Horst Petschenig, BSc BSc

Reinforcement Learning with Spiking Neural Networks

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

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

Institute for Theoretical Computer Science

Graz, February 2019

Affidavit

Date	Signature

Acknowledgements

Foremost, I would like to thank my thesis supervisor Prof. Robert Legenstein for his continuous support and patience. His insightful explanations and comments about my research shaped this work; he carefully directed me towards successful completion of this thesis.

My sincere thanks go to Prof. Wolfgang Maass and Prof. Denis Helic for joining the examination board of my thesis.

Furthermore I would like to thank Guillaume Bellec for introducing me to long short-term memory spiking neural networks, pointing out mistakes and supporting me on the way.

Last but by no means least, I would like to express my gratitude to my family, who have supported and encouraged me unconditionally throughout my years of study. Without you, this thesis would not have been possible. Thank you!

Abstract

Reinforcement learning provides a rigorous theory for learning processes of biological and artificial agents through trial-and-error interaction with environments. The goal of the agent is to maximize some notion of cumulative reward by exploring possible actions to find and exploit good actions. In recent years, artificial neural networks have been successfully applied to reinforcement learning by estimating the value of actions in a given situation and choosing the best action thereafter. Nature has evolved animals into efficient learners, thus biologically plausible spiking neural networks are an obvious alternative to artificial neural networks. However, spiking neural networks have not found widespread application to reinforcement learning so far due to lack of efficient learning rules. Through the extension of backpropagation through time to spiking neural networks the efficient optimization of spiking neural networks for many tasks became possible. The aim of this thesis is to show that spiking neural networks can be used in state-of-the-art reinforcement learning methods as drop-in replacement for artificial neural networks with memory capabilities. Furthermore we investigate the impact of recurrent connections in the network and show that recurrent connections are not necessary to solve most of the investigated environments. We compare the performance of agents using spiking neural networks or artificial neural networks in an actor-critic setting on benchmark tasks such as the Roboschool robotic environment suite.

Kurzfassung

Reinforcement Learning (Vestärkendes Lernen) beschreibt jene Lernprozesse, die ein Agent (sowohl biologisch als auch künstlich) in der Interaktion mit seiner Umwelt durch Versuch und Irrtum durchläuft. Das Ziel des Agenten ist dabei die langfristige Maximierung von Belohnungen, die er von seiner Umwelt erhält. Dafür müssen möglichst viele verschiedene Aktionen ausprobiert werden, um jene mit höchstmöglichem zukünftigen Ertrag zu entdecken und auszunutzen. In den letzten Jahren wurden künstliche neuronale Netzwerke erfolgreich für das Reinforcement Learning eingesetzt: Sie approximieren den Wert einer Aktion in einer bestimmten Situation und wählen daraufhin die bestmögliche. Gehirne höherer Lebewesen wurden mittels natürlicher Evolution zu effizient lernenden Systemen geformt. Daher sind biologisch plausible spikende (gepulste) neuronale Netzwerke eine naheliegende Alternative zu künstlichen neuronalen Netzwerken. Aufgrund fehlender oder ineffizienter Lernalgorithmen sind spikende neuronale Netzwerke noch von keiner besonderen Bedeutung für Reinforcement Learning. Durch die Erweiterung von "backpropagation through time" wurde die effiziente Optimierung spikender neuronaler Netzwerke für viele Anwendungen möglich. In dieser Arbeit zeigen wir die Anwendbarkeit spikender neuronaler Netzwerke in modernen Methoden des Reinforcement Learnings als direkten Ersatz von künstlichen neuronalen Netzwerken. Außerdem untersuchen wir den Einfluss rekurrenter Verbindungen im Netzwerk und zeigen, dass rekurrente Verbindungen für die meisten untersuchten Umgebungen nicht notwendig sind. Schließlich werden anhand verschiedener Experimente Agenten mit spikenden neuronalen Netzwerken und künstlichen neuronalen Netzwerken in der simulierten Robotikumgebung Roboschool und dem Cart-Pole Kontrollproblem verglichen.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Humans try to unravel the mysteries of rational thinking and acting at least since the days of Aristotle in ancient Greece. Our wish of understanding intelligence and the inner workings of human brains, the most advanced intelligent systems we know of, influenced fields such as cognitive sciences, mathematics, philosophy and biology (Russell and Norvig, 2009).

Even today, after decades of focused research, we still do not have a unified theory of what intelligent systems are and how to build artificial general intelligence. There are many systems that solve problems requiring narrow intelligence with superhuman performance, such as the recent success of computers beating world-class players in the game of Go (Silver et al., 2016). Artificial intelligence is now one of the most active and best funded fields of research; technology giants such as Microsoft, Google, Facebook, IBM, Intel and Baidu have keen interest in advancing the state of the art.

When we think about traits or inherent properties of intelligence, one of the first that come to mind include thinking and acting rationally, planning and learning from experience to reach goals. Many animals exhibit forms of reward-based learning that reinforces or discourages certain behaviour based on pleasure or pain following some action. Psychologists and biologists have studied this type of learning and conditioning since Thorndike, 1898. An animal or actor has to choose actions based upon observations of the environment to maximize the cumulative reward in the future. Typical supervised methods known from machine learning are not immediately suitable to model and solve these problems; there the goal is approximating an unknown function by learning from correct examples of input and output values. However, in this animal-like learning scenario there are no correct examples of behaviour to be learned from, just a form of reward is presented to the actor after an action has been taken. Thus, a separate field of machine learning called reinforcement learning tries to formalize and solve these problems.

Naturally, the fields of reinforcement learning and neuroscience are closely intertwined; both profited from another and inspired new and fascinating ideas in both fields. For example, in recent years popular function approximation methods known from supervised learning have been successfully applied to reinforcement learning: Artificial neural networks (ANNs) were used to estimate whether observations perceived by an actor will yield high expected cumulative reward in the future after following some course of action. As image classification and speech translation workhorse ANNs have found widespread application. Circling back to biological brains, we want to find out how spiking neural networks (SNNs) can solve these difficult tasks seemingly with ease. ANNs process real-valued data without a concept of time; SNNs operate in continuous time by sending and receiving short pulses encoding information. This completely different method of information transmission is highly energy-efficient and inherently capable of processing temporal data. Thus spiking neural network models not only provide insights about the principles of brain computation but also make the construction of neuromorphic hardware possible. This is not only interesting for low-power applications; large-scale neuromorphic hardware platforms may finally allow simulating large mammalian brains and find out how animals learn and process information.

In this work we set out to show that state-of-the-art reinforcement learning methods are not only compatible with spiking neuron networks but are able to perform as well as artificial neural networks in this setting. The problem of training is solved using backpropagation through time, a gradient-based optimization technique that has not been applicable to spiking neural networks until recently. Neither the training method nor the network architecture proposed in this work are claimed to be biologically plausible. However, once trained, these spiking neural networks could be used in neuromorphic hardware for scenarios described above. We show experimentally, that spiking neural networks can solve current benchmark problems of reinforcement learning such as the Roboschool robotic environment as well as classic control problems. Further, we find that recurrent connections degrade convergence speed and performance of SNNs for tasks where no memory is necessary. Surprisingly, SNNs performed better than state-of-the-art ANN configurations augmented with memory capabilities when applied to partially-observed environments. To the best of our knowledge no previous work solved reinforcement learning problems of this difficulty with spiking neural networks.

In Chapter 2 we introduce basic terminology and concepts of spiking neuron models, artificial neural networks, backpropagation through time in spiking neural networks and reinforcement learning. With this in place, we can put these individual parts together to perform the experiments described in Chapter 4. Works related to reinforcement learning with spiking neural networks are briefly summarized in Chapter 3 and the thesis is concluded in Chapter 5 with an outlook on future work.

2 Background

The aim of this chapter is to give an introductory overview of concepts and tools necessary for the experiments conducted in later chapters. This will include concepts of theoretical nature such as basic introductions to spiking neuron models and backpropagation.

In the following sections, the inner workings of the simple phenomenological integrate-and-fire model of neurons will be introduced. Then, one of the primal reasons for the success of artificial neural networks in the last decades, aptly named backpropagation, will be reviewed. Building upon this we will discuss how we can apply backpropagation to spiking neural networks, which has not been possible until recently. The next step will be to introduce the foundations of elementary reinforcement learning concepts without covering biologically plausible ideas of reinforcement learning in the brain. This will lead up to recent reinforcement learning algorithms such as proximal policy optimization (Schulman, Wolski, et al., 2017).

2.1 Spiking Neuron Models

In this section several important notions of neuroscience will be introduced, including mathematical abstractions and simplifications of biological neurons necessary for simulation on computers.

There are many good reasons for studying the brain and its inner workings in detail; a non-exhaustive list may include points such as: philosophical questions regarding the emergence of intelligence from biological building blocks communicating with each other, biological and evolutionary aspects including the development of brains from nerve cells, the computational architecture with which difficult tasks such as pattern recognition, planning, learning, abstraction and more are solved and last but not least the energy efficiency of the computations performed in the brain. Some of these reasons led to a variety of recent advances in the field of neuromorphic hardware design (Davies et al., 2018; Furber et al., 2014; Merolla et al., 2014) with the goal of accurately simulating artificial brains with enough computing power in silico to solve complex problems in an energy-efficient manner that are very difficult to solve for computers with a von Neumann architecture.

2.1.1 Neuron structure

On an abstract level, neurons can be seen as tiny compute units with inputs, a processing step and an output; this corresponds to a multivariate function. Each part of the neuron has a specific purpose in terms of producing a function output (Gerstner, Kistler, et al., 2014), a schematic illustration of a single neuron can be seen in Figure 2.1.

Dendrites receive information from other neurons in the neural network. They connect to the output terminals (axons) of other neurons via synapses and relay the information to the cell body (soma).

Cell Bodies receive their input via synaptic connections either to input dendrites or directly to their soma. Once a certain threshold value of cumulated inputs is exceeded, a spike, also called action potential, is created in the axon hillock and transmitted via the axon to other neurons. This non-linear function of thresholding is of key importance as the network would otherwise collapse to a single linear function.

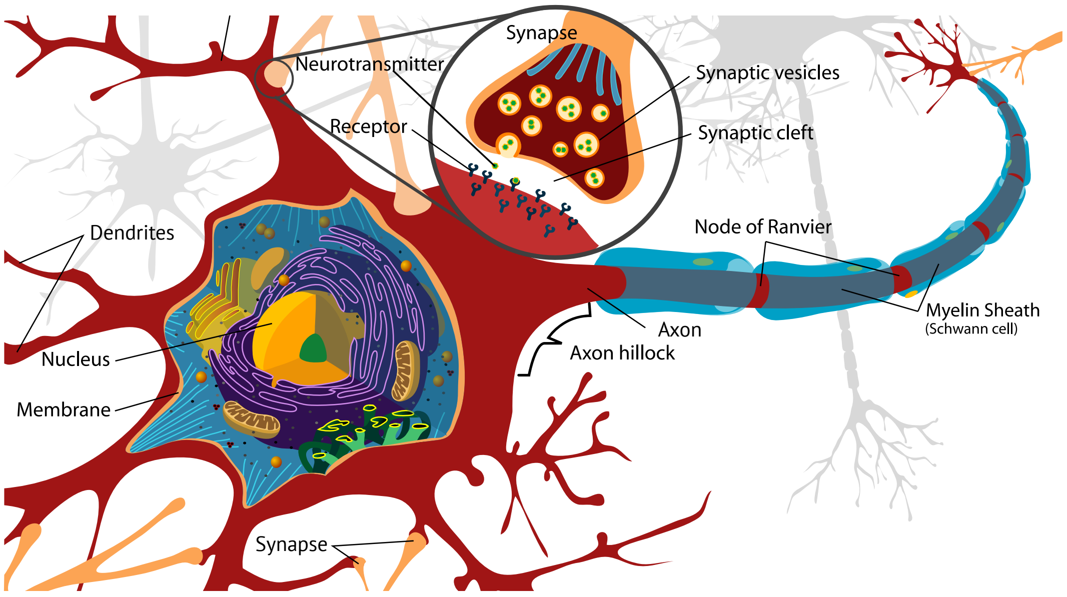


Fig. 2.1: Illustration of a single neuron. Neurons receive incoming information via dendrites, process it in the cell body and relay the generated action potentials to other neurons.[a].

[a]Mariana Ruiz Villarreal, 2007. Complete neuron cell diagram. Public Domain, `https://commons.wikimedia.org/wiki/File:Complete_neuron_cell_diagram_en.svg`

Axons relay the action potentials generated in the soma to often more than 10^4 other neurons in the network.

Synapses are the connections between axons and dendrites, cell bodies or other axons. Most of the synapses in the brain operate chemically, meaning they use special messenger substances called neurotransmitters. Neurons operate both chemically and electrically.

Myelin sheaths provide insulation to the axon "wire" and prevent losses in the membrane potential.

Nodes of Ranvier regenerate action potentials. (Bear, Connors, and Paradiso, 2016; Gerstner, Kistler, et al., 2014).

The aforementioned action potentials are generated by a complex biochemical process which produces signals with the same distinct shape over and over again as in Figure 2.2 on the right-hand side. In the brain, chemical and electrical processes support each other for information transmission. The cell membrane separates fluids inside and outside of the cell from each other. These fluids have different concentrations of specific electrically charged ions, which cause the time-dependent membrane potential $u(t)$. Ions of importance for the membrane potential are sodium (Na^+), potassium (K^+), calcium (Ca^{2+}) and chloride (Cl^-). Voltage-dependent ion channels spread over the axon and the axon hillock allow passage of specific ions between the fluid outside and inside of a neuron. Action potential generation is caused by time-dependent conductance changes in the membrane; sodium inflow depolarizes the membrane (voltage increase) whereas potassium outflow polarizes the membrane (voltage decrease) as visualized on the left side of Figure 2.2. Once the threshold voltage ϑ of the neuron has been passed, sodium channels open, increasing the membrane potential. At a large membrane potential, the sodium channels close and potassium channels open and the membrane potential decreases again to the resting potential u_{rest} (the membrane potential without incoming spikes). This produces the distinct action potential shape. A very similar process occurs for most synapses; instead of operating with voltage-gated ion channels, they use neurotransmitter-gated ion channels to transmit information (Bear, Connors, and Paradiso, 2016).

Due to the fixed shape of action potentials the information transmitted from neuron to neuron is not captured in the amplitude or the width of the spike but in the relative time duration between spikes. They last usually for 1–2 ms, during this time they cannot fire; this is called the absolute refractory period. After that, the membrane voltage undershoots its resting state, called the relative refractory period, making firing again unlikely, but possible (Gerstner, Kistler, et al., 2014).

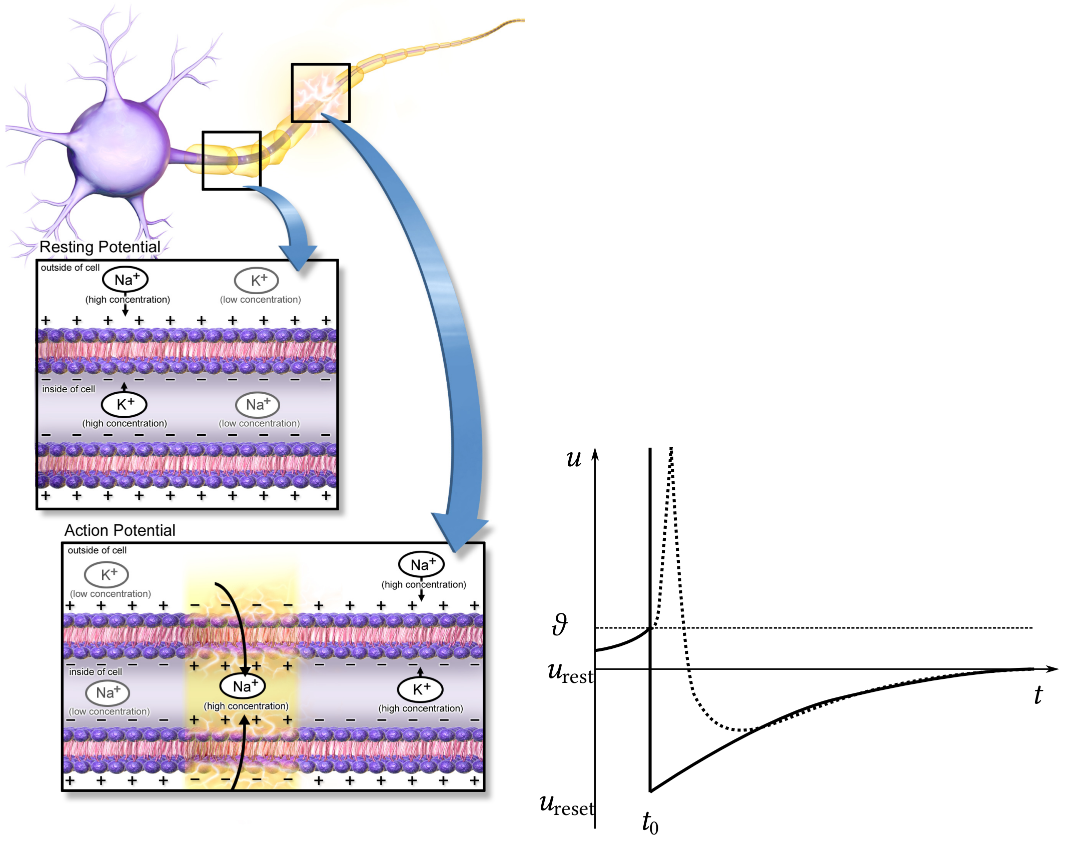


Fig. 2.2: Action potential generation of a neuron. **Left:** The chemical process by which the action potential (voltage difference between inside and outside of the cell membrane) changes[a]. **Right:** Once the threshold voltage has been overcome by the input stimulus, the neuron produces a signal following the shape of the dashed curve. Leaky integrate-and-fire neurons produce a spike-event after reaching the threshold potential and immediately reset the membrane voltage to a certain value. Adapted from Gerstner, Kistler, et al., 2014.

[a]Blausen.com staff, 2014. Medical gallery of Blausen Medical 2014. WikiJournal of Medicine 1 (2). CC BY 3.0, `https://commons.wikimedia.org/w/index.php?curid=29452220`

2.1.2 Leaky integrate-and-fire models

The original phenomenological integrate-and-fire model (Lapicque, 1907) is the foundation for many modern spiking neuron models. The cell membrane was described to be a simple circuit with a capacitor C and a resistor R in parallel, as seen in Figure 2.3. Overall, the dynamic of the membrane potential $u(t)$ between the inside and the outside of the neuron was described with a single linear differential equation

$$\tau_m \frac{du}{dt} = -(u(t) - u_{\text{rest}}) + RI(t) \tag{2.1}$$

where $\tau_m = RC$ is the membrane time constant. The input current $I(t)$ represents the input the neuron receives from other neurons; it can be interpreted as the weighted sum of all incoming action potentials of the neuron. The spikes are weighted because they can either increase or decrease the membrane potential of the receiving neuron. Synapses that relay spikes increasing the membrane potential are called excitatory synapses whereas synapses decreasing the membrane potential are called inhibitory synapses. The summation of the input spikes – in continuous time integration – gives the integrate-and-fire model its name.

Until now, the dynamics of the membrane potential are only described until a spike occurs; with just a single linear differential equation the entire action potential shape cannot be modelled because the behaviour of the membrane potential is different after the neuron crosses the firing threshold. Therefore we still need to describe the action potential shape after the firing threshold. Without introducing another differential equation, we can abstract the concept of an action potential into a single event in time that occurs immediately after crossing the firing threshold. This means that we do not model the full action potential shape, just the membrane potential before the spike. The final missing piece is to have an external reset mechanism that hard-resets the membrane potential to a fixed value $u_{\text{reset}} \leq \vartheta$ that will eventually decay back to the resting potential if no input is presented to the

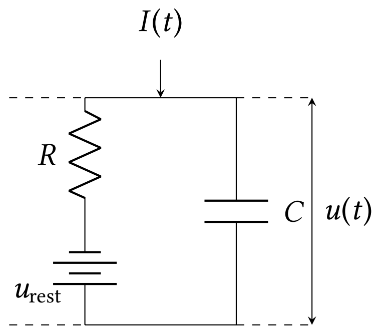


Fig. 2.3: The neuron membrane as a circuit. It consists of a parallel capacitor C and a resistor R representing the capacitance and leakage resistance. Adapted from Gerstner, Kistler, et al., 2014.

neuron.

Solving the differential equation under the assumptions that we have a constant input current $I(t) = I$, no refractory period and a spike has been produced at $t = 0$ yields

$$u(t) - u_{\text{rest}} = - \exp\left(-\frac{t}{\tau_m}\right)(u_{\text{rest}} + RI - u_{\text{reset}}) + RI. \tag{2.2}$$

Discretizing the differential equation for time intervals of width Δt, we can rewrite the solution above in a discrete-time setting as

$$u(t + \Delta t) = (1 - \alpha_m)(u_{\text{rest}} + RI(t)) + \alpha_m u(t), \tag{2.3}$$

$$\alpha_m = \exp\left(-\frac{\Delta t}{\tau_m}\right). \tag{2.4}$$

Using equation 2.2 to simulate the membrane potential of a neuron with constant input current I and with an external reset mechanism the membrane potential develops as seen in Figure 2.2. If the input current is too low, the membrane potential will saturate to a constant value without producing any spikes. However, if the threshold voltage has been passed, increasing the input current leads to an increase in the spike frequency. This non-linear behaviour creates the expressiveness of the neural network. A nonlinear function called the rectified linear unit has been proposed as a model for the firing rate of cortical neurons (Hahnloser et al., 2000), see Figure 2.4. In fact, it has become quite popular in the context of artificial neural networks and will be further discussed in Section 2.2.

Obviously the simple integrate-and-fire model does not capture all details of real biological neurons. A prominent short-coming is its missing ability to produce different spike patterns. Different areas of the brain contain different types of neurons that evolved to produce a diverse range of spike patterns. The leaky integrate-and-fire model discussed so far is unable to produce distinct spike patterns for a constant input current because the cell membrane is reset to u_{reset} immediately after spiking. Therefore no memory about the previous neuron state persists after this hard-reset. There are several ways to circumvent this issue; one of the simplest and most intuitive is to introduce a dynamic time-dependent threshold $\vartheta(t)$. After each spike $z(t)$, the threshold slightly increases by some fixed amount θ, if no spike occurs the dynamic threshold $\vartheta(t)$ will decay to some base threshold ϑ_0. The dynamics of this behaviour can be expressed as

$$\tau_a \frac{d\vartheta}{dt} = -(\vartheta(t) - \vartheta_0) + \theta z(t). \tag{2.5}$$

Assuming the threshold at time t_0 is $\vartheta(t_0) = \vartheta_0$, we can solve the differential equation and model the behaviour of the dynamic threshold in continuous time as

$$\vartheta(t) = \vartheta_0 + \theta \left(z(t) - \exp\left(-\frac{t - t_0}{\tau_a} \right) z(t_0) \right). \tag{2.6}$$

If the threshold at time t_0 is not at the base threshold ϑ_0, then the result is

$$\vartheta(t) = \vartheta_0 + \theta z(t) - \exp\left(-\frac{t - t_0}{\tau_a} \right) (-\vartheta(t_0) + \vartheta_0 + \theta z(t_0)) \tag{2.7}$$

In a discrete time setting, the result above can be rewritten as

$$\vartheta(t + \Delta t) = (1 - \alpha_a)(\vartheta_0 + \theta z(t)) + \alpha_a \vartheta(t) \tag{2.8}$$

$$\alpha_a = \exp\left(-\frac{\Delta t}{\tau_a}\right) \tag{2.9}$$

In combination with implementing a refractory period $\Delta_{\text{abs}} > 0$ this model is able to accurately predict the spike pattern of many biological neurons, even for longer periods of time (Gerstner, Kistler, et al., 2014). Spike frequency adaptation corresponds closely to the mechanism of dynamic thresholds. In Section 2.3 the leaky integrate-and-fire model with dynamic threshold and refractoriness will be revisited in a discrete-time setting suitable for simulation.

For the remainder of this thesis, we will keep to this simple neuron model. This means, that we will not cover many important biological aspects of neurons such as models describing the full action potential shape, synapse models, plasticity and many more. For a more in-depth analysis, please refer to Gerstner, Kistler, et al., 2014.

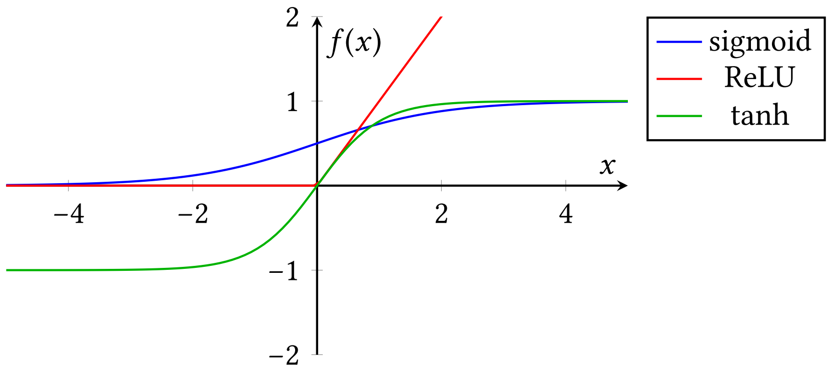


Fig. 2.4: Common non-linear functions for ANNs. The sigmoid function is defined as $\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}$, the Rectified Linear Unit function as $\text{ReLU}(x) = \max(0, x)$ and the hyperbolic tangent as $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

2.2 Artificial Neural Networks

In this section we will give a short introduction to artificial neural networks (ANN) and the backpropagation algorithm that enabled the success of deeper neural network architectures in fields such as image classification, language modelling, machine translation and many more.

Artificial neural networks are a class of models suitable for various tasks such as supervised learning, unsupervised learning and reinforcement learning. ANNs can be seen as an mathematical abstraction of biological neurons, having similar features: both receive some form of input, apply a non-linear transformation to the input and propagate the output to other neurons. One of the most common types of ANN, called feed-forward neural network, is completely stateless. Instead of simulating neurons in a biologically plausible way, the computation of the output is time-invariant and not dependent on the current state of the neuron. Figure 2.5 shows the structure of a simple feed-forward artificial neural network. It consists of an input layer, one or more hidden layers and an output layer. Each hidden layer applies a non-linear function $\phi(z^{(l)}) = \boldsymbol{a}^{(l)}$ to the linear function of the previous

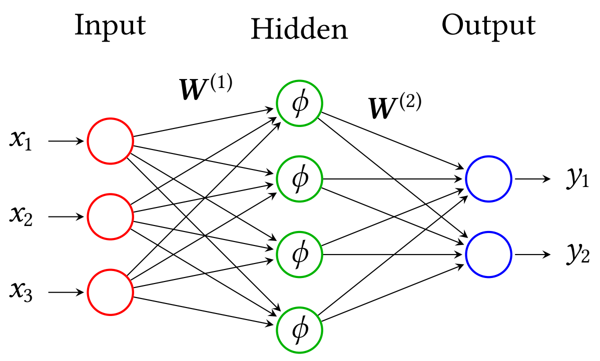


Fig. 2.5: Structure of an artificial neural network.

layer's output $z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)}$. A selection of common non-linear functions, often referred to as activation functions, can be seen in Figure 2.4. Mathematically, the network in Figure 2.5 can be represented as a function $f(x, \theta)$ where $x = a^{(1)}$ is the input vector and $\theta = [W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}]$ is the tensor holding the parameters $W^{(l)}, b^{(l)}$ of layer l.

$$f(x, \theta) = W^{(2)}\phi(W^{(1)}x + b^{(1)}) + b^{(2)} \tag{2.10}$$

2.2.1 Backpropagation

The training setup of a neural network is intimately related to the class of the problem we want to solve; for now, we will only consider supervised learning problems. In this context, we have a training set $X = \{x_1, \dots, x_n\}$ with known ground-truth values $Y^* = \{y_1^*, \dots, y_n^*\}$ of the function we want to approximate or class we want to assign. This allows the construction of loss functions $\mathcal{L}(Y, Y^*)$ that evaluates how well the models prediction $Y = \{y_i \,|\, y_i = f(x_i, \theta)\}$ fits the ground truth $\boldsymbol{Y}^*$. A common choice is the mean-squared-error loss

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} \left\| f(\boldsymbol{x}_i, \boldsymbol{\theta}) - \boldsymbol{y}_i^* \right\|_2^2 \tag{2.11}$$

for regression, where n is the number of samples in the dataset. Formulating the training of the neural network as optimization problem, we want to find a set of parameters $\boldsymbol{\theta}^*$ such that

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}}(\boldsymbol{Y}, \boldsymbol{Y}^*). \tag{2.12}$$

A popular method of solving these kinds of problems is to use gradient methods, since the loss surface is usually non-convex (Beck, 2014; Bishop, 2006) and an analytical solution is computationally infeasible or may not even exist. The arguably most commonly used gradient method today is gradient descent which updates the parameters in an iterative manner giving

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \eta \nabla \mathcal{L}_{\boldsymbol{\theta}^k}(\boldsymbol{Y}, \boldsymbol{Y}^*) \tag{2.13}$$

as the new parameters under the condition that the step size η is chosen sufficiently small enough to guarantee convergence to a local minimum and the loss function is differentiable (Beck, 2014). Intuitively, this means that we will take small steps on the loss surface in the direction of steepest descent towards some stationary point where the gradient vanishes and the loss decreases.

This leaves the problem of calculating the derivative $\nabla_{\boldsymbol{\theta}^k} \mathcal{L}(\boldsymbol{Y}, \boldsymbol{Y}^*)$ with respect to the parameters $\boldsymbol{\theta}$ for the data in the training set. All that is necessary is some basic calculus and the chain rule of calculus. Starting with the last layer of the example network in Figure 2.5, we will later generalize to arbitrary layers. The most intuitive way to perform this computation is to first visualize the computational graph of the entire network, as seen in Figure 2.6; then it becomes obvious on how to apply

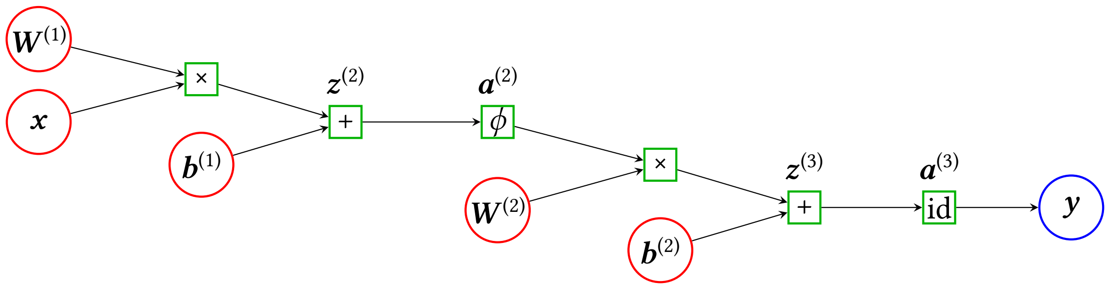


Fig. 2.6: Computational graph of the 2-layer neural network from Fig. 2.5. Parameters and input values are represented by ellipses whereas operations are represented as rectangles. Nodes with "×" refer to the inner product $\langle \cdot, \cdot \rangle$; "id" refers to the identity operation.

the chain rule to obtain the derivatives

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(2)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(3)}} \frac{\partial \boldsymbol{a}^{(3)}}{\partial \boldsymbol{z}^{(3)}}}_{\boldsymbol{\delta}^{(3)}} \frac{\partial \boldsymbol{z}^{(3)}}{\partial \boldsymbol{W}^{(2)}}, \tag{2.14}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(2)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(3)}} \frac{\partial \boldsymbol{a}^{(3)}}{\partial \boldsymbol{z}^{(3)}}}_{\boldsymbol{\delta}^{(3)}} \frac{\partial \boldsymbol{z}^{(3)}}{\partial \boldsymbol{b}^{(2)}} \tag{2.15}$$

The next step is to realize that the computation of the gradients of $\boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}$ with respect to $\mathcal{L}$ share a common term we will call $\boldsymbol{\delta}^{(3)}$. Not only can this term be used for the computation of both parameter derivatives; it is also very useful for computing gradients in earlier layers of the network:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(3)}} \frac{\partial \boldsymbol{a}^{(3)}}{\partial \boldsymbol{z}^{(3)}} \frac{\partial \boldsymbol{z}^{(3)}}{\partial \boldsymbol{a}^{(2)}} \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}} \frac{\partial \boldsymbol{z}^{(2)}}{\partial \boldsymbol{W}^{(1)}} \tag{2.16}$$

$$= \underbrace{\boldsymbol{\delta}^{(3)} \frac{\partial \boldsymbol{z}^{(3)}}{\partial \boldsymbol{a}^{(2)}} \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}}}_{\boldsymbol{\delta}^{(2)}} \frac{\partial \boldsymbol{z}^{(2)}}{\partial \boldsymbol{W}^{(1)}} \tag{2.17}$$

$$= \boldsymbol{\delta}^{(2)} \frac{\partial \boldsymbol{z}^{(2)}}{\partial \boldsymbol{W}^{(1)}} \tag{2.18}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(1)}} = \boldsymbol{\delta}^{(2)} \frac{\partial \boldsymbol{z}^{(2)}}{\partial \boldsymbol{b}^{(1)}} \tag{2.19}$$

Since all of the layers of the neural network are structured the same way, we can take advantage of this and solve some of the partial derivatives for general neural network architectures with L layers:

$$\frac{\partial \boldsymbol{z}^{(l+1)}}{\partial \boldsymbol{a}^{(l)}} = \boldsymbol{W}^{(l)^T} \tag{2.20}$$

$$\frac{\partial \boldsymbol{z}^{(l+1)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{a}^{(l)^T} \tag{2.21}$$

$$\frac{\partial \boldsymbol{z}^{(l+1)}}{\partial \boldsymbol{b}^{(l)}} = \mathbf{1} \tag{2.22}$$

Using these results, we can now formulate generalized derivatives for the parameters of the network using

$$\boldsymbol{\delta}^{(l)} = \begin{cases} \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l)}} \frac{\partial \boldsymbol{a}^{(l)}}{\partial z^{(l)}} & \text{if } l = L \\ \boldsymbol{\delta}^{(l+1)} \boldsymbol{W}^{(l)^T} \frac{\partial \boldsymbol{a}^{(l)}}{\partial z^{(l)}} & \text{else} \end{cases} \tag{2.23}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}} = \nabla_{\boldsymbol{W}^{(l)}} \mathcal{L} = \boldsymbol{\delta}^{(l+1)} \boldsymbol{a}^{(l)^T} \tag{2.24}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} = \nabla_{\boldsymbol{b}^{(l)}} \mathcal{L} = \boldsymbol{\delta}^{(l+1)}. \tag{2.25}$$

The derivatives $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}}$ and $\frac{\partial \boldsymbol{a}^{(l)}}{\partial z^{(l)}}$ depend on the choice of loss function and nonlinear transformation, respectively, and cannot be reduced any further.

Considering common loss functions such as mean-squared-error loss, their output is defined using the value $\boldsymbol{a}^{(L)}$ of the network; the derivative of the weights $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}}$ also depends on $\boldsymbol{a}^{(l)}$. The calculation of $\boldsymbol{\delta}^{(l)}$ depends on $\frac{\partial \boldsymbol{a}^{(l)}}{\partial z^{(l)}}$. Thus, in order to compute all required partial derivatives to update the parameters of the network, it is necessary to use all intermediary values $\boldsymbol{a}^{(l)}, \boldsymbol{z}^{(l)}$. The straightforward way to compute and store all these values is to evaluate the network; this is also referred to as the forward-pass, because the evaluation starts with the first network layer and ends with the last one. After that, by using the intermediary values to calculate parameter updates, the calculation of the partial derivatives starts at the last layer and ends at the first layer – this is called the backward-pass. Applying the forward- and the backward-pass together is called the backpropagation algorithm (Bishop, 2006) and can be seen in Algorithm 1. In the listing, some abuse of notation is used to refer to the derivatives of the parameters $\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}$ as $\nabla \boldsymbol{W}^{(l)}, \nabla \boldsymbol{b}^{(l)}$ instead of $\nabla_{\boldsymbol{W}^{(l)}} \mathcal{L}, \nabla_{\boldsymbol{b}^{(l)}} \mathcal{L}$ for sake of readability. Furthermore, the derivatives $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l)}}, \frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{z}^{(l)}}$ have been replaced with $\nabla \mathcal{L}(\boldsymbol{a}^{(l)}, \boldsymbol{y}^{*}), \nabla \phi^{(l)}(\boldsymbol{z}^{(l)})$.

One of the key factors of the success of backpropagation besides its simplicity and formulation as a general optimization problem is its efficiency. As seen in Algorithm 1, it is clear that the GRADIENT_DESCENT procedure is linear in its computation cost in the number of parameters of the network and the number of training samples. Both the FORWARD and BACKWARD procedure use the elements in the parameter tensor $\boldsymbol{\theta}$ once, thus having $\mathcal{O}(|\boldsymbol{\theta}|)$ computation time cost when thinking of inner products as operations with per-element cost $\mathcal{O}(1)$. The overall computational cost increases linearly with the number of training samples $|\boldsymbol{X}|$ and the number of training iterations K. On modern GPUs, matrix operations can be executed in parallel very efficiently, reducing the training time of large data sets considerably.

An alternative to the gradient descent method described above is called stochastic gradient descent. Instead of iterating over the entire training set, small subsets called minibatches are selected at random from the shuffled training set and these are processed alike standard gradient descent. This has several advantages: First of all, it inherently makes the gradient more noisy allowing the method to escape local stationary points. Second, it usually converges faster if there is some amount of redundancy in the training data. Many large datasets contain similar samples; if the dataset gets shuffled randomly or preferably according to the underlying distribution of the data, performing updates on the minibatches instead of the full training dataset produces similar effects. Thus it is possible to perform more updates in the same amount of time leading to much faster convergence.

Algorithm 1 Backpropagation algorithm with gradient descent

```
procedure FORWARD($\boldsymbol{x}, \boldsymbol{\theta}$)    ▷ Calculates forward-pass for one sample
    for $l \leftarrow 1, \dots, L$ do
        $\boldsymbol{z}^{(l+1)} \longleftarrow \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}$
        $\boldsymbol{a}^{(l+1)} \longleftarrow \phi^{(l+1)}(\boldsymbol{z}^{(l+1)})$
    end for
    return $\boldsymbol{a}, \boldsymbol{z}$
end procedure

procedure BACKWARD($\boldsymbol{a}, \boldsymbol{z}, \boldsymbol{y}^*, \boldsymbol{\theta}$)    ▷ Calculates backward-pass for one sample
    $\boldsymbol{\delta}^{(L)} \longleftarrow \nabla\phi^{(L)}(\boldsymbol{z}^{(L)}) \nabla\mathcal{L}(\boldsymbol{a}^{(L)}, \boldsymbol{y}^*)$
    for $l \leftarrow L-1, \dots, 1$ do
        $\boldsymbol{\delta}^{(l)} \longleftarrow \boldsymbol{\delta}^{(l+1)} \boldsymbol{W}^{(l)^T} \nabla\phi^{(l-1)}(\boldsymbol{z}^{(l)})$
    end for
    for $l \leftarrow 1, \dots, L$ do
        $\nabla \boldsymbol{W}^{(l)} \longleftarrow \boldsymbol{\delta}^{(l+1)} \boldsymbol{a}^{(l)^T}$
        $\nabla \boldsymbol{b}^{(l)} \longleftarrow \boldsymbol{\delta}^{(l+1)}$
    end for
    return $\nabla \boldsymbol{W}, \nabla \boldsymbol{b}$
end procedure

procedure GRADIENT_DESCENT($\boldsymbol{X}, Y^*, K, \eta$)
    initialize $\boldsymbol{\theta}$
    for $k \leftarrow 1, \dots, K$ do
        set $\nabla \boldsymbol{W}, \nabla \boldsymbol{b}$ to $\boldsymbol{0}$
        for $i \leftarrow 1, \dots, |X|$ do
            $\boldsymbol{a}_i, \boldsymbol{z}_i \longleftarrow$ FORWARD($\boldsymbol{x_i}, \boldsymbol{\theta}$)
            $\nabla \boldsymbol{W}_i, \nabla \boldsymbol{b}_i \longleftarrow$ BACKWARD($\boldsymbol{a}_i, \boldsymbol{z}_i, \boldsymbol{y}_i^*, \boldsymbol{\theta}$)
            $\nabla \boldsymbol{W} \longleftarrow \nabla \boldsymbol{W} + \frac{1}{|X|} \nabla \boldsymbol{W}_i$
            $\nabla \boldsymbol{b} \longleftarrow \nabla \boldsymbol{b} + \frac{1}{|X|} \nabla \boldsymbol{b}_i$
        end for
        $\boldsymbol{W} \longleftarrow \boldsymbol{W} - \eta \nabla \boldsymbol{W}$
        $\boldsymbol{b} \longleftarrow \boldsymbol{b} - \eta \nabla \boldsymbol{b}$
    end for
end procedure
```

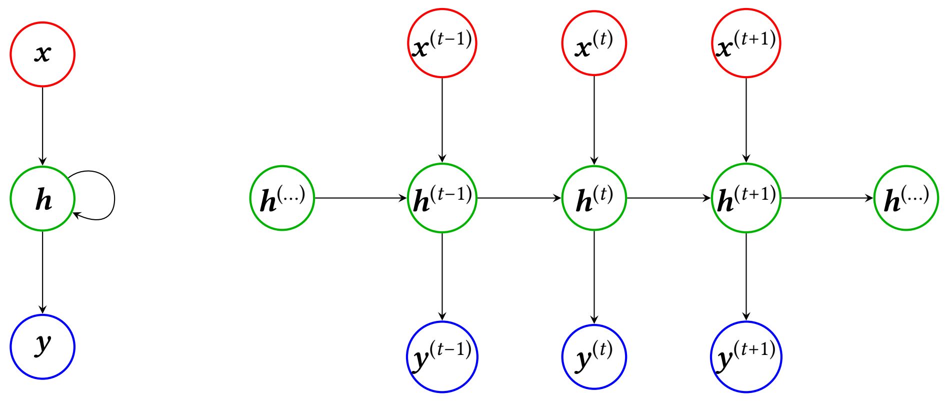


Fig. 2.7: A recurrent neuron. The input $\boldsymbol{x}$ and the last state of the neuron is processed by a function f to produce the next state. From this new state, the output $\boldsymbol{y}$ is derived. **Left:** The original recurrent version of the network where the recurrent connection is delayed for one time step. **Right:** The unrolled version of the same neuron where the computation of the next hidden state becomes clear. Adapted from Goodfellow, Bengio, and Courville, 2016.

2.2.2 Backpropagation Through Time

Another important class of ANNs besides the feed-forward network is called recurrent neural network (RNN). As the name suggests, some parts of the output of the neuron are used as input again; this is called the state of the neuron. Whereas regular feed-forward networks are completely state-less, meaning that they do not "remember" anything between two evaluations of different input values, recurrent neural networks can store or encode parts of the input they receive in their state.

This can be useful for a wide variety of applications that require recalling the occurrence or absence of certain observations such as words in sentences or the tracking of objects in a scene. A graphical representation of this can be seen in Figure 2.7 on the left. Much like regular feed-forward networks, it still produces output that could be passed on to other layers. The functional dependence on the previous state directly leads to a recursive definition of the state calculation and a sequence of state values where each element in the sequence depends only on its immediate predecessor. Although the elements in the sequence do not necessarily follow a certain temporal ordering, the element indices of the sequence are usually referred to as time steps. The sequence of states and their transition to follow-up states can be seen in Figure 2.7 on the right-hand side. This step-wise transitioning from state to state up to a fixed length is called unrolling. A common definition of the state (Goodfellow, Bengio, and Courville, 2016), also called hidden unit $\boldsymbol{h}$, is

$$\boldsymbol{h}^{(t+1)} = f(\boldsymbol{h}^{(t)}, \boldsymbol{x}^{(t)}, \boldsymbol{\theta}) \tag{2.26}$$

where t refers to the t-th element in the sequence of states and input values and f can be seen as a function akin to that of a feed-forward neural network as in equation (2.10). One aspect to consider is that even if the input values $\boldsymbol{x}$ do not follow a specific meaningful order on their own, the calculation of the state values structures the input values in a sequence indirectly. If one thinks of the recurrent neural network as a natural extension of the feed-forward network where the previous state of the network is used as input again, we can immediately formalize this relation as follows:

$$\boldsymbol{z}^{(t)} = \boldsymbol{b}_z + \boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} \tag{2.27}$$

$$\boldsymbol{h}^{(t)} = \phi(\boldsymbol{z}^{(t)}) \tag{2.28}$$

Here, the superscript notation $\cdot^{(t)}$ does not refer to the t-th layer in the feed-forward network but to the time step t. The last missing piece is the calculation of the "visible" output value

$$\boldsymbol{y}^{(t)} = \boldsymbol{b}_y + \boldsymbol{W}_y \boldsymbol{h}^{(t)} \tag{2.29}$$

The relationship to the feed-forward network can be prominently seen in the formal definition in equations (2.27) to (2.29). This leaves the problem of a learning rule for training this kind of artificial neural network. First, it is important to observe the structure of equation (2.26). The parameter vector $\boldsymbol{\theta}$ does not depend on the current time step t; therefore the same parameters are used for all time steps in the calculation. This reduces the number of parameters considerably compared to using different $\boldsymbol{\theta}^{(t)}$ for every time step.

The relation of recurrent neural networks to feed-forward neural networks brings the use of backpropagation as the choice of learning rule to mind at once. However, in the feed-forward setting there is a clear structure of the network and how to apply backpropagation with respect to the individual layers. The process of unrolling the computational graph as in Figure 2.8 for a certain chosen sequence length gives a result very similar to that of a feed-forward network. Now, each time step in the computational graph represents one "layer" of an equivalent feed-forward network; for some fixed chosen unrolled sequence length the same backpropagation algorithm can be applied to training the network.

The trainable parameters $\boldsymbol{W_h}$, $\boldsymbol{W_x}$, $\boldsymbol{W_y}$, $\boldsymbol{b_z}$, $\boldsymbol{b_y}$ are used for all time steps in the computational graph without allowing individual sets of parameters for certain time steps; this is commonly referred to as parameter sharing (Goodfellow, Bengio, and Courville, 2016). This leads to a set of unique problems for recurrent neural networks that is not immediately obvious. Considering Figure 2.8, during the computation of $\boldsymbol{z}^{(t)}$, we apply the weight $\boldsymbol{W_h}$ for each time step in the graph. Disregarding the other elements of the computation of $\boldsymbol{z}^{(t)}$ and the non-linear function ϕ the state is computed as $\boldsymbol{h}^{(t)} = \boldsymbol{W}_{\boldsymbol{h}}^{t}\boldsymbol{h}^{(0)}$. Repeated potentiation of the same weight matrix will force all values towards 0 or ∞. In the literature, this problem is known as the vanishing or exploding gradients problem. The impact of this can be seen in Algorithm 1 in line 11, where $\boldsymbol{\delta}^{(l)}$ is calculated based on $\boldsymbol{W}^{(l)}$. Several solutions exist to address these issues somewhat successfully. A noteworthy example is gradient clipping where the exploding gradient problem is solved by hard-resetting the gradient to some threshold value if this values is exceeded.

Regular feed-forward networks do not suffer from the vanishing or exploding gradient problem as much because they employ different sets of weights in different

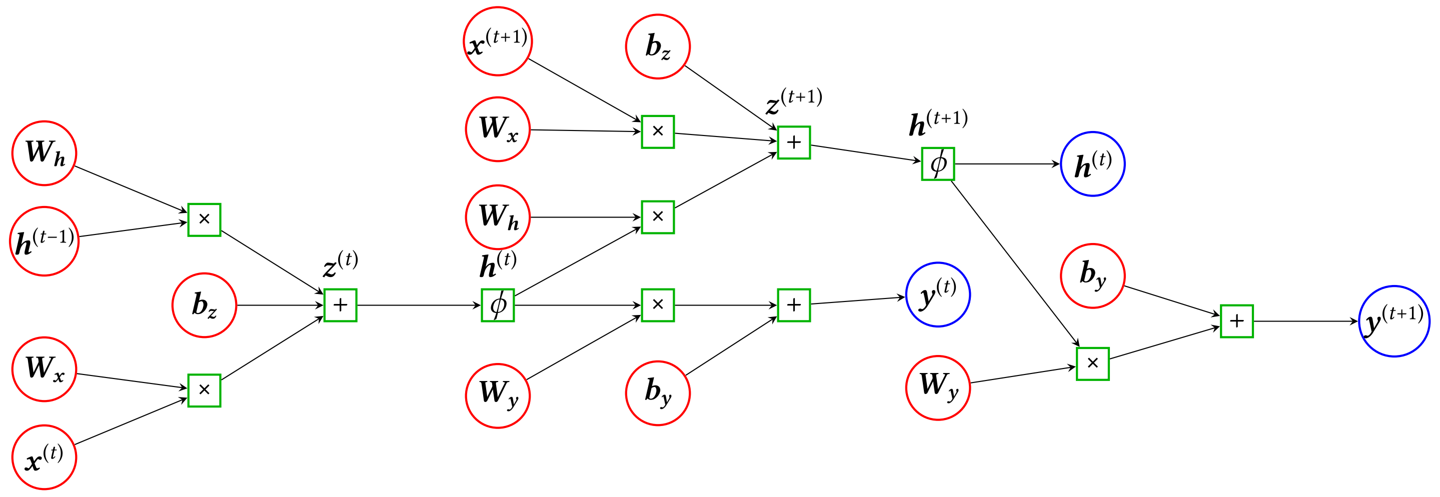


Fig. 2.8: Computational graph of the unrolled recurrent neural network from Fig. 2.7. Parameters and input values are represented by ellipses whereas operations are represented as rectangles. Nodes with "×" refer to the inner product $\langle \cdot, \cdot \rangle$.

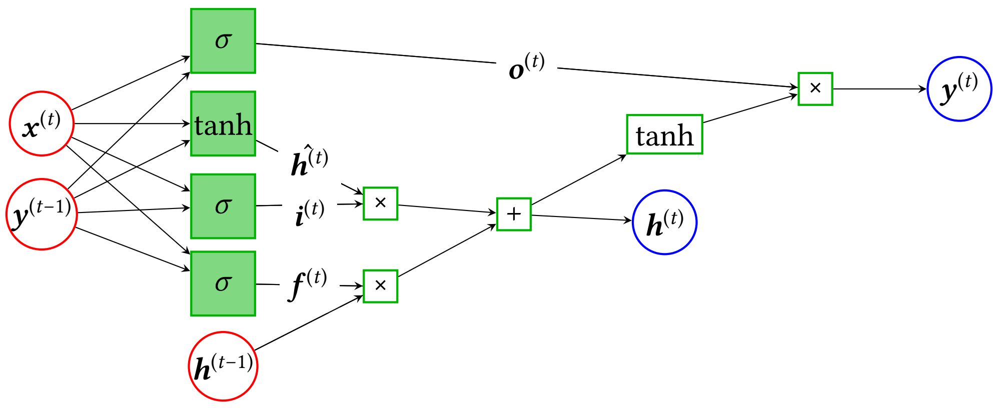


Fig. 2.9: Computational graph of the LSTM network. The filled green squares represent a complete feed-forward layer with their own set of weights and biases used in a linear function followed by the non-linear function specified in the label of the square. $\hat{h}^{(t)}$, $i^{(t)}$, $f^{(t)}$, $o^{(t)}$ refer to the state update, the input gate, the forget gate and the output gate, respectively.

layers; thus the problem of repeated self-potentiation does not occur in this extreme form. The most prominent solution to both the vanishing and exploding gradient problem for RNNs are gated recurrent neural networks such as the long short-term memory (LSTM) network (Hochreiter and Schmidhuber, 1997). Observing the behaviour of feed-forward networks, the idea is to adapt the flow of information such that the calculation of the state does not suffer the same repeated potentiation of weights as regular RNNs. This, in effect, produces the same behaviour as having different weights at each time step in the unrolled computational graph.

In Figure 2.9 an overview of the LSTM network can be seen. To modulate the flow of information dynamically depending on the previous state and the input, four new separate feed-forward layers called gates are introduced. Each of these gates serves a different purpose and produces vectors that control each element of the state vector independently. The input gate $i^{(t)}$ controls how much of the state update $\hat{h}^{(t)}$ actually gets added to the old state $h^{(t-1)}$ to create the new state $h^{(t)}$. The forget gate $f^{(t)}$ adjusts the amount of information that is kept from the old state for the computation of the new state. The output gate $o^{(t)}$ regulates what parts of the state processed by the hyperbolic tangent function make up the output of the LSTM network.

The sigmoid unit produces values in the range $[0, 1]$ and intuitively is a suitable function for controlling the amount of information flow as gate; if it is zero no information flows, if it is one all information is kept. The hyperbolic tangent used for the state update $\hat{\boldsymbol{h}^{(t)}}$ with range $[-1, 1]$ adding to and subtracting from the current state to compute the new state; a sigmoid function would not make sense. Both the sigmoid and the hyperbolic tangent function are well-behaved functions with regard to first- and second-order derivatives.

In this section we have argued that recurrent neural networks are good choices when operating on sequences of data. In fact, spiking neural networks in the brain can be seen as recurrent neural networks; in the brain there are no feedforward-only layers of neurons similar to artificial neural networks. Thus it is highly interesting to explore approaches to extending backpropagation through time to spiking neural networks to benefit from the advantages of efficient gradient descent optimization. In the following section we will review a recent method of approximating the derivative of spikes with respect to the membrane voltage of neurons.

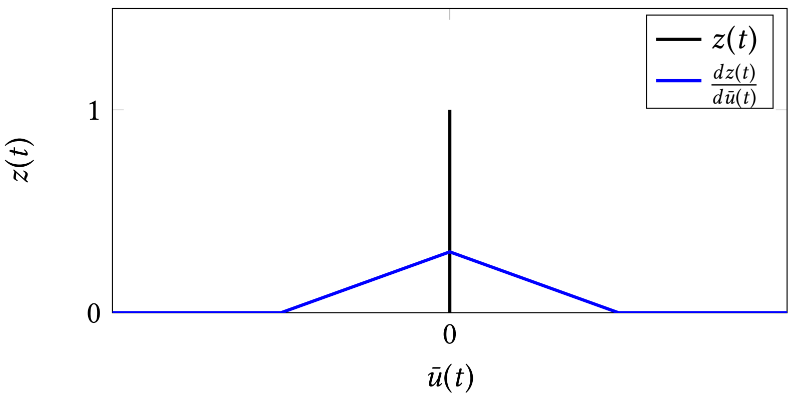


Fig. 2.10: Illustration of pseudoderivative of the spike function.

2.3 Backpropagation in SNNs

In Section 2.1 use cases for spiking neural networks such as neuromorphic hardware and a generally better understanding of the mammalian brain have been motivated. However, explicit strategies for training more biologically plausible neural network models such as the leaky integrate-and-fire model were not mentioned. There is a plethora of neuron models for describing neural dynamics; learning rules for networks of biological neurons with varying degrees of complexity have been proposed. Common choices of describing long term changes in the brain are spike-time dependent plasticity (STDP) observed by Markram et al., 1997 or the Clopath learning rule (Clopath et al., 2010).

In previous works, biologically plausible learning rules had severe drawbacks. Either they were only suitable for a very small number of neurons, had very slow convergence rates or worked only when applied to specific neuron models in certain configurations. Although backpropagation and especially backpropagation through time is not biologically plausible, the success of artificial neural networks in combination with backpropagation make this method highly interesting for spiking neural networks. Being able to accurately assign error contributions of single neurons in a population is superior to local plasticity mechanisms. The reason why backpropagation is not applicable to spiking neuron models such as the integrate-and-fire model is that spikes are not differentiable. If we consider the output behaviour of LIF neurons, their output $z(t) = 0$ everywhere except when the membrane voltage $u(t)$ reaches the firing threshold ϑ where $z(t) = 1$; it is clear that this function is not differentiable. Thus it has been proposed by Esser et al., 2016; Courbariaux and Bengio, 2016 to approximate the derivative of the spike function for binary neurons. This idea has been extended to adaptive leaky integrate-and-fire neurons by Bellec et al., 2018 where the derivative is given as

$$\frac{dz(t)}{d\bar{u}(t)} = \phi \max\{0, 1 - |\bar{u}(t)|\} \tag{2.30}$$

where $\bar{u}(t) = \frac{u(t)-\vartheta(t)}{\vartheta(t)}$ is the normalized membrane potential. A dampening factor $\phi = 0.3$ seems to be necessary to keep the training of recurrent spiking neural networks stable. An illustration of the derivative is given in Figure 2.10. It smoothly increases and decreases with $\bar{u}(t)$, it is not linearly based on t alone.

With this in mind, we can put all previous sections on background information together and train leaky integrate-and-fire neurons with backpropagation through time in a reinforcement learning setting to solve interesting tasks. In the next chapter, we will use this unified framework to solve classic and state-of-the-art benchmarks of reinforcement learning with spiking neural networks.

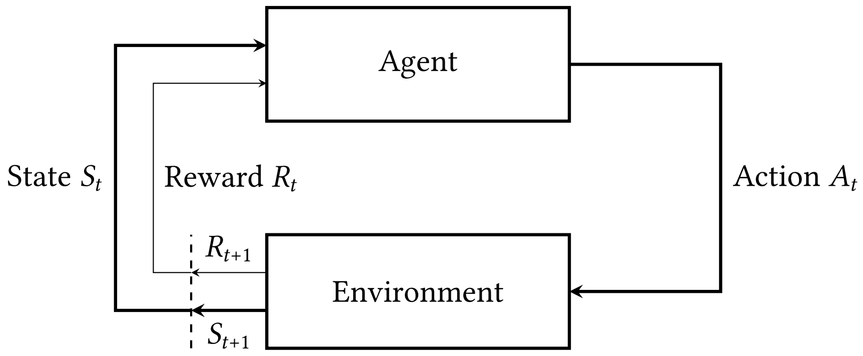


Fig. 2.11: Typical reinforcement learning problem. Given a state S_t and a reward R_t determine the next action A_t. This in turn leads to the new state S_{t+1} and reward R_{t+1}. The reward signal is a scalar whereas both the state and the action can be vector-valued. Adapted from Sutton and Barto, 2018.

2.4 Reinforcement Learning

In the literature reinforcement learning is described as one of the three broad categories of machine learning: supervised learning, unsupervised learning and reinforcement learning (Sutton and Barto, 2018; Bishop, 2006; Goodfellow, Bengio, and Courville, 2016). They differ both in their goal and in the way the input data is structured. In this section an introduction to basic reinforcement learning terminology including an overview of the algorithm used for all experiments (Schulman, Wolski, et al., 2017) will be given.

As discussed in Section 2.2 in the supervised learning paradigm the goal is to train a model to map given input data to some known target value or target class and achieve high accuracy on unseen test data. For the unsupervised learning scenario, the training dataset does not contain any target values and the goal is to find hidden structures or groups within the data. Reinforcement learning is different insofar that it does not contain target values for specific examples in the training dataset; however a special signal, usually called a reward signal, is given by the environment to an actor. Figure 2.11 shows the general setup of the reinforcement learning problem. Here the goal is to maximize the reward achieved over all timesteps choosing optimal actions to reach subsequent states yielding high rewards. The difficulty lies in choosing actions such that the state space is sufficiently explored to find states with high reward. An additional layer of difficulty may arise from the structure of the reward signal; some problems might award non-zero rewards only after many time steps. Thus learning to choose good actions is completely decoupled from the immediate reward received after taking the action. Therefore the overarching goal is to balance exploring and exploiting good states in order to maximize the long-term cumulative reward.

The behaviour of the agent in the environment is described using a policy π. This policy can be seen as a function $\pi(a \mid s) : \mathcal{S} \longrightarrow \mathcal{A}(s), a \in \mathcal{A}(s), s \in \mathcal{S}$ mapping from values in the state space $\mathcal{S}$ to values in the action space $\mathcal{A}$. Generally policies can be stochastic or deterministic. States in $\mathcal{S}$ may include partial observations of the environment or specific aspects of the actor in the environment and should only include relevant information to solve the given problem. Examples are external information such as the absolute position of the robot in the world or internal information such as sensor readings like joint angles of extremities or camera images. Thus the state can be high-dimensional and difficult to explore efficiently. Actions in $\mathcal{A}$ may also be high-dimensional and vector-valued; controlling multiple actuated joints of a robotic platform in a continuous setting or choosing whether to go left or right in a discrete setting may be seen as simple examples. Choosing a particular action transitions the environment into a new state, similar to the sequence of states discussed in Section 2.2.2 this transition immediately gives a notion of discrete timesteps.

In order to formalize the abstract description of the system described in Figure 2.11 some assumptions need to be made. From the study of probability theory the concept of Markov chains and Markov processes come to mind, allowing to reason about the future state of a dynamical system solely based upon its current state in a probabilistic fashion. This formalism can be directly applied to the problem of reinforcement learning when taking the chosen action and the given reward into

$$\begin{tikzcd}
S_t \arrow[r, "A_t"] & (S_{t+1}, R_{t+1}) \arrow[r, "A_{t+1}"] & (S_{t+2}, R_{t+2}) \arrow[r, "A_{t+2}"] & \cdots \arrow[r, "A_{T-1}"] & (S_T, R_T)
\end{tikzcd}$$

Fig. 2.12: Trajectory of observed states. In the setting of this thesis, we will mostly be concerned with the actually observed trajectory and return after the agent reached a terminal state instead of the expected trajectory most probable or profitable before choosing an action.

account. In this extended definition, Markov chains are known as Markov decision processes. Given a state S_t and an action A_t we want to know the probability of future state S_{t+1} and reward R_{t+1} occurring. We will follow the notation of Sutton and Barto, 2018 for the remainder of this section. The dynamics of the Markov decision process for random variables s, a, s', r are given by

$$p(s', r \mid s, a) = \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}, \tag{2.31}$$

where

$$\Pr\{S_t = s', R_t = r \mid S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \dots, S_1 = s_1, A_1 = a_1\} \tag{2.32}$$
$$= \Pr\{S_t = s', R_t = r \mid S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \tag{2.33}$$

fulfilling the Markov property. In practice it would be infeasible to store and process all previous states for all new actions to be decided upon indefinitely, especially for continuous state spaces. Thus it is reasonable to assume that the current state should contain all the information necessary to make informed decisions. From now on, we will mostly be concerned with the actual values of s, a, s' and r taken at time t, which are S_t, A_t, S_{t+1} and R_{t+1}, respectively. This chain is illustrated in Figure 2.12.

Previously we defined our goal to maximize the cumulative reward in the long-term. This definition is fuzzy; it does not specify what long-term means. More formally we want to maximize the return G_t defined over an sequence of states called episode

$$G_t = R_{t+1} + \cdots + R_T \tag{2.34}$$

where T is the length of the sequence. This means that the agent acts in the environment and produces a trajectory of states and rewards until termination of the episode; this is visualized in Figure 2.12. More generally, for continuous tasks where T can be infinity, it is practical to discount future rewards

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \tag{2.35}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.36}$$

such that infinite sums in the case $T = \infty$ have finite values if $\gamma < 1$ and the reward sequence $\{R_1, \dots, R_T\}$ is bounded. It allows to control how much emphasis is put on future rewards; if potential rewards are too far in the future they might be less relevant even for certain finite tasks.

With this notion of return as the weighted sum of future rewards starting from a certain time step t in a state s we can express the value of a state. The value of a state is given as

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{2.37}$$

which is the expected return of a state s and policy π when starting in state s and following π for choosing future actions. This concept can be extended to evaluate state-action pairs (s, a) as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{2.38}$$

where the expected return of state s after taking action a and subsequently following policy π is considered (Sutton and Barto, 2018). For a small number of states, it may be possible to continuously visit all states and keep accurate estimates of the value in lookup tables; for large continuous state spaces this is infeasible. Thus approximations of state values have to be made using parametrized functions $\hat{v}(s, \boldsymbol{\theta}_v) \approx v_\pi(s)$ or $\hat{q}(s, a, \boldsymbol{\theta}_q) \approx q_\pi(s, a)$. Accurately estimating value functions is essential for evaluating policies and improving them consistently. A popular choice of function approximators are artificial neural networks (Mnih, Kavukcuoglu, et al., 2015; Schulman, Levine, et al., 2015; Schulman, Wolski, et al., 2017; Hessel et al., 2018; Espeholt et al., 2018).

A common extension of returns G_t is the idea of bootstrapping. Instead of finishing an episode until termination to get the full exact return, one may wish to update parameters earlier to incorporate changes faster. To still get sensible information about the future rewards of a state, the estimated value function is incorporated into the return

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}) \tag{2.39}$$

to form the n-step return $G_{t:t+n}$. The n-step return can be directly used in the definition of state-value (2.37) and action-value (2.38) functions.

The goal of optimizing the parameters $\boldsymbol{\theta}_v$ such that $\hat{v}(s, \boldsymbol{\theta}_v) \approx v_\pi(s)$ immediately gives multiple choices of objective function that can be minimized, a default choice is of course the sum of squared errors

$$\mathcal{L}^v(\boldsymbol{\theta}_v) = \sum_{s \in \mathcal{S}} \left(v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}_v)\right)^2 . \tag{2.40}$$

Obviously the true value of $v_\pi(s)$ is not known a priori, but noisy estimates based on values observed by sampling a trajectory of states and returns can be made by acting in the environment according to policy π. One choice of an noisy estimate would be $v_\pi(s) \approx G_t$; this is known as Monte Carlo sampling in reinforcement learning literature. This loss is then defined over the observed returns for all visited states during acting in the environment and can be written as

$$\mathcal{L}^v(\boldsymbol{\theta}_v) = \sum_{t=0}^{T} \left(G_t - \hat{v}(S_t, \boldsymbol{\theta}_v)\right)^2 . \tag{2.41}$$

With the assumption that our choice of value function approximator is differentiable, the plethora of non-linear optimization methods based on gradient information is available (Beck, 2014); this has the advantages that convergence guarantees can be made and the speed of convergence is usually faster. Similarly, it would be convenient to use a parametrized differentiable policy $\pi(a \mid s, \boldsymbol{\theta}_\pi)$ instead of a per-state policy $\pi(a \mid s)$ discussed so far.

As discussed in Section 2.2.1 we want to optimize our parameters $\boldsymbol{\theta}_\pi$ according to an objective function $\mathcal{L}(\boldsymbol{\theta}_\pi)$ and forms of gradient descent are the method of choice to solve this problem. According to the policy gradient theorem (Sutton, McAllester, et al., 2000; Sutton and Barto, 2018), a good choice of objective function we want to maximize is

$$\mathcal{L}(\boldsymbol{\theta}_\pi) = v_\pi(s_0). \tag{2.42}$$

Thus, we need a differentiable value function v_π in order to apply gradient ascent steps

$$\boldsymbol{\theta}_\pi^{k+1} = \boldsymbol{\theta}_\pi^{k} + \eta \nabla v_\pi(s_0). \tag{2.43}$$

The policy gradient theorem states that asymptotic equality of $\nabla v_\pi(s_0)$ is given by

$$\begin{aligned} \nabla v_\pi(s_0) &\asymp \sum_s \mu(s) \sum_a \nabla \pi(a \mid s) q_\pi(s, a) && (2.44) \\ &= \mathbb{E}_\pi \left[\sum_a \nabla \pi(a \mid s) q_\pi(s, a) \right] && (2.45) \end{aligned}$$

which is used in the REINFORCE algorithm (Williams, 1992) to update the policy parameters, where $\mu(s)$ is the probability of state s occurring. At each environment step (t) for training iteration k the update

$$\boldsymbol{\theta}_\pi^{k,(t+1)} = \boldsymbol{\theta}_\pi^{k,(t)} + \eta G_t \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_\pi^{k,(t)}) \tag{2.46}$$

is applied. With this, iterative updates of both the parametrized policy and value function are possible.

Another common improvement on standard policy gradient methods is including a baseline $b(s)$ into the gradient $\nabla v_\pi(s)$ as

$$\nabla v_\pi(s_0) \asymp \sum_s \mu(s) \sum_a \nabla \pi(a \mid s) \left(q_\pi(s,a) - b(s)\right) \tag{2.47}$$

An intuitive choice of baseline $b(s)$ is $\hat{v}(s)$; similarly to subtracting the mean value from a set of data it puts emphasis on values above the baseline and vice versa. Without the baseline, choosing an action from a set of similarly valued actions is difficult. The difference $\mathbb{A}(S_t, A_t) = q_\pi(S_t, A_t) - \hat{v}(S_t, \boldsymbol{\theta}_v)$ is known as advantage in the literature. Using bootstrapping whilst learning both policy and value function parameters simultaneously in one update belongs to the class known as actor-critic methods. The update of n-step actor-critic methods looks as follows:

$$\boldsymbol{\theta}_\pi^{k,(t+1)} = \boldsymbol{\theta}_\pi^{k,(t)} + \eta \mathbb{A} \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_\pi^{k,(t)}) \tag{2.48}$$

This leaves the issue of choosing parametrized functions $\pi(a \mid s, \boldsymbol{\theta}_\pi)$ that are differentiable and well-behaved. For discrete action spaces Sutton and Barto, 2018 suggest using a parametrized soft-max distribution

$$\pi(a \mid s, \boldsymbol{\theta}_\pi) = \frac{e^{h(s,a,\boldsymbol{\theta}_\pi)}}{\sum_b e^{h(s,b,\boldsymbol{\theta}_\pi)}} \tag{2.49}$$

with $h(s, a, \boldsymbol{\theta}_\pi)$ being a differentiable function for given state-action pairs under the given parametrization; examples for h include artificial neural networks or spiking neural networks. In the continuous action space case, a common approach is to learn the parameters of a chosen distribution such as a Gaussian distribution

$$\pi(a \mid s, \boldsymbol{\theta}_\pi) = \frac{1}{\sigma(s, \boldsymbol{\theta}_\sigma)\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s, \boldsymbol{\theta}_\mu))^2}{2\sigma(s, \boldsymbol{\theta}_\sigma)^2}\right) \tag{2.50}$$

where $\boldsymbol{\theta}_\pi = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^T$ are the parameters to be learned. Much alike to $h(s, a, \boldsymbol{\theta}_\pi)$, $\mu(s, \boldsymbol{\theta}_\mu)$ and $\sigma(s, \boldsymbol{\theta}_\sigma)$ are differentiable functions.

With this basic knowledge of policy gradient methods and actor-critic methods, a brief overview of the proximal policy optimization (Schulman, Wolski, et al., 2017) algorithm (PPO) will be given. The main contribution of PPO is a new objective function based on conservative policy iteration (Kakade and Langford, 2002) and trust region policy optimization (Schulman, Levine, et al., 2015) suitable for general stochastic policies. Conservative policy iteration (CPI) investigates a major issue of standard policy gradient methods resulting in insufficient exploration without a large number of samples from the environment to estimate the gradient direction and magnitude well.

They propose optimizing a different objective function

$$\mathcal{L}^{CPI}(\boldsymbol{\theta}_\pi) = \mathbb{E}_{\boldsymbol{\theta}_{\text{old}}}\left[\frac{\pi_{\boldsymbol{\theta}}(A_t \mid S_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(A_t \mid S_t)}(q_\pi(S_t, A_t) - \hat{v}(S_t, \boldsymbol{\theta}_v))\right] = \mathbb{E}_{\boldsymbol{\theta}_{\text{old}}}[\rho\mathbb{A}] \tag{2.51}$$

where $\rho = \frac{\pi_{\boldsymbol{\theta}}(A_t \mid S_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(A_t \mid S_t)}$ is known as the importance sampling ratio in the literature. This ratio is commonly used in off-policy sampling, where the goal is to learn about a target policy by behaving according to a different but similar policy. This means that the behaviour policy at least sometimes needs to visit the same states as the target policy. In the CPI setting, the target and behaviour policy are the current and previous iterates of the same policy and the ratio describes how much the new policy differs from the old one.

Schulman, Levine, et al., 2015 found that performing updates on this new objective function without complicated constraints results in too large updates to the policy.

Algorithm 2 Proximal Policy Optimization (PPO)

```
1: while termination condition not met do
2:     for actor $1, \dots, N$ do
3:         Sample with policy $\pi_{\theta_{\text{old}}}$ for $T$ steps from the environment
4:         Compute the advantage $\mathbb{A}$ for all steps
5:     end for
6:     Optimize objective $\mathcal{L}^{CLIP+v+H}(\boldsymbol{\theta}_\pi)$ for $K$ epochs and minibatch size $M$
7: end while
```

In their newer work (Schulman, Wolski, et al., 2017), they limit the update by clipping the importance sampling ratio to a ε-bound

$$\mathcal{L}^{CLIP}(\boldsymbol{\theta}_\pi) = \mathbb{E}_{\boldsymbol{\theta}_{\text{old}}}\left[\min\left(\rho\mathbb{A}, \text{clip}(\rho, 1-\varepsilon, 1+\varepsilon)\mathbb{A}\right)\right]. \tag{2.52}$$

preventing changes to $\boldsymbol{\theta}_\pi$ that move ρ too fast from 1; the closer $\boldsymbol{\theta}_\pi$ is to $\boldsymbol{\theta}_{\pi_{\text{old}}}$, the closer it is to 1. The advantage of this new structure is that it does not need explicit constraints like trust region policy optimization.

When the policy objective (2.52) and the value function objective (2.40) get combined, we get

$$\mathcal{L}^{CLIP+v+H}(\boldsymbol{\theta}_\pi) = \mathbb{E}_{\boldsymbol{\theta}_{\text{old}}}\left[\mathcal{L}^{CLIP}(\boldsymbol{\theta}_\pi) - c_1\mathcal{L}^{v}(\boldsymbol{\theta}_v) + c_2 H(\pi(S_t, \boldsymbol{\theta}_\pi))\right] \tag{2.53}$$

where $H(\pi_{\theta_\pi})$ is the entropy of policy π. Adding this entropy bonus was inspired by Mnih, Badia, et al., 2016 and improves exploration. If the entropy is low, the policy is close to deterministic; to force the agent to keep exploring the entropy needs to be kept high. The coefficients c_1, c_2 are hyperparameters that need to be tuned separately.

The full PPO algorithm used for all experiments conducted during this thesis is given as Algorithm 2. In this setting, M, N and T are hyperparameters that need to be chosen separately. Each actor, typically one CPU core per actor with N cores in total, samples experience from the environment for T steps under the current policy parametrization. Then the T-step return $G_{t:T}$ is used to calculate the advantage $\mathbb{A}$ necessary for calculating the gradient of $\boldsymbol{\mathcal{L}}^{CLIP}(\boldsymbol{\theta}_\pi)$. The termination condition may be reaching a certain number of iterates, time steps or a pre-defined mean return.

3 Related Work

The idea of trial-and-error learning in the brain dates back to Thorndike, 1898 and a whole body of research covering different aspects of learning in animals has been created. The reward prediction error hypothesis of dopamine neuron activity (Schultz, Dayan, and Montague, 1997) has many supporters among neuroscientists and may be closely related to computational reinforcement learning. However, the role and interplay of neurotransmitters and neuromodulators in the brain facilitating reinforcement learning is not clear. Thus, we do not consider broad or isolated mechanisms describing general changes in synaptic efficacy, even if there is strong experimental evidence. Unsupervised Hebbian learning rules such as standard spike-time dependent plasticity are without doubt of significant importance, but they lack treatment of reward signals. Therefore the focus will be on works that have clear links to (computational) reinforcement learning that may not necessarily be biologically plausible. Most of the works presented below are able to solve simple reinforcement learning problems with small populations of spiking neurons and training may take long. We will show, that populations of a few hundred neurons can be trained successfully to solve more demanding benchmark tasks found in the reinforcement learning literature. We are not aware of other research groups solving reinforcement learning problems with spiking neural networks and backpropagation-through-time at this level of performance.

Seung, 2003 proposes the existence of "hedonistic" stochastic synapses that are modulated by a global reward signal that is the immediate effect of choosing a particular action. He defines synapses as hedonistic if they adapt the probability of reliably releasing neurotransmitters upon depolarization based on the action chosen previously. Similarly to the necessity of randomness in the process of evolution, the unreliability of synapses is postulated essential to learning in the brain. Integrate-and-fire neurons with hedonistic synapses estimate gradient learning in a biologically plausible manner. This gradient information is used in the REINFORCE algorithm to solve problems such as the XOR task, where the XOR of two binary numbers encoded as Poisson spike-trains is to be computed. Note that the probability of release of a neurotransmitter is adapted, not the synaptic connectivity itself.

After the development of the hedonistic synapse model, Xie and Seung, 2004 propose a learning rule estimating the gradient of the expected reward with respect to the synaptic weights. It is formulated under the assumption that all neurons in the population follow a Poisson process where the firing rates, not the individual spikes, are a function of the input current. This neural model greatly simplifies neural dynamics and is not physiologically plausible. They show experimentally, that this learning rule is also applicable to integrate-and-fire neurons if they add white noise to the input current all neurons receive. To justify this, they argue that near-Poisson distributed spike patterns can arise naturally due to the background noise in the brain.

A more general version of the learning rule developed by Xie and Seung, 2004 was postulated by Florian, 2007. He bases his derivation of the learning rule on the spike response model (Gerstner, Ritz, and Hemmen, 1993) and does not require Poisson-distributed firing rates. It is an extension of standard STDP and is called reward-modulated STDP that is biologically plausible and has been confirmed experimentally in vivo later (Yagishita et al., 2014; Yang and Dani, 2014; Cassenaer and Laurent, 2012). During experiments he solves the XOR task and show that specific patterns encoded via firing-rates can be learned. He also gives learning rules that include eligibility traces allowing to solve reward-delayed problems and they produce solutions with lower firing rates. Using the same reinforcement learning approach (Baxter and Bartlett, 2001) in a similar fashion, Baras and Meir, 2007 also find update rules for synaptic weights and relate their findings to the BCM plasticity rule (Bienenstock, Cooper, and Munro, 1982). Izhikevich and Desai, 2003 have shown that the BCM rule and the STDP rule are closely related and in certain cases even identical. Yet another work very similar to the methods in this paragraph was published in the same year by Farries and Fairhall, 2007.

An entirely different set of approaches has been reviewed by Floreano, Dürr, and Mattiussi, 2008. They claim that neural evolution efficiently solves both the problem of network architecture search and learning rule design. However, most of the work relates to conventional ANNs and not SNNs. Still, the fact that evolved architectures solve tasks such as cart-pole and robot movement is highly interesting. In our work, we focus on the same set of problems. The work of Di Paolo, 2003 shows that evolution and STDP can coexist to train a simulated 2D two-wheel robot to drive towards a light source with only 6 neurons and two light sensors.

Urbanczik and Senn, 2009 show that including an additional population response signal into the previously discussed reward-modulated plasticity mechanisms improves performance when increasing neuron population size. Previous works perform well for small neural assemblies but decrease in performance when the number of neurons in the network increases. This is due to the credit assignment problem, where the global reward signal is not informative enough to provide feedback to single neurons on how to change. In contrast, the backpropagation algorithm discussed with regard to artificial neural networks in Section 2.2.1 assigns a customized feedback signal to every unit in the network. With this in mind, the population response signal increases the performance when the number of neurons is increased; without it, the performance quickly degrades with increasing number of neurons. They argue that this method can be seen as a form of gradient descent.

Implementing a full actor-critic system with spiking neural networks was first presented by Potjans, Morrison, and Diesmann, 2009. They address several important issues: The reward prediction error hypothesis of dopamine neuron activity (Schultz, Dayan, and Montague, 1997) has been related to a method known in reinforcement learning literature as temporal difference learning. The neuromodulator dopamine can be seen as a temporal difference learning error $\delta_t = R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(S_t)$ that relates the successive value estimates through time. They find a biologically plausible synaptic weight update rule based on this temporal difference reward prediction error and propose a neural architecture consisting of an actor, a critic and a population of neurons representing different states. With this system they succeeded in solving the well-known grid-world environment, where the actor has to find positions of high reward and remember how to get to these positions from random starting positions. Another work by Frémaux, Sprekeler, and Gerstner, 2013 follows a similar approach and also produces synaptic update rules for continuous time actor-critic temporal difference learning. However, they extend this idea to continuous state and action spaces and their performances on the Morris water-maze navigation task matches the performance of rats. Their approach to handling continuous state and action spaces by place cells only works well for small spaces.

This issue has been addressed by Legenstein, Wilbert, and Wiskott, 2010 by proposing hierarchical slow feature analysis as a biologically plausible method of dimensionality reduction. The aim of slow feature analysis is to find a set of features that vary slowly over time, have high information content and are uncorrelated. Features such as position and identity of objects vary slowly over time, thus this method seems well suited for real-world scenes. Arguably hierarchical information processing is evident in many areas of the brain; experimental studies backing this claim are given to justify the use of hierarchical slow feature analysis. The experiments have been conducted using linear neurons that resemble artificial neural networks but model dendritic effects. With this, they solve tasks such as the Morris water-maze using high-dimensional visual input; the scene is rendered as a top-down view on the maze.

Karamanis, Zambrano, and Bohté, 2018 propose a novel multi-layer architecture for implementing a SARSA-like temporal difference reinforcement learning algorithm with working memory in spiking neural networks in continuous time. It consists of sensory, association, Q-value and action-readout layers connected in a feedforward manner with error feedback connections. This hierarchical design might potentially allow dimensionality reduction, although this was not explored. They use a general spike response model for their neuron populations and the synaptic weight update rule is biologically plausible. Notably, the mean firing rate of trained networks for tasks where memory is necessary is usually below 20 Hz.

The proximal policy optimization algorithm used for the experiments conducted in this thesis has also been applied by Tieck et al., 2018 to solve robotic environments in a spiking configuration. The environment they describe uses a model of a human arm with six muscles where the task is to control the arm to reach a target position. They use a liquid state machine as an input layer to transform the observations into a high-dimensional space to make use of its inherent memory properties; this makes liquid state machines or LSTMs highly suitable for processing partially observed or sequential data. This transformed input is passed on to ANN layers that act as readout of the liquid state machine. Regular ANNs are used as parametrized policy and value functions and are trained using gradient descent. The remainder of the configuration is identical to the parameters suggested by Schulman, Wolski, et al., 2017.

Of course the work of Bellec et al., 2018 is closely related to this thesis. They proposed the long short-term memory spiking neural network (LSNN) model used for all experiments conducted in this thesis involving spiking neural networks. Furthermore they investigated the capabilities of LSNNs in a meta reinforcement learning setting, where the task is learning to learn solving a family of similar tasks. Their tasks describe a 2D world where an agent is randomly placed in an arena and has to locate a fixed target area. This area is relocated after some episodes according to an unknown distribution within the arena. Over the course of many episodes the agent should remember the fixed position of the target area and subsequently move towards the goal efficiently.

4 Experiments

In this chapter we present the experiments conducted during this thesis and the corresponding experiment setup including a description of the environments the agent was trained in. We will further discuss the results obtained using spiking neural networks and compare them with similar artificial neural network architectures employing feedforward networks. All experiments have been conducted using the Ray reinforcement learning library (Liang et al., 2017) allowing distributed computing and efficient cluster usage.

4.1 Environments

In this section we will give a brief overview of the environments used. This includes reviews of the observation and action space, the reward structure and the task the agent is intended to solve. Although the description of the observation and action space are of no relevance for learning good policies, it allows the reader to better understand how the actor behaves in the environment.

4.1.1 Cart-Pole

The first environment is the well-known cart-pole task (Sutton and Barto, 2018) which features both a small observation space and a small discrete action space. It was mainly chosen for its simplicity allowing fast prototyping and testing of

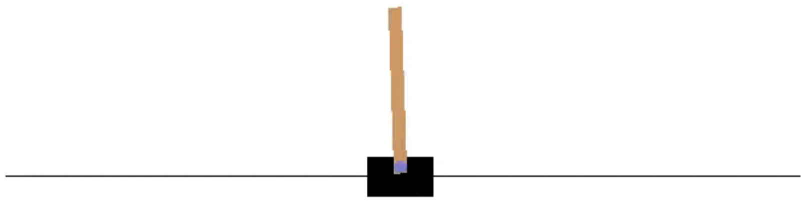


Fig. 4.1: The well-known cart-pole environment. The brown pole has to be balanced on the black rectangle by either pushing the cart to the left or to the right.

different network architectures and hyperparameters. As seen in Figure 4.1 a pole is attached with a joint to a cart moving on a track. The task is to learn to balance the pole such that it does not fall over. There are several conditions which terminate the current episode:

1. The pole angle exceeds $\pm 12^\circ$ measured from the upright position.
2. The cart position exceeds ± 2.4 measured from the starting position. Otherwise, the pole on the cart could stay balanced by indefinitely pushing the cart in one direction.
3. After 200 environment steps, the episode is considered solved and terminates.

The observations given to the agent are four floating-point numbers corresponding to the cart position, the cart velocity, the pole angle and the pole velocity at the tip of the pole. With this information, the agent needs to decide whether to choose one of the two possible options for each environment step: To push the cart to the left or to the right with a fixed amount of force not known to the agent. For each environment step, the agent receives a reward of $+1$, thus the maximal reward equals the maximal episode length of 200. Overall, the environment is considered solved, when the agent exceeds a mean reward of 195 averaged over 100 consecutive episodes.

4.1.2 Roboschool Hopper

The second environment under consideration is the more recent 3D Roboschool Hopper environment part of the OpenAI Gym suite (Plappert et al., 2018). This poses a more challenging task that is also used by state-of-the-art algorithms as benchmark. This is one of the main reasons this environment has been chosen; whilst providing a reasonably small observation state size it also features a continuous action space. After finding suitable architectures and hyperparameters for the cart-pole task these configurations are transferred to the new Hopper task. This allows to greatly reduce the necessary computation time for grid search to find good choices of hyperparameters.

In this environment the task is to learn efficient movement patterns used to propel a unipedal robot forward as far as possible within a given amount of time without falling over as seen in Figure 4.2. In theory, the Roboschool environment permits specifying the direction the robot should travel to; this is used for environments such as “Humanoid Flagrun” where a humanoid robot has to move towards a randomly chosen target. In the Hopper environment, the target is fixed and the robot cannot move freely in the environment; it just moves along an axis.

The robot consists of four limbs joined with motors in hinge joints. Each of these motors can be controlled by the agent issuing torque commands as actions for each environment time step. Informally speaking, if the robot does not manage to use its motors to keep an upright position, the episode terminates. More specifically, if the centroid z-coordinate (the height of the body off the ground) of the robot falls below a threshold of 0.8 or the absolute pitch of the body (rotation along the transverse axis) exceeds 1.0 the robot is considered to be fallen over and the episode terminates.

The actor receives 15 floating-point values that directly correspond to the state of the robot in the environment and could be measured by the robot itself. An overview

Index	Name	Description
1	centroid z-coord.	z-coordinate of the robot body centroid
2–3	angle	Sine and cosine of the angle between robot and target
4–6	velocity	Global robot velocity
7	roll	Roll of robot body
8	pitch	Pitch of robot body
9–10	thigh joint	Relative angle position and speed of thigh joint
11–12	leg joint	Relative angle position and speed of leg joint
13–14	foot joint	Relative angle position and speed of foot joint
15	floor contact	1.0 if the foot has contact with the floor, else 0.0

Tab. 4.1: Description of hopper observation space. Names and description have been devised by the author of this thesis based on the source code of the Roboschool suite.

including a short description of each observation value can be seen in Table 4.1. The individual observations are clipped to the range $[-5, 5]$ by the environment.

As mentioned earlier, the robot has three motors it can use to change its pose. By carefully activating the motors in certain sequences, the robot can learn a hopping motion suitable for locomotion. There may exist other movement patterns that propel the robot forward, however in this very constrained setting hopping forward is likely the best option.

The reward structure for the Roboschool environments is rather complex. The scalar reward given after each time step is the sum of the values described in Table 4.2. The choice of the compound reward has severe implications. A valid policy allows the robot to stand completely still and receive a cumulative reward of 1000 without learning anything else. This is an example of potentially bad reward signal design that rewards exploiting sub-goals that do not directly contribute to solving the designated task.

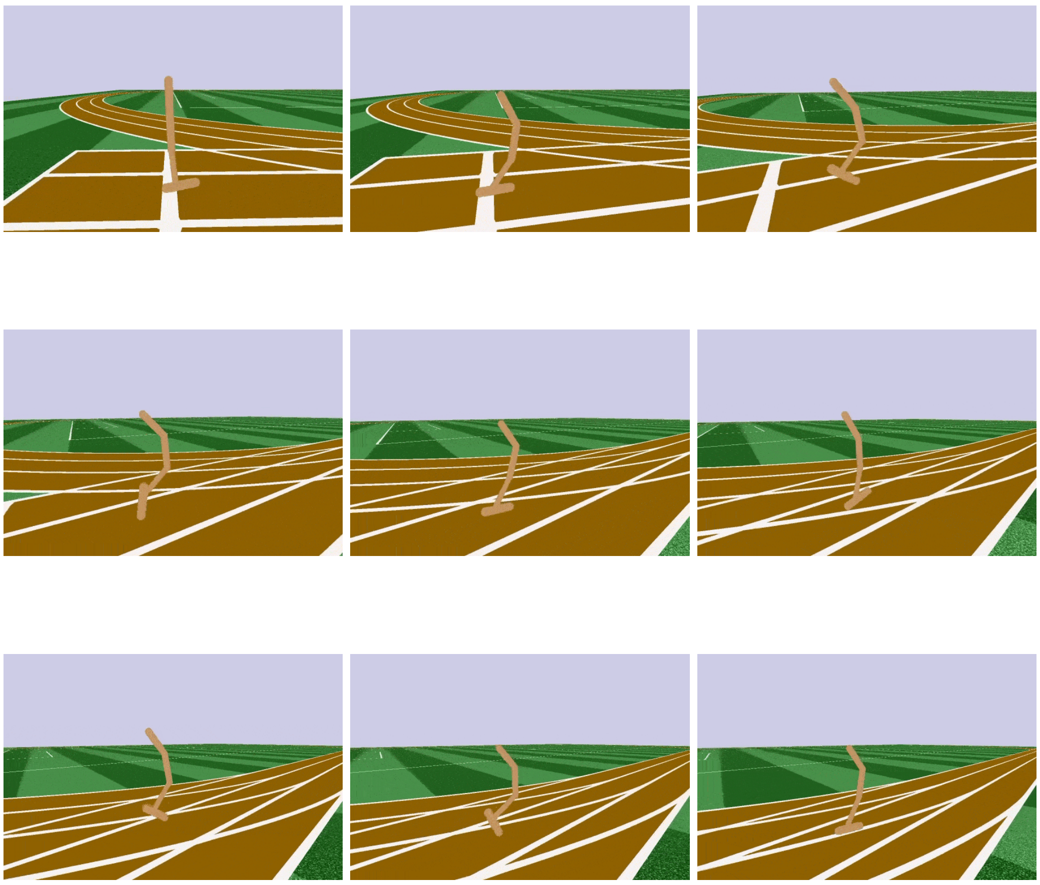


Fig. 4.2: Sequence of images from a fully trained "Hopper"-robot propelling itself forward as far as possible within a given amount of time.[a]

[a]Images have been extracted from video clips found on `https://blog.openai.com/roboschool/`.

Name	Description
alive bonus	+1 if not fallen over, else −1
progress	speed of the robot towards the target
electricity cost	cost of using the motors depending on joint speed and cost of running current through motor even if there is zero rotational speed
joints at limit cost	if joints are close to their maximum range of motion, give negative reward to discourage stuck joints

Tab. 4.2: Description of hopper reward structure. Names and description have been devised by the author of this thesis based on the source code of the Roboschool suite.

4.2 Preprocessing

This section describes the approach taken to transform the raw observation input values into a more suitable form. The problem with general floating-point input is that each of the observations has a different range of possible values, i.e. the floor contact observation has values between 0 and 1 whereas the z-coordinate of the global robot velocity has values between −5 and 5.

The range of possible values for each observation is not known beforehand and is not documented. Therefore it was necessary to experimentally extract these ranges by using a non-spiking model such as an artificial feedforward network and record all observations to determine the bounds. It is not sufficient to use a pre-trained model to find the bounds for each observation; during training it is possible that exploratory actions yield observations not produced by the fully trained model.

After the bounds of the observation space have been determined, it is possible to encode floating point values by passing them through a structure similar to a receptive field. In this setup, the raw observation is presented to ω Gaussian functions

$G = [g_1(x_i^{(t)}), \dots, g_\omega(x_i^{(t)})]^T$ spaced equally within the previously determined bounds. This process is visualized in Figure 4.3 for $\omega = 4$. For our experiments ω is usually in the range $[20, 40]$. Depending on the value of the input $x_i^{(t)}$ at time step t, each Gaussian function produces values between 0 and 1. Thus for any given input, with a sufficient number of Gaussian functions within the specified bounds most remain silent while only a few are active; this provides a sparse input representation.

Formally, the shape of the Gaussian function is given by

$$g_j(x_i^{(t)}) = a \exp\left(-\frac{(x_i^{(t)} - b_j)^2}{2c^2}\right) \tag{4.1}$$

with parameters

$$a = 1 \tag{4.2}$$

$$b_j = o_l + (j-1)\frac{o_h - o_l}{\omega - 1} \tag{4.3}$$

$$c = \frac{b_{j+1} - b_j}{2\sqrt{-2\log\left(\frac{1}{2}\right)}} \tag{4.4}$$

where o_l, o_h describe the experimentally determined bounds for a single specific observation value $x_i^{(t)}$ part of the vector $\boldsymbol{x}^{(t)}$. This observation vector $\boldsymbol{x}^{(t)}$ represents the information the agent receives at time t. The parameters a, b_j, c describe the height, center and width of the curve, respectively. For our purposes, a has been chosen such that $g_j : \mathbb{R} \longrightarrow [0, 1]$, the positions b_j of the Gaussian functions are equally distributed in the range $[o_l, o_h]$ and the width c has been chosen such that at the point of overlap between two Gaussian functions the function value is $\frac{a}{2} = \frac{1}{2}$. In the following section we will describe how these transformed values are used in the spiking neural network.

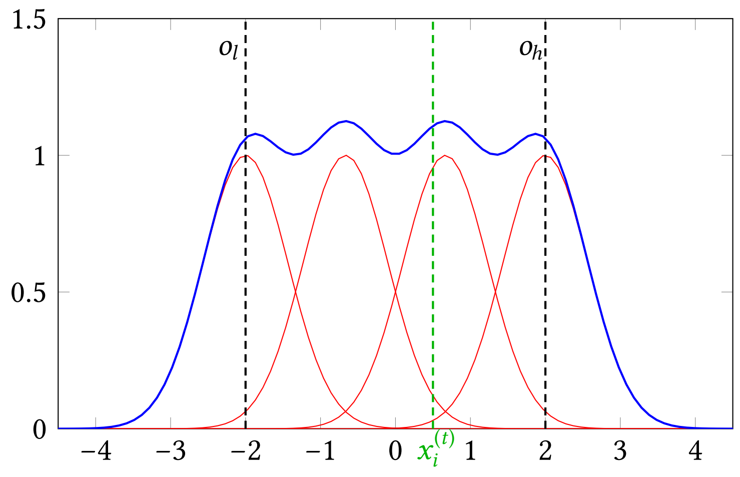


Fig. 4.3: Non-linear transformation of SNN inputs. A scalar input value $x_i^{(t)}$, represented by the green line, gets processed by a vector-valued function $G = [g_1(x_i^{(t)}), \dots, g_\omega(x_i^{(t)})]^T$, where g_j are Gaussian functions. Between defined bounds $[o_l, o_h]$ the cumulative activation of all Gaussian functions g_j is roughly the same.

4.3 Network Architecture

In this section we describe the general spiking network architecture used as a model in the reinforcement learning setup alongside the simulation details including simulation time and general hyperparameters used.

The network architecture mimics the usual setting of an artificial feedforward network and can be seen in Figure 4.4. The observations $x_1^{(t)}, \dots, x_n^{(t)}$ from the environment at time step t are preprocessed by the method described in Section 4.2. The preprocessed values are then fed into one or more layers of LIF neurons that depending on the experiment have recurrent connections within the same layer.

In Section 2.1.2 we described the membrane voltage dynamics of a leaky integrate-

and-fire neuron j as

$$u_j(t + \Delta t) = (1 - \alpha_a)(u_{\text{rest}} + RI_j(t)) + \alpha_a u_j(t) \tag{4.5}$$

where $I(t)$ is the input current. In our simulations, we define the input current as the weighted sum of observation input and recurrent spike-based input

$$I_j(t) = \sum_i^n \sum_k^\omega w^{\text{in}}_{i,k,j}\, g_k(x_i^{(t)}) + \sum_{m \in M_j} w^{\text{rec}}_{j,m}\, z_m^{(t)} \tag{4.6}$$

where M_j is the index set of neurons m recurrently connected to neuron j. This is the formulation used for the first spiking layer; if multiple spiking layers are used the observation input is replaced by the previous layers' spiking output. The last layer of spiking neurons is connected to an artificial linear readout layer that performs a linear transformation of the exponentially filtered spike train with time constant τ_f.

The final output $y_1^{(t)}, \dots y_k^{(t)}$ corresponds to the actions the actor should take for the next environment step. This depends on the action space of the environment; we follow the common action selection methods presented in equations (2.49) and (2.50) for discrete and continuous action spaces, respectively.

To better simulate the dynamics of a biological neural network considering the recurrent connections within the LIF layers, it is necessary to simulate the LIF neurons for multiple internal discrete time steps within a single environment step. Additionally the output of the non-linear transformations $G(x_i) = [g_1(x_i^{(t)}), \dots, g_\omega(x_i^{(t)})]^T$ is only shown at the first internal time step to approximate the sparse firing behaviour in the brain. At the last time step of the internal simulation the values of the readout layer are used to determine the action the agent should take in the next environment step. A visualization of this can be seen in Figure 4.5.

As discussed in Section 2.4 policy gradient methods provide a very general framework suitable for optimizing general parameterized policies. Artificial neural net-

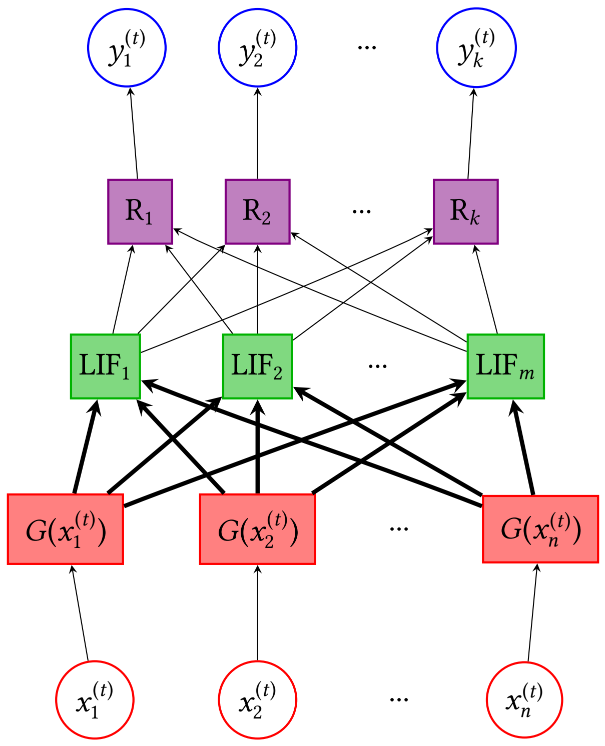


Fig. 4.4: Network architecture for all experiments. The input $x_i^{(t)}$ at time step t gets transformed by a non-linear transformation $G(x_i^{(t)})$ as seen in Figure 4.3. This produces a vector-valued output fed into the population of LIF neurons arranged in one or more layers. The output of the last layer of LIF neurons is then processed by a linear readout layer. Bold arrows indicate vector-valued output whereas thin arrows represent scalar-valued output.

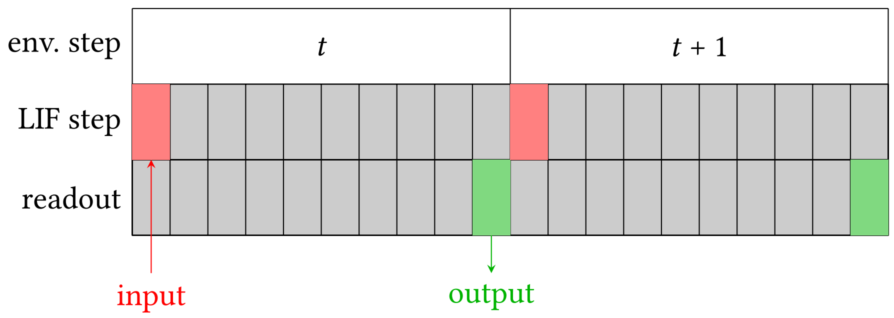


Fig. 4.5: Illustration of the computational process. The input to the LIF neuron is only available on the first time step whereas the output of the readout layer should only be accessed during the last internal time step.

works are the default choice in many settings (Schulman, Wolski, et al., 2017) and usually outperform linear policies. In this setting, it is only natural to extend policy gradient methods such as proximal policy optimization to spiking policies. With the network architecture described in this section spiking neural networks provide a drop-in replacement for artificial neural networks. For all experiments conducted in this thesis the proximal policy optimization algorithm will be used.

What remains to examine is the calculation of the loss function. Again, the suggested objective for PPO consists of

$$\mathcal{L}^{CLIP+v+H+R}(\boldsymbol{\theta}_\pi) = \mathbb{E}_{\boldsymbol{\theta}_{\text{old}}}\left[\mathcal{L}^{CLIP}(\boldsymbol{\theta}_\pi) - c_1\mathcal{L}^{v}(\boldsymbol{\theta}_v) + c_2 H(\pi_{\boldsymbol{\theta}_\pi}) + c_3\sum_j\left(z_{j,t} - f^0\right)^2\right] \tag{4.7}$$

where the additional term $c_3\sum_j\left(z_{j,t} - f^0\right)^2$ is a firing rate regularization term where $z_{j,t}$ is 1 if neuron j fired at time t. The clipped conservative policy iteration loss $\mathcal{L}^{CLIP}(\boldsymbol{\theta}_\pi)$, the value function loss $\mathcal{L}^{v}(\boldsymbol{\theta}_v)$ and the entropy bonus $H(\pi_{\boldsymbol{\theta}_\pi})$ remain unchanged. By modifying the PPO objective we can incorporate additional sub-objectives relevant for altering the behaviour of the spiking neural network such as regularizing the mean firing rate. Since the mean firing rate of the neurons should remain small, we can introduce a regularization objective that penalizes neurons for deviating from a chosen target firing rate $f^0 = 10$ Hz. This not only enforces low firing rates but also encourages neurons that would otherwise remain silent to fire more. The regularization is computed across many time steps to allow bursts of activity and short periods of silence of each neuron.

The hyperparameters used during the experiments are described individually for each experiment in the following sections. They were partially adapted from Schulman, Wolski, et al., 2017 and partially obtained via grid search or random search. Much emphasis has to be put on the fact that results for single trials of experiments are not reliable; based on different random number generator seeds used to initialize weights in both the spiking and non-spiking networks the results vary greatly. Therefore the experimental results had to be averaged over many trials to get an estimate of the true performance. To keep the computational cost relatively low, the each experiment was averaged over five trials. This makes tuning hyperparameters via grid search or random search very time consuming, even if computations are performed on a cluster.

4.4 Experimental Results

In this section we present the experimental results obtained using the previously described settings with proximal policy optimization. Depending on the environment the actor operates in, the results are discussed separately.

As a rule of thumb, we aim for similarly sized spiking network configurations compared to artificial feedforward network models found in the literature (Schulman, Wolski, et al., 2017; Espeholt et al., 2018; Hessel et al., 2018; Horgan et al., 2018). Thus, networks with one to two layers of $\{64, 128, 256\}$ neurons each are considered. Additionally, most of the hyperparameters used in the literature can be directly adapted for the spiking network configuration; this allows direct comparison of the performance and saves on computational time for hyperparameter search.

4.4.1 Cart-Pole

The cart-pole environment provides an excellent playground for experimenting with different network architectures and hyperparameter settings in a time-efficient manner. In Table 4.3 an overview of the hyperparameters used including the ranges of possible values is given.

Figures 4.6 and 4.7 show the typical stages of training a spiking neural network for the cart-pole task; the performance over episodes can be seen in Figure 4.8. A similar learning curve can be produced for a 1-layer feedforward ANN. Although firing rate regularization is enabled in this experiment, some neurons remain silent as can be seen in Figure 4.7 where a tradeoff between convergence and overall firing rate has to be made. Without firing rate regularization the neurons will fire almost immediately after leaving their refractory period.

To make the task more interesting, a new cart-pole task has been devised. To test the memory capabilities of LSNNs and LSTMs, the velocity information of the cart and the pole tip have been removed from the observations. Thus, only the absolute cart position and the pole angle are known to the agent. The results of this experiment can be seen in Figure 4.9. In this experiment, the one-layer spiking neural network with recurrent connections outperformed the memory augmented two-layer ANN with an LSTM attached; not only is the final SNN performance better, it even converges fast to good solutions.

Overall, we can see that the LSNN performs at least as well as combinations of ANN and LSTM for the cart-pole problem. Surprisingly, the spiking neural network usually seems to learn the task about as fast as the corresponding ANN configuration; however it seems that the convergence is not as smooth and the standard deviation around the mean performance is usually larger for the SNN.

	Cart-Pole Hyperparameters		
	Name	**Range**	**Chosen Value**
General	Training batch size ≥ NT	[1000, 4000]	2000
	Sample batch size T	[100, 200]	100
	SGD batch size M	[100, 400]	200
	SGD epochs K	[10, 20]	20
	Learning rate	[0.1, 0.0001]	0.001
PPO	Number of Actors N	[1, 10]	6
	GAE parameter λ	[0.95, 1]	0.98
	Discount γ	[0.9, 1]	0.99
	Clipping ε	[0.1, 0.5]	0.2
	KL coefficient β	[0.1, 0.5]	0.3
	$\hat{v}$ loss coefficient c_1	[0, 5]	1.0
	entropy bonus coefficient c_2	[0, 1]	0.0
	firing rate coefficient c_3	[0, 100]	10
SNN	NLT number of kernels ω	[10, 40]	40
	Show input for · steps	[1, 10]	1
	Threshold ϑ	[0.03, 0.05]	0.03
	Number of internal SNN steps	[1, 10]	8
	RNN maximum sequence length	[10, 20]	10
	Exponential filter τ_f	[5, 30]	8

Tab. 4.3: Hyperparameters used for the cart-pole experiments. The range indicates values considered during grid search whereas the chosen value represents the value found in the best-performing configuration.

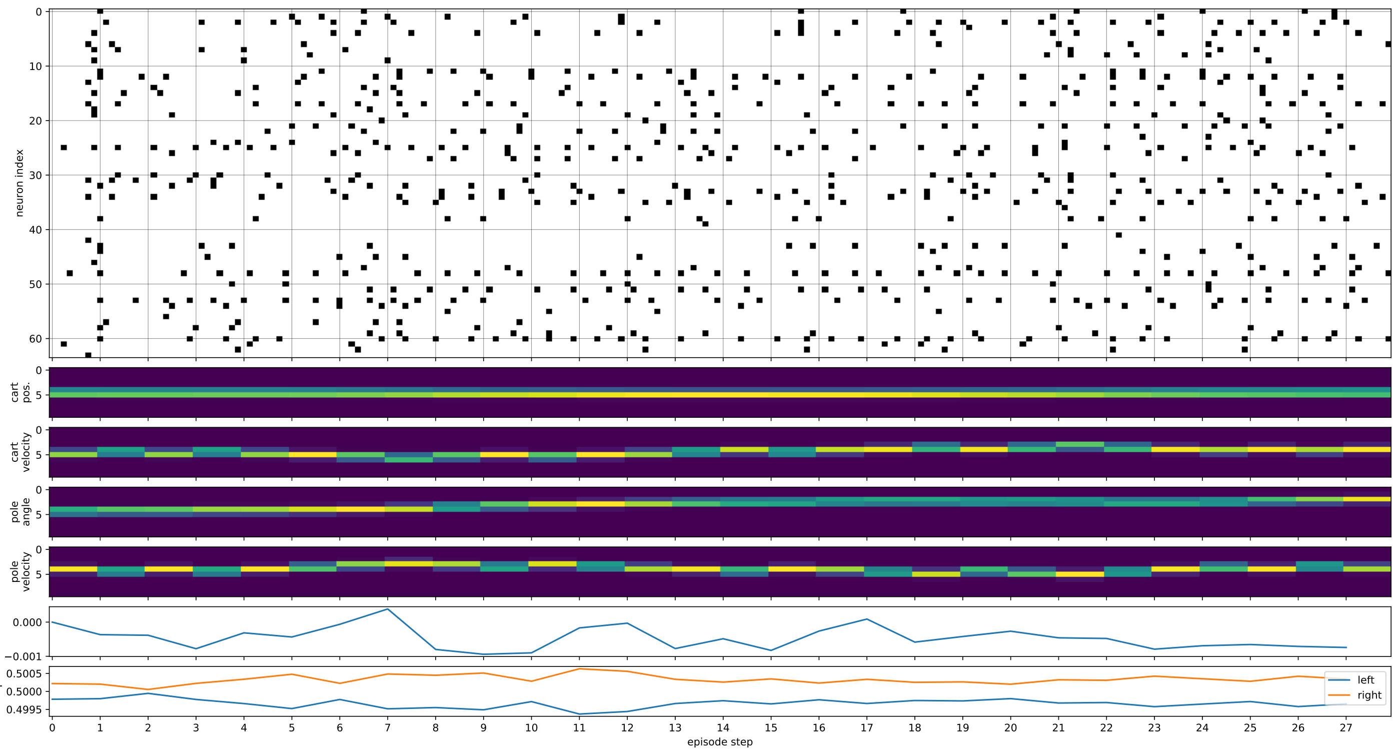


Fig. 4.6: Spiking behaviour of **untrained** SNN at cart-pole task. Below the spike raster plot a representation of the observations encoded by the Gaussian functions can be seen. Below the four observations the current value of the state is given, whereas the action probability refers to the suggested action recommended by the value-function SNN. In this setting, a low number of Gaussian functions (ω = 10) has been chosen.

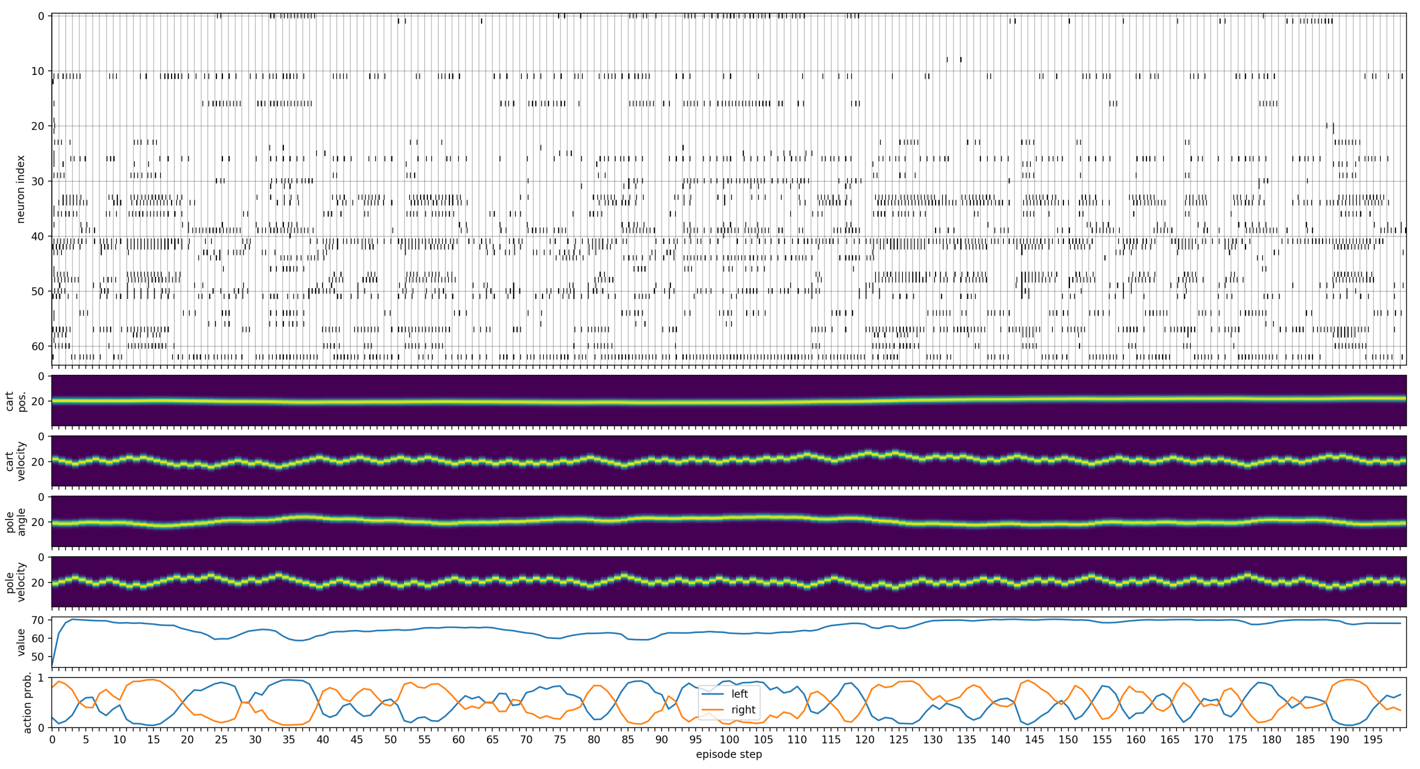
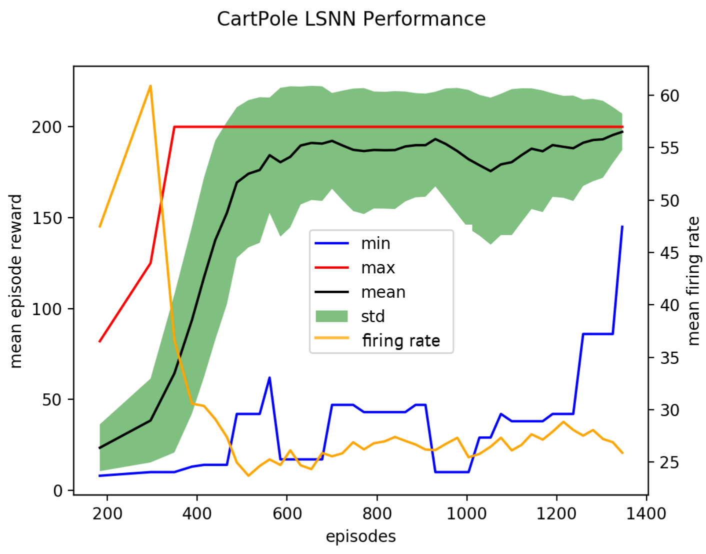


Fig. 4.7: Spiking behaviour of **trained** SNN at cart-pole task. Below the spike raster plot a representation of the observations encoded by the Gaussian functions can be seen. Below the four observations the current value of the state is given, whereas the action probability refers to the suggested action recommended by the value-function SNN. In this setting, a higher number of Gaussian functions (ω = 40) has been chosen.



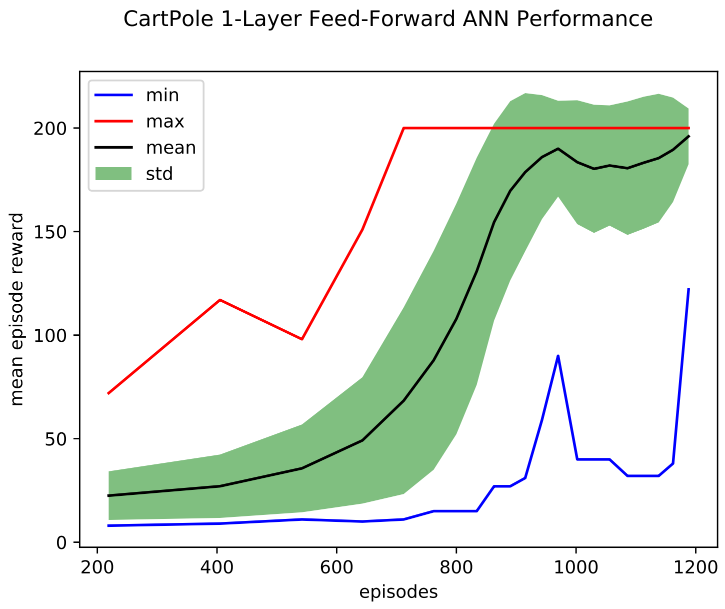


Fig. 4.8: Performance of SNN (top) and ANN (bottom) at cart-pole task over episodes. The black, blue and red lines correspond to the mean, minimum and maximum performance over the episodes during a training step. The green area around the mean reward shows the standard deviation. The orange line over the SNN performance chart shows the mean firing rate of the neurons with the dual axis on the right side of the figure.

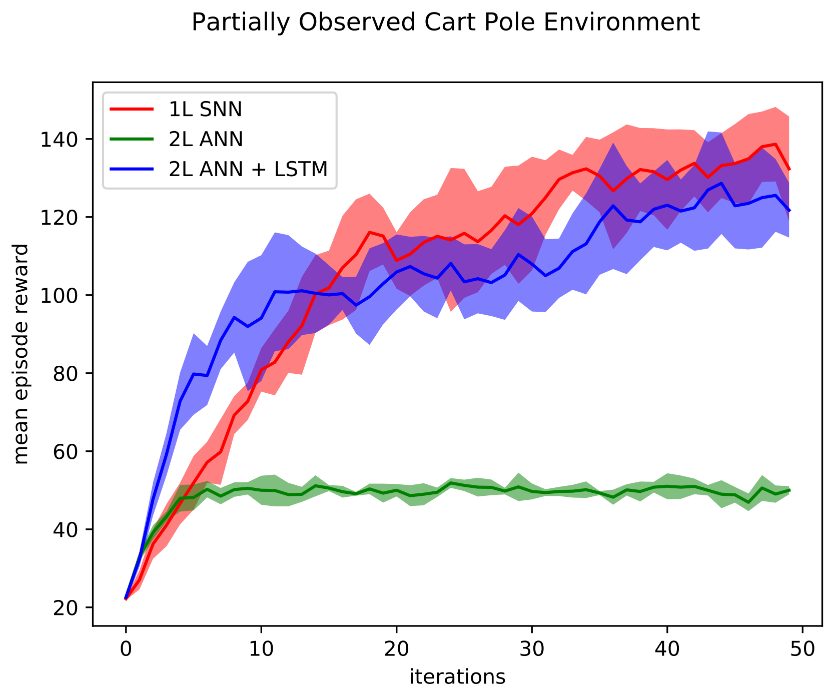


Fig. 4.9: Performance comparison of different network architectures on the new partially observed cart-pole environment. The ANN configurations have two feedforward layers of 64 neurons each, where one architecture additionally has a 64 unit LSTM layer attached. The SNN architecture has one layer of 128 recurrently connected neurons. The hyperparameters correspond to the chosen parameters in Table 4.3; each result is the average of five experiment runs. One iteration corresponds to 2000 environment steps.

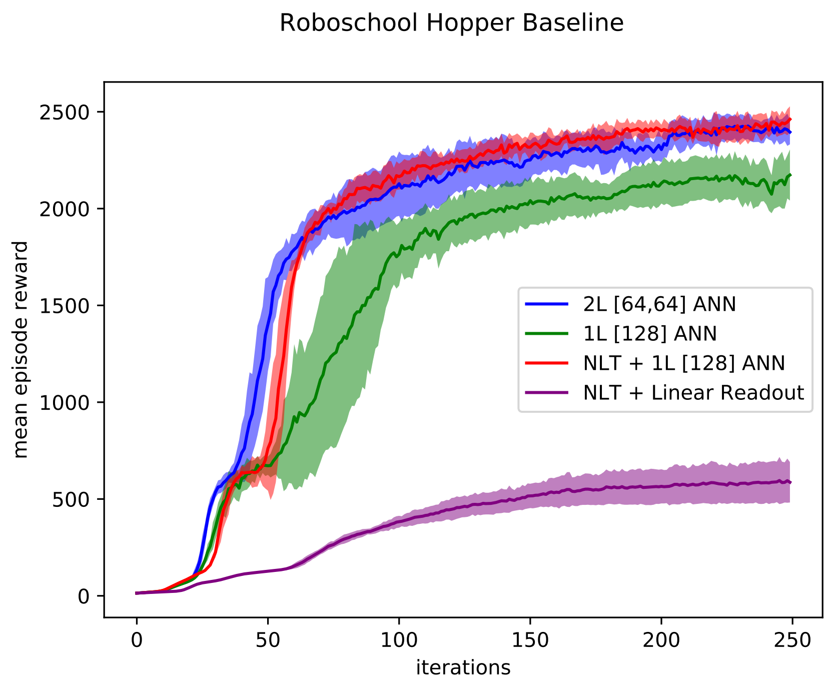


Fig. 4.10: Roboschool Hopper experiments with feedforward ANNs exploring impact of additional non-linear input transformations with 20 Gaussian functions. A two-layer ANN with 64 neurons in each layer performs as well as a one-layer ANN with non-linear input transformation. The numbers in brackets in the legend indicates the number of neurons per layer.

4.4.2 Roboschool Hopper

Since the cart-pole task can be considered as an introductory example of reinforcement learning problems that can be solved readily, the more demanding Roboschool environment suite allows to better compare the performance potential of spiking neural networks and artificial neural networks. Again, an overview of the hyperparameters including ranges used during grid and random search and the final chosen value are given in Table 4.4. These chosen values are used for all experiments. Only the parameters under investigation are modified, all the other parameters stay the same for all experiments to produce comparable results. The following experiments are conducted five times each and the mean performance including the standard deviation is given in the figures. As mentioned, this is due to the high variance of possible results in the Hopper environment discussed in Section 4.1.2. The training batch size has been deliberately chosen to be large enough to produce good results while keeping the overall computation time acceptable. Depending on the chosen hyperparameters, one experiment run typically takes 12 to 36 hours to complete.

To provide a fair comparison to state-of-the-art results of PPO in combination with regular feedforward ANNs the impact of using the same non-linear Gaussian function input transformation as in the spiking network setup needs to be discussed. The higher number of weights seems to be beneficial to the network and greatly reduces the variance of results observed over multiple trials as seen in Figure 4.10. Our two-layer ANN with 64 units per layer achieves better results than the same architecture used by Schulman, Wolski, et al., 2017; this is due to the larger number of environment steps sampled per training iteration.

The next experiment inspects the impact of recurrent connections within one layer of spiking neurons. In the literature feedforward ANNs are typically used for policies in reinforcement learning problems. In the Roboschool suite the observations from the environment include temporal information about the state of the robot in the scene, as seen in Table 4.1. This information should be sufficient to choose good next actions from an analytical perspective. Recurrent connections would make sense if the problem requires memory such as in partially observed environments and environments that require very non-linear actions where the recurrent spiking neural network projects the input in a high-dimensional space like a liquid state machine (Maass, Natschläger, and Markram, 2002). In Figure 4.11 a direct comparison of recurrent and feedforward spiking neural networks is shown. The number of weights in the feedforward setting is much lower, therefore it is to be expected that the network converges faster. However, the recurrently connected network achieves a performance close to that of the feedforward network in the

	Roboschool Hopper Hyperparameters		
	Name	**Range**	**Chosen Value**
General	Training batch size $\geq NT$	[5000, 320000]	160000
	Sample batch size T	[200, 1000]	1000
	SGD batch size M	[64, 32768]	16384
	SGD epochs K	[10, 30]	15
	Learning rate	[0.01, 0.0001]	0.001
PPO	Number of Actors N	[8, 16]	10
	GAE parameter λ	[0.9, 0.99]	0.95
	Discount γ	[0.8, 0.999]	0.995
	Clipping ε	[0.1, 0.3]	0.2
	KL coefficient β	[0.1, 5]	1.0
	$\hat{v}$ loss coefficient c_1	[0, 5]	1.0
	entropy bonus coefficient c_2	[0, 1]	0.0
	firing rate coefficient c_3	[0, 100]	10
SNN	NLT number of kernels ω	[10, 40]	20
	Show input for $\cdot$ steps	[1, 10]	1
	Threshold ϑ	[0.03, 0.05]	0.03
	Number of internal SNN steps	[1, 10]	8
	RNN maximum sequence length	[10, 20]	10
	Exponential filter τ_f	[10, 30]	10

Tab. 4.4: Hyperparameters used for the Roboschool Hopper experiments. The range indicates values considered during grid search whereas the chosen value represents the value found in the best-performing configuration.

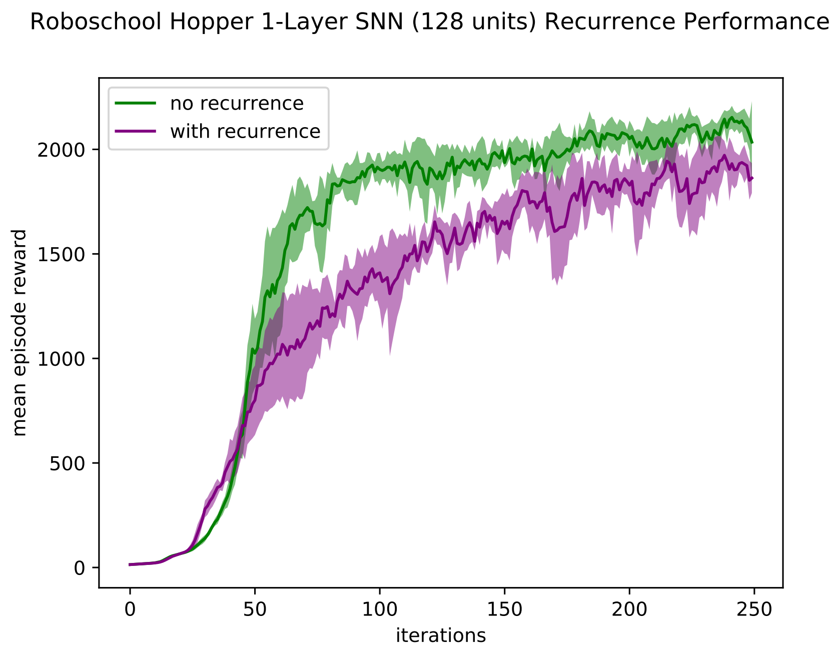


Fig. 4.11: Roboschool Hopper experiments with one-layer SNN with 128 neurons exploring the impact of recurrent connections for the performance. As the observations from the environment contain sufficient information to decide upon actions on a step-by-step basis, recurrent connections may be a hindrance concerning convergence of the network due to the larger number of weights.

end. The standard deviation of the mean performance is rather low in both settings, implying that the results are robust to different initializations of the weights.

Another experiment investigates the influence of the training batch size on the overall performance. As with most problems in supervised learning, a standard way of improving performance is to increase the dataset size where available. In our reinforcement learning scenario with simulated robotics, it is easy to generate more training data simply by sampling from more often from the environment. However, increasing the training set size not only significantly increases computation cost to update the policy but there are only diminishing incremental gains possible.

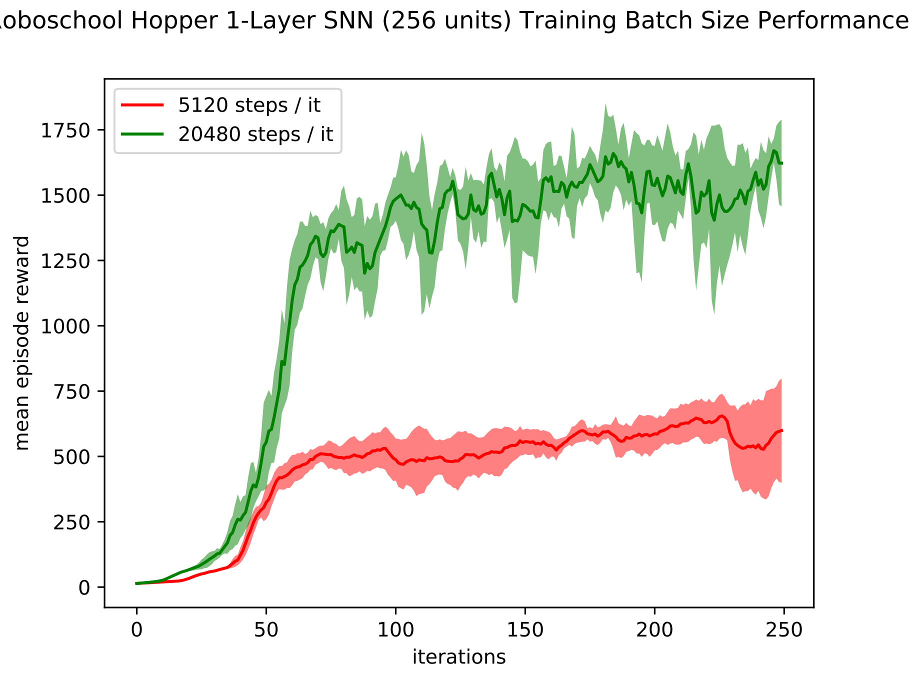


Fig. 4.12: Roboschool Hopper experiments with 1-Layer SNN with 256 neurons exploring the impact of the training batch size for the performance. The parameters were adapted directly from Schulman, Wolski, et al., 2017 and scaled according to the number of workers in use. It is clear that the linearly scaling the training batch size by the number of workers only serves as a rule of thumb; however, the impact of different training batch sizes is enormous.

Figure 4.12 shows the results of training the spiking network with different training batch sizes inspired by the hyperparameters chosen by Schulman, Wolski, et al., 2017. The training batch size is of utmost importance for achieving good performance over many trials. As seen in Figure 4.12 even over five independent trials there was no run where a the training batch size of 5120 environment steps produced mean episode rewards above 1000.

5 Conclusion and Outlook

In this thesis we have investigated the applicability of spiking neural networks to current reinforcement learning methods. We have devised several experiments exploring different aspects of spiking neural networks both in feed-forward and recurrent configurations.

The first group of experiments was performed in the classic cart-pole environment. There we have shown that both ANN and SNN can solve the task within 1500 training episodes; surprisingly the SNN converges comparably fast to good solutions. We have introduced a partially observed version of the cart-pole environment, where the velocity of cart and pole are excluded. Here, a standard ANN performs poorly compared to the SNN; once a LSTM network is attached to the ANN, its performance approaches that of the SNN.

In the second experiment group, we investigated the performance of ANNs and SNNs in the more demanding Roboschool Hopper environment with a four times larger observation space and a continuous action space. There we found that the preprocessing step used for the SNN is beneficial to the ANN and improves its performance due to an increased number of weights to be learned. We further confirmed our hypothesis that the observations given by the Hopper environment contain sufficient temporal information to be solved without recurrent connections. The larger number of weights in the recurrent configuration of the SNN reduces convergence speed and the performance.

We have shown that the recent actor-critic method proximal policy optimization is well suited to employ spiking neural networks for its parametrized policy and value function. In the Roboschool environment, the agent achieves good performance compared to state-of-the-art artificial neural network architectures. In relation to environments described in other works in Chapter 3 the Roboschool robotic suite is more difficult to solve. No previous works using spiking neural networks solved problems of this difficulty to the best of our knowledge.

Exploring the hyperparameter space with a fine-grained grid for the Roboschool environment is computationally infeasible and would take months even on a cluster. This is mainly due to the fact that results are noisy and need to be averaged over multiple trials to estimate the true performance of a specific hyperparameter setting. Thus random search in combination with hand-tuning was the only viable option to find reasonable parameters comparably fast. The number of environment samples used for one training iteration is one of the most important hyperparameters besides the learning rate of gradient descent. Without a large number of samples from the environment, proximal policy optimization with spiking neural networks performs poorly.

We want to point out that the spiking network architectures used in this work are simple; more elaborate hierarchical networks similar to convolutional neural networks could be investigated. Slow-feature analysis may be a biologically plausible model for hierarchical abstraction of high-dimensional features. This might allow SNN-based agents to solve environments with visual input that are typically more difficult to solve and could bridge the gap to current state-of-the-art methods. Another interesting method of finding good architectures might involve evolutionary methods related to the work of Floreano, Dürr, and Mattiussi, 2008 or newer evolutionary approaches (Salimans et al., 2017). Another method that may not be biologically plausible but produced outstanding results is an augmented random search algorithm (Mania, Guy, and Recht, 2018). These approaches might be useful for finding better hyperparameters, new network architectures or distributions of weights to initialize networks with, even if they might not be suitable to optimize the network weights themselves.

Appendix

Bibliography

Baras, Dorit and Ron Meir (2007). “Reinforcement learning, spike-time-dependent plasticity, and the BCM rule.” In: *Neural Computation* 19.8, pp. 2245–2279 (cit. on p. 39).

Baxter, Jonathan and Peter L Bartlett (2001). “Infinite-horizon policy-gradient estimation.” In: *Journal of Artificial Intelligence Research* 15, pp. 319–350 (cit. on p. 39).

Bear, Mark, Barry Connors, and Michael Paradiso (2016). *Neuroscience: Exploring the Brain.* Wolters Kluwer (cit. on pp. 6, 7).

Beck, Amir (2014). *Introduction to Nonlinear Optimization: Theory, Algorithms, and Applications with MATLAB.* MOS-SIAM (cit. on pp. 15, 33).

Bellec, Guillaume, Darjan Salaj, Anand Subramoney, Robert Legenstein, and Wolfgang Maass (2018). “Long short-term memory and Learning-to-learn in networks of spiking neurons.” In: *CoRR* abs/1803.09574. arXiv: `1803.09574` (cit. on pp. 27, 42).

Bienenstock, Elie, Leon Cooper, and Paul Munro (1982). “Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex.” In: *Journal of Neuroscience* 2.1, pp. 32–48 (cit. on p. 40).

Bishop, Christopher (2006). *Pattern Recognition and Machine Learning.* Springer (cit. on pp. 15, 18, 28).

Cassenaer, Stijn and Gilles Laurent (2012). “Conditional modulation of spike-timing-dependent plasticity for olfactory learning.” In: *Nature* 482.7383, pp. 47–52 (cit. on p. 39).

Clopath, Claudia, Lars Büsing, Eleni Vasilaki, and Wulfram Gerstner (2010). "Connectivity reflects coding: a model of voltage-based STDP with homeostasis." In: *Nature Neuroscience* 13.3, pp. 344–352 (cit. on p. 26).

Courbariaux, Matthieu and Yoshua Bengio (2016). "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1." In: *CoRR* abs/1602.02830. arXiv: `1602.02830` (cit. on p. 27).

Davies, Mike, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang (2018). "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning." In: *IEEE Micro* 38.1, pp. 82–99 (cit. on p. 5).

Di Paolo, Ezequiel (2003). "Evolving spike-timing-dependent plasticity for single-trial learning in robots." In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 361.1811, pp. 2299–2319 (cit. on p. 40).

Espeholt, Lasse, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu (2018). "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures." In: *CoRR* abs/1802.01561. arXiv: `1802.01561` (cit. on pp. 32, 54).

Esser, Steven, Paul Merolla, John Arthur, Andrew Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David Berg, Jeffrey McKinstry, Timothy Melano, Davis Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra Modha (2016). "Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing." In: *Proceedings of the National Academy of Sciences* 113.41, pp. 11441–11446 (cit. on p. 27).

Farries, Michael and Adrienne Fairhall (2007). "Reinforcement learning with modulated spike timing-dependent synaptic plasticity." In: *Journal of Neurophysiology* 98.6, pp. 3648–3665 (cit. on p. 40).

Floreano, Dario, Peter Dürr, and Claudio Mattiussi (2008). "Neuroevolution: from architectures to learning." In: *Evolutionary Intelligence* 1.1, pp. 47–62 (cit. on pp. 40, 67).

Florian, Răzvan (2007). "Reinforcement Learning Through Modulation of Spike-Timing-Dependent Synaptic Plasticity." In: *Neural Computation* 19.6, pp. 1468–1502 (cit. on p. 39).

Frémaux, Nicolas, Henning Sprekeler, and Wulfram Gerstner (2013). "Reinforcement learning using a continuous time actor-critic framework with spiking neurons." In: *PLoS Computational Biology* 9.4, pp. 1–21 (cit. on p. 41).

Furber, Steve, Francesco Galluppi, Steve Temple, and Luis Plana (2014). "The SpiNNaker Project." In: *Proceedings of the IEEE* 102.5, pp. 652–665 (cit. on p. 5).

Gerstner, Wulfram, Werner Kistler, Richard Naud, and Liam Paninski (2014). *Neuronal Dynamics.* Cambridge University Press (cit. on pp. 5–8, 10, 12).

Gerstner, Wulfram, Raphael Ritz, and Leo van Hemmen (1993). "Why spikes? Hebbian learning and retrieval of time-resolved excitation patterns." In: *Biological Cybernetics* 69.5–6, pp. 503–515 (cit. on p. 39).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning.* MIT Press (cit. on pp. 20–22, 28).

Hahnloser, Richard, Rahul Sarpeshkar, Misha A Mahowald, Rodney Douglas, and Sebastian Seung (2000). "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit." In: *Nature* 405, pp. 947–951 (cit. on p. 11).

Hessel, Matteo, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver (2018). "Rainbow: Combining improvements in deep reinforcement learning." In: *Thirty-Second AAAI Conference on Artificial Intelligence* (cit. on pp. 32, 54).

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory." In: *Neural Computation* 9.8, pp. 1735–1780 (cit. on p. 24).

Horgan, Dan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver (2018). "Distributed Prioritized Experience Replay." In: *CoRR* abs/1803.00933. arXiv: `1803.00933` (cit. on p. 54).

Izhikevich, Eugene and Niraj Desai (2003). "Relating STDP to BCM." In: *Neural Computation* 15.7, pp. 1511–1523 (cit. on p. 40).

Kakade, Sham and John Langford (2002). "Approximately optimal approximate reinforcement learning." In: *International Conference on Machine Learning*. Vol. 2, pp. 267–274 (cit. on p. 35).

Karamanis, Marios, Davide Zambrano, and Sander Bohté (2018). "Continuous-Time Spike-Based Reinforcement Learning for Working Memory Tasks." In: *International Conference on Artificial Neural Networks*. Springer, pp. 250–262 (cit. on p. 41).

Lapicque, Louis (1907). "Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization." In: *Journal de Physiologie et de Pathologie Generalej* 9, pp. 620–635 (cit. on p. 9).

Legenstein, Robert, Niko Wilbert, and Laurenz Wiskott (2010). "Reinforcement Learning on Slow Features of High-Dimensional Input Streams." In: *PLoS Computational Biology* 6.8, pp. 1–13 (cit. on p. 41).

Liang, Eric, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica (2017). "Ray RLLib: A Composable and Scalable Reinforcement Learning Library." In: *CoRR* abs/1712.09381. arXiv: 1712.09381 (cit. on p. 43).

Maass, Wolfgang, Thomas Natschläger, and Henry Markram (2002). "Real-time computing without stable states: A new framework for neural computation based on perturbations." In: *Neural Computation* 14.11, pp. 2531–2560 (cit. on p. 62).

Mania, Horia, Aurelia Guy, and Benjamin Recht (2018). "Simple random search provides a competitive approach to reinforcement learning." In: *CoRR* abs/1803.07055. arXiv: 1803.07055 (cit. on p. 67).

Markram, Henry, Joachim Lübke, Michael Frotscher, and Bert Sakmann (1997). "Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs." In: *Science* 275.5297, pp. 213–215 (cit. on p. 26).

Merolla, Paul, John Arthur, Rodrigo Alvarez-Icaza, Andrew Cassidy, Jun Sawada, Filipp Akopyan, Bryan Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron Flickner, William Risk, Rajit Manohar, and Dharmendra Modha (2014). "A million spiking-neuron integrated circuit with a scalable communication network and interface." In: *Science* 345.6197, pp. 668–673 (cit. on p. 5).

Mnih, Volodymyr, Adria Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu (2016). "Asynchronous Methods for Deep Reinforcement Learning." In: *International Conference on Machine Learning*, pp. 1928–1937 (cit. on p. 36).

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. (2015). "Human-level control through deep reinforcement learning." In: *Nature* 518, pp. 529–533 (cit. on p. 32).

Plappert, Matthias, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba (2018). "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research." In: *CoRR* abs/1802.09464. arXiv: `1802.09464` (cit. on p. 45).

Potjans, Wiebke, Abigail Morrison, and Markus Diesmann (2009). "A spiking neural network model of an actor-critic learning agent." In: *Neural Computation* 21.2, pp. 301–339 (cit. on p. 40).

Russell, Stuart and Peter Norvig (2009). *Artificial Intelligence: A Modern Approach.* Prentice Hall Press (cit. on p. 1).

Salimans, Tim, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever (2017). "Evolution Strategies as a Scalable Alternative to Reinforcement Learning." In: *CoRR* abs/1703.03864. arXiv: `1703.03864` (cit. on p. 67).

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz (2015). "Trust Region Policy Optimization." In: *International Conference on Machine Learning*. Vol. 37, pp. 1889–1897 (cit. on pp. 32, 35).

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). "Proximal Policy Optimization Algorithms." In: *CoRR* abs/1707.06347. arXiv: `1707.06347` (cit. on pp. 4, 28, 32, 35, 36, 42, 53, 54, 62, 65).

Schultz, Wolfram, Peter Dayan, and Read Montague (1997). "A neural substrate of prediction and reward." In: *Science* 275.5306, pp. 1593–1599 (cit. on pp. 38, 40).

Seung, Sebastian (2003). "Learning in Spiking Neural Networks by Reinforcement of Stochastic Synaptic Transmission." In: *Neuron* 40.6, pp. 1063–1073 (cit. on p. 38).

Silver, David, Aja Huang, Chris Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587, pp. 484–489 (cit. on p. 1).

Sutton, Richard and Andrew Barto (2018). *Reinforcement Learning: An Introduction.* MIT Press (cit. on pp. 28, 30, 31, 33, 34, 43).

Sutton, Richard, David McAllester, Satinder Singh, and Yishay Mansour (2000). "Policy Gradient Methods for Reinforcement Learning with Function Approximation." In: *Advances in Neural Information Processing Systems*, pp. 1057–1063 (cit. on p. 33).

Thorndike, Edward (1898). "Animal intelligence: An experimental study of the associative processes in animals." In: *The Psychological Review: Monograph Supplements* 2.4, pp. i–109 (cit. on pp. 1, 38).

Tieck, Juan, Marin Pogančić, Jacques Kaiser, Arne Roennau, Marc-Oliver Gewaltig, and Rüdiger Dillmann (2018). "Learning Continuous Muscle Control for a Multi-joint Arm by Extending Proximal Policy Optimization with a Liquid State Machine." In: *International Conference on Artificial Neural Networks*, pp. 211–221 (cit. on p. 42).

Urbanczik, Robert and Walter Senn (2009). "Reinforcement learning in populations of spiking neurons." In: *Nature Neuroscience* 12.3, p. 250 (cit. on p. 40).

Williams, Ronald (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning." In: *Machine Learning* 8.3, pp. 229–256 (cit. on p. 33).

Xie, Xiaohui and Sebastian Seung (2004). “Learning in neural networks by reinforcement of irregular spiking.” In: *Physical Review E* 69.4, p. 041909 (cit. on p. 39).

Yagishita, Sho, Akiko Hayashi-Takagi, Graham Ellis-Davies, Hidetoshi Urakubo, Shin Ishii, and Haruo Kasai (2014). “A critical time window for dopamine actions on the structural plasticity of dendritic spines.” In: *Science* 345.6204, pp. 1616–1620 (cit. on p. 39).

Yang, Kechun and John Dani (2014). “Dopamine D1 and D5 receptors modulate spike timing-dependent plasticity at medial perforant path to dentate granule cell synapses.” In: *Journal of Neuroscience* 34.48, pp. 15888–15897 (cit. on p. 39).