducibility challenging. Backpropagation with surrogate gradients needs more time for the training process and training is more unstable, while compared to conversion, inference is more efficient and generalization to unseen data is better. Direct conversion has an especially high inference time, while first training a classifier that learns the policy of a DQN and then converting it is more efficient, but can come with a performance loss and induces a small runtime overhead. Training a SNN classifier on the policy using surrogate gradients brings down the inference time even more, but can likewise cause a performance loss and increases the runtime overhead.
Our results suggest that neural networks trained with cross entropy loss (e.g. classifiers or policy gradient networks) are easier to convert (need less simulation time to produce similar results) than DQNs which use the temporal difference loss. Further, our results indicate that the performance of a DQN can potentially be improved through conversion, but that no generic approach to achieve this exists.
We analyzed various conversion approaches and find that constant input currents, potential outputs, robust normalization, and reset-by-subtraction are the most generic methods and usually lead to the best results. In some cases, introducing noise, for example by using Poisson spike trains, can improve the performance.
Additionally, we showed how to run the trained SNNs on sophisticated neuron simulators and neuromorphic hardware (NEST [46], PyNN [4], and SpiNNaker [63]) and outlined the challenges that come from different neuron models.
Combining reinforcement learning and spiking neural networks is a so far only little explored research area which promises not only energy efficiency but also novel algorithms and a deeper understanding of the human brain. We successfully showed that various training methods of spiking neural networks can be applied to deep reinforcement learning. Nevertheless, further research is needed to exploit the described advantages. We hope our thesis can help to understand the current developments and to point the way for future work.

Appendices

Appendix A
Code Repository

A major part of this thesis is the code repository which is available from https://github.com/vhris/Deep-Spiking-Q-Networks.git. The repository contains a code, an experiment, and a result folder. The CartPole experiments are provided in the form of jupyter notebooks [67] and are conceptualized as tutorials which explain how to use the code. They are intended to be viewed in the following order:

1. DQN-Training (How to train a conventional DQN and a spiking DQN using Surrogate Gradients (DSQN).)
2. Load-DQN (How to load a previously saved D(S)QN and how to save a replay dataset.)
3. Train-Classifier (How to train a spiking or non-spiking classifier on the saved replay data set.)
4. SNN-Conversion (How to convert a DQN and a Classifier to a SNN.)
5. Load in NEST (How to load a converted or directly trained spiking network in NEST.)
6. Conversion in PyNN with NEST or SpiNNaker (How to load a spiking network in PyNN using NEST or SpiNNaker as backend.)

The Breakout experiments consist of the experiment conducted in this thesis (see section 4.3.3) and a draft for training and converting the larger network according to Mnih et al. [53].
Additionally, we report the networks used throughout this thesis with initial and final weights in the results folder. Within the experiments we also report the seeds we used to ensure reproducibility, however, for the networks trained earlier in this thesis we did not save the seeds.
Finally, we provide an extensive list of the libraries we used together with Python 3.5 (a short version is provided in table B.7) as well as this thesis itself and the mid- and endterm presentations held at TUM.

Appendix B
Hyperparameters

Parameter	CartPole	CartPole-SNN	MountainCar	MountainCar-SNN
Spiking	No	Yes	No	Yes
Architecture	[4,16,16,2]	[5,17,17,2]	[2,64,64,3]	[3,64,64,3]
Loss	NLL-loss	NLL-loss	NLL-loss	NLL-loss
Batch Size	50	50	50	50
Number of Batches	$2 * 10^5$ (50 epochs)	$2 * 10^5$ (50 epochs)	$2 * 10^5$ (50 epochs)	$2 * 10^5$ (50 epochs)
Optimizer	Adam	Adam	Adam	Adam
Learning Rate	0.0001	0.0001	0.0025	0.0025
Betas	0.9,0.999	0.9,0.999	0.9,0.999	0.9,0.999
Epsilon	10^{-8}	10^{-8}	10^{-8}	10^{-8}
Simuation Time	–	20	–	20

Table B.1: Training details of the classifiers. The conventional classifiers use ReLu activations in the hidden layers and a linear activation (no activation) in the output layer. As SpyTorch does not support biases, a constant input is added for the SNN classifiers and 1 neuron is added in each hidden layer to compensate the missing biases. SNNs use Non-Leaky Integrate-And-Fire neurons. Adam [41] is an optimizer based on stochastic gradient descent.

Parameter	Specification
Architecture	[4,16,16,2]
Optimizer	Adam (epsilon=10^{-8}, betas=(0.9,0.999))
Learning Rate	0.01
Discount Factor	0.99

Table B.2: Hyperparameters of the policy gradient network for CartPole.

Parameter	Breakout (small network)
Architecture	[6400,1000,4]
Optimizer	RMS-Prop (epsilon=0.001,alpha=0.95)
Learning Rate	0.00025
Discount Factor	0.99
Epsilon Start	1
Epsilon End	0.1
Epsilon Decay	0.9999977
Replay Memory Size	$2 * 10^5$
Initial Replay Size	50000
Sample Size	32
Target Update Frequency	10000 (iterations)
Double Q Learning	no
Gradient Clipping	clipped at 1
Update Frequency	4
Frameskip	4
No-Op Max	30
Prioritized Experience Replay	no
Dueling Architecture	no

Table B.3: Breakout hyperparameters. Compared to CartPole and MountainCar, Breakout has some additional hyperparameters. Initial replay size describes the minimum size of the replay memory before training starts, an update frequency of 4 means that training happens only every 4 steps, the frameskip describes how often an action is repeated where the states in-between repetitions are not seen by the agent, No-Op max describes the maximum number of a random amount of "do-nothing" operations performed at the start of each episode. Additionally, the target update is not done every x episodes but instead every x iterations. We use the same hyperparameters as Mnih et al. [53] except for the network architecture of the small network (from Patel et al. [66]) and the parameters of RMS-Prop [32] as the implementation details are different in PyTorch compared to TensorFlow (we use Pytorch, Mnih uses TensorFlow). For a more detailed description of the other parameters also see caption of table B.8.

Parameter	iaf_psc_delta
V_th	1.0 (inf for last layer)
t_ref	0.0
V_min	$-\inf$
C_m	1.0
V_reset	0.0
tau_m	10^{10}
E_L	0.0
V_m (initial value)	0.0
I_e	set to bias

Table B.4: Hyperparameters of the NEST neuron model iaf_psc_delta. The parameters are set in such a way that a Non-Leaky Integrate-and-Fire neuron is approximated.

Parameter	pp_psc_delta
C_m	1.0
tau_m	10^{10}
E_L	0.0
V_m (initial value)	0.0
q_sfa	1.0
tau_sfa	10^{10}
dead_time	1e-8
dead_time_random	False
dead_time_shape	1
t_ref_remaining	0.0
I_e	set to bias
c_1	10^{10}
c_2	10.0
c_3	5.0
with_reset	False

Table B.5: Hyperparameters of the NEST neuron model pp_psc_delta. tau_m is set high to approximate non-leaky behavior. tau_sfa, the adaptive threshold time constant, is set high such that the threshold immediately jumps to its next value (old value plus q_sfa). This approximates reset-by-subtraction. Finally, c_1, c_2, c_3 are set such that stochastic firing is minimized (see section 4.5).

Parameter	iaf_psc_delta
v_thresh	1.0 (10000.0 for last layer)
c_m	1.0
v_reset	0.0
v_rest	0.0
tau_m	100.0
tau_refrac	0.0
v (initial value)	0.0
i_offset	set to bias

Table B.6: Hyperparameters of the SpiNNaker neuron model IFCurDelta. The parameters are set in such a way that a Non-Leaky Integrate-and-Fire neuron is approximated. Compared to the corresponding NEST model (see table B.4), tau_m must not be set too large, else the SpiNNaker chip risks running into an overflow which ruins the results. For a similar reason, the maximum potential for the last layer must not be set too high, but not too low either such that neurons in the output layer would spike.

Library	Version
python3	3.5
gym	0.15.4
matplotlib	3.0.3
numpy	1.17.4
torch	1.3.1
NEST	2.16.0
PyNN	0.9.5
sPyNNaker8	5.1.0

Table B.7: This table lists the most important libraries one needs to install to run our scripts. An extensive list of installed packages is available from our code repository (see Appendix A). Versions are reported to ensure good reproducibility.

Parameter	CartPole A	CartPole B/C	CartPole D/E (SNN)	MountainCar A	MountainCar B/C	MountainCar D/E
Spiking	no	no	yes	no	no	yes
Architecture	[4,16,16,2]	[4,16,16,2]	[5,17,17,2]	[2,64,64,3]	[2,64,64,3]	[3,65,65,3]
Optimizer	Adam	Adam	Adam	Adam	Adam	Adam
Learning Rate	0.001	0.001/0.0005	0.001/0.0005	0.001	0.001/0.0005	0.001/0.0005
Discount Factor	0.99	0.999	0.999	0.99	0.999	0.999
Epsilon Start	1	1	1	1	1	1
Epsilon End	0.05	0.05	0.05	0.05	0.05	0.05
Epsilon Decay	0.99	0.999	0.999	0.99	0.999	0.999
Replay Memory Size	10^6	$4 * 10^4$	$4 * 10^4$	10^6	10^3	10^3
Sample Size	64	128	128	64	128	128
Target Update Frequency	20	10	10	20	5	5
Double Q Learning	yes	yes	yes	yes	yes	yes
Gradient Clipping	clipped at 1	clipped at 1	clipped at 1	clipped at 1	clipped at 1	clipped at 1
Prioritized Experience Replay	no	no	no	no	no	no
Dueling Architecture	no	no	no	no	no	no
Simulation Time	–	–	20	–	–	20

Table B.8: Training details of the DQNs. CartPole A describes the network trained for the conversion experiments, while CartPole B-E describe the parameters used for comparing DQN and Surrogate Gradient convergence (likewise for MountainCar). The architecture [4,16,16,2] describes a network with 4 inputs, two hidden layers with 16 neurons, and 2 outputs (actions). The DQNs use ReLu activations in the hidden layers and a linear activation in the output layers. As SpyTorch does not support biases, the spiking networks get an additional constant input and one additional neuron in the hidden layers to compensate for the missing bias. SNNs use Non-Leaky Integrate-And-Fire neurons, constant input currents, and potential outputs. For the Adam optimizer [41] we use the standard values of Pytorch for the parameters except the learning rate (Betas: (0.9,0.999), ϵ: 10^{-8}). The epsilon decay is a linear decay of the form $\epsilon_i = max((\epsilon_{decay})^i \epsilon_{start}, \epsilon_{end})$. A target update of 10 means the policy net is duplicated in the target net every 10 episodes.

Appendix C
Additional Plots

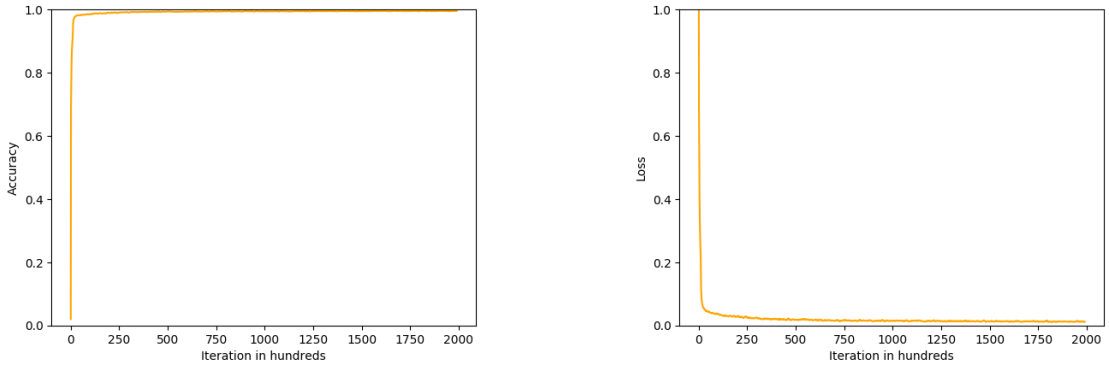

Figure C.1: Accuracy (left) and Loss (right) during training of the classifier for MountainCar.

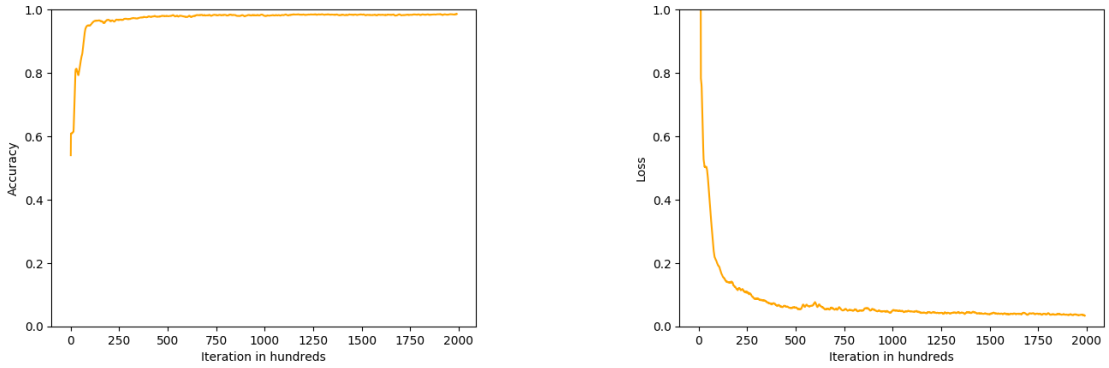

Figure C.2: Accuracy (left) and Loss (right) during training of the SNN classifier for MountainCar.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[2] Sneha Aenugu, Abhishek Sharma, Sasikiran Yelamarthi, Hananel Hazan, Philip S. Thomas, and Robert Kozma. Reinforcement learning with spiking coagents. (October), 2019.

[3] Arnon Amir, Brian Taba, David Berg, Timothy Melano, Jeffrey Mckinstry, Carmelo Di Nolfo, Tapan Nayak, Alexander Andreopoulos, Guillaume Garreau, Marcela Mendoza, Jeff Kusnitz, Michael Debole, Steve Esser, Tobi Delbruck, Myron Flickner, and Dharmendra Modha. A low power, fully event-based gesture recognition system. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-January:7388–7397, 2017.

[4] P. Davison Andrew, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, and Dejan Pecevski. PyNN : a common interface for neuronal network simulators. 2(January):1–10, 2009.

[5] Pierre Baldi, Peter Sadowski, and Zhiqin Lu. Learning in the machine: Random backpropagation and the deep learning channel. *Artificial Intelligence*, 260:1–35, 2018.

[6] AG Barto, RS Sutton, and CW Anderson. Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

[7] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *IJCAI International Joint Conference on Artificial Intelligence*, 2015-January:4148–4152, 2015.

[8] Yoshua Bengio and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. 86(11), 1998.

[9] Yoshua Bengio, Thomas Mesnard, Asja Fischer, Saizheng Zhang, and Yuhuai Wu. STDP-Compatible Approximation of Backpropagation in an Energy-Based Model. *Neural Computation*, 2017.

[10] Sander M Bohte, Joost N Kok, and Han La Poutr. SpikeProp : Backpropagation for Networks of Spiking Neurons. (January 2000), 2014.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Open AI Gym. *CoRR*, abs/1606.01540, 2016.

[12] John A. Bullinaria. Recurrent Neural Networks Neural Computation : Lecture 12. pages 1–20, 2015.

[13] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. 2016.

[14] Balázs Csanád Csáji. Approximation with Artificial Neural Networks. 2001.

[15] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

[16] Peter U. Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9(AUGUST):1–9, 2015.

[17] Peter U. Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih chii Liu, and Michael Pfeiffer. Fastclassifying, high-accuracy spiking deep networks through weight and threshold balancing. In *inNeural Networks (IJCNN), 2015 International Joint Conference on (Killarney: IEEE)*, 2015.

[18] Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo Di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences of the United States of America*, 113(41):11441–11446, 2016.

[19] Mazdak Fatahi, Mahmood Ahmadi, Mahyar Shahsavari, Arash Ahmadi, and Philippe Devienne. evt_MNIST: A spike based version of traditional MNIST. 2016.

[20] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of human genetics*, pages 179–188, 1936.

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

[22] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement Learning Using a Continuous Time Actor-Critic Framework with Spiking Neurons. *PLoS Computational Biology*, 9(4), 2013.

[23] Angus Galloway, Graham W. Taylor, and Medhat Moussa. Attacking Binarized Neural Networks. pages 1–14, 2017.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[25] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement Learning with Deep Energy-Based Policies. 2017.

[26] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *35th International Conference on Machine Learning, ICML 2018*, 5:2976–2989, 2018.

[27] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 13 January 2017.

[28] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. BindsNET: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics*, 12:1–25, 2018.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-December:770–778, 2016.

[30] Professor David Heeger. Poisson Model of Spike Generation. September 5, 2000.

[31] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pages 3207–3214, 2018.

[32] Geoffrey Hinton. RMSProp (Lecture), Retrieved on 8th January 2020. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[33] A. L. Hodgkin and A. F. Huxley. Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo. *The Journal of Physiology*, 116(4):449–472, 1952.

[34] Yangfan Hu, Huajin Tang, Yueming Wang, and Gang Pan. Spiking Deep Residual Network. 2018.

[35] Eric Hunsberger and Chris Eliasmith. Training Spiking Deep Networks for Neuromorphic Hardware. pages 1–10, 2016.

[36] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003.

[37] Jacques Kaiser, Alexander Friedrich, J. Camilo Vasquez Tieck, Daniel Reichard, Arne Roennau, Emre Neftci, and Rüdiger Dillmann. Embodied Neuromorphic Vision with Event-Driven Random Backpropagation. pages 1–8, 2019.

[38] Saeed Reza Kheradpisheh, Mohammad Ganjtabesh, and Timothée Masquelier. Bio-inspired unsupervised learning of visual features leads to robust invariant object recognition. *Neurocomputing*, 205:382–392, 2016.

[39] Saeed Reza Kheradpisheh, Mohammad Ganjtabesh, Simon J. Thorpe, and Timothée Masquelier. STDP-based spiking deep convolutional neural networks for object recognition. *Neural Networks*, 99:56–67, 2018.

[40] Minje Kim and Paris Smaragdis. Bitwise Neural Networks. 37, 2016.

[41] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.

[42] Giri P Krishnan, Timothy Tadros, Ramyaa Ramyaa, and Maxim Bazhenov. Biologically inspired sleep algorithm for artificial neural networks. 2019.

[43] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *University of Toronto*, 05 2012.

[44] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training Deep Spiking Neural Networks using Backpropagation. pages 1–10.

[45] Yuxi Li. Deep reinforcement learning. pages 19–21, 2018.

[46] Charl Linssen, Mikkel Elle Lepperd, Jessica Mitchell, Jari Pronold, Jochen Martin Eppler, Chrisitan Keup, Alexander Peyser, Susanne Kunkel, Philipp Weidel, Yannick Nodem, Dennis Terhorst, Rajalekshmi Deepu, Moritz Deger, Jan Hahne, Ankur Sinha, Alberto Antonietti, Maximilian Schmidt, Luciano Paz, Jess Garrido, Tammo Ippen, Luis Riquelme, Alex Serenko, Tobias Khn, Itaru Kitayama, Hkon Mrk, Sebastian Spreizer, Jakob Jordan, Jeyashree Krishnan, Mario Senden, Espen Hagen, Alexey Shusharin, Stine Brekke Vennemo, Dimitri Rodarie, Abigail Morrison, Steffen Graber, Jannis Schuecker, Sandra Diaz, Barna Zajzon, and Hans Ekkehard Plesser. NEST 2.16.0, August 2018.

[47] Charl Linssen, Mikkel Elle Lepperd, Jessica Mitchell, Jari Pronold, Jochen Martin Eppler, Chrisitan Keup, Alexander Peyser, Susanne Kunkel, Philipp Weidel, Yannick Nodem, Dennis Terhorst, Rajalekshmi Deepu, Moritz Deger, Jan Hahne, Ankur Sinha, Alberto Antonietti, Maximilian Schmidt, Luciano Paz, Jess Garrido, Tammo Ippen, Luis Riquelme, Alex Serenko, Tobias Khn, Itaru Kitayama, Hkon Mrk, Sebastian Spreizer, Jakob Jordan, Jeyashree Krishnan, Mario Senden, Espen Hagen, Alexey Shusharin, Stine Brekke Vennemo, Dimitri Rodarie, Abigail Morrison, Steffen Graber, Jannis Schuecker, Sandra Diaz, Barna Zajzon, and Hans Ekkehard Plesser. NEST 2.16.0 documentation: pp_psc_delta., Retrieved on 8th January 2020. https://www.nest-simulator.org/helpindex/cc/pp_psc_delta.html.

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

[49] 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.

[50] Michael McCloskey and Neal J Cohen. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press, 1989.

[51] Claus Meschede. Training Neural Networks for Event-Based End-to-End Robot Control. *Master thesis at the Technical University of Munich*, 28 July 2017.

[52] Volodymyr Mnih, Adria Puigdomenech Badia, Lehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *33rd International Conference on Machine Learning, ICML 2016*, 4:2850–2869, 2016.

[53] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, and Shane Legg & Demis Hassabis. Human-level control through deep reinforcement learning. *Nature, Volume 518*, 26 February 2015.

[54] Andrew Moore. Efficient Memory-Based Learning for Robot Control. 06 2002.

[55] Hesham Mostafa. Supervised learning based on temporal coding in spiking neural networks. (Nips), 2016.

[56] Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, and Timothe Masquelier. SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks with at most one Spike per Neuron. 6 March 2019.

[57] Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, Simon J. Thorpe, and Timothée Masquelier. Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks. *Pattern Recognition*, 94:87–95, 2019.

[58] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. 2010.

[59] Emre Neftci, Charles Augustine, Somnath Paul, and Georgios Detorakis. Neuromorphic Deep Learning Machines. pages 1–25, 2016.

[60] Emre Neftci, Friedemann Zenke, Lorric Ziegler, and Ankur Sinha. Auryn, Available from https://gitlab.com/eneftci/erbp_auryn, Retrived on 16.12.2019.

[61] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate Gradient Learning in Spiking Neural Networks. *IEEE SPM WHITE PAPER FOR THE SPECIAL ISSUE ON NEUROMORPHIC COMPUTING*, 28 January 2019.

[62] Peter O Connor and Max Welling. Deep Spiking Networks. 48, 2016.

[63] University of Manchester, University of Southampton, University of Cambridge, University of Sheffield, ARM Ltd., Silistix Ltd, and Thales. SpiNNaker webpage: http://apt.cs.manchester.ac.uk/projects/SpiNNaker/, retrieved on 04.10.2019.

[64] Priyadarshini Panda and Kaushik Roy. Unsupervised Regenerative Learning of Hierarchical Features in Spiking Deep Networks for Object Recognition.

[65] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[66] Devdhar Patel, Hananel Hazan, Daniel J. Saunders, Hava T. Siegelmann, and Robert Kozma. Improved robustness of reinforcement learning policies upon conversion to spiking neuronal network platforms applied to Atari Breakout game. *Neural Networks*, 2019.

[67] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[68] Michael Pfeiffer and Thomas Pfeil. Deep Learning With Spiking Neurons: Opportunities and Challenges. *Frontiers in NeuroScience*, 25 October 2018.

[69] S. Phaniteja, Parijat Dewangan, Pooja Guhan, Abhishek Sarkar, and K. Madhava Krishna. A deep reinforcement learning approach for dynamically stable inverse kinematics of humanoid robots. *2017 IEEE International Conference on Robotics and Biomimetics, ROBIO 2017*, 2018-January:1818–1823, 2018.

[70] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. (2015):1–21, 2018.

[71] Wiebke Potjans and Abigail Morrison. A Spiking Neural Network Model of an Actor-Critic Learning Agent. *Neural Computation*, 339:301–339, 2009.

[72] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, and Michael Pfeiffer. Theory and Tools for the Conversion of Analog to Spiking Convolutional Neural Networks. December 2016.

[73] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[74] J. Sawada, F. Akopyan, A. S. Cassidy, B. Taba, M. V. Debole, P. Datta, R. Alvarez-Icaza, A. Amir, J. V. Arthur, A. Andreopoulos, R. Appuswamy, H. Baier, D. Barch, D. J. Berg, C. d. Nolfo, S. K. Esser, M. Flickner, T. A. Horvath, B. L. Jackson, J. Kusnitz, S. Lekuch, M. Mastro, T. Melano, P. A. Merolla, S. E. Millman, T. K. Nayak, N. Pass, H. E. Penner, W. P. Risk, K. Schleupen, B. Shaw, H. Wu, B. Giera, A. T. Moody, N. Mundhenk, B. C. V. Essen, E. X. Wang, D. P. Widemann, Q. Wu, W. E. Murphy, J. K. Infantolino, J. A. Ross, D. R. Shires, M. M. Vindiola, R. Namburu, and D. S. Modha. TrueNorth Ecosystem for Brain-Inspired Computing: Scalable Systems, Software, and Applications. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 130–141, Nov 2016.

[75] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. pages 1–21, 2015.

[76] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. *32nd International Conference on Machine Learning, ICML 2015*, 3:1889–1897, 2015.

[77] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. pages 1–12, 2017.

[78] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. A Survey of Neuromorphic Computing and Neural Networks in Hardware. pages 1–88, 2017.

[79] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going Deeper in Spiking Neural Networks: VGG and Residual Architectures. *Frontiers in Neuroscience*, 13(1998):1–16, 2019.

[80] Sumit Bam Shrestha and Garrick Orchard. Slayer: Spike layer error reassignment in time. *Advances in Neural Information Processing Systems*, 2018-Decem(NeurIPS):1412–1421, 2018.

[81] Thomas Simonini. An intro to Advantage Actor Critic methods: lets play Sonic the Hedgehog! https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/, accessed on 18. November 2019.

[82] Thomas Simonini. An introduction to Policy Gradients with Cartpole and Doom. https://www.freecodecamp.org/news/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f/, accessed on 18. November 2019.

[83] Taylor Simons and Dah-Jye Lee. A Review of Binarized Neural Networks. *Electronics*, 8(6):661, 2019.

[84] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. pages 1–14, 2014.

[85] Hagen Soltau, Hank Liao, and Hasim Sak. Neural speech recognizer: Acoustic-To-word LSTM model for largevocabulary speech recognition. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2017-August:3707–3711, 2017.

[86] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning:An Introduction. 2014.

[87] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy. *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, pages 2625–2631, 2017.

[88] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111(April):47–63, 2019.

[89] Amirhossein Tavanaei and Anthony Maida. BP-STDP: Approximating backpropagation using spike timing dependent plasticity. *Neurocomputing*, 330:39–47, 2019.

[90] Amirhossein Tavanaei, Timothee Masquelier, and Anthony S. Maida. Acquisition of visual features through probabilistic spike-timing-dependent plasticity. *Proceedings of the International Joint Conference on Neural Networks*, 2016-October:307–314, 2016.

[91] Simon Jonathan Thorpe. Rate Coding Versus Temporal Order Coding: What the Retinal Ganglion Cells Tell the Visual Cortex. *Neural Computation*, July 2001.

[92] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning. *Google DeepMind*, 8 December 2015.

[93] Kancheng Wang. Deep Reinforcement Learning with Binarized Neural Network. 05 2017.

[94] Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. Learning Channel-wise Interactions for Binary Convolutional Neural Networks. *Cvpr*, pages 4321–4330, 2019.

[95] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Frcitas. Dueling Network Architectures for Deep Reinforcement Learning. *33rd International Conference on Machine Learning, ICML 2016*, 4(9):2939–2947, 2016.

[96] Martha White and Richard S Sutton. Off-Policy Actor-Critic. (May), 2012.

[97] Wikipedia. List of Nobel laureates in Physiology or Medicine, Retrieved on 26th November 2019. https://en.wikipedia.org/wiki/List_of_Nobel_laureates_in_Physiology_or_Medicine.

[98] Wikipedia. Softmax function, Retrieved on 8th January 2020. https://en.wikipedia.org/wiki/Softmax_function.

[99] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for ConnectionistReinforcement Learning. *Machine Learning*, 8:229–256, 1992.

[100] Jibin Wu, Yansong Chua, Malu Zhang, Guoqi Li, Haizhou Li, and Kay Chen Tan. A Hybrid Learning Rule for Efficient and Rapid Inference with Spiking Neural Networks. pages 1–12, 2019.

[101] Si Wu and Shun ichi Amari. Population Coding and Decoding in a Neural Field: A Computational Study. *Neural Computation*, June 2002.

[102] Friedemann Zenke and Surya Ganguli. SuperSpike: Supervised learning in multi-layer spiking neural networks. October 14, 2017.