Gradient-Free Optimization of Artificial and Biological Networks using Learning to Learn

Alper Yeğenoğlu

IAS Series
Band / Volume 55
ISBN 978-3-95806-719-6

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Schriften des Forschungszentrums Jülich
IAS Series Band / Volume 55

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Gradient-Free Optimization of Artificial and Biological Networks using Learning to Learn

Alper Yeğenoğlu

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 55

ISSN 1868-8489

ISBN 978-3-95806-719-6

Bibliografische Information der Deutschen Nationalbibliothek.
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten
sind im Internet über http://dnb.d-nb.de abrufbar.

Herausgeber und Vertrieb:	Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag 52425 Jülich Tel.: +49 2461 61-5368 Fax: +49 2461 61-6103 zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Umschlaggestaltung:	Grafische Medien, Forschungszentrum Jülich GmbH
Titelbild:	Vecteezy.com
Druck:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2023

Schriften des Forschungszentrums Jülich
IAS Series, Band / Volume 55

D 82 (Diss. RWTH Aachen University, 2023)

ISSN 1868-8489
ISBN 978-3-95806-719-6

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess.

Contents

Abstract

Understanding intelligence and how it allows humans to learn, to make decision and form memories, is a long-lasting quest in neuroscience. Our brain is formed by networks of neurons and other cells, however, it is not clear how those networks are trained to learn to solve specific tasks. In machine learning and artificial intelligence it is common to train and optimize neural networks with gradient descent and backpropagation. How to transfer this optimization strategy to biological, spiking networks (SNNs) is still a matter of research. Due to the binary communication scheme between neurons of an SNN via spikes, a direct application of gradient descent and backpropagation is not possible without further approximations.
In my work, I present gradient-free optimization techniques that are directly applicable to artificial and biological neural networks. I utilize metaheuristics, such as genetic algorithms and the ensemble Kalman Filter, to optimize network parameters and train networks to learn to solve specific tasks. The optimization is embedded into the concept of meta-learning and learning to learn respectively. The learning to learn concept consists of a two loop optimization procedure. In the first, inner loop the algorithm or network is trained on a family of tasks, and in the second, outer loop the hyper-parameters and parameters of the network are optimized.
First, I apply the EnKF on a convolution neural network, resulting in high accuracy when classifying digits. Then, I employ the same optimization procedure on a spiking reservoir network within the L2L framework. The L2L framework, an implementation of the learning to learn concept, allows me to easily deploy and execute multiple instances of the network in parallel on high performance computing systems. In order to understand how the network learning evolves, I analyze the connection weights over multiple generations and investigate a covariance matrix of the EnKF in the principle component space. The analysis not only shows the convergence behaviour of the optimization process, but also how sampling techniques influence the optimization procedure. Next, I embed the EnKF into the L2L inner loop and adapt the hyper-parameters of the optimizer using a genetic algorithm (GA). In contrast to the manual parameter setting, the GA suggests an alternative configuration. Finally, I present an ant colony simulation foraging for food while being steered by SNNs. While training the network, self-coordination and self-organization in the colony emerges. I employ various analysis methods to better understand the ants' behaviour.

With my work I leverage optimization for different scientific domains utilizing meta-learning and illustrate how gradient-free optimization can be applied on biological and artificial networks.

Zusammenfassung

Die Intelligenz zu verstehen und wie sie den Menschen ermöglicht zu lernen, Entscheidungen zu treffen und Erinnerungen zu bilden, ist ein langfristiges Bestreben in den Neurowissenschaften. Unser Gehirn besteht aus Netzwerken von Neuronen und anderen Zellen, jedoch ist es nicht klar wie diese Netzwerke trainiert werden können, um bestimmte Aufgaben zu lösen. Im Bereich des maschinellen lernen und künstlicher intelligenz ist es üblich neuronale Netzwerke mittels Gradientenabstieg und Backpropagation zu trainieren und zu optimieren. Wie diese Optimierungsstrategie auf biologische, gepulste neuronale Netzwerke übertragen werden kann, ist wissenschaftlich noch offen. Wegen der binären Kommunikation zwischen den Neuronen des gepulsten Netzwerke mittels Spikes, ist eine direkte Anwendung des Gradientenabstieg und Backpropagation ohne Approximationen nicht möglich.
In meiner Arbeit präsentiere ich gradientfreie Optimierungstechniken, welche auf künstliche und biologische Netzwerke direkt anwendbar sind. Ich benutze Metaheuristiken, wie zum Beispiel Genetische Algorithmen oder Ensemble Kalman Filter (EnKF), um die Netzwerkparameter zu optimieren und die Netzwerke zu trainieren bestimmte Aufgaben zu lösen. Die Optimierung wird in das Konzept des Meta-Lernens bzw. learning to learn eingebettet. Das learning to learn Konzept besteht aus einem zwei Schleifen Optimierungsprozess. In der ersten, inneren Schleife wird ein Algorithmus oder ein Netzwerk auf einer Aufgabe aus einer Familie von Aufgaben trainiert. In der zweiten, äußeren Schleife werden die Parameter und Hyper-Parameter des Netzwerkes optimiert.
Als Erstes optimiere ich mit dem EnKF ein Convolutional Neural Network und erreiche eine hohe Erfolgsrate bei der Klassifizierung von Zahlen. Danach, wende ich die gleiche Optimierungsstrategie innerhalb der L2L Bibliothek auf ein gepulstes Reservoir Netzwerk an. L2L, eine Implementierung des learning to learn Konzepts, ermöglicht es mir auf eine einfache Weise mehrere Netzwerkinstanzen parallel auf Hochleistungsrechnern auszuführen. Um zu verstehen wie sich das Lernen des Netzwerkes entwickelt, analysiere ich die Netzwerkgewichte über mehrere Generationen und untersuche eine Kovarianzmatrix des EnKF im Hauptkomponentenraum. Die Analyse zeigt nicht nur das Konvergenzverhalten der Optimierung, sondern auch wie zum Beispiel Sampling Techniken die Optimierung beeinflussen. Danach bette ich den EnKF in die innere Schleife des L2L ein und passe die Hyper-Parameter des Optimierers mit einem genetischem Algorithmus (GA) an. Im Kontrast zur manuellen

Parametereinstellung findet der GA eine alternative Konfiguration. Zum Schluss präsentiere ich eine simulierte Ameisenkolonie, die nach Futter sucht und vom einem gepulstem Netzwerk gesteuert wird. Während des Netzwerktrainings entwickelt sich eine Selbstkoordination und Selbstorganization innerhalb der Kolonie. Ich verwende verschiedene Analysemethoden, um das Ameisenverhalten besser zu verstehen.
Mit meiner Arbeit zeige ich wie Optimierung und Meta-Lernen auf einfache Weise in verschiedenen Wissenschaftsbereichen benutzt werden können und wie gradientfreie Optimierung auf biologische und künstliche Netzwerke angewendet werden kann.

Acknowledgments

First of all I would like to thank my supervisors Prof. Dr. Abigail Morrison and Prof. Dr. Michael Herty. They were always helpful and guided me when I asked them and otherwise gave me enough space and time to try out new things. Our discussions helped me to improve my work whenever I was stuck.

Another big thanks goes to Dr. Sandra Diaz-Pier, Dr. Kai Krajesk and Dr.Giuseppe Visconti. I had fruitful discussions with Kai regarding deep neural networks and performance optimization. Giuseppe, one of the best math lecturers I know, helped me a lot in understanding the ensemble Kalman filter. I always enjoyed our long discussions which often lead to a productive outcome. The idea for the first paper presented in this work originated when I was explaining deep networks to Giuseppe, while he was explaining me Kalman filters. You were a great supervisor. Sandra supervised me a lot as well. She was there when I needed help with my work and gave me a lot of pointers. Of course, we had great discussions and coding sessions, too. She was open to every topic I came up with and diligently answered my questions. I am happy to have you as my supervisors and hope to work with you in future as well.

I also want to thank to Dr. Nicole Voges and Dr. Piero Giovanni Luca Porta-Mana. Both of you constantly listened to my ramblings and also gave me scientific advice. I am glad to have you as friends who are always there for me.

I express my appreciation to all the collaborators and my colleagues of the SDL neuroscience and IGPM. Also a thank you to all my friends around the globe who supported me.

Finally, I want to thank my family, who always supported me and gave me strength and motivation to do my best.

Eidesstattliche Erklärung

Ich, Alper Yegenoglu, erkläre hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt,

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;

2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;

3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;

4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;

5. Diese Dissertation is vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;

6. Alle wesentlichen Quellen von Unterstützung wurden benannt;

7. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;

8. Ein Teil oder Teile dieser Arbeit wurden zuvor veröffentlicht und ersichtlich im Abschnitt *Publications and contributions*

Publications and contributions

Journal publications and contributions

- **Ensemble Kalman Filter Optimizing Deep Neural Networks: An Alternative Approach to Non-performing Gradient Descent** Alper Yegenoglu, Kai Krajsek, Sandra Diaz-Pier and Michael Herty, International Conference on Machine Learning, Optimization, and Data Science. LOD 2020. Springer, Cham 2020.

  https://doi.org/10.1007/978-3-030-64580-9_7

  Chapter 4 *Deep Neural Networks Optimized by the Ensemble Kalman Filter* is based on this publication.

  Contributions: Under the supervision of Prof. Herty, the author implemented the workflow and carried out the experiments. All authors contributed to the design of the experimental workflow and jointly wrote the manuscript.

- **Exploring Parameter and Hyper-Parameter Spaces of Neuroscience Models on High Performance Computers With Learning to Learn** Alper Yegenoglu, Anand Subramoney, Thorsten Hater, Cristian Jimenez-Romero, Wouter Klijn, Aarón Pérez Martín, Michiel van der Vlag, Michael Herty, Abigail Morrison and Sandra Diaz-Pier, Front. Comput. Neurosci. 2022, 16:885207.

  https://doi.org/10.3389/fncom.2022.885207

  Chapter 5 *Learning to Learn* is based on this publication.

  Contributions: Under the supervision of Prof. Morrison and Dr. Diaz-Pier the author worked on and contributed to all use cases. He especially, designed, implemented and simulated the reservoir network. In collaboration with Dr. Cristian Jimenez-Romero, he worked on the ant colony use case. Furthermore, he contributed to the implementation of the network in the mountain car example using the spiking simulator NEST. All authors jointly wrote the manuscript.

- **Emergent Communication enhances Foraging Behaviour in Evolved Swarms controlled by Spiking Neural Networks** Cristian Jimenez-Romero, Alper Yegenoglu, Aarón Pérez Martín, Sandra Diaz-Pier and Abigail Morrison

  Cristian Jimenez Romero, Alper Yegenoglu, Aarón Pérez Martín, Sandra Diaz-Pier, Abigail Morrison, 2022, arXiv:2212.08484

  https://doi.org/10.48550/arXiv.2212.08484

  Chapter 6 *Optimizing Spiking Neural Networks enhances Foraging Behaviour of Swarm-Agents* is based on this publication

  Contributions: Under the supervision of Prof. Morrison, Dr. Cristian Jimenez Romero and the author contributed to this work equally. They designed and implemented the ant colony simulation. While Cristian Jimenez-Romero executed the simulations, the author analyzed the network activity, the colony behaviour and performed the benchmarks. All authors jointly wrote the manuscript.

1 Introduction and Motivation

1.1 Motivation

Understanding intelligence and how it emerges from a clump of soft matter is one of the greatest questions in neuroscience. This question occupies all kinds of scientists that wish to explore and exploit the mechanisms of intelligence. The quest to understand biological intelligence in living organisms can be compared to the motivation to create an artificial intelligence (AI). For the AI community it is interesting to explore how mechanisms observed in the human brain can be leveraged to create a general artificial intelligence, which exhibits humanlike intelligence (Yoshua Bengio, Lee, et al. 2015; Lake et al. 2017; Hassabis et al. 2017; Hasson et al. 2020).

In recent years, work intertwining neuroscience and artificial intelligence has seen substantial progress. Researchers are drawing inspiration from neuroscientific observations and ideas to enhance learning in AI with the hope to create intelligent machines. For example, Guerguiev et al. (2017) combine deep neural networks with multi-compartment neurons and enable the network to utilize neurons in different layers to coordinate synaptic weight updates. This approach allows the network to exhibit a better performance in classifying images than a single layered network. One of the authors' goals is to better understand the mechanisms of the neocortex and to comprehend how the neocortex may optimize cost functions. DiCarlo et al. (2012) and Yamins et al. (2016) relate deep learning to specific functions of the human visual system and draw conclusions on how learning in deep networks could be enhanced.

Neuroscience is able to profit from insights in AI as well. Khaligh-Razavi et al. (2014) test a variety of computational models to perform categorization and investigate whether the internal representations of those models can explain the representation of the inferior temporal (IT) cortex. They compare the representational dissimilarity matrices (RDMs) of the model representations with the RDMs obtained from electrophysiological recordings of human and monkey IT. Interestingly, their results show the necessity for supervised training when explaining the behaviorally induced categorical divisions of the IT. Richards et al. (2019) divide deep learning into three basic components: objective functions, learning rules and architec-

ture. They argue that focusing on these components will benefit systems neuroscience, which investigates how the brain implements perceptual, cognitive and motor tasks. The authors conclude the framework has to be rooted in **optimization**. For example, they mention that in order to solve complex tasks, local plasticity rules (e.g. Hebbian learning) must incorporate objective functions and the appropriate design of network architectures. They conclude that a top-down framework of systems neuroscience will profit when guided by machine learning insights.

As illustrated in the works above, there is an interest in the neuroscience and artificial intelligence community to understand the potential of combining biological networks with artificial ones to solve complex tasks. In neuroscience, spiking neural networks (SNNs) offer an understandable and biologically realistic way to explore certain dynamics observed in the brain. The potential of those networks is not fully understood and, as aforementioned, still subject to research within the neuroscience and artificial intelligence communities. Despite the computational success of artificial neural networks (ANNs), SNNs are more energy and memory efficient, since they utilize sparse and temporal coding via electrical signals (Yamazaki et al. 2022). This efficiency may be extremely helpful when leveraged for applications that require real time responses and a low power consumption such as drones or swarm robots.

The real challenge lies in optimizing SNNs for machine learning tasks, such as image classification or steering multiple collaborative agents to navigate in (changing) 2D or 3D environments. Mimicking the learning process of the brain is not yet feasible, since the application of learning rules or objective functions in the brain is not well understood (Marblestone et al. 2016; Richards et al. 2019). Optimizing SNNs is a complex task and, due to the sheer amount of parameters, requires manual adjustments which is very time consuming and error prone. Additionally, the non-linear relationship between the neuron's input and its output makes the optimization process non-intuitive (Russell et al. 2010). Gradient descent, a popular optimization technique in machine learning, is not applicable without further approximations and structural changes to the network (Whittington et al. 2019; Surace et al. 2020). Training deep neural networks requires significant amounts of computational resources (Chowdhery et al. 2022). Similarly, scaling up spiking networks to simulate a small percentage of the brain requires computational power as well (J. Jordan et al. 2018). This increases the importance of algorithms and optimization techniques that scale well on high performance computers. There is a need to find alternative ways to optimize SNNs in order to increase their performance.

In machine learning and data sciences black box optimizers are successfully applied on optimization problems in order to automate the search for optimal parameters. **Gradient-free** optimization techniques, such as evolutionary algorithms or Kalman filtering, are less common utilized when training artificial networks. However, they provide a powerful alternative

to optimize biological and artificial networks, since they do not require the explicit calculation of gradients and can be directly applied without changing any property of the network. Moreover, gradient-free methods avoid the problem of vanishing or exploding gradients, which can occur when optimizing deep artificial networks with gradient descent. This problem hinders the network to learn by disabling the parameter updates (vanishing gradients) or by extremely increasing them (exploding gradients), so that the network cannot convergence into a local optimum.

The applied optimization methods have to be **model agnostic**, i.e. the optimization process has to be generic and employable on many problems without the need of knowing the underlying model or structure. That means, the optimization target does not have to be the model parameters, it can be any parameter configuration as well as hyper-parameters. **Hyper-parameters** are parameters that influence and control the learning process of an algorithm or model. Often, hyper-parameter optimization is applied after the training in an offline, grid-search fashion. A range of parameters is tested to observe if the algorithm performance increases. However, this approach just iterates over all available parameter ranges and does not adapt very well to any parameter modification, which is necessary in an online training scenario. Furthermore, due to the simple iteration over the parameter range, grid-search's run time performance is inefficient. Thus, an alternative and intelligent approach is desirable, which adapts the hyper-parameters in an interchangeable manner with the model parameters. Meta-learning or learning to learn is a technique to update hyper-parameters while an algorithm or model is trained to solve a problem. The optimization workflow consists of a two loop structure. While in the first (also called the inner loop) the algorithm is trained and optimized on a task from a family of tasks, the second (the outer loop) adjusts the hyper-parameters. In the inner loop an objective or fitness function evaluates the performance of the algorithm. The fitness and parameters of the inner optimization process, and optionally the algorithm parameters, are sent to the outer loop optimizer, which returns optimized parameters for the next iteration. The outer loop usually consists of black box optimizers, which are agnostic to the inner loop algorithm.

In my thesis I focus on the optimization of biological and artificial networks utilizing the learning to learn concept to solve machine learning tasks. My work interconnects neuroscience and artificial intelligence by exploring how gradient-free optimizers can enhance the performance of networks and how they can be used to overcome problems such as vanishing gradients. The presented, gradient-free optimizers are metaheuristics and optimize their targets based on population decisions, leading to an overall increase in performance. Utilizing a framework based on the implementation of the learning to learn concept, I am able to efficiently scale the algorithms to high performance computers (HPCs), which let me run multiple inner loop models in parallel easing the compute load. Furthermore, I analyze the optimized parame-

ters and provide an interpretation of the learning process within convolutional and spiking networks. This helps to understand the relationship between ANNs and SNNs and how the acquired insight in one of those fields can be transferred to the other. Finally, I explore and analyze a setup that includes SNNs controlling multiple agents. While optimizing the network, the swarm evolves and starts to exhibit a collective behaviour and self-organization, which increases their performance to solve a task in a 2D environment.

1.2 Contribution of the Thesis

The main contribution of this thesis is to leverage optimization for artificial and biological networks. My work focuses on bridging computational neuroscience and artificial intelligence to better understand how to utilize optimization to solve complex tasks which are applicable within both fields. In the same spirit of Richards et al. (2019), I describe and embed the problem of solving (machine learning) tasks with biological and artificial networks within an optimization framework, namely the L2L framework, which is an implementation of the learning to learn concept. Following this principle, my work allows scientists to better understand and compare optimization problems of different domains. On the one hand, this cross-fertilization can help create an easier comprehension for optimization in biological networks, which is an emergent field. On the other hand this work can help enhance learning in artificial networks if biological processes are understood better and made transferable to artificial networks. By utilizing gradient-free methods I can apply the optimization methods directly on the network parameters without the need for any computational approximation or structural change of the network topology.

The optimization process needs to scale well to complex problems while still being efficient and robust. Here, I use metaheuristic optimizers which are model agnostic and applicable on different problems and tasks. This automates the search for optimal parameter settings of SNNs, which otherwise is a tedious task due to the sheer amount of variables to set. I incrementally increase the complexity of the problems from image classification with ANNs and SNNs, to hyper-parameter optimization on top of the classification and finally to optimizing a spiking neural network to steer a swarm of agents collectively foraging food. With this work, I also show the potential of gradient-free optimization on networks of different domains solving complex tasks.

1.3 Scope and Structure of this Thesis

In Chapter 2, I provide an introduction to artificial and biological networks. I discuss different types of networks and explain basic concepts in order to model and simulate them. A short introduction into biological processes of neurons will establish the basics to differentiate between ANNs and SNNs. The chapter ends with an outlook to bio-inspired learning, which intersects learning in neuroscience and AI and illustrates how inspirations drawn from neuroscience enhances learning in AI.

The next chapter (Chapter 3) dives into the topic of optimization. After a short primer into the theory of optimization, I discuss gradient descent and backpropagation as popular and established optimization methods when training ANNs. I continue explaining the problem of vanishing or exploding gradients when applying gradient descent as an optimization technique. Moreover, it is not clear how gradient descent could be implemented in biological networks. The discussion is not only from a biological but also a computational point of view. In contrast to ANNs, in SNNs a gradient cannot be calculated without any approximation or without changing the structure of the network. Instead, gradient-free methods, in particular metaheuristics, can provide a suitable solution, which I explicate in more detail.

In Chapter 4, I investigate the problem of vanishing gradients by analyzing activation values and gradients of a convolutional neural network when it is trained with gradient descent. I discuss the ensemble Kalman filter (EnKF) as an alternative optimizer for training deep neural networks and present the performance of the network classifying digits and letters. The EnKF provides a stable optimization solution even under ill-conditioned network settings. Additionally, I analyze the effects of different hyper-parameters of the EnKF and provide a simple algorithm to adapt the hyper-parameters. Finally, I discuss the convergence behaviour and benchmark the computational time of the EnKF.

Chapter 5 discusses the importance of hyper-parameter optimization and introduces the learning to learn concept and the corresponding L2L Python implementation. Here, I explain in detail how an experiment with the L2L framework is conducted. I present optimization results of digit classification using a spiking reservoir network optimized by the Kalman filter within the L2L framework. Afterwards, I explore the learning process of the reservoir by analyzing the connection weights and the covariance matrix of the EnKF. In order to automatically find the optimal parameter configuration for the optimizer I embed the EnKF optimization into the inner loop of L2L. I optimize the hyper-parameters of the EnKF using a genetic algorithm (GA). The hyper-parameter optimization finds a value similar to the manual setting for one parameter, but suggests a more efficient configuration for the other parameters.

In Chapter 6 I present the optimization of SNNs steering multiple agents, in particular an ant colony, foraging for food. While the SNN is evolving over the generations, self-organization and self-coordination emerges within the ant colony. The actions of the ants are not manually encoded and there are no pre-defined rules, i.e. the optimization procedure has to find an efficient network configuration to enhance the foraging behaviour. For example, the ants are able to deposit chemical signals, pheromones, to create a trail other ants can follow to a food source. This behaviour is not encoded into the network and the network has to learn how to utilize the pheromones. After describing the L2L optimization workflow I analyze the ant foraging behaviour. I compare the performance between a simple, rule-based system and the SNN based model. The SNN performs better than the simple model, but disabling the ants' pheromone sensing the performance deteriorates significantly. This highlights the importance of using pheromones for communication and organization purposes. The ants deposit the pheromones on their way to the food source and not necessarily when they visually perceive the food source or sense their nest. To have a better understanding of the ant behaviour, I correlate the input to output spike activity of the SNN over several or all generations and thus, relate the ants' actions to their visual and sensing perceptions.

The last chapter of this thesis, Chapter 7, provides conclusions and discusses the future developments and applications of gradient-free optimization and learning to learn applied on biological and artificial networks. I explore and suggest techniques which can be used in the future to overcome current limitations when optimizing networks.

2 Artificial and Biological Neural Networks

Although the structure and process units of artificial neural networks (ANNs) are inspired by the brain, there are significant differences between ANNs and biological spiking neural networks (SNNs). This chapter aims to introduce important types of ANNs and SNNs mainly utilized in my simulations. First, I will describe the structure of feed-forward networks such as multilayer perceptrons (MLP, Section 2.1.1) and Convolutional Neural Networks (CNN, Section 2.1.3). CNNs are loosely inspired by the visual system of the brain and provide state of the art results in image classification and image vision tasks. A short primer into biological neurons and their communication processes (Section 2.2) will help to understand the elementary differences between these types of networks. Then, I will introduce the leaky integrate and fire (LIF) neuron, which is the basic unit I mainly use in my biological models. Most models found in nature are (complex) dynamical systems and time is a substantial parameter in such systems. Inspired by the recurrent structure of the brain, recurrent neural networks can process sequential or temporal data, i.e. they incorporate time as a variable (Section 2.3). Based on the LIF neuron, I will describe the reservoir computing (RC, Section 2.3.2) approach, specifically the Liquid State Machine which is the network I utilize for my biological simulations. The last section (Section 2.5) concludes with a short excursion exploring the intersection between artificial intelligence and neuroscience and how neuroscientific phenomena inspire learning in artificial systems.

2.1 Overview of Artificial Neural Networks

Most types of neural networks can be described as processing systems which receive input, do computations and output a result. The multilayer perceptron and the convolutional neural network are feed-forward networks loosely inspired by the brain. The origins of artificial neural networks and the initial quest to represent the brain as a logical processing system can be tracked back to the works of McCulloch et al. (1943), Rosenblatt (1957), and Widrow et al. (1960).

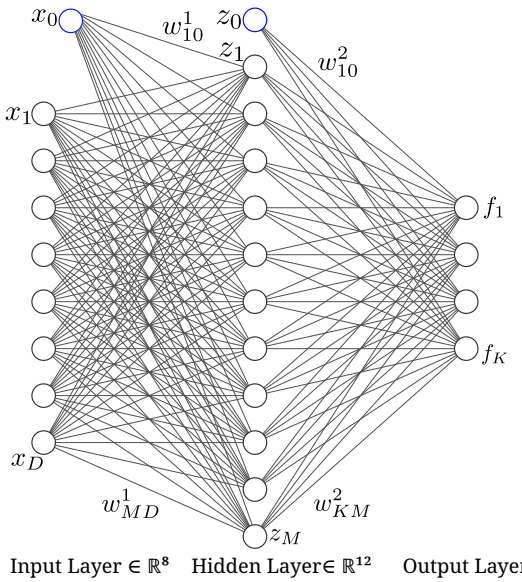


Figure 2.1: A typical multilayer perceptron (MLP). This feed-forward network has a stacked architecture with input, hidden and output layers. The circles are neurons which are processing units. Bias neurons are depicted in blue. Adding more layers makes the network deeper, known as a deep neural network.

2.1.1 Multilayer Perceptron

In its simplest form an MLP consist of 3 different layers, the input, hidden and output layer as depicted in Figure 2.1[1]. Each layer has nodes called neurons or units (circles in Figure 2.1). A nonlinear activation function (see Section 2.1.2) is applied to every neuron. The activation function is not used on neurons of the input layer in order to not transform the input data. Often, bias units (or simply biases, blue circles in Figure 2.1) are added to the network as well. Biases shift the output of the activation function by adding a constant value, e.g. to shift the result towards positive values. In most cases the network is trained in a supervised manner via the backpropagation method (Rumelhart et al. 1995) and optimized by gradient descent. The network training is separated into two steps: a feed-forward phase, where the data propagates from the input to the output layer and a feed-back phase, where the error, calculated in the last layer by comparing the output of the network with a target or label, is propagated back through the network. Every connection between neurons has weights attached to it, which are updated in the backpropagation step.

[1]The figure was drawn with the tool provided by Alex Lenail at https://alexlenail.me/NN-SVG. Descriptions were added afterwards.

In the following is a short mathematical description of neural networks based on the formulations by Bishop (2007). A feed-forward neural network can be constructed from linear combinations of fixed nonlinear basis functions $\phi_j(\mathbf{x})$:

$$f(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x}) \right) \tag{2.1}$$

where $\sigma(\cdot)$ is a nonlinear activation function, e.g. the logistic function (see Equation 2.7). Given a simple network with input, hidden and output layer like in Figure 2.1, the basic network structure can be described as functional transformations of M linear combinations. Thus, a formulation for the first layer with variables $x_1, \ldots, x_D$ can be written as:

$$a_j = \sum_{i=1}^{D} w_{ji}^{1} x_i + w_{j0}^{1} \,, \tag{2.2}$$

with $j = 1, \ldots, M$, where i is the index of the neuron and the superscript 1 is indicating parameters of the first layer, w_{j0} is a bias and w_{ji} is a connection weight. The a_j are called activations and can be transformed by applying the nonlinear activation function $\sigma(\cdot)$:

$$z_j = \sigma(a_j) \,. \tag{2.3}$$

The final description for this simple network is:

$$f(\mathbf{x}, \mathbf{w}) = \sigma_2 \left(\sum_{j=0}^{M} w_{kj}^{2} \, \sigma_1 \left(\sum_{i=0}^{D} w_{ji}^{1} x_i \right) \right) \,, \tag{2.4}$$

where $k = 1, \ldots, K$ is the index of the output neurons. σ_1 and σ_2 indicate that different activation functions can be utilized for each layer. The bias parameter is absorbed into the weight vectors by clamping an additional input variable to $x_0 = 1$ (c.f. Equation 2.2):

$$a_j = \sum_{i=0}^{D} w_{ji}^{1} x_i \,. \tag{2.5}$$

In supervised learning setting the output, e.g. the prediction in a classification task, can now be compared against the label (or target) $\mathbf{y}$ from the dataset. The total error can be calculated as:

$$E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} ||\hat{\mathbf{y}}_n - \mathbf{y}||^2 \tag{2.6}$$

where $\hat{\mathbf{y}}_n = \mathbf{f}(\mathbf{x}_n, \mathbf{w})$ is the network output for a set of input vectors $\{\mathbf{x}_n\}$, $n = 1, \ldots, N$. Equation 2.6 is the Mean Squared Error. This error can be minimized with different optimization techniques such as gradient descent (GD). While Chapter 3 will primarily cover

gradient-free optimization techniques, gradient descent is described in detail in Section 3.2.

2.1.2 Activation Functions

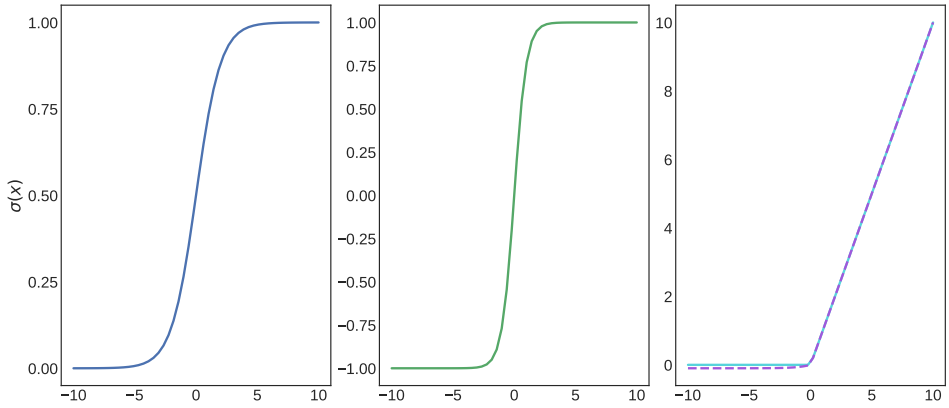


Figure 2.2: Activation functions. Left in blue is the graph of the logistic or sigmoidal function. In the middle the tanh function's graph in green and on the right the graph of the rectified linear unit (ReLU) in cyan. The dashed line in purple depicts the exponential linear unit (ELU), which allows negative values, while ReLU clips them to 0.

An activation function introduces nonlinearities into a neural network. Without activation functions the output signal of a neural network is only a combination of linear functions of the input. Thus, they allow the network to solve nontrivial, nonlinear problems. There is a plethora of different functions; three exemplary types are shown in Figure 2.2. On the left the logistic (or sigmoidal) function applied on a variable $x \in [-10, 10]$ is formulated as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.7}$$

The results are in the interval of $[0, 1]$. The hyperbolic tangent function (tanh, Figure 2.2 middle)

$$\sigma(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.8}$$

has a shape similar to the logistic function. According to Yann LeCun, Bottou, et al. (2012) the tanh is symmetric around the origin on the x and y axis, which produces in average values closer to zero and thus converges faster.

Sigmmoidal function were the de facto standard applied in feed-forward neural networks. Their use in deep neural networks is discouraged due to the vanishing and exploding gradient

problem, which is discussed in Section 3.2.3. Instead, the rectified linear unit ReLU (Figure 2.2 right, cyan line)

$$\sigma(x) = \max(0, x) \tag{2.9}$$

replaced the logistic function as the standard activation function for neural networks (Jarrett et al. 2009) to tackle the vanishing gradient issue. The ReLU is a piece-wise linear function, which preserves properties for an easier optimization with gradient descent. It helps linear models to generalize better, since models can be optimized easier when their behavior is linear (Goodfellow et al. 2016; Nair et al. 2010). However, the drawback of ReLU is the non differentiability at 0. Additionally, neurons can fall into a state where they are inactive for all inputs. In this state gradients cannot "flow" back through the neuron in the back-propagation step, which leads to an inactive state where the neuron "dies". To counteract this issue smoothing can be applied, such as the exponential linear unit (Clevert et al. 2015) (ELU, Figure 2.2 right, dashed purple line):

$$\sigma(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0 \end{cases} \tag{2.10}$$

with $\alpha > 0$.

2.1.3 Convolutional Neural Networks

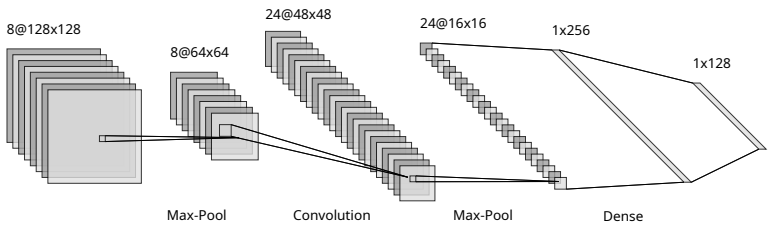


Figure 2.3: A typical convolutional neural network with 2 max-pool layers, one convolutional layer and 2 fully-connected (dense) layers. Note, the network input is omitted.

One of the first published and nowadays widely used Convolutional Neural Network (CNN) is the classical LeNet architecture (Yann LeCun, Boser, et al. 1989). Its performance on image classification sparked wide attention in the computer vision community. This model, depicted in Figure 2.3, was introduced by Yann LeCun to recognize handwritten digits, namely the MNIST dataset (Yann LeCun, Cortes, et al. 2010). The network consists of an input layer, several hidden layers and an output layer. Often the penultimate layer is fully connected (dense) with the last layer, similar to an MLP. The main idea of a CNN is to extract local features from the input image since neighboring pixels are more correlated than distant ones.

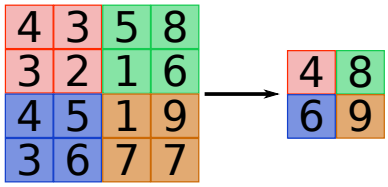


Figure 2.4: Max pooling over 16 pixels divided into 4 subgroups. The kernel size of the max-pool operation is 2, thus the result is a 2×2 (sub-sampled) group with the maximum value of each subgroup.

These low-level features (e.g. edges) are found in the early layers of the network. They correspond to a small region of the image and help to detect higher-order features (e.g. features of a face) in deeper layers, which in the end provides information of the image as a whole (Erhan et al. 2009). There are three important mechanisms in CNNs (Bishop 2007): 1. local receptive fields, 2. weight sharing and 3. subsampling (i.e. pooling). The name CNN already implies that the network employs the convolution operation, i.e. convolving the input with a filter or kernel and passing it to the next layer. The output of this operation is often called feature map. The idea is that units in the kernel only take a small subregion and that all units share the same weight values. The kernel is slid over the image and convolved with the corresponding pixels. Instead of learning the weights like in MLPs, the kernels' weights and biases are learned in a CNN. For a two-dimensional image $I(m, n)$ and a kernel K, the formula of the convolution operation can be stated as follows (Goodfellow et al. 2016):

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n)\,. \tag{2.11}$$

The distance from one position to the next when moving the kernel over the image is called stride. The striding distance reduces the learnable parameters for the filters and the size of the next layer and is an important element when designing the architecture of the CNN.

A further typical operation in CNNs is the sub-sampling known as pooling. Patches of a small receptive field from the previous layer are reduced to one value either by taking the maximum (max pooling) or the average (average pooling). An example of max-pooling over a patch of 4×4 pixels is shown in Figure 2.4. The last layer is often a fully connected layer as depicted in Figure 2.3. Similar to MLPs, activation functions are applied on the output of the neurons. In case of multi-class predictions a softmax nonlinearity (for details see section 3.2) can be applied to the last layer. The machine learning community considered CNNs to be shift invariant, i.e. the displacement or translation in the image is correctly projected in the output prediction. However, recent work shows that CNNs are only equivariant or need several modifications to be shift-invariant (Azulay et al. 2018; Myburgh et al. 2021).

2.2 Introduction into Neurons & Spiking Neural Networks

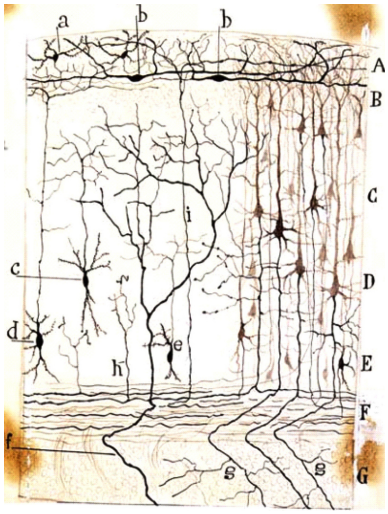


Figure 2.5: A drawing of the cerebral cortex by Ramón y Cajal. **A-D** depict different cell layers, while **F** shows white matter and **G** the striatum. In this picture, Cajal compiled his observations from small mammals such as rabbit or mouse. Modified from Gil Fernández et al. (2014)

The central processing units in our brains and nervous system are neurons. Neurons are highly interconnected with each other. An early drawing of a fragment depicting the cerebral cortex can be seen in Figure 2.5. This picture was drawn by Ramón y Cajal in the 1890s who is one of the pioneers in neuroscience. The figure shows neurons with triangular and circular cell bodies with their extensions partly spanning over different layers. In this drawing, layers **B, C** contain pyramidal cells which have a triangularly shaped cell body. It is estimated that the cerebral cortex has more than 10 billion neurons and the brain in total around 100 billion (Von Bartheld et al. 2016).

2.2.1 Structure of a Neuron

Figure 2.6 is a schematic view of a neuron. The neuron can be separated into three parts: the cell body (soma), dendrites, and an axon. The cell body contains the nucleus, mitochondria, ribosomes, Golgi apparatus and other organelles, which are not shown in the figure. The main function of the soma is to supply the neuron with energy and proteins. The axon extends the body of the neuron from the axon hillock, which controls the electrical signal transmission along the axon. Electrical signals are the primary form of communication between neurons.

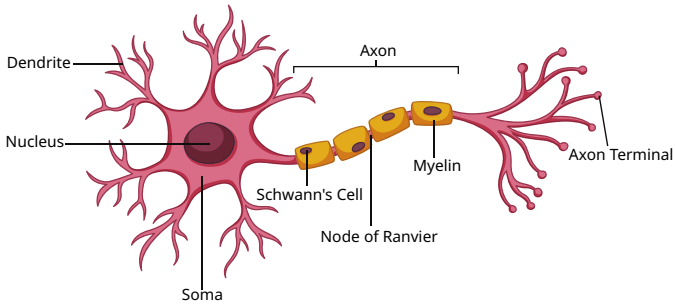


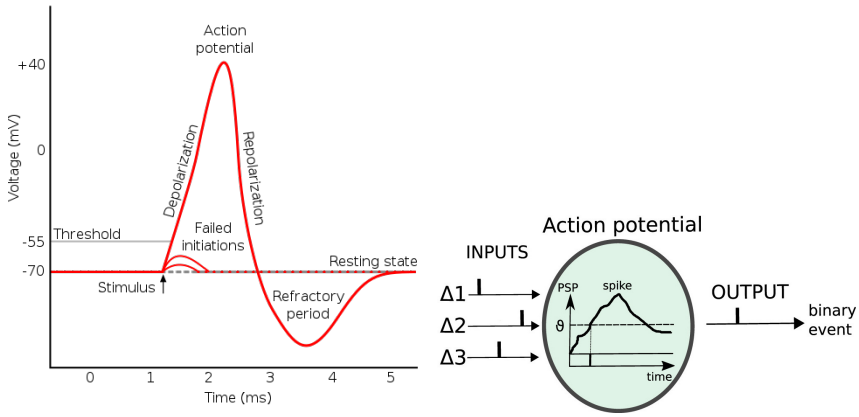
Figure 2.6: A schematic view of a neuron. Modified from Vecteezy.com

The axon can branch into smaller parts and ends at the nerve or axon terminals. Axons can be covered with a myelin sheath, to accelerate the signal transmission. The sheath consists of glia cells, e.g. the Schwann cell (Figure 2.6). Glia cells have many supporting functions for the neurons, however, their role is not fully understood and remains a topic of research (Jäkel et al. 2017). Dendrites extend from the soma and are connected via their synapses to other neuronal axons. In summary, a neuron receives electrical signals from other neurons via its dendrites and passes information to other neurons through its axon. There are other methods to transmit electrical signals such as gap junctions or synapses which are directly attached to the soma, however, they are not as common. The electric signals pass from one neuron to the other through junctions known as synapses. A synapse is located between the end of the axon (presynaptic) and the dendrite of the other neuron (postsynaptic), the space between the two neurons is called synaptic cleft. Signal transmission occurs via chemical activity. The voltage change induced by the action potential (see Section 2.2.2) at the presynaptic neuron triggers the release of chemicals, also called neurotransmitters. These transmitters bind to receptors at the membrane of the postsynaptic cell. This process opens ion channels, ions flow in, depolarize the cell to trigger an action potential and, thus, continue the transmission.

2.2.2 Properties of a Neuron

Every cell has a membrane potential, which is a voltage difference between the cell body and its surroundings. Ion pumps and ion channels at the membrane of a cell are responsible to maintain or change the potential. As aforementioned, neurons communicate via electrical signals. These signals are called action potentials or spikes. Before an action potential can occur, the membrane potential has to be depolarized from its resting state at around $-70\,\mathrm{mV}$ as depicted in Figure 2.7a[2]. Incoming spikes are depolarizing the neuron and only

[2]The shape and amplitude is dependent on the experiment and the location where the measurement was taken.



(a) (b)

Figure 2.7: (a) Typical phases of an action potential, indicating how a spike is evoked over time. Source: Wikimedia Commons. (b) The summation of incoming signals elucidate a spike in the postsynaptic neuron if it passes the threshold. This is abstracted as a binary event in time. Figure from Lobo et al. (2020).

after reaching a threshold potential at around $-55\,\text{mV}$ the action potential is triggered. The membrane potential can rise up to $40\,\text{mV}$, afterwards it repolarizes, i.e. the potential drops below $-70\,\text{mV}$. In the refractory period the neuron cannot be excited, until it reaches its resting state again. The whole process takes less than $5\,\text{ms}$, while the action potential lasts around $1-2\,\text{ms}$.

In computational neuroscience the action potential is described as a "all or nothing" rule. Only if the incoming input is able to excite the neuron, so that the polarization passes the threshold, a spike is emitted. For computational purposes it is useful to abstract the biophysical characteristics of a spike as a binary event in time. As depicted in Figure 2.7b the summation of incoming spikes in a certain time frame stimulates the neuron and triggers it to produce one spike, i.e. one event in time. Spikes elucidated from one neuron in successive order are called a spike train $s(t)$:

$$s(t) = \sum_{i=1}^{n} \delta(t - t_i) \tag{2.12}$$

for n spikes with t as time and δ as the dirac function. Note that, a spike train can contain spikes from several neurons. Fortunately in simulations the origin of a spike can clearly be attributed to a specific neuron. There exists a plethora of mathematical models to describe the dynamics of neurons. In this work, I use the Leaky-Integrate-and-Fire neuron (LIF). The

LIF neuron incorporates a leak term for the diffusion of ions through the membrane. The model can be formulated as (Dayan et al. 2005):

$$\tau_m \frac{dV}{dt} = E_L - V + R_m I_e \,, \tag{2.13}$$

where τ_m is the membrane time constant, R_m the membrane resistance, E_L the resting potential, V the voltage and I_e the input current. In contrast, the resistance R_m in a non-leaky-integrate-and-fire model is infinite, i.e. the membrane is not leaky and is a perfect insulator. The equation can be extended to generate an action potential, i.e. whenever a threshold V_{th} is reached a spike is emitted and afterwards the potential is reset to V_{reset}. When $I_e = 0$ the membrane potential is exponentially relaxed to $V = E_L$ with the time constant τ_m. The next action potential occurs when the membrane potential $V(t)$ reaches the threshold at time step Δt:

$$V(t + \Delta t) = E_L + R_m I_e + (V_{reset} - E_L - R_m I_e) \exp(\Delta t / \tau_m) \,. \tag{2.14}$$

2.3 Recurrent Neural Networks

When we humans speak, the words we use to build our sentences are based on previous words and sentences. Neural Networks such as MLPs or CNNs are not able to process data in this manner. They can only handle vectors of fixed size as inputs such as images or videos and they ouput fixed-sized vectors. Recurrent neural networks (RNNs) are a family of networks which can process sequential or temporal data. Furthermore, they have feed-back (recurrent) connections, which enable them to store information and, thus, have an internal state or memory. By utilizing their internal state, RNNs can work on sequential inputs with variable lengths.

2.3.1 The Basics of Recurrent Networks

Different versions of RNNs have been proposed, such as Elman or Jordan networks (Elman 1990; M. I. Jordan 1997). In the Elman network the output of the previous hidden layer is provided as the input alongside the initial input to the actual layer. In contrast, the Jordan network uses the output of the last layer as the input alongside the normal input. The Elman network can be formulated as:

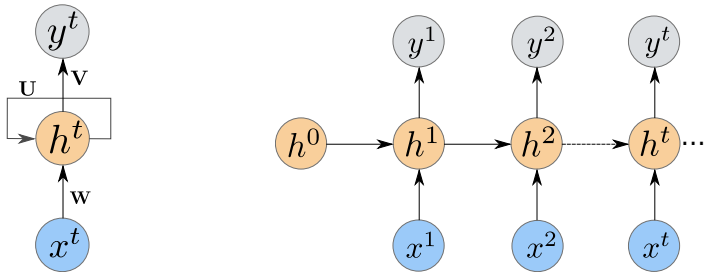


Figure 2.8: (a) A standard recurrent neural network. The arrow to itself in the middle layer indicates a recurrent connection. (b) The computation graph of an unrolled RNN. x^t is the input (blue), h^t the hidden layer activity (orange) and y^t the output (gray). $\mathbf{W}$, $\mathbf{U}$, $\mathbf{V}$ are input, hidden and output layer weight matrices.

$$\mathbf{h}^t = \sigma_h(\mathbf{W}x^t + \mathbf{U}\mathbf{h}^{t-1} + \mathbf{b}_h) \tag{2.15}$$

$$\mathbf{y}^t = \sigma_y(\mathbf{V}\mathbf{h}^t + \mathbf{b}_y) \tag{2.16}$$

where $\mathbf{x}^t$ is an input vector of time t, $t = 1, \ldots, T$ with T the total time, $\mathbf{h}^t$ is a hidden layer vector, $\mathbf{y}^t$ is an output vector. $\mathbf{W}, \mathbf{U}, \mathbf{V}$ are input, hidden and output layer weight matrices, $\mathbf{b}$ is a bias vector, σ_y, σ_h are activation functions. In the original implementation Elman uses context units to store the internal state $\mathbf{h}^{t-1}$. These units can be bypassed by directly sending the internal state to the next layer's $\mathbf{h}^t$. To describe the Jordan network, $\mathbf{h}^{t-1}$ in Equation 2.15 needs to be replaced by $\mathbf{y}^{t-1}$. Figure 2.8a depicts a standard RNN with the input $\mathbf{x}^t$, the hidden layer state $\mathbf{h}^t$ and its output y^t. The internal computing process of an RNN can be described using a computational graph as shown in the diagram Figure 2.8b. In every time step t, the output (following Elman's description) of the previous hidden layer is provided in addition to the vector $\mathbf{x}$ as an input to the actual hidden layer. This process is called unrolling the network. In machine learning, RNNs are typically trained with gradient descent and a technique named as backpropagation through time (BPTT, for details see Section 3.2.1).

2.3.2 Reservoir Computing

Reservoir computing (RC) is a computational approach to process sequential or temporal data. It was independently developed by Jaeger (2001, Echo State Network) and Maass et al. (2002, Liquid State Machine). The system consists of a reservoir of non-linear neurons to project input signals into a higher dimensional space and a readout layer to read the state of

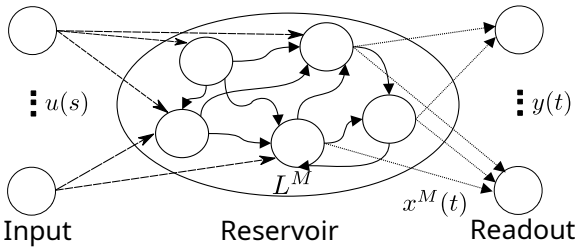


Figure 2.9: A typical reservoir computing network with recurrent connections. The weights inside the reservoir are fixed, while the readout weights are trained.

the network and map it to a target output. The weights inside the reservoir are fixed, but the readouts can be trained with a simple algorithm such as linear regression. In comparison to other recurrent neural networks, a reservoir network can faster learn and has lower training costs. According to Maass et al. (2002) the reservoir computing approach exhibits two major properties:

1. The separation property which is the ability to separate the reservoir's internal states caused by different types of input streams from each other and

2. The approximation property, which is the capability of the readouts to distinguish but also map the internal state(s) to the target outputs.

The separation property depends on the complexity of the reservoir, i.e. the structure of the reservoir such as number of connections, neurons and layers or columns. The concept of separation is comparable to the kernel trick in machine learning (e.g. in support vector machines), where the problem space is elevated and computed in a higher dimensional space (Shi et al. 2007; Hermans et al. 2012). The approximation property depends on the adaptability of the readouts to the given task, i.e. the readouts should be able to understand the internal state of the reservoir and map it correctly to the outputs. However, for different types of inputs, i.e. inputs coming from distinct tasks, the readouts usually have to be retrained.

The reservoir dynamics can be formalized as (Maass et al. 2002):

$$x^M(t) = (L^M u)(t)\,, \tag{2.17}$$

where t is the time, x^M is the (liquid) state of the reservoir, $u(s)$, $s \leq t$, is the input function and L^M is a filter which transforms the input to the reservoir state. The output is obtained as follows:

$$y(t) = f^M(x^M(t))\,, \tag{2.18}$$

f^M is a memory-less readout map that transforms in every time step t the state x^M to the output $y(t)$. In contrast to the filter L^M, the design choice of f^M depends on the specific task. Since the readout maps are memory-less, all information to produce the target output $y(t)$ at time step t has to be contained in the internal state $x^M(t)$ (Maass et al. 2002). It was shown that RC systems exhibit universal approximation properties (Maass et al. 2002; Gonon et al. 2019).

From a biological point of view RC is an interesting approach, since it provides a compelling interpretation about computations in columns of neocortical microcircuits (Buonomano et al. 2009). The connections are not hard-coded to a specific task and the same reservoir can be trained for different input types (only the readouts are task specific). Continuous time inputs are handled in a more "natural" fashion. There is no need to learn (recurrent) weights with biologically implausible learning rules such as backpropagation through time. For instance many dynamical models allow to use the (infinitely) full history of the input for computations. RC have a continuity property called fading memory which states that for any input $u(\cdot)$ the output $(Fu)(0)$ can be approximated by the outputs $(Fv)(0)$. $v(\cdot)$ is any other input function which approximates u on a sufficiently long time interval $[-T, 0]$ (Maass 2011). This means that it is not required to precisely know the value of the input function at any time or the whole history in time. A study conducted by Nikolić et al. (2009) indicated that the primary visual cortex is endowed with the fading memory property.

2.4 Simulation Tools for Neural Networks

Simulations are one of the main tools computational neuroscientist are utilizing to understand the mechanisms of the brain. **NEST** is a popular library for simulating spiking neural networks. The design focuses on the efficient and accurate simulations of point neuron models. In NEST the morphology of a neuron is abstracted into a single, iso-potential compartment, i.e. the morphology of axons and dendrites do not have physical extents. The library supports parallelization with MPI and multi-threading, and thus, scales very well on high performance computing systems (HPCs). Simulations can either be conducted on local machines such as laptops or efficiently be scaled up to large scale runs on HPCs (J. Jordan et al. 2018).

In the artificial intelligence and machine learning community PyTorch (Paszke et al. 2019) is a popular framework for designing and running ANNs. **PyTorch** is an open-source Python package and provides tensor computing on CPUs and GPUs. It allows to build deep neural networks using an automatic differentiation system to calculate gradients, therefore it is efficient and scales well on HPCs. PyTorch implements tensors, which are multidimensional and homogeneous instances of vectors. In this context, a tensor is a data structure and should not

be confused with the mathematical notion of a tensor. PyTorch's core is written in `C++` and has an interface to support Python. Implementing networks in PyTorch follows the design principles of Python, thus the framework is easy to use for programmers who are already familiar with Python.

In this work, the reservoir network in Chapter 5 and the spiking network in Chapter 6 are implemented in NEST. The convolutional neural network in Chapter 4 is designed with PyTorch and the network is trained utilizing PyTorch's gradient descent optimizers, as well as PyTorch's automatic differentiation system.

2.5 Neuro-inspired Learning

The two domains of artificial intelligence and neuroscience are interconnected and many properties of AI models are inspired by neuroscientific principles. This section is a brief excursion to explore the intersection and discusses some examples following the argumentation found in Van Gerven (2017) and Hassabis et al. (2017). One goal in artificial intelligence is to achieve a general or strong intelligence (artificial general intelligence, AGI), comparable to human intelligence. Therefore, it is obvious to copy and adapt principles of neuronal processes from the human brain into artificial models. In order to understand complex biophysical systems Marr et al. (1976) postulated three requirements to be fulfilled. These are known as information processing systems:

1. The *computational* level requires to understand the system on a *functional* level, i.e. what the system does.

2. The *algorithmic* level is how the system solves the problem.

3. The *implementation* level is how the system is physically realized.

According to Hassabis et al. (2017) the first two points help us gain insights into the general processes of the biological brain which can be transferred to artificial machines or agents. The third point helps to understand the physical boundaries of the system, e.g. with regards to energy efficiency, and how it interacts with the environment it is set into.

2.5.1 Visual Cortex as Inspiration for Convolutional Networks

A popular example for a model inspired by the primary visual cortex (V1) is the CNN. V1 is known for advanced preprocessing of visual input captured by the eye. A model of a CNN is

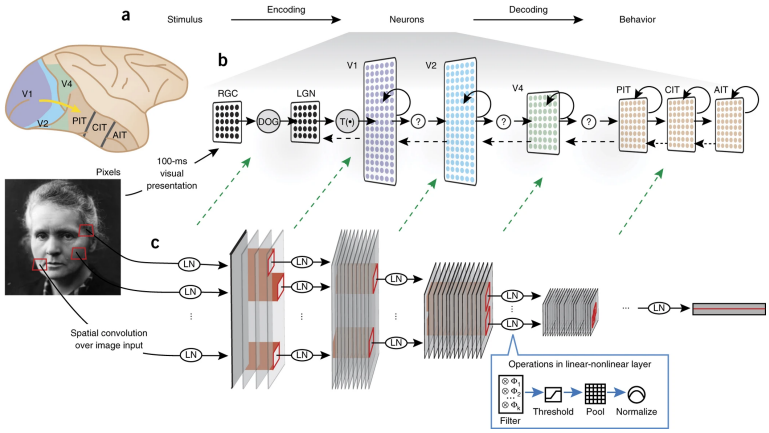


Figure 2.10: Hierarchical CNN as a model for the visual cortex. **a** depicts a macaque brain. The yellow arrow indicates the stream through different areas of the visual system. **b** shows the ventral visual pathway. After the eye captures the image at the bottom, a cascade of operations between individual layers are executed and information is encoded or decoded along the pathway. **c** depicts the corresponding CNN. Green dashed arrows indicate the corresponding layers and operations to the visual pathway. Figure from Yamins et al. (2016).

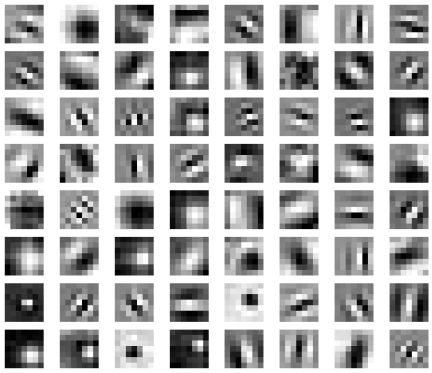


Figure 2.11: The visualization shows the first layer's feature detection filters of a pre-trained ResNet-50 model from PyTorch.

compared to the visual cortex of a macaque monkey in Figure 2.10. Figure 2.10**a** shows the macaque brain and the yellow arrow indicates the visual stream which goes through different areas, i.e. V1, V2, V4 and IT The IT area is further divided into posterior, central and anterior inferior temporal cortex (PIT, CIT, AIT). Neurons in the retina (retinal ganglion cells, RGC in Figure 2.10**b**) perform a simple preprocessing of the image. The difference of Gaussians (DOG) is a preprocessing step, which mimics the feature extraction capabilities

of the retina, to obtain features such as edges. This resembles the feature detector units in early layers of CNNs which are able to identify lines, edges and rough objects as depicted in Figure 2.11, while later layers can detect more complex structures.

The lateral geniculate nucleus (LGN) is responsible for spatial-temporal correlations and relays the information to the primary visual cortex. In Figure 2.10 operations applied in the LGN and V1 are symbolized as $\mathbf{T}(\cdot)$. The corresponding functions in the CNN are marked as **LN** and include filtering, thresholding, pooling and normalization (Figure 2.10**c**). These functions are applied in every layer of the CNN, however, in the deeper layers of the visual pathway these operations are much more complex and not entirely understood. The whole cascade of operations from V1 to IT happens within the first 100 ms of glimpsing on the picture. The IT area can be compared to the last layers of CNNs, in which objects are finally predicted or recognized (DiCarlo et al. 2012).

2.5.2 Other Neuro-Inspired Methods

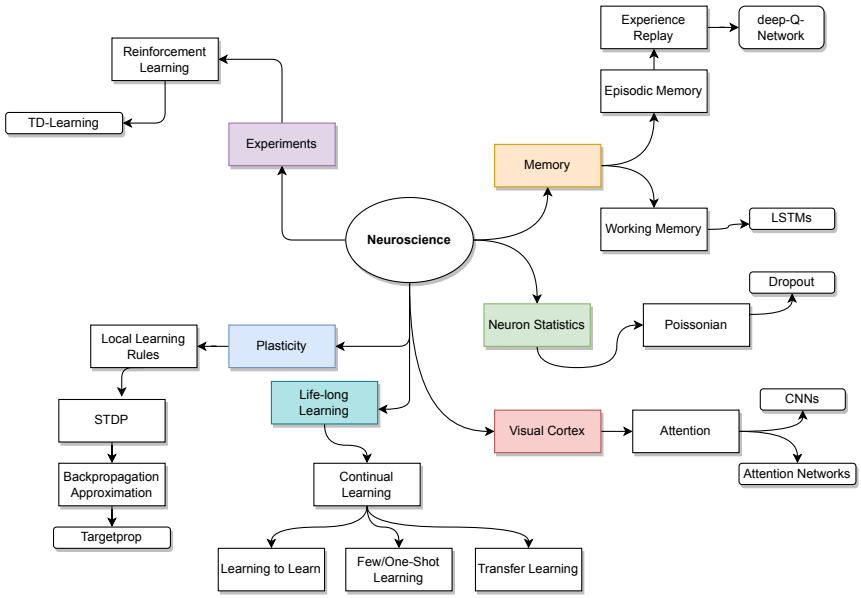


Figure 2.12: Many neuroscientific principles inspired models and methods in artificial intelligence. This figure gives an overview regarding the topics and methods explicated in Section 2.5.2.

In this section I describe other methods influenced by neuroscience, following the examples mentioned in the work of Hassabis et al. (2017). Figure 2.12 gives an overview of the discussed

methods.

Dropout is a regularization method to prevent overfitting when training neural networks. In the training phase neurons are "turned' off, i.e. with a certain probability their output is multiplied with zero. This is motivated by the stochasticity of neurons with Poissonian firing rates.

Reinforcement learning and **temporal difference learning** were inspired by animal behaviour and conditioning experiments. For example, Schultz (1998) conducted reach and grasp experiments with monkeys which were rewarded with juice if they successfully completed a task He recorded the neuronal activity of reward-related dopamine neurons after the monkeys learned the task. The reward was either predicted due to the preceding presentation of a conditioned stimulus (CS) or not predicted. If the reward was not predicted (no additional CS), the dopamine neurons responded upon receiving the reward. However, if the monkeys predicted the reward (due to a CS), the neurons responded to the CS and not to the reward. The observations of such reward based behaviour were incorporated into the temporal difference learning.

Synaptic Plasticity strengthens or weakens the connection between synapses depending on the firing activity. In the backpropagation step weight updates are dependent on non-local errors, i.e. error signals acquired in downstream layers. In contrast, plasticity is based on local information, the pre- and post-synaptic neural activity. Authors of Yoshua Bengio, Lee, et al. (2015) and Yoshua Bengio, Mesnard, et al. (2017) approximate stochastic gradient descent and backpropagation using mechanisms found in spike-timing-dependent-plasticity (STDP). They compare STDP to the delta rule and formulate an algorithm called targetprop which can obtain the gradient in one layer, while not relying on the updates from deeper layers.

Attention modules mimic the primate visual system. Instead of processing the whole input, primates shift their attention from location to location and center it on specific regions. This mechanism is implemented in AI models which glimpse at the input image in every step and update their internal states to select the next location to shift their attention to (Larochelle et al. 2010; Mnih et al. 2015; Vaswani et al. 2017).

Episodic memory is an instance based mechanism, allowing learned experiences to be encoded and stored. The network stores a subset of already trained data and is able to replay the experience in an offline fashion (deep-Q-network; Mnih et al. 2015). One goal is to counteract catastrophic forgetting, a status in which the optimization overwrites the weights when training on a new data set and thus, performs poorly on previously learned data samples (McCloskey et al. 1989; Ratcliff 1990). The episodic learning mechanism to process and store new memories is similar to the one in the hippocampus which encodes new learned information and

consolidates the experience to the neocortex. The learned memory is replayed or reactivated while sleeping and resting (O'Neill et al. 2010).

Working memory is a system with the ability to temporarily store and manipulate information and is important for behavioural actions such as reasoning and planning (Baddeley 2003). Research indicates that working memory is endowed within the prefrontal cortex and interconnected regions (Goldman-Rakic 1991). Long-short-term memory (LSTM) networks, are able to store information into their internal state and maintain it until needed or replaced using a gating mechanism (Hochreiter and Schmidhuber 1997).

Often, models are trained to solve a specific task. However, to reach AGI, the system needs to learn continuously. **Continual learning** is an approach to mimic life-long learning. Learning to learn, transfer learning, one-/few-Shot learning and continuous learning are part of this category. The idea is to quickly generalize and perform well on new data sets using the experience from previously learned examples. Humans and other primates have the ability to learn with only a few examples presented (Harlow 1949). How this higher-level of learning is achieved is still unknown. Theories state that the formation of conceptual representation is emerging, which encode abstract and relational information (Doumas et al. 2008).

2.6 Summary

Artificial neural networks are inspired by their biological counterparts. While the multilayer perceptron reflects the functionality of biological neural networks in an abstract and simplified manner, the convolutional neural network takes its inspirations from the visual system. For example, activation functions are non-linear functions and resemble the thresholding function in spiking neurons to create an action potential. An action potential or a spike is an electrical signal a biological neuron uses to communicate with other neurons. In computational neuroscience this signal is often represented as a binary event. A spike is only elucidated if the incoming spikes at the neuron exceed a certain threshold.

Spiking neural networks are dynamical systems with recurrent connections. Recurrent neural networks are able to process sequential or temporal data. The reservoir network is a specific type of recurrent networks, where the neurons in the reservoir are randomly connected. When adapting the network, only the connection weights from the reservoir to the output are changed. The reservoir network has two characteristics, the separation property, which is the ability of the network to separate internal states, and the approximation property, which maps the internal states to the outputs.

The intersection between neuroscience and artificial intelligence is an ongoing field of research.

From an artificial intelligence perspective, this interconnection is called neuro-inspired learning. Many machine learning methods and frameworks draw inspiration from neuroscience and incorporate neuronal properties into their design. Bio-inspired learning helps to overcome limitations and provides, among other things, the ability for local, temporal or continual learning and efficient training via memorization (e.g. working and episodic memory).

3 Optimization methods applied on Neural Networks

In the previous chapter I introduced different types of artificial neural networks, components and properties of biological neurons and discussed recurrent networks, in particular the reservoir network. Since both fields complement each other, I shortly discussed the intersection between neuroscience and artificial intelligence. The previous chapter was considered mostly from a bottom-up perspective – the building blocks to construct and execute neural networks were represented. However, to gain an understanding of how such models work, we also have to account the top-down or the functional view. In light of this, our models need to learn to do meaningful tasks so we can evaluate their performance, which can provide an interpretation about the usefulness of the model. Training neural networks requires optimization, which is the focus of this chapter.

After a general introduction into optimization, I will explicate the gradient descent algorithm (Section 3.2), a powerful and efficient optimization technique frequently used in machine learning when training neural networks. To train feed-forward neural networks, gradient descent is applied in conjunction with the backpropagation algorithm. A second version is called backpropagation through time and is used in recurrent neural networks to incorporate time as a variable (Section 3.2.1). Although gradient descent optimization is a very popular technique, it can lead to issues such as vanishing gradients when applied on recurrent neural networks. Moreover, from a biological standpoint the direct application of gradient descent is implausible and cannot be used to optimize spiking neural networks without complex approximations (Section 3.2.4). Fortunately, a plethora of alternative optimization techniques exists. I will describe different metaheuristic variants, such as the genetic algorithm (Section 3.3.1), a nature inspired evolutionary algorithm, and the ensemble Kalman filter (Section 3.3.3), an iterative filtering strategy and a suitable technique for solving inverse problems. Additionally, I introduce the ant colony optimization (Section 3.3.2) as a technique for finding shortest paths. In a later chapter, this introduction will help to understand the difference between training SNNs to control swarm agents and applying classical ant colony optimization with pre-defined, rule-based models to steer swarms.

3.1 Introduction into Optimization

Optimization describes the process of finding the best decision from a selection with regards to certain constraints. This results in an optimal performance given a set of optimal parameters of a (real-world) model formulated in a mathematical way. For example, maximizing or minimizing a function is a typical optimization problem. An optimization problem can be mathematically described as (Malik et al. 2021):

$$\min_{\mathbf{x}} f_i(\mathbf{x}),\ i = 1, 2, \ldots, P \tag{3.1}$$

Subject to

$$g_j(\mathbf{x}) = 0,\ j = 1, 2, \ldots, Q \tag{3.2}$$

$$h_k(\mathbf{x}) \leq 0,\ k = 1, 2, \ldots, R \tag{3.3}$$

where $f_i(\mathbf{x}), g_j(\mathbf{x}), h_k(\mathbf{x})$ are functions for the input or decision vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ and $\mathbf{x} \in \mathbb{R}^n$ and each $x_i \in \mathbb{R}$. The functions f_i are called objective functions, if $P = 1$ then there is a single objective. In Equation 3.2 $g_j(\mathbf{x})$ denotes the equality constraint function, while $h_k(\mathbf{x})$ in Equation 3.3 denotes the inequality constraint function. The space spanned by $\mathbf{x}$ is the search space and the space formed by the objective functions is the solution space. Objective functions can be linear or non-linear. To maximize an objective function the inequality constraints can be expressed as ≥ 0, since the maximization of $f_i(\mathbf{x})$ is the minimization of $-f_i(\mathbf{x})$ with $-h_k(\mathbf{x}) \geq 0$. For a decision variable x_i, $x_{i,\min} \leq x_i \leq x_{i,\max}$ are the bounds. The maximization $\max(\cdot)$ can replace the $\min(\cdot)$ operation depending on the optimization target. The functions $f_i(x)$ can conflict with each other forcing the multi-objective optimization to find a trade-off to satisfy the functions in the best possible manner. For example, this can result in reducing the error of one function while increasing the error of the other function. A Pareto optimal solution is desirable, i.e. solutions which degrades the performance of one or more objectives if the other objective function is improved. In multi-objective optimization it is important to find the right measure which considers all the objectives and their importance or priority in the optimization process. A classical measure is the weighted sum, also called weighted fitness F:

$$F(\mathbf{x}) = \sum_{m=1}^{M} w_m f_m(\mathbf{x}), \tag{3.4}$$

where M is the set of objectives and the w_m are factors which weight the importance of the objective for the overall fitness. Thus, the weighted sum measure influences the performance of the optimization and the final result.

3.1.1 Optimization algorithms

Most of the optimization algorithms are iterative methods. In every iteration of the optimization process the optimizer updates the decision vector $\mathbf{x}$ with the goal to provide a solution which performs better than the solution of the previous iteration. The optimization step incorporates the objective function and its constraints as well as the performance of the optimization target. Requirements for optimization algorithms are (Malik et al. 2021):

1. Efficiency: They should be resourceful. For example their computations should be lightweight and fast. Here, parallelization can be helpful as well, it eases the compute load by distributing calculations to different processing units, e.g. cores or compute nodes on high performance computers, or architectures, e.g. CPU and GPU.

2. Accuracy: They need to be precise, but not sensitive to errors, e.g. to rounding errors or outliers in the data.

3. Robustness: They should be applicable to different optimization problems and perform well.

3.2 Gradient Descent and Backpropagation

Gradient descent (GD) is a first-order optimization method, used to find the local minima of an objective function. Gradient descent is a popular choice to update the connection weights w in neural networks. To optimize the weights GD does a small step in the direction of the negative gradient:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t) \tag{3.5}$$

where the hyper-parameter $\eta > 0$ is the learning rate, which determines how fast the gradient can descend towards a local minima. t is the iteration step and E is a function, e.g. the error function when training a neural network (see Section 2.1.1). GD is an iterative method, after every update the gradients are re-evaluated and the optimization procedure is repeated. In this case, the whole dataset is required to calculate the new gradients, this is known as a *batch* method. In contrast, stochastic gradient descent (SGD) uses a mini-batch, i.e. only a small partition of the dataset is taken to update the new gradients. In comparison to the standard gradient descent, SGD provides an efficient solution with low computational costs (Yann LeCun, Boser, et al. 1989).

To evaluate the error function $E(\mathbf{w})$ of a neural network a technique called backpropagation (BP; Rumelhart et al. 1995) – also called backprop – can be applied. We can apply the

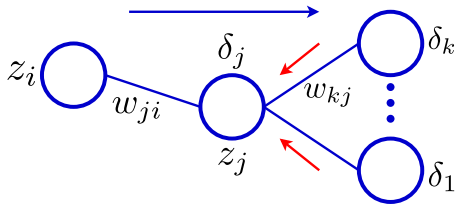


Figure 3.1: Illustration of the backpropagation algorithm. The blue arrow indiciates a forward pass of the network while the red arrows depict the error propagation back to neuron j of a hidden layer. δ_k is the backpropagated error of neuron k. z_j is the activation of neuron j. Modified from Bishop (2007).

backpropagation algorithm to the network presented in Section 2.1.1. The derivative of E_n needs to be calculated with respect to the weight w_{ji}, where $E_n\,, n = 1, \ldots, N$, is the error for one data point. Since E_n depends on the weight w_{ji} via the input a_j to neuron j, the chain rule can be applied to calculate the partial derivative (Bishop 2007, see also Figure 3.1):

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}\,. \tag{3.6}$$

Equation 3.6 can be shortened as $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$, with $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$ and $z_i = \frac{\partial a_j}{\partial w_{ji}}$. The δ's are often referred as errors. For the output we have $\delta_k = \hat{y}_k - y_k$, with $\hat{y}_k$ the prediction value and y_k the target value for output neuron k. The gradients for the hidden layers can be calculated as:

$$\frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}\,, \tag{3.7}$$

where k are all neurons to which neuron j has connections to. By further substituting, the final backpropagation formula becomes:

$$\delta_j = \sigma'(a_j)\sum_k w_{kj}\delta_k\,. \tag{3.8}$$

3.2.1 Backpropagation Through Time

Backpropagation through time (BPTT) is a special application of the backpropagation algorithm for recurrent neural networks. The computation graph of an RNN is unrolled for each time step as depicted in Figure 2.8b. Then, the BP algorithm is applied to the unrolled graph. The gradients need to be differentiated with respect to the weights of the matrices $\mathbf{W}, \mathbf{U}, \mathbf{V}$ (see Equation 2.15 and Equation 2.16).

Let $\mathcal{L}\left(\mathbf{y}, \hat{\mathbf{y}}\right)^t$ be the error or loss function (e.g. MSE) for the time step t, where $\mathbf{y}$ is the target

and $\hat{\mathbf{y}}$ the prediction of the network. Then the gradient for the loss with respect to the weight matrix $\mathbf{U}$ can be derived as:

$$\frac{\partial \mathcal{L}^t}{\partial \mathbf{U}} = \frac{\partial \mathcal{L}^t}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}^t}{\partial \mathbf{h}^t} \frac{\partial \mathbf{h}^t}{\partial \mathbf{U}} . \tag{3.9}$$

The term backpropagation through time becomes visible in the term $\frac{\partial \mathbf{h}^t}{\partial \mathbf{U}}$, since all gradients of the previous states of $\mathbf{h}^t$ have to be obtained and summed up:

$$\frac{\partial \mathbf{h}^t}{\partial \mathbf{U}} = \sum_{k=1}^{t} \frac{\partial \mathbf{h}^t}{\partial \mathbf{h}^k} \frac{\partial \mathbf{h}^k}{\partial \mathbf{U}} , \tag{3.10}$$

$$\frac{\partial \mathbf{h}^t}{\partial \mathbf{h}^k} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^i}{\partial \mathbf{h}^{i-1}} \tag{3.11}$$

Equation 3.10 is recursively applied from $k = t$, while in every update step k is decremented. It is important to note that the same weights are reused at every time step. The gradients with respect to $\mathbf{V}$ and $\mathbf{W}$ can be calculated in a similar manner as in Equation 3.9. The multiplications depicted in Equation 3.11 can cause the vanishing or exploding gradient problem. A direct implementation of the BPTT algorithm following the equations above is tedious, not very efficient and error prone. Fortunately, machine learning frameworks like TensorFlow (Martín Abadi et al. 2015) and PyTorch (Paszke et al. 2019) provide modules for automatic differentiation to compute the gradients by using computational graphs. For example the autograd module [1] is such a graph implemented in PyTorch.

3.2.2 Adaptive Moment Estimation

The adaptive moment estimation (Adam) is an alternative gradient descent algorithm proposed by Kingma et al. (2014). It computes adaptive learning rates for each weight by using estimations of the first and second moments of the gradient and keeps an exponentially decaying average m^t of past gradients $g_t = \nabla_\omega L_t$ with L_t the loss:

$$\begin{aligned} m^t &= \beta_1 m^{t-1} + (1 - \beta_1) g^t \\ v^t &= \beta_2 v^{t-1} + (1 - \beta_2)(g^2)^t \end{aligned} \tag{3.12}$$

where m^t and v^t are estimates of the first and the second moment of the gradients respectively and t is the iteration step. β_1 , β_2 are the decay factors for the moving average. To finally obtain the new weights w, the gradient is calculated:

$$w^{t+1} = w^t - \frac{\eta}{\sqrt{\hat{v}^t} + \epsilon} \hat{m}^t , \tag{3.13}$$

[1] https://pytorch.org/docs/stable/autograd.html

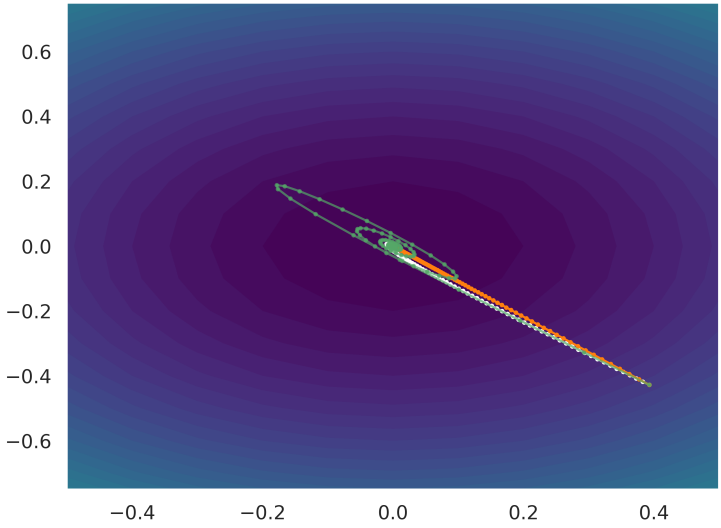


Figure 3.2: Gradient descent and Adam applied on an elliptic paraboloid function. The orange line depicts vanilla gradient descent, the green and white lines show the Adam optimizer with different learning rates. The green line is oscillating around the minimum in the center of the image due to a high learning rate.

with

$$\hat{m}^t = \frac{m^t}{1-\beta_1^t}, \quad \hat{v}^t = \frac{v^t}{1-\beta_2^t}, \tag{3.14}$$

where $\epsilon > 0$ is a small scalar to prevent division by 0 and $\eta > 0$ is the learning rate parameter. Root-squaring and squaring is applied element-wise.

Figure 3.2 depicts the gradient descent (white line) and Adam (green and orange lines) optimizer applied on an elliptic paraboloid function $z = x^2 + y^2$ with $x, y \in [-1, 1]$. The gradient descent has a learning rate parameter of $\eta = 0.02$ and the white line shows the minimization process from a randomly chosen point to the global minimum in the center of the image. The orange line depicts the optimization process using Adam, with $\beta_1 = 0.9, \beta_2 = 0.999, \eta = 0.01$. It follows the gradient straight to the minimum, similar to the normal gradient descent optimization. Increasing the learning rate to $\eta = 0.1$, results in a big step size and the gradient descent oscillates around the minimum as the green line depicts. According to Kingma et al. (2014), the Adam optimizer is computationally efficient with little memory requirements and performs well on noisy problems with sparse gradients or on large data sets and models with lots of parameters. However, the authors of Keskar et al. (2017) recommend to switch to stochastic gradient descent in later stages of the training, since it enhances the performance and generalizes better.

3.2.3 The Problem of Exploding and Vanishing Gradients

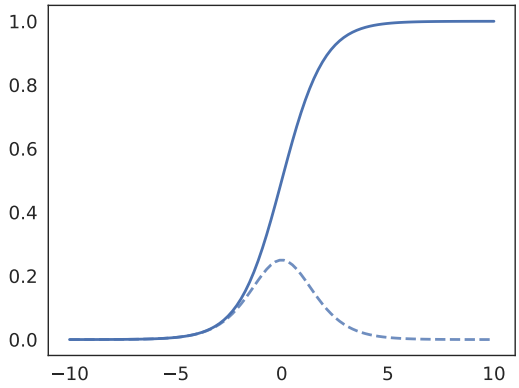


Figure 3.3: The blue line depicts the sigmoid function, the dashed blue line is its derivative.

The problem of vanishing or exploding gradients does not only occur within RNNs but also in very deep feed-forward networks. In every step of the backpropagation the weights are updated following the gradient of the error function with respect to the weights. The sigmoid activation function (see Section 2.1.2) can be utilized to understand the issue in an intuitive way. Figure 3.3 depicts its derivative $\sigma(x)(1-\sigma(x))$ as a dashed blue line. If the neuron gets saturated, i.e. the input to the sigmoid function is big and the output is close to 0 or 1, the gradient becomes very small. This can immensely decelerate the training process (Y. LeCun et al. 1998; Yann LeCun, Bottou, et al. 2012). Similarly, the exploding gradient problem appears if the gradients keep getting larger during the backpropagation step, which results in large weight updates and leads to an unstable network. According to Goodfellow et al. (2016) the parameter space contains sharp non-linearities, which resemble steep cliffs and give rise to large gradients. The parameters close to such regions can "jump" to suboptimal values after the gradient descent update is applied, thus, hindering or stopping the optimization process (Pascanu et al. 2013). A simple solution to counteract these issues, is to clip or scale down the gradients $\mathbf{g}$ by its norm whenever it exceeds a threshold v (Pascanu et al. 2013):

$$\mathbf{g} \leftarrow \frac{\mathbf{g}v}{\|\mathbf{g}\|}, \text{ if } \|\mathbf{g}\| > v\,. \tag{3.15}$$

Several studies investigate why the logistic function is not suitable as an activation function (Y. LeCun et al. 1998; Pennington et al. 2017). Instead, different activation functions such as ReLU and ELU (see Section 2.1.2) are suggested. However, small numerical instabilities due to possible singular instabilities can occur for small smoothing parameters. Furthermore, alternative weight initialization schemes are proposed to tackle the vanishing

and exploding gradient problem (Glorot et al. 2010; K. He et al. 2015). Glorot et al. (2010) investigate different activation functions regarding the saturation of the gradients and introduce the Xavier initialization. Especially, K. He et al. (2015) incorporate the number of incoming and outgoing neurons from the previous and next layer to normalize the weights and thus, stabilize the training.

The effects of vanishing and exploding gradients on deep neural networks will be presented and further analyzed in Section 4.1.

3.2.4 Biological Plausibility of Gradient Descent and Backpropagation in Spiking Neural Networks

Gradient descent and backpropagation are efficient techniques to optimize artificial neural networks. However, their biological plausibility and application in the mammalian brain is still debated and open questions remain how backpropagation can be implemented in the cortex (Crick 1989; Grossberg 1987).

Backpropagation requires **synaptic symmetry** in the forward and backward pass. The error δ of Equation 3.8 needs to be sent from the post-synaptic to the pre-synaptic neuron. In ANNs this is realized as a flow back through the layers along the weights w. However, the error is dependent on the feedback weights which are symmetric to the weights of the forward pass (see also Figure 3.1). Having the same set of weights for two different connection paths is biological implausible. This is known as the "weight transport problem" (Grossberg 1987). Furthermore, to update one synaptic connection, distant error signals and the weights of other connections need to be specified, however, this information is not locally available for the synapse. Recent work showed alternative solutions and approximations using plausible computations. For example, Lillicrap et al. (2016) propose feedback-alignment; a method which uses fixed random weight feedback matrices between the network layers and allows backpropagation to update the feed-forward weights to align with the feedback matrices. A modification suggested by Nøkland (2016) utilizes the random feedback to pass directly the error obtained at the network output to every layer. Samadi et al. (2017) apply the direct feedback alignment to spiking neural networks. However, an effective application of feedback alignment algorithm is limited to shallow networks, deep networks lose accuracy under such a procedure (Bartunov et al. 2018).

Training ANNs is divided into the feed-forward step and the backwards step. Only at the output of the network the error is evaluated and propagated back. From a biological viewpoint this approach is problematic. In mammalian brains there is no such a separation into two distinct phases. For example, plasticity between synapses occurs locally and in a constant

manner. Recently, Sacramento et al. (2018) propose a framework in which a neuron can be simultaneously used for activity propagation, error encoding and error propagation in different locations of the neuron without the need for different learning stages. The weights update depends on the difference between the predictive input steered by synaptic plasticity and error feedback from deeper layers. Although the framework is biological plausible in many aspects, the implemented model requires a strict connectivity between the layers, which does not align with neuroscientific experiments.

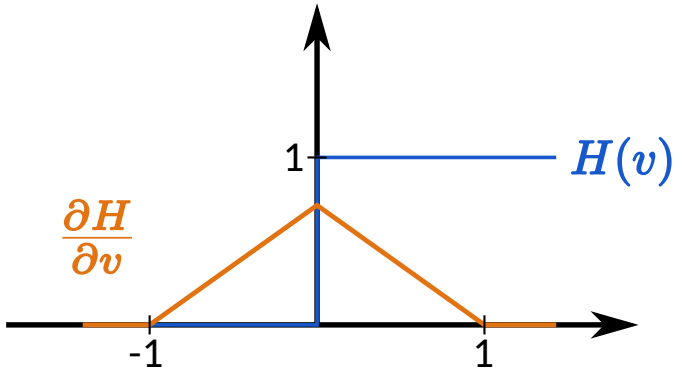


Figure 3.4: The all-or-nothing behaviour of a spiking neuron is expressed as a Heaviside step function $H(v)$ (blue), its derivative is zero except at 0, where it is ill-defined. Piece-wise linear functions (orange) can approximate the gradient at 0.

Another challenge is the **non-differentiability** of the spiking non-linearity. In the backpropagation step and in BPTT the activation function needs to be derived as well (see Equation 3.8 and Equation 3.9). In the spiking neuron, however, the all-or-nothing behaviour to emit a spike can be understood as a Heavside step function, thus the derivative is zero except at 0 (blue line in Figure 3.4), where it is ill-defined. Thus, an exact calculation of the gradient is intractable and often approximations are suggested. The simplest solution, is a piece-wise linear approximation as depicted in the schematic Figure 3.4 (orange line). In recent work surrogate gradients were introduced (Neftci et al. 2019; Bellec et al. 2019; Woźniak et al. 2020) to overcome this issue. Surrogate gradient learning involves the membrane potential to obtain gradients or is based on the recent spiking activity of the neuron. Other approaches apply smoothing on the neuron by altering the synapse properties. For example, Huh et al. (2018) modify an IF neuron and replace the non-linearity by a continuous-valued gating function, thus, they are able to directly apply backpropagation.

Finally, backpropagation requires to calculate the error between the target and the prediction. How such a **target signal** could be realized in the brain is not clear.

3.3 Metaheuristics

Heuristic algorithms are based on a trial and error approach. In general, they should be applicable for many problems and agnostic of the task itself. Metaheuristics are based on heuristics, however, they use a trade-off between local and random search. In this context, also taking the term "meta" into account, metaheuristic can be understood as a higher-level strategy to discover solutions of a problem and to guide the search process. Metaheuristics perform better and computationally more efficient than simple heuristic approaches. They are approximating and often non-deterministic (Blum et al. 2003). Most metaheuristic optimization algorithms are inspired by nature, they mimic biological or evolutionary processes and consist of two fundamental components:

1. **Selection of the best** ensures the algorithm converges to an optimal solution by ranking the problem solutions and selecting the best.

2. **Randomization** avoids that the algorithm gets trapped in a local optima and enlarges the pool of possible solutions, i.e. it increases the diversity of the possible solutions.

In most cases, a trade-off between these two components will lead to a suitable local optimum or even to the global optimum. Furthermore, in metaheuristic optimization a balance between exploration and exploitation of the search process is very important. **Exploration** is the divergence of the search space, i.e. the search for possible better solutions than the already found ones by exploring the search space maximally. For example, increasing the search space when initializing the optimization algorithm is a strategy to cover a broad range of search solutions. **Exploitation** is a concept to strengthen the search space by exploring better performing solutions in the neighborhood of found solutions. For instance, perturbing the solutions with noise to acquire new search candidates is a common applied technique.

Metaheuristic algorithms can be classified into two types, population based and trajectory based. **Population** based algorithms use several individuals or agents to explore the search space, in the mathematical framework described in Section 3.1 they are different instances of the vector $\mathbf{x}$. These individuals are often randomly initialized according certain distributions and interact with each other in every optimization step. Examples of such algorithms are ant colony optimization, evolutionary strategies including genetic algorithms, ensemble Kalman filter and particle swam optimization.
Trajectory based methods are single point search techniques. They do not rely on a population of individuals but use a single agent or solution which explores the search space in a piece wise manner, thus, describing a single trajectory. Simulated annealing, tabu search, iterated local search and variable neighborhood search are realizations of trajectory based algorithms.

3.3.1 Evolutionary Strategies

Algorithm 1: Basic genetic algorithm. To create a new population the crossover and mutation steps are applied on a selection of fittest individuals.

```
Initialize population P with M individuals
while convergence condition not met do
    f ← Fitness(P)
    for i in P do
        O ← Select(P, f)
    end
    for o in O do
        P′ ← Crossover(o)
    end
    for j in P′ do
        P″ ← Mutate(j)
    end
    f ← Fitness(P″)
    P ← Select(P ∪ P″, f)
end
```

Evolutionary strategies are metaheuristic search techniques and are inspired by biological evolution. Within the family of evolutionary strategies the genetic algorithm (GA) is the most prominent gradient free, optimization algorithm. GA was introduced by Holland (1992) and mimics evolutionary processes such as selection, reproduction and survival of the fittest (Simon 2013). Because the algorithm is based on biology the terminology is inspired by nature as well. The elements spanning the search space (c.f. Section3.1) are called individuals, the set of individuals is the population $\mathbf{P}$. The performance of the population is evaluated by a fitness function (the objective function) and results in a fitness value $\mathbf{f}$. The fitness can be a vector or a scalar and depends on the objective function as well as the corresponding task. The design of the fitness of is the one the most challenging but important parts, since it defines the success of the whole optimization. Single elements of the individuals (mathematically x_i) are the chromosomes. A selection phase ensures the best or fittest individuals are passed to the next generation (also known as elite strategy). This is in order to keep a high fitness within the population and not to coincidentally remove high performing individuals. In the tournament phase the individuals from the actual generation are compared with each other and ranked according to their fitness. The Hall of Fame (HoF) keeps track of the best individuals in every generation. In the HoF, the individuals with a higher fitness replace worse performing ones and are stored over the generations. Crossover and mutation are two techniques to randomly recombine the chromosomes of the individuals to generate a new population. Crossover takes two parent chromosomes and exchanges them with each other. Mutation perturbs chromosomes, e.g. by adding Gaussian noise. Algorithm 1 illustrates a

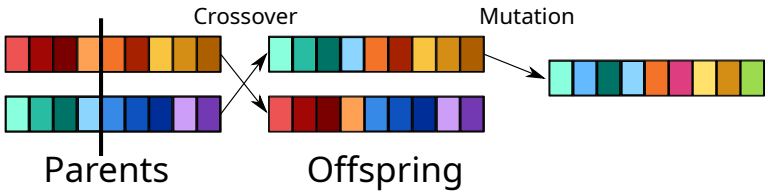


Figure 3.5: A sketch of the recombination process. The left side shows two parental individuals, the colors depict chromosomes. In the first steps crossover is applied. The second step shows the mutation of an individual. Given a probability the chromosome is changed, e.g. by adding Gaussian noise. The crossover step does not have to be applied beforehand.

simple implementation, written in pseudo-code. The algorithm iterates until a convergence criterion is met, for example this can be a predefined generation number or a target fitness value. In every generation the fitness is evaluated before selection and recombination. Depending on its value, the fitness defines which individuals go into crossover and mutation and which individuals are kept for the next generation. This ensures to search for good solution candidates in the local neighborhood (exploitation step) but allows also to increase the search space (exploration for better optima). Depending on the fitness the final step mixes the newly obtained generation with the old kept generation.

Different, alternative versions to the classical GA exist. In a recent work Salimans et al. (2017) create a new population by applying Gaussian noise to the fittest individuals and calculate a stochastic gradient estimate over several episodes to update the parameters. Similarly, the authors of Wierstra et al. (2014) apply the natural evolution strategies (NES) optimizer, a sampling technique which utilizes a multivariate Gaussian distribution to create new individuals.

3.3.2 Ant Colony Optimization

The ant colony optimization algorithm (ACO) is based on the movement of ants to randomly forage for food in the environment and quickly bring it back to the nest. The ants are able to deposit chemical signals, pheromones, on the ground to mark the path and guide other cohorts from the nest to the food source. The pheromone leaves a fading trail and the more ants follow this trail the higher the concentration. ACOs are applied in travel salesman problems or graph problems and can be described as follows. η_{ij} is a heuristic value when moving between two points i and j (e.g. hive and food patch, usually set as $1/d_{ij}$ with d the distance), then p_{ij}^k is the probability that an ant k will move from point i to j using the

amount of pheromone τ_{ij} (Iba et al. 2020):

$$p_{ij}^k = \frac{\tau_{ij}\eta_{ij}^\alpha}{\sum_{h \in J_i^k} \tau_{ih}\eta_{ih}^\alpha}\,, \tag{3.16}$$

where J is the set of points the ants can still visit and α is a factor to weight the distance to travel. The equation expresses the condition that the ants will move towards the higher pheromone concentration, but also includes past searches as well as the heuristic to further explore the environment. The pheromone trails are updated when every ant completed her turn. The amount is increased or decreased depending on the found solution:

$$\tau_{ij} = (1-\rho)\tau_{ij} + \sum_k \Delta\tau_{ij}^k\,, \tag{3.17}$$

where ρ is the pheromone evaporation coefficient. $\Delta\tau_{ij}$is the amount of pheromone deposited by ant k:

$$\Delta\tau_{ij}^k = \begin{cases} Q/L_k & \text{if ant } k \text{ goes from } i \text{ to } j, \\ 0 & \text{otherwise}\,. \end{cases} \tag{3.18}$$

Q is a constant (e.g. $Q = 1$) and L_k is the length of the tour the kth ant takes. Because the pheromone evaporates with time, long tours contribute to a low density of concentration. In contrast a high pheromone amount indicates a high transit of ants from point i to j. This results in finding the shortest path between the food sources and the nest.

3.3.3 Ensemble Kalman Filter

The ensemble Kalman filter (EnKF; Evensen 1994) is an iterative numerical method for nonlinear dynamic filtering problems under noise. It is suitable for problems with a large number of variables, for example partial differential equations in geophysical models such as weather forecast simulations and similar data-assimilation applications (Evensen 1994; Janjic et al. 2014; Schwenzer et al. 2019) as well as in mathematical studies (Schillings et al. 2018; Herty et al. 2019). Recently, Iglesias et al. (2013) applied the EnKF also to inverse problems. The EnKF consists of a set of particles, the ensemble, which approximate the state distribution (Katzfuss et al. 2016). The state distribution is the distribution of a state space model in time which consists of an observation and an evolution part. The ensemble is propagated through time and updated with new incoming data. Thus, the method can be divided into different phases as depicted in Figure 3.6: The **initialization** which takes a priori knowledge into account, the **prediction** and the **update** step. If a priori knowledge is available, it can be incorporated into the EnKF in form of state matrices. At the start

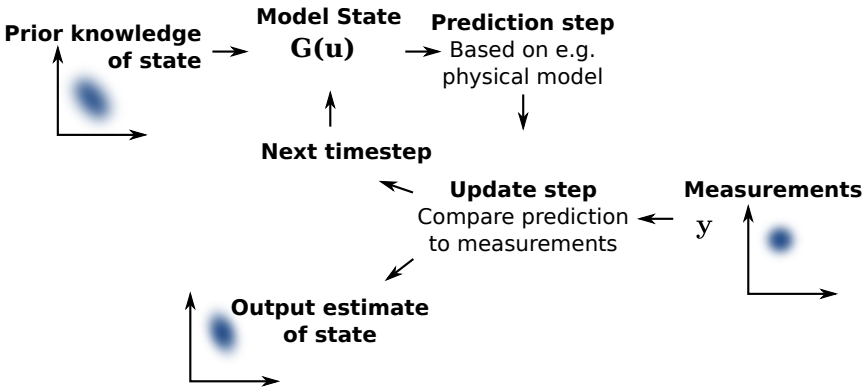


Figure 3.6: The prediction and update step of the ensemble Kalman filter. A priori knowledge state matrices, e.g. the weather state over the last days for a weather forecasting scenario, or a certain initialization scheme of the ensembles can be incorporated into the method. In the next step the model makes a prediction to approximate the solution. The prediction is compared in the update step with new incoming observations or measurements. Afterwards, the method converges or continues the iteration. Modified from Wikimedia Commons

of the iteration, assumptions on the ensemble's initialization, such as the state distribution being a Gaussian normal distribution with a specified mean and standard deviation, can be accounted for. The prediction, is the model output $\mathcal{G}(\mathbf{u})$ and provides a solution or an approximation to the given optimization problem. Afterwards, the output is compared with new incoming observations (or measurements) to update the ensemble to incorporate new data. The ensemble is shifted by moving its mean towards the distribution of the state, this is schematically indicated in the Figure 3.6 as a moving point cloud from the prior state to the output estimate. The method converges if the convergence criterion is met or continues with the next iteration. The EnKF has its root in Bayesian inference, the method uses linear update rules to convert the prior ensemble distribution to a posterior distribution after each update (Katzfuss et al. 2016). This means, in the next iteration the posterior is accounted as the prior knowledge of the state.

Following the formulation by Iglesias et al. (2013) the EnKF can be described as:

$$\mathbf{u}_j^{n+1} = \mathbf{u}_j^n + \mathbf{C}(\mathbf{U}^n)\left(\mathbf{D}(\mathbf{U}^n) + \mathbf{\Gamma}^{-1}\right)^{-1}\left(\mathbf{y}^{n+1} - \mathcal{G}(\mathbf{u}_j^n)\right) \tag{3.19}$$

where $\mathbf{U}^n = \{\mathbf{u}_j^n\}_{j=1}^J$ is the set of the ensemble, n is the iteration index, $\mathbf{u}_j \in \mathbb{R}^d$ is an ensemble member (or state vector) and J is the total number of the members in the ensemble and $\mathcal{G}(\mathbf{u}) \in \mathbb{R}^{K\times d}$. $\mathbf{\Gamma} \in \mathbb{R}^{K\times K}$ is the covariance matrix related to the measurement of noise

and often multiplied with the scaling factor $\frac{1}{\Delta t}$. In this work, $\mathbf{\Gamma}$ is an identity matrix multiplied with a small scalar, i.e. $\mathbf{\Gamma} = \gamma \mathbf{I}$. $J, \gamma,$ and n are the hyper-parameters of the EnKF.

The matrices $\mathbf{C}(\mathbf{U}^n)$ and $\mathbf{D}(\mathbf{U}^n)$ are sample covariance matrices defined by:

$$\begin{aligned} \mathbf{C}(\mathbf{U}) &= \frac{1}{J} \sum_{j=1}^{J} (\mathbf{u}_j - \overline{\mathbf{u}}) \otimes \left(\mathcal{G}(\mathbf{u}_j) - \overline{\mathcal{G}} \right) \in \mathbb{R}^{d \times K}, \\ \mathbf{D}(\mathbf{U}) &= \frac{1}{J} \sum_{j=1}^{J} \left(\mathcal{G}(\mathbf{u}_j) - \overline{\mathcal{G}} \right) \otimes \left(\mathcal{G}(\mathbf{u}_j) - \overline{\mathcal{G}} \right) \in \mathbb{R}^{K \times K}, \\ \overline{\mathbf{u}} &= \frac{1}{J} \sum_{j=1}^{J} \mathbf{u}_j, \;\; \overline{\mathcal{G}} = \frac{1}{J} \sum_{j=1}^{J} \mathcal{G}(\mathbf{u}_j), \end{aligned} \tag{3.20}$$

where $\otimes$ is the tensor-product. Both matrices are important for the convergence behaviour of the EnKF ,which can be compared to gradient descent. The matrices define in which direction the mean in the feature space shifts and how the ensemble variance contracts.

Equation 3.19 is the update step of the EnKF, while the latter part requires the model output $\mathcal{G}(\mathbf{u})_j$ and the observations $\mathbf{y}$ obtained in an earlier step. The later part subtracts the model output from the observation, thus it corrects the model prediction by calculating the distance to the observations. The greater the distance, the larger the prediction error, which has a greater weight in the update step. Kovachki et al. (2018) have shown that under simplified assumptions it exists $\overline{\mathbf{u}}^\star = \text{argmin}(\Phi(\mathbf{u}, \mathbf{y}))$ and $\frac{1}{N} \sum_{j=1}^{J} \mathbf{u}_j^n \xrightarrow{n \to \infty} \overline{\mathbf{u}}^*$. The first equation describes the minimization of the ensemble members given the labels $\mathbf{y}$ in a supervised learning setting and will be utilized in Chapter 4 to optimize a convolutional network. The latter, more theoretical term indicates that a local or global minimum exists if the number of members goes towards infinity. Then, the optimum is the mean of all ensemble members.

3.4 Summary

Optimization is the process of obtaining the best set of parameters with regards to one or more criteria. In mathematical terms an optimizer minimizes or maximizes a problem expressed as a function. In machine learning and deep learning, gradient descent and backpropagation is commonly used when training neural networks. Backpropagation through time incorporates the notion of time and is applicable on artificial recurrent neural networks. For every time step the recurrent network is unrolled so that backpropagation can calculate the derivatives with respect to the weights, which is a time and compute intensive operation. In deep networks the exploding or vanishing problem can appear, which leads to a saturation of the neurons,

thus, hindering any further network learning.

The process of how biological networks are trained in the brain is still an ongoing area of research. Reinforcement learning in combination with synaptic plasticity (see Section 2.5.2) are developed theories indicating how biological networks may be trained in the brain. Moreover, neurotransmitters and neuromodulators, such as dopamine, serotonin, and norepinephrine, play a role in regulating synaptic plasticity. These chemicals can influence the strengthening or weakening of neuronal synapses based on the brain's dynamics and (external) environmental conditions. However, optimizing biological networks using standard backpropagation is biologically implausible and a direct application is not possible. Instead, metaheuritics as optimization techniques can provide alternative solutions. They do not require the calculation of a gradient and are also model agnostic. Many population based optimization methods, such as the ant colony optimization and genetic algorithm, are inspired by biological observations. Especially, the ensemble Kalman filter and the genetic algorithm are two techniques, which will be utilized in the following chapters when optimizing spiking neural networks.

4 Deep Neural Networks Optimized by the Ensemble Kalman Filter

The vanishing and exploding gradient problem was introduced in the previous chapter (Section 3.2.3), in this chapter I will discuss the effects of vanishing gradients on neural networks (Section 4.1) and present the ensemble Kalman filter as an alternative, gradient-free optimization technique applied on a convolutional neural network. The problem is connected to the parameter settings for neural networks, such as the initialization scheme of the weights and the selected activation functions. Applying the EnKF the problem of vanishing gradients does not occur, because the method does not explicitly calculate gradients and allows the employment of non differentiable activation functions. Even with ill-conditioned settings the EnKF is able to optimize the network and an overall satisfactory classification performance on the MNIST and letters dataset is achieved (Section 4.4).

The main aim of this chapter is to analyze network parameter configurations which lead to the vanishing gradient problem when training the network with gradient based methods and how the EnKF provides an alternative and stable solution, since it does not require to calculate gradients. Therefore, I will explicate a setup where a CNN learns to classify a task and compare the performance of two gradient descent methods and the EnKF while optimizing the weights of the network during the training process. Analyzing the gradients and weights over the training iterations will show how vanishing gradients affect the learning when optimized with gradient descent. In contrast, this effect does not appear when optimizing with the EnKF.

A typical hyper-parameter for the ensemble Kalman filter is the number of state vectors in the ensemble, which influences the overall performance of the network. In Section 4.4.2, I will show that, additionally, the accuracy of the network on the classification tasks depends to a great extent on the ensemble member size. Similarly, one other hyper-parameter is the repetition size, i.e. how often the same sample of data-points is presented to the network while training and before a new set is shown. To automatically adjust the repetition and ensemble member size, I will introduce an adaptive technique based on the previously obtained accuracy (Section 4.4.4).

4.1 The Effects of Vanishing Gradients in Deep Neural Networks

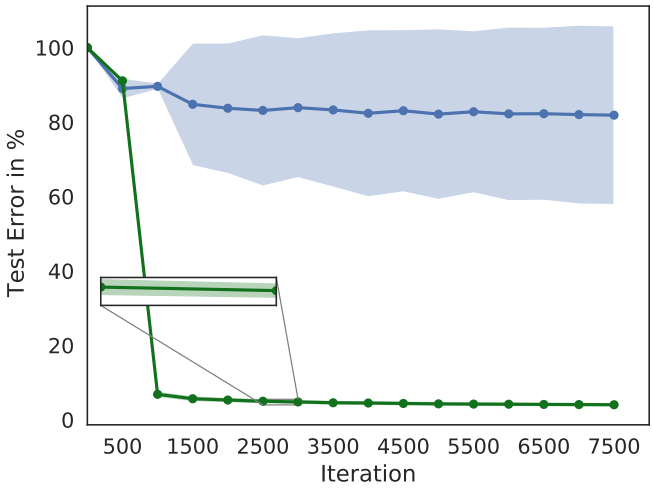


Figure 4.1: Mean test error of a CNN optimized by SGD (blue line) and EnKF (green line). The network is trained for one epoch and the mean error over 10 separate runs is depicted. The shaded area is the standard deviation of the error. The weights are sampled from the normal distribution with a standard deviation of $\sigma = 1$. The test is performed on a test dataset every 500 iterations.

The initialization of weights and the selected activation functions are factors which determine the performance of deep artificial neural networks (Xie et al. 2017; Hayou et al. 2019). Poorly selected parameters can result in loss of information in the feedforward step of the network training or lead to the vanishing or exploding gradient issue in the backpropagation phase (Y. Bengio, P. Frasconi, et al. 1993; Y. Bengio, Simard, et al. 1994; Hochreiter, Yoshua Bengio, et al. 2001; Sutskever et al. 2013, see also Section 3.2.3). Figure 4.1 illustrates the vanishing gradient problem for a CNN (see Section 4.2) trained to classify the MNIST (Yann LeCun, Cortes, et al. 2010) dataset over one epoch for 10 different runs. The dark colored line is the mean test error, on a test dataset in every 500th iteration. The shaded area depicts the standard deviation of the error of all 10 runs. If the initialization of the weights is non-optimal the SGD is not able to train the network due to vanishing gradients (a detailed explanation is given in Section 4.3). In contrast, the EnKF optimizes the network to perform well. In all 10 runs the error stays low as the standard deviation in the zoomed view displays.

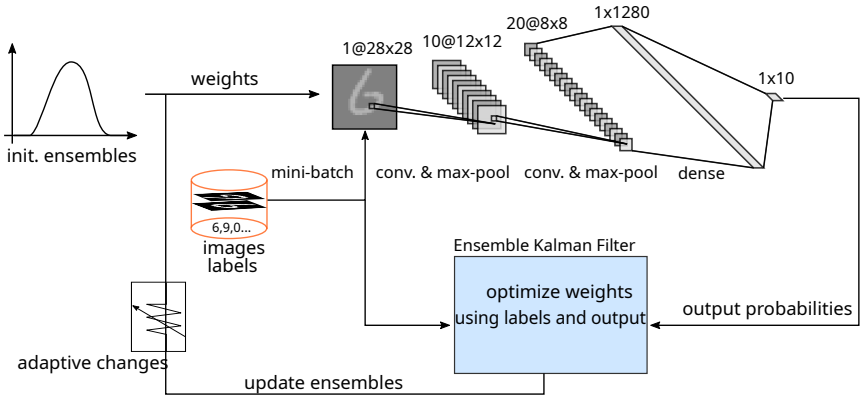


Figure 4.2: The iterative workflow depicts a training run of a CNN. The network is optimized by the EnKF. In the initialization the network weights are sampled form a normal distribution. A mini-batch of digits is presented to the network, which classifies the input. Afterwards, the weights are updated by the EnKF and fed back to the network. The hyper-parameters of the optimizer are adjusted in an adaptive manner.

4.2 Experimental Setup and Network Optimization

In this setting, the network is a convolutional neural network, consisting of two convolutional layers and a fully connected output layer, as depicted in Figure 4.2. The activation function is the logistic function and applied on all layers. The kernel size of the convolution is of size 5×5 and has a stride of 1. On both convolution layers max pooling is applied with a kernel size of 2 and a stride of 2. The experiments are conducted on a compute node with an NVIDIA Tesla K20c graphic card, an Intel i74770 CPU, the operating system is Scientific Linux 7.4. The network is implemented with PyTorch v1.2.0.

4.2.1 Ensemble Kalman Filter Optimization

Parameter estimations fall under the domain of inverse problems (Tarantola 2004), thus the EnKF can be applied here. The ensemble Kalman filter (see Section 3.3.3) requires the evaluation of the feedforward pass, and omits the backpropagation step when training deep neural networks. In contrast to gradient based methods, the EnKF is easily parallelizable, since the model outputs are not dependent on each other and the covariance matrix calculations can be executed in parallel as well.

Kovachki et al. (2018) and Mirikitani et al. (2008) presented first results training deep neural

networks and recurrent neural networks with the EnKF. Kovachki et al. (2018) formulate the network training as a minimization problem for Φ:

$$\Phi(\mathbf{u}, \mathbf{y}) = \|\mathcal{G}(\mathbf{u}) - \mathbf{y}\|_{\Gamma}^2 , \tag{4.1}$$

where $\mathcal{G}(\mathbf{u})$ is the model output, i.e. the output of the feed-forward network, and $\mathbf{y}$ is the target or label. In this setting, the ensemble $\mathbf{u}_j$ is the weight matrix of a convolutional neural network.

The optimization workflow is depicted in Figure 4.2, the experiments are performed on the MNIST dataset. In Section 4.4.3, classification results using the letters dataset are shown as well. In the initialization, the network biases are set to 0 and the weights $\mathbf{W} = (W_{i,k})_{i,k}$ for each layer are drawn from a normal distribution:

$$W_{i,k} \sim \mathcal{N}(\mu, \sigma^2) \tag{4.2}$$

with $\mu = 0$, $\sigma \in [0.01, 10]$, where $\mathcal{N}$ is the normal distribution with mean μ and variance σ.

Every member of $\mathbf{u}^n$ at $n = 0$ is initialized J times, i.e. a single member j of the ensemble corresponds to a random initialized weight matrix $\mathbf{W}$ drawn from the normal distribution $\mathcal{N}(\mu, \sigma^2)$. This matrix corresponds to $\mathbf{u}_j^0 := \mathbf{W} \in \mathbb{R}^{\text{layer} \times \text{weights}}$ for $j = 1, \ldots, J$. $\mathbf{u}_j^n$ for $n > 0$ is obtained by the update step in Equation 3.19. The initial ensemble member size is $J = 5000$. Different settings will be discussed in Section 4.4.2. For J iterations the j-th ensemble is initialized as weights for the network and one classification step is performed for a mini-batch of size 64. The scaling factor γ is a constant in all runs and set to $\gamma = 0.01$.

The EnKF receives the output of the network and the targets and a new set of the ensemble J is calculated according to Equation 3.19. A repetitive presentation of the mini-batch before switching to a new set of images increased the network performance and helped the optimization process to converge faster. To reach a high accuracy on the training set the initial number of repetitions is 8. A smaller number or even omitting the repetition is possible after 500 iterations and when a high test accuracy is obtained (see Section 4.4.4). Every 500th iterations, after completing the training, the test errors are acquired. The mean vector $\overline{\mathbf{u}}$ of the ensemble is calculated and set as the network weights in order to test the network on a test set in order to obtain the classification error.

4.2.2 Training with Gradient Descent

In a second setting, SGD and Adam are chosen as optimizers when training the network. The training procedure is similar to the workflow described as above. The cross entropy loss, a

combination of the negative log-likelihood and log softmax, is applied on the model output $\mathbf{z}_j = \mathcal{G}(\mathbf{u}_j)$:

$$\sigma(\mathbf{z}_j) = -\log\left(\frac{e^{z_j}}{\sum_i e^{z_i}}\right). \tag{4.3}$$

However, for longer runs the network is not able learn or improve its accuracy on the test set when optimized by SGD (see Section 4.3.1).

4.3 Numerical Results with Gradient Descent Optimizers

In this section, the focus is on how the gradients and activation values are affected by the vanishing gradient problem when optimized with SGD and Adam. The initialization scheme described in Section 4.2 is applied here as well. Afterwards, the influence of different sigmoidal activation functions, i.e. the logistic function, ReLU, Tanh, on the network performance on the basis of the test error is discussed. Similarly, a varying number of the ensemble members affect the performance of the EnKF. Furthermore, training results on the letters dataset (Bulatov 2018) are presented.

4.3.1 Optimization with SGD: Non evolving Gradients and Activation Values

In the aforementioned experimental setup applying SGD as an optimizer leads to a saturation of the neurons in the early training stage. Even in later phases the units cannot recover as depicted in Figure 4.3a. Here, the evolution of the mean in dark blue for the first, orange for the second and green line for the last layer and standard deviation of the activation values, respective in light blue, orange and green vertical bars, is shown. The logistic function is applied on the hidden layers over all mini-batch iterations. The activation values saturate around 200 iterations for the first and second layer. The mean activation value of layer one is 0.4597, with a standard deviation of 0.2976, for layer two the mean activation is 0.3178 and with 0.4624 as the standard deviation. The mean activation value in layer three (0.1) stays constant through the whole training and has a standard deviation of 0.2908.

The gradients saturate as depicted in the histogram of Figure 4.3b. The plot shows additionally a kernel density estimation (KDE) to highlight peaks in the distribution. After 800 mini-batch iterations the distribution does not change and the mean of the gradients is close to zero, the gradients are saturated.

After more than 50 epochs the network is still not able to learn. This is shown in Figure 4.4a, the gradients have zero mean for all three layers. While the first two layers have a standard

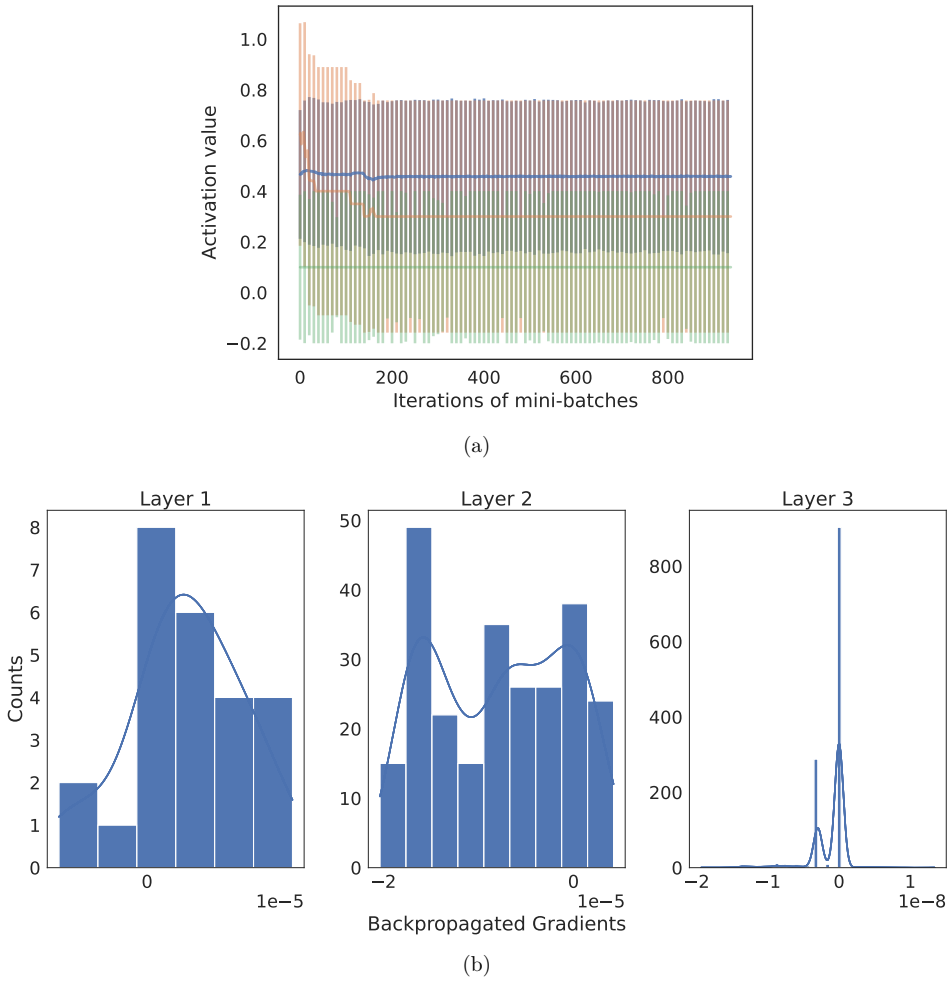
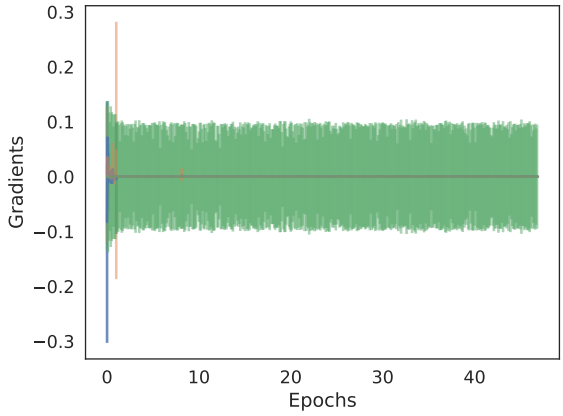


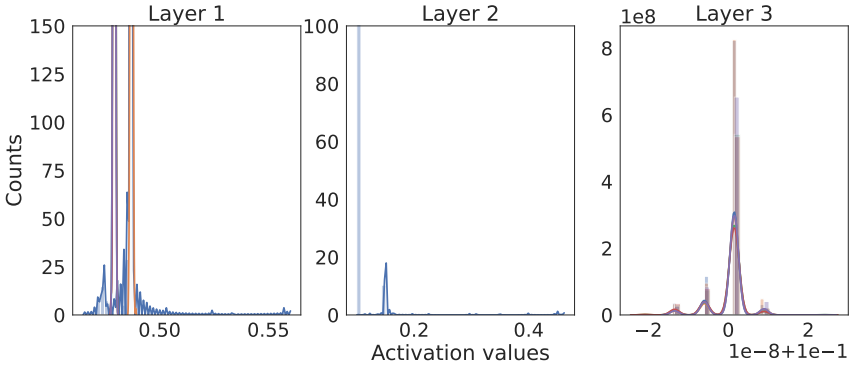
Figure 4.3: Activation values and gradient distributions in one epoch using **SGD** as an optimizer. (a) Mean and standard deviation of the activation values during training. The lines depict the means of the three layers (blue: first layer, orange: second layer, green: third layer) and the corresponding vertical bars are the standard deviations. Every 8th entry is displayed. (b) The distribution of gradients is depicted as a histogram with a KDE plot. The mean value is close to 0 for all three layers. One epoch is run with a mini-batch size of 64.

deviation around the value of 0, the third layer has an oscillating standard deviation around the value of 0.1. Similarly, the activation functions stay between a mean of 0 and 0.5, as it can be seen in the distribution of the mean activation (Figure 4.4b). A small shift in the

mean from 0.475 towards 0.5 occurs in layer one, however, the values of the other layers do not change and are close to zero.



(a)



(b)

Figure 4.4: Activation values and gradient distributions over epochs using **SGD** as an optimizer. (a) Mean of the backpropagated gradients (blue: first layer, orange: second layer, green: third layer) and the standard deviation (blue, orange and green vertical bars) for all three layers over 50 epochs. (b) Mean distributions of activation values over 50 epochs depicted for all layers. Colors indicate the epoch. Blue: epoch 0, orange: epoch 10, green: epoch 20, violet: epoch 40.

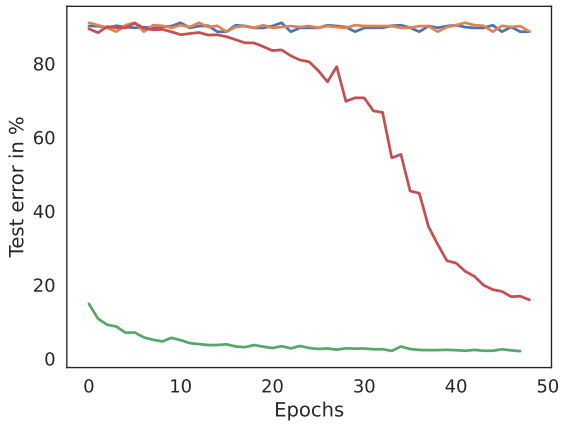


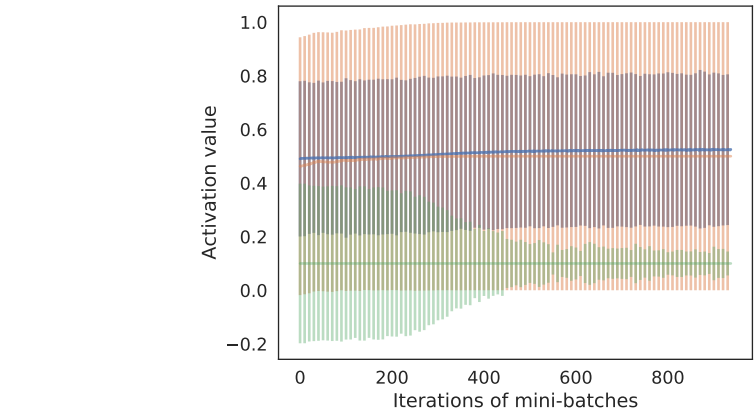
Figure 4.5: Network test error. The network is trained for 50 epochs on the MNIST dataset and optimized by **SGD** and **Adam**. The weights are initialized using a normal distribution with different standard deviations σ. Blue: SGD, $\sigma = 1$; orange: SGD, $\sigma = 3$; green: Adam, $\sigma = 1$; red line: Adam, $\sigma = 3$.

4.3.2 Optimization with Adam: Slowly evolving Gradients and Activation Values

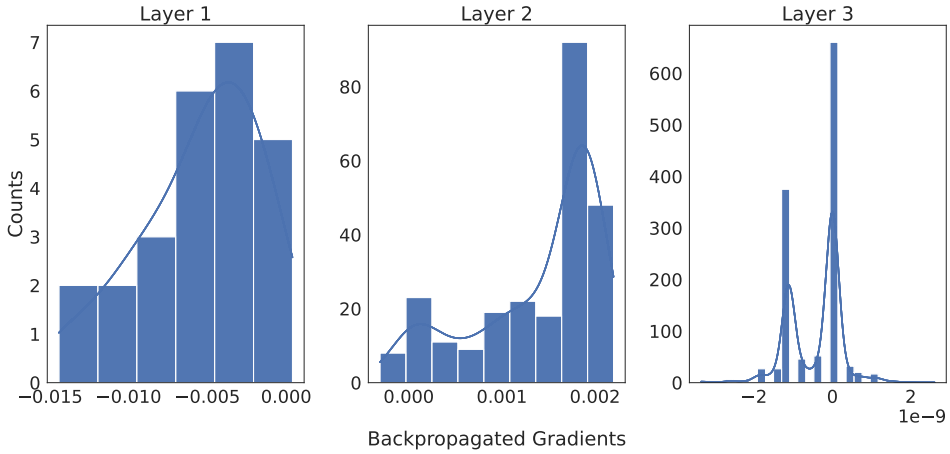
In contrast to SGD, Adam is more robust regarding the performance when optimizing the network. For a standard deviation of $\sigma = 1$ the network starts to correctly classify the dataset. However, this process is slow, after 10 epochs the network reaches a test accuracy of 95% (see Figure 4.5, green line). Fixing $\sigma = 3$, the behaviour resembles the SGD setting in the first epochs (Figure 4.5, red line). Until the 30th epoch, the network is not able to classify the dataset, afterwards it slowly starts to perform better but does not reach the same accuracy as the setting with $\sigma = 1$.

In the setting with $\sigma = 3$, the mean and standard deviation of the activation values within in the first epoch for layer one does not show a fast saturation as depicted in Figure 4.6a (dark blue). An increase of the mean value of the activation values from 0.49 up to 0.53 can be observed (an overall mean value of 0.5124), with a standard deviation of 0.2860. Analyzing layer two, a slow saturation in the longer run can be seen. From the first iteration on, layer two increases its mean activation value (0.47) to the value 0.5, but it stagnates at around the 400th iteration with a value of 0.5 (overall mean is 0.4957, standard deviation is 0.4965). The mean value of layer three stays constant at 0.1. Figure 4.6b shows the mean distribution of gradients after one epoch and obtained after the backpropagation step. The mean is close to zero for all layers.

Figure 4.7a shows how the mean activation values evolve within 50 epochs. Only after 30



(a)

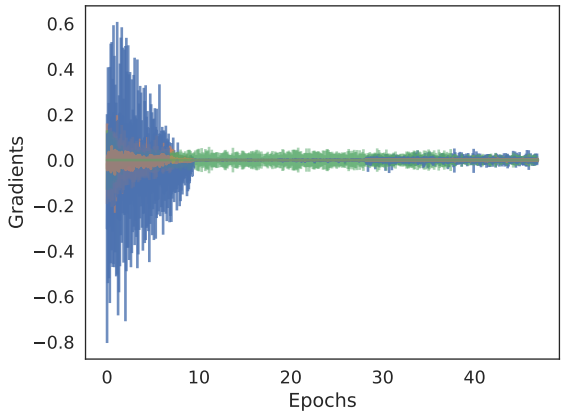


(b)

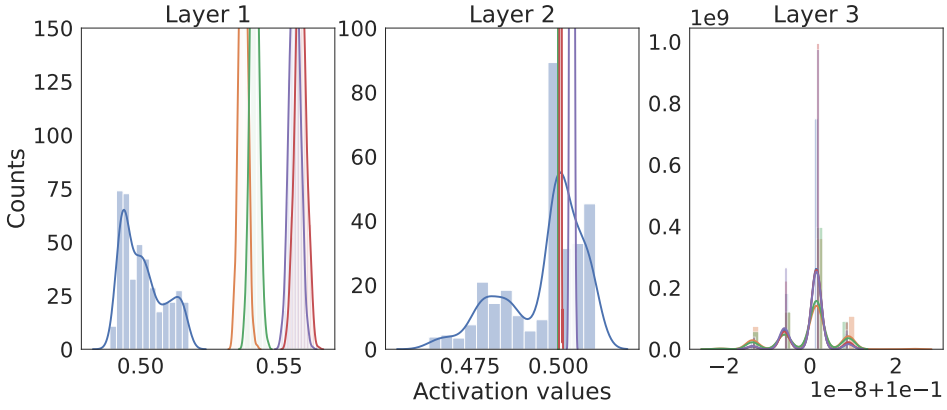
Figure 4.6: Activation values and gradient distributions in one epoch using **Adam** as an optimizer, $\sigma = 3$. (a) Mean (shown as lines, layer 1: dark blue, layer 2: orange and layer 3: green) and standard deviation (blue, orange and green vertical bars) of the activation values of the three network layers over one epoch. (b) Histogram depicting the distribution of the gradients. The mean value for all three layers is close to 0. The input has a mini-batch size of 64.

epochs the performance is starting to increase as shown in Figure 4.5. Before 30 epochs the network is not able to classify the dataset, i.e. the test error is higher than 60%. The mean activations in Figure 4.7a explain this issue. In the first epoch the distribution is around 0.49, after 40 epochs the distribution evolves around 0.56. In contrast, the distribution of the

activation values of layers two and three do not change, they stay around 0.5 and 0.0.



(a)



(b)

Figure 4.7: Activation values and gradient distributions over epochs using **Adam** as an optimizer. (a) Mean (line) and standard deviation (vertical bar) of the backpropagated gradients for all three layers for 50 epochs. (b) Mean distributions of activation values over 50 epochs for all three layers. Colors indicate the epoch. Blue: epoch 0, orange: epoch 10, green: epoch 20, violet: epoch 40.

However, after 30 epochs the network is able to classify the dataset and exhibits a better performance (less than 15% test error). Up to epoch 10 the mean and standard deviations of the backpropagated gradients in layer one and layer two do not fluctuate much as depicted in Figure 4.7a. Until then, the network is not capable to learn (test accuracy below 10%). Mean values in the first epochs for layer one are between -0.3 and 0.2 and standard deviations

are between -0.8 and 0.6 with decreasing trend. For layer two the mean values are between -0.1 and 0.1 and the standard deviations are between -0.2 and 0.2. From epoch 10 up to epoch 28 mean and standard deviation of layer one and two decrease and stay close to zero. After the 28th epoch the network performs better. An increase and a small variance in the mean and standard deviations of layer one is observable (layer one means around: $[-0.02, 0.02]$, standard deviations: $[-0.01, 0.06]$). There is a small shift in layer two (means around $[-0.01, 0.01]$, standard deviation: $[-0.01, 0.03]$). However, the mean activation value of layer three is constant in all iterations with a small value close to zero. In contrast, the standard deviation starts in the first epoch at 0.16 and decreases to 0.01. The results can be interpreted as follows:

1. A network can recover from the vanishing gradient problem, i.e. the saturation of its gradients, if given enough time to train.

2. The network can converge to classification rates with high accuracy only for suitable initial settings.

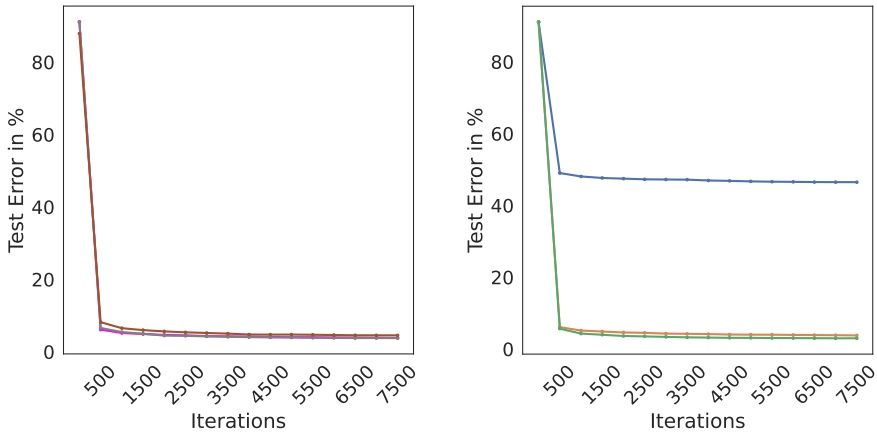
4.4 Optimization Results with the EnKF

For the same, above mentioned setup, the network reaches a high test accuracy when using the ensemble Kalman filter as an optimizer, which shown in Figure 4.8a and Figure 4.9. After the first epoch the network has already a small test error of 3.8% on the MNIST dataset.

In the following, the performance with a varying number of ensemble members and the sensitivity to different activation functions is analyzed.

4.4.1 The Effect of Different Activation Functions

Figure 4.8a depicts the classification test errors on the MNIST test dataset using different activation functions and the EnKF as an optimizer. A minor difference in the test errors is observable when ReLU (Figure 4.8a, gray line) or the logistic function (Figure 4.8a, purple line) as activation functions are applied. The test errors obtained with ReLU and the logistic function in the 500th iteration are 6.56% and 6.11% with a difference of only 0.45%. After one epoch (60000 training samples $\times$ 1/8 mini-batch samples $\times$ 8 repetitions = 7500 iterations) the error is at 3.8% with logistic function applied and at 3.75% for ReLU. The performance with Tanh as the activation function is slightly worse, with a test error of 8.15% at the 500th iteration and 4.51% after one epoch (brown line). This discrepancy may be due to the range of the Tanh function, which is in $[-1, 1]$, whereas the range of the normalized images is



(a) (b)

Figure 4.8: EnKF optimization test results with different activation values and ensemble member sizes. (a) Test error on the MNIST dataset for different activation functions within one epoch. The purple line is the logistic function, gray line is ReLU and the brown line shows the Tanh activation function. (b) Test error on the MNIST dataset for different ensemble member sizes within one epoch. Every 500th iteration is shown. The blue line depicts the error for 100 ensemble members, the orange line for 5000 members and the green line for 10000 members.

between $[0, 1]$. A similar observation is described in Y. LeCun et al. (1998). Due to the fast convergence of the EnKF , the network reaches a high accuracy already after 500 iterations. Afterwards, the improvement slows down, the accuracy between the 500th and the 7500th iteration increases approximately around 3% for all settings.

4.4.2 Varying Number of the Ensemble Members

The number of ensemble members is a hyper-parameter. The convergence behavior of the EnKF is dependent on this value. Figure 4.8b depicts the test error for a different number of ensemble members. A number of hundred ensemble members is not sufficient to optimize the network and to reach a suitable accuracy. The test error fluctuates around 47%, which is just above chance level. In contrast, with a higher number of ensemble members, e.g. 5000 members, it is possible to achieve a good performance on the test set. After one epoch the test error is at 3.8%. In comparison, a higher number of 10000 ensemble members decreases the error to 3%. This means, that a higher number does not necessarily lead to a better performance.

4.4.3 Performance on the Letters Dataset

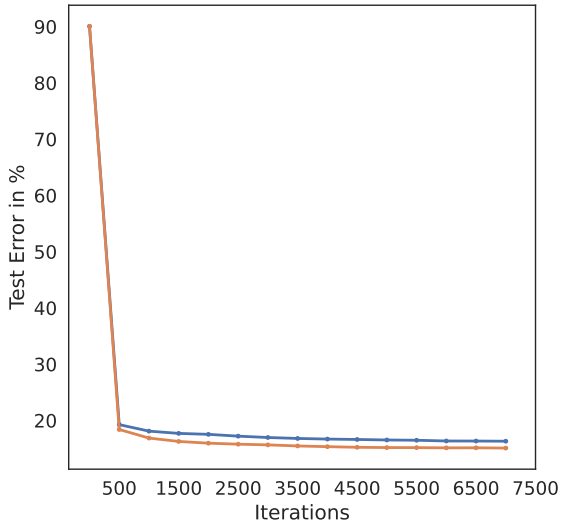


Figure 4.9: Test error on the letters dataset for different sizes of the ensemble members. The blue line depicts the error obtained with 5000 members and the orange line for 10000 members. Every 500th iteration the error is shown.

Figure 4.9 depicts the test error on the letters dataset. The letters dataset is similar to MNIST, it contains 10 types of handwritten letters, with a total set of 70000 samples. This dataset is little bit more difficult to classify than the MNIST dataset. The setup is the same as described in Section 4.2.2. Figure 4.9 shows the test errors obtained with 5000 and 10000 ensemble members. A higher number of ensemble members achieves a slightly better performance with a test error of 15.1% within one epoch. The test error for 5000 member is at 16.32%. A higher number of ensemble members enables a better performance but is not necessarily an overall solution to increase the accuracy. An experiment to test if a run with a smaller size less than 10000 members could achieve the same error rate was not conducted.

4.4.4 Adapting the Hyper-Parameters

The obtained results indicate the possibility to adapt the hyper-parameters of the EnKF. Therefore, this section describes an algorithm to decrease the number of ensemble members, repetitions and iterations. The main idea is to compare the actual test accuracy with the previous one (t_n, t_{n-1}) and adjust the value of the parameters within predefined upper and lower boundaries (b_u, b_l). A detailed description is given in Algorithm 2 and the procedure is

Algorithm 2: Algorithm adapting the hyper-parameters of the EnKF based on the test error.

Input: Lower bound b_l, upper bound b_u, error threshold ϵ, test error t_n, mini-batch size m, length of dataset D_s, update interval τ, ensemble update step υ, error difference threshold κ

Output: Number of ensemble members J, total iterations N, repetitions r

```
if n mod τ = 0 and n > 0 then
    if t_n < ε and t_n < t_{n−1} then
        if J > b_l and t_{n−1} − t_n ≤ κ then
            J = J − υ
            N = (D_s / m) · r
            r = r − 1
        end
        else if n < b_u and t_{n−1} − t_n > κ then
            J = J + υ
            N = (D_s / m) · r
            r = r + 1
        end
    end
end
```

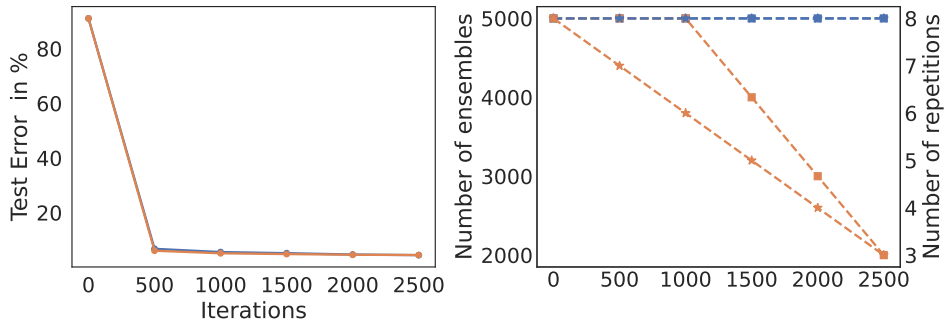


Figure 4.10: Two runs on the MNIST dataset with (adaptive, orange graphs) and without (fixed, blue graphs) adaptive changes. In the adaptive setting, every 500 iteration the values of the parameters required by the EnKF are dynamically changed depending on the test errors.

explained in the following.

Figure 4.10 depicts two EnKF optimization runs on the MNIST dataset. The orange line displays the adaptive changes and the blue line shows a run without any parameter adjustments and follows the setup described in Section 4.2 (c.f. Figure 4.8a). In the latter setting, the values of the hyper-parameters are fixed. The left plot shows the test error, the number

of ensemble members and repetitions are on the right plot. In every 500th iteration ($\tau = 500$ in Algorithm 2) the test errors determine the parameter change if $t_n < \epsilon$ with $\epsilon = 10\%$ test error. The iteration of the test error of the fixed parameter run goes only up to iteration 2500, because by then the adaptive setting has completed one epoch. The adaptive parameter adjustments (Figure 4.10 left, orange line with dots) do not increase the test error and reach the same accuracy as the fixed run (blue line with dots). Due to the fast converge of the optimizer it is possible to decrease the repetition number r by 1 in every 500th iteration (Figure 4.10 right, orange line with stars). Additionally, after some iterations and after a high accuracy is reached, more repetitions of the same mini-batch do not increase the performance any further. The number of ensemble members N decreases after 1500 iterations by $\upsilon = 1000$ (orange line with squares). This setting is rather conservative since it increases or decreases the parameters slowly. For instance, the number of ensemble members decreases only by $\upsilon = 1000$ and not, e.g., by half to ensure smaller changes and stability. The lower bound b_l is initialized with a minimal number of 1000 ensemble members and the upper bound b_u is set to a maximal number of 10000 ensemble members. Another condition to decrease the values is the difference between the previous and actual test error which has to be equal or less than $\kappa = 1$, i.e. $t_{n-1} - t_n \leq \kappa$. This setting is restrictive and causes a slow decrease regarding the number of ensemble members after 1500 iterations. In the conducted experiments, an increase in the test error does not occur, thus no parameter is increased.

4.4.5 Convergence of the EnKF

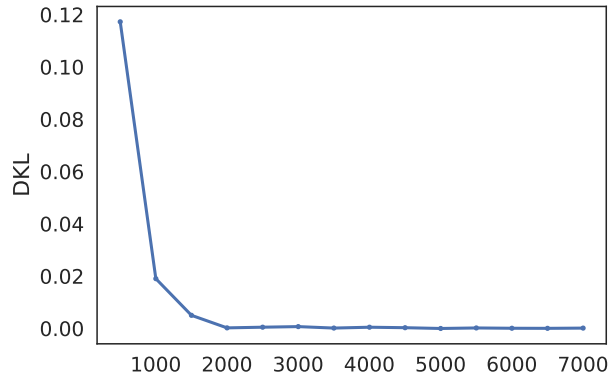


Figure 4.11: Kullback-Leibler divergence (DKL) between ensemble members. Every 500th iteration the DKL is calculated between the previous and the actual test iteration. After the 1500th iteration the DKL varies minimally. The EnKF algorithm has converged. The network is trained on the MNIST dataset.

The convergence behavior of the EnKF algorithm by means of the Kullback-Leibler divergence is analyzed in this section. The Kullback-Leibler divergence (DKL; MacKay 2003) is defined as:

$$D_{KL} = (P||Q) = \sum_x P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{4.4}$$

for two probability distribution $P(x)$ and $Q(x)$ over the probability space $\mathcal{X}$.

Figure 4.11 shows the Kullback-Leibler divergence applied on the network connection weights, i.e. the ensemble members. Before the DKL between the ensemble members of the previous ($P(x)$) and actual test ($Q(x)$) iteration is computed (e.g. between the 500 and 1000th iteration), the distribution over the members is calculated. Additionally, the ensemble members are normalized, so that the sum of the distribution is equal to one. As visible in Figure 4.11, after the 500th step the DKL varies only a little bit. The little fluctuations between the iterations around 0, especially after the 1500th iteration, show that the EnKF optimization has converged, which aligns with the slowly evolving network performance after 2000 iterations (compare with Figure 4.8a). This can be also observed in the test errors of Figure 4.8a: Between the 500th and the 7500th iteration the accuracy improves slightly, approximately 3%. It is possible to soften the strong convergence in order to increase the performance by adding more variance to the ensemble members in the update step of the EnKF as suggested by Kovachki et al. (2018), e.g. by adding Gaussian noise.

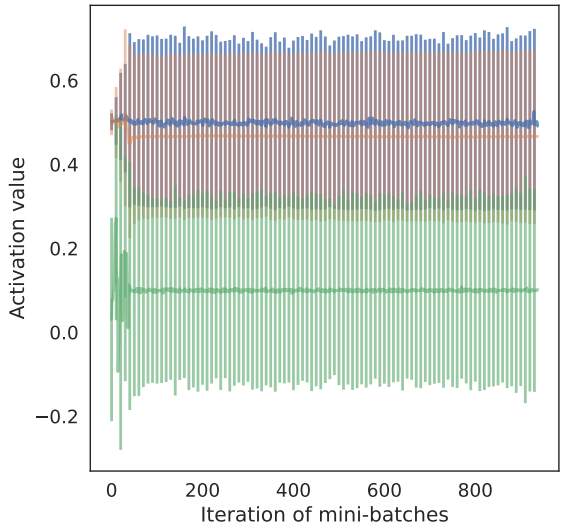


Figure 4.12: Mean (dark line) and standard deviation (vertical bar) of the activation values for the three hidden layers of the network during training on the MNIST dataset. The optimizer is the **EnKF**.

A problem described in Section 4.3.1 and Section 4.3.2 are the non evolving gradients or activation function values, which hindered the network to learn. In a similar way the mean and standard deviation of the activation values during training while being optimized by the EnKF are investigated. As seen in Figure 4.12 the means and standard deviations of all layers do vary. For layer one, the mean oscillates around 0.54 with a maximal standard deviation of 0.2. Layer two shows less variance in its mean of 0.53 after the first iterations with a standard deviation of 0.19. A similar situation can be observed for layer three, after the first iteration the mean varies around 0.1 and a maximal standard deviation of 0.4. The overall behavior corresponds to the slow changing distribution as described in the previous Section 4.4.5, which is due to the fast convergence of the EnKF method.

4.4.6 Benchmarking the EnKF

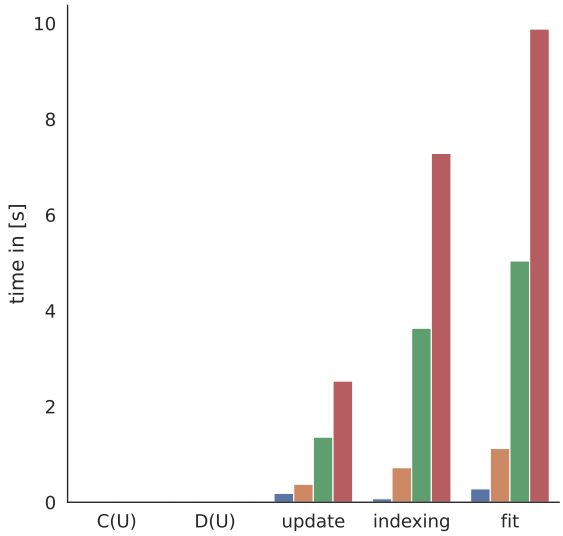


Figure 4.13: Benchmark results of the EnKF fit function. The colors indicate the number of ensemble members, with blue for 100, orange for 1000, green for 5000 and red for 10000 ensemble members. The run time for different operations is depicted in seconds.

Although the ensemble Kalman filter converges fast and is a suitable optimizer for network training, the compute time performance is inadequate and needs to be improved for big datasets with millions of samples. One epoch on the MNIST dataset with the aforementioned setup takes up to 11 hours on a NVIDIA Tesla K20 graphics card. To find out which component causes a high compute load over time, individual functions of the EnKF method

are benchmarked. Figure 4.13 depicts the calculation time for the two covariance matrices (Equation 3.20), the update step (Equation 3.19), indexing samples from the model output and the whole function (`fit()`), which includes every operation such as tensor assignments, copy processes and setting the correct dimensions. The setup configuration is the same as presented in section 4.2 and the benchmark is run for one iteration on the MINST dataset with a mini-batch size of 64. The colors in Figure 4.13 indicate the number of ensemble members, with blue for 100, orange 1000, green 5000 and red 10000 ensemble members. The copy and set operation times are neglectable and do not last longer than a few milliseconds, for instance for 5000 members cloning takes 0.01516s, setting dimensions $9.775e^{-6}$s and converting to PyTorch tensors 0.00796s. Here, the time for every operation is summed up and obtained with the Python profiling tool *cProfile*[1]. Similarly, the time for calculating the covariance matrices $\mathbf{C}(\mathbf{U})$ and $\mathbf{D}(\mathbf{U})$ is negligible with 0.01094s and 0.00917s for 5000 ensemble members. Surprisingly, in comparison to the other operations, the indexing takes the most amount of time, with 3.63357s for 5000 members and close to 10s for 10000 ensemble members. This may be due to the structure of the object which holds the output. The object is a 3D PyTorch array with the size of $\mathbb{R}^{J\times K\times b}$, i.e. the number of ensemble members $\times$ model output $\times$ mini-batches. The main time consuming component in the update step is obtaining the inverse, $\left(\mathbf{D}(\mathbf{U}) + \Gamma^{-1}\right)^{-1}$, with 1.3593s for 5000 ensemble members.

4.5 Discussion

The performance of networks is dependent on the weight initialization and the selection of activation functions. Improperly selected, the vanishing gradient problem occurs, especially in deep neural network trained with gradient descent and backpropagation, as presented in Section 4.3.

For different parameter settings, gradients and activation function values are investigated by analyzing their distributions, mean and standard deviation per layer over iterations and over epochs. The optimizers are SGD and Adam, the weights are sampled from a random weight distribution and the activation function is the logistic function. With this setting, the CNN could not or only slowly learn to classify the MNIST dataset. In contrast to SGD, Adam is more robust and shows a better performance for different parameter configurations.

As an alternative optimizer the ensemble Kalman filter is analyzed. The EnKF does not calculate gradients and requires only the feed-forward step of the network. Given the same settings, the EnKF provides an optimization solution to train the network on the MNIST and letters dataset. The network performance is able to reach a high accuracy above 96% and

[1] https://docs.python.org/3/library/profile.html

85% for the MNIST and letters dataset. Similarly to the gradient based optimization, the activation values of different network layers are investigated by analyzing the distribution, mean and standard deviations over iterations within one epoch. Furthermore, the results show that a larger ensemble member size does not necessarily improve the performance as initially expected (Section 4.4.2).

Due to the fast convergence of the EnKF, the network can classify the MNIST dataset after 500 iterations. However, depending on the optimization problem, the fast convergence can cause the EnKF to be stuck in a local minimum and hinder exploration of other possible optima. Adding noise to the ensemble members in every update step can counteract this issue (Kovachki et al. 2018). Another approach is to dynamically adjust the scaling (γ) of the identity matrix $\mathbf{\Gamma}$, either by a simple algorithm as presented in Section 4.4.4 or with a more intelligent approach such as learning-to-learn (see Chapter 5).

One bottleneck is the compute time performance within the update step of the EnKF. The EnKF is easy to parallelize, since the update step can be calculated for each ensemble member, which is only dependent on the previous iteration. A limiting factor seems to be the calculation of the covariance matrices, but the benchmarks in Section 4.4.6 show that the matrices can be quickly computed. It is open, whether parallelizing the covariance matrices enhances the compute speed. Interestingly, the bottleneck is due to the indexing operation to acquire the model output. A simplified structure, which decreases the dimensions of the data object, may already help to reduce the indexing time.

The EnKF approach provides an alternative solution to optimize neural networks and can overcome the issues introduced by different activation functions and initialization settings. It provides a basis to further investigate gradient free methods to train neural networks. The set of analyzed parameters is chosen rather small to control the optimization process and to strengthen the conclusions drawn from the experiments. However, the EnKF can be advantageous when exploring different settings and is an unique option to overcome specific problems introduced by optimization methods based on gradients.

5 Learning to Learn

Successfully training an algorithm requires not only a suitable optimizer, but the optimizer's hyper-parameters need to be selected in an appropriate fashion, too. Hyper-parameters influence the learning process of the training. Finding and setting them in a manual fashion can be a tedious and error prone process. As aforementioned in Section 4.5, a smart approach to automatically tune the optimizer's hyper-parameters is desirable. Fortunately, methods such as learning to learn exist to automate the learning and optimization of parameters and hyper-parameters. Learning to learn or meta-learning is a technique to learn via experience (Thrun et al. 2012). The learning performance of an algorithm improves by generalizing over a family of tasks. A task specific function evaluates the performance of the algorithm. Learning to learn can be decomposed into a two-loop structure. The inner loop consists of an optimizee (e.g. an artificial network or a simulation of a biological neuronal network) with learning capabilities trained on specific tasks. The outer loop is an optimizer algorithm which optimizes parameters that improve the optimizee's learning performance over the iterations. In this work, the aim is to improve the overall performance of the algorithm by optimizing the learning parameters, i.e. the parameters and hyper-parameters of the algorithm. The concept of learning to learn is inspired by biological evolution, thus, the performance measure is called fitness and the iterations are called generations.

First, I will embed learning to learn into its historical context and describe the state of the art work (Section 5.1). Then, I will continue with the basic concepts and theoretical formulation of learning to learn (Section 5.2), followed by a technical description of the L2L framework, which implements the concepts of meta-learning (Section 5.3). I will describe a use case, in which the parameters of a spiking reservoir network are optimized by the L2L framework, in particular utilizing the EnKF optimizer. Additionally, I will analyze the parameters of the evolved network (Section 5.5). Finally, I explore hyper-parameters of the EnKF, while it optimizes the connection weights of the reservoir network. In this setting, the EnKF is run within the inner loop and its hyper-parameters are adapted in the outer loop (Section 5.6). This approach will highlight the advantages of an automated hyper-parameter exploration to find a suitable optimizer configuration, which is comparable to the manually tuned EnKF parameter setup for the reservoir network.

5.1 From the Historical Context to the State of the Art

Already in 1976, Rice (1976) mentioned the problem of effectively choosing an algorithm for a given problem. He explored if approximation theory could be applied to select algorithms. This approach is a meta-learning approach that tests out different methods to solve the problem tasks. Rendell et al. (1987) introduced the Variable Bias Management System (VBMS), which is able learn about the relationship between induction problems and biases, i.e. the systems understands the problem and how to solve it by using meta-learning. Schmidhuber (1987) first applied meta-learning on neural networks. He described a set of methods with the learning to learn idea by using self-referential learning. In this case, artificial neural networks are able to learn their own weights and predict them. Schmidhuber used evolutionary algorithms to optimize the networks. Thrun et al. (1998) formalized the term learning to learn in a theoretical way and demonstrated different practical implementations of learning to learn. Yoshua Bengio, S. Bengio, et al. (1990) and S. Bengio et al. (1995) propose to learn new, synaptic rules for ANNs using a learning to learn approach. They consider the rule as a parametric function and learn the parameters of this function. Early work on learning to learn focused on using the computational power of recurrent networks to learn dynamics of the inner loop network (Hochreiter, Younger, et al. 2001). The weights of the recurrent networks were treated as the hyper-parameters and trained/learned in the outer loop, whilst being kept fixed in the inner loop. This allowed the dynamics of the recurrent network to perform the learning. The supervised learning paradigm of Hochreiter, Younger, et al. (2001) was later extended to reinforcement learning (Y. Duan et al. 2016; Wang et al. 2016).

Recent work focuses on optimizing gradients of the inner loop network with networks which learn the gradients or apply additional optimization techniques in the outer loop. For example, Andrychowicz et al. (2016) is using an LSTM (long short term memory network) to optimize the top-level gradients and update the weights of the network inside the inner loop. They utilize the outer loop network to replace the gradient descent optimizer of the inner loop. The weights in the inner loop network are fixed and considered as hyper-parameters and are learned in the outer loop. Inspired by the work of Andrychowicz et al. (2016), Ravi et al. (2017) incorporates the test error into the optimization step. This leads to fewer unrollings of the LSTMs and the optimization process converged faster, thus, reducing the computational cost.

Finn, Abbeel, et al. (2017) introduce the Model Agnostic Meta-Learning (MAML) framework for feed-forward models. MAML learns initial parameters of the inner loop network. They utilize gradient descent in the outer loop and are able to train a network to generalize well on the validation set. The method is agnostic to the model in the inner loop, and thus, can be applied to a variety of different tasks. Finn and Levine (2017) show that sophisticated

optimization methods such as recurrent networks in the outer loop can be replaced by learning the initialization parameters and updating them as hyper-parameters with gradient descent. Several works extend the MAML framework in order to enhance the performance of the learning and the computation time (Finn, Xu, et al. 2018; Finn, Rajeswaran, et al. 2019).

Population based algorithms for the outer loop optimization are a popular choice as well. For example, Cao et al. (2019) propose to use particle swarm optimization (Kennedy et al. 1995) to train a meta-optimizer that continuously learns point-based and population-based optimizers. They employ LSTMs to train and learn the fitness function for a sample population. Their learning incorporates two attention mechanisms, the feature-level ("intra-particle") and sample-level ("inter-particle") attentions. The intra-particle module is utilizing the hidden state of the corresponding LSTM to weight the features and the inter-particle attention module learns to update the particle information based on the previous particles.

In a similar manner Jaderberg et al. (2017) employ a parallelized, population based approach. They use a random search optimizer to update the hyper-parameters of neural networks. First, they randomly sample the network parameter and hyper-parameters in the initialization phase. Then, they execute the optimization procedure in parallel and asynchronously evaluate every training run. An underperforming network is exchanged by a more successful network. To expand the search space, Jaderberg et al. (2017) additionally perturb the hyper-parameters of the replacing network.

Learning to learn can be extended to search for the architecture of neural networks, which is known as neural architecture search (Zoph and Q. V. Le 2016). In combination with evolutionary algorithms, neural architecture search has been shown to be very powerful in finding network architectures for different tasks (Stanley and Miikkulainen 2002).

Bergstra et al. (2012) showed that random search is surprisingly effective for hyper-parameter searches for a wide variety of tasks. Automated hyper-parameter search is a part of Automated Machine Learning or in short AutoML (Hutter et al. 2019; X. He et al. 2021).

5.2 Basic concepts of Learning to Learn

Learning to learn or meta-learning is a technique to learn and generalize from experience (Thrun et al. 2012). The learning to learn process is divided into two loops, the inner and outer loop as depicted in Figure 5.1. The inner loop consists of an algorithm with learning capabilities (e.g. an artificial or spiking neural network), the algorithm runs a specific task T from a family $\mathcal{F}$ of tasks. Tasks can range from classification, e.g. the MNIST dataset, or training agents to solve optimization problems in an autonomous fashion.

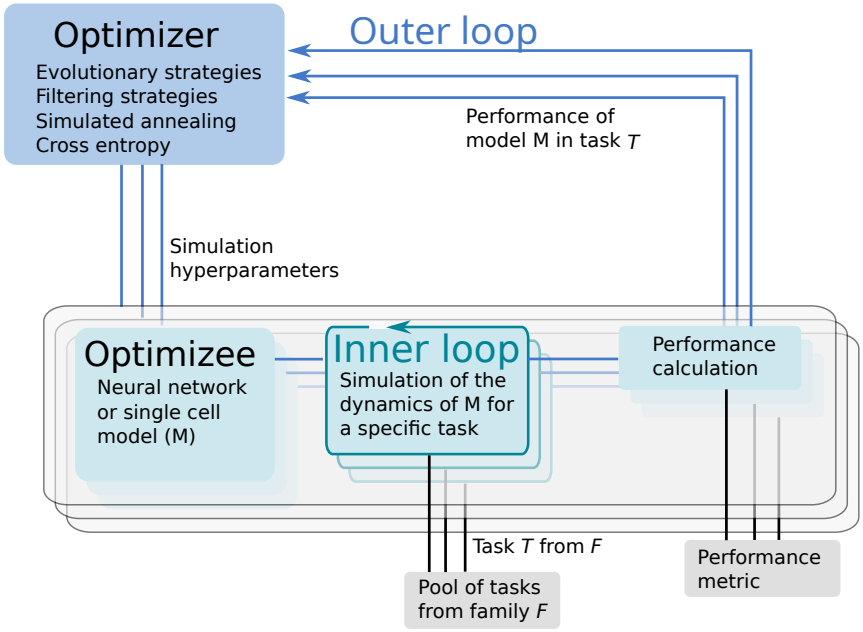


Figure 5.1: The two loop, parallel structure of L2L. In the inner loop an algorithm is learning a task from a family of tasks. A fitness function evaluates the performance of the algorithm in every generation. The (hyper-) parameters together with the fitness are sent to outer loop where they are optimized.

A fitness function evaluates the performance of the algorithm. This function is specifically designed and calculates a fitness value f or a fitness vector $\mathbf{f}$. In general, the function is adjusted for every model and dependent on the task.

The outer loop receives parameters and hyper-parameters, together with the fitness value of the optimizee. In order to improve the optimizee's performance, different optimization algorithms based on meta-heuristics, such as evolutionary algorithms, filtering methods or gradient descent optimize the hyper-parameters. After the optimization step, the hyper-parameters are fed back to the inner loop algorithm and a new generation iteration (i.e. the inner loop) starts. From a technical point of view, the optimizee orchestrates the inner loop. Each optimizee starts a simulation process. After the optimization phase, it accepts optimized (hyper-)parameters coming from the outer loop and executes the inner loop process to run the simulation. Lastly, it calculates the fitness and sends everything together to the outer loop optimizer.

L2L's terminology is inspired by evolutionary algorithms, thus, the parameter set which is

optimized, is called an individual and the algorithm performance is the fitness.

L2L follows the hyper-parameter optimization process and the fitness f for a loss or fitness function $\mathcal{L}$ is theoretically formulated as (Brazdil et al. 2022):

$$f = \mathcal{L}(A(M(\alpha, \theta, d_{train})X_{test}), y_{test}) \,, \tag{5.1}$$

where $M(\alpha, \theta, d_{train})$ is a trained model M, α is the algorithm with hyper-parameters θ trained on a dataset d_{train}. $A(M(\alpha, \theta, d_{train})X_{test})$ is the model output on a test sample coming from a test set. This is a partition of the dataset which is separated before the training and is used in order to evaluate the model performance on unseen data samples. To obtain the loss $\mathcal{L}$, the model output on the test set is compared against the target y_{test}. However, this is task specific and often labels are not available, thus y_{test} is omitted. In this formulation the algorithm α is interchangeable and a parameter itself. A shorter form for the loss with a fixed algorithm, which includes only the input parameters, is $\mathcal{L}(\alpha, \theta, d_{train}, d_{test})$. The outer loop has to optimize the hyper-parameters α and θ with respect to a set of algorithms A and possible parameter configurations Θ:

$$(\alpha^*, \theta^*) = \underset{\theta \in \Theta,\, \alpha \in A}{\operatorname{argmin}} \mathcal{L}(\alpha, \theta, d_{train}, d_{test}) \,. \tag{5.2}$$

The model performance, i.e. the loss $\mathcal{L}$, is evaluated on the test set d_{test}. In Equation 5.2 the algorithm selection can be incorporated into the parameter configuration, this shortens the term (α^*, θ^*) to (θ^*). The evaluation of θ over many generations is represented as a trajectory $\mathcal{T}$, which keeps track of the optimized parameters and fitnesses over the generations n and per individual i:

$$\mathcal{T}_{\theta,L} \equiv (\theta_i L_i)^n \,. \tag{5.3}$$

5.3 Technical description of L2L

In L2L the optimizers are based on population based methods which enables simulations to be run embarrassingly parallel, i.e. each individual is initialized independently. It is possible to easily distribute every individual via the message passing interface (MPI) on several computing nodes and thus exploit high-performance computing systems (HPC). Additionally, the framework supports multi-threading on single nodes or local/non-HPC machines. The hardware resources can be set in the initialization phase of L2L and the framework will automatically take care over the resource distributions and collection of results.

5.3.1 Executing an L2L run

```
1  from l2l.utils.experiment import Experiment
2  from l2l.optimizees.optimizee import Optimizee, OptimizeeParameters
3  from l2l.optimizers.optimizer import Optimizer, OptimizerParameters
4
5  experiment = Experiment(root_dir_path='/home/user/L2L/results')
6  jube_params = {"exec": "srun -n 1 -c 8 --exact python"}
7  traj, all_jube_params = experiment.prepare_experiment(name='L2L-Run',
8                                                        log_stdout=True,
9                                                        jube_parameter=jube_params)
10
11 ## Inner loop simulator
12 # Optimizee class
13 optimizee = Optimizee(traj)
14 optimizee_parameters = OptimizeeParameters()
15
16 ## Outer loop optimizer initialization
17 optimizer_parameters = OptimizerParameters()
18 optimizer = Optimizer(traj,
19                       optimizee_prepare=optimizee.create_individual,
20                       fitness_weights=(1.0,),
21                       optimizee_bounding_func=optimizee.bounding_func,
22                       parameters=optimizer_parameters)
23
24 experiment.run_experiment(optimizee=optimizee,
25                           optimizee_parameters=optimizee_parameters,
26                           optimizer=optimizer,
27                           optimizer_parameters=optimizer_parameters)
28 experiment.end_experiment(optimizer)
```

Listing 1: Template script to execute an L2L run. The experiment class manages the run. The optimizer and optimizee are initialized here.

To create the optimizee and start the L2L run, the user needs to work on two files. The first file, the **run script**, is the entry script to invoke the L2L run. The **optimizee** is the second file and handles the inner loop simulation.

In the run script the user sets the optimizee and optimizer, their configurations and configures the hardware settings, e.g. the user can define if the simulation should run on an HPC or a local computer. Listing 1 shows an exemplary code template to invoke an L2L run. Lines 1–3 import the necessary Python modules, which are the experiment, optimizee and the optimizer. For simplicity the names of the modules in the template are called Optimizee and Optimizer, in a real run they have to be adapted to their corresponding class names. The **experiment** class steers the run. In line 5 the class experiment is initialzed and the result path is set inside the constructor. All outputs produced by the L2L run are stored in this folder. `prepare_experiment` in line 7 is a method to prepare the run. The user sets the name

of the run, enables logging and configures further parameters for the Juelich Benchmarking Environment (JUBE; Speck et al. 2020). The version of JUBE in L2L is stripped down to submit and manage parallel jobs on HPCs, it offer an interface to the job management system SLURM (Yoo et al. 2003). The `prepare_experiment` method returns the trajectory object, which in the template is named as `traj`. This object is a container which holds the history of the parameter space exploration, the parameters to be explored and the optimization results of every generation. The optimizee and optimizer have read and write access to the trajectory, thus the trajectory acts as an interface between both classes to exchange the parameters and the fitnesses. The trajectory is modelled after PyPet's trajectory[1]. In line 6 the directives for the HPC jobs are displayed. `exec` is the command to invoke a run, followed by a `srun` directive for SLURM. In the template, one task (`-n 1`) will be executed on 8 cores (`-c 8`). The `python` command indicates that the optimizee and optimizer are Python executables. On local machines the run can be executed without any parameters by invoking the command `{'exec':'python'}`. Internally, JUBE creates a job script and passes it to SLURM, which then invokes the optimizees and the optimizer. It is possible to execute the script either as a batch script or as an interactive job on an HPC.

The **optimizee** is initialized in line 13, it requires the trajectory `traj`. `OptimizeeParameters` is a `namedtuple` object, which is a Python tuple with a keyword and a corresponding item, similar to a Python dictionary. It accepts the parameters for the optimizee. For the optimizee, the `namedtuple` appears as a parameter object and can be accessed using the parameter's name, i.e. as `parameters.name`.

The optimizee requires at least two functions to be implemented:

1. The function `create_individual()` defines the individual. Here, the parameters which are going to be explored are initialized and returned as a Python dictionary.

2. `simulate()` invokes the simulation run. The L2L framework and especially the optimization process is agnostic to the application carrying out the simulation. It only requires that a fitness value or fitness vector is returned.

3. `bounding_func()` is an optional function in which parameters before and after the optimization are restricted to defined ranges. For example, in a spiking neural network delays have to be strictly positive and greater than zero. The function is applied only on parameters which are defined in `create_individual()`.

The **optimizer** is created in line 18. It requires the optimizer parameters (line 17), the method `optimizee.create_individual` and optionally the bounding function `optimizee.bounding_func`.

[1] https://github.com/SmokinCaterpillar/pypet

Additionally, a tuple of weights (`fitness_weights`, in the template $(1.0,)$) can be specified, which weights the optimizee's fitness by multiplying those values with the fitness itself. For example, in case of a two dimensional fitness vector, a tuple of $(1.0, 0.5)$ would weight the first fitness fully and the second one only half. The L2L framework offers different optimization techniques, such as cross-entropy, genetic algorithm (GA), evolutionary strategies (Salimans et al. 2017), gradient descent, grid-search, ensemble Kalman filtering, natural evolution strategies (Wierstra et al. 2014), parallel tempering and simulated annealing. The results of the optimization runs and the trajectory are saved in a user specified folder as Python binary files. Additionally, the user can store the results from within the optimizee in any format and location they wish.

The method `run_experiment` (line 24) requires that the optimizee, the optimizer and their parameters are initialized. Finally, the `end_experiment` method is responsible to end the simulation and to stop any logging processes.

5.4 Optimizing with L2L

This section presents an L2L optimization run to classify digits using a spiking neural network, in particular a reservoir network (see Section 2.3.2). A fitness function evaluates the performance of the classification and the EnKF optimizes the connection weights. After multiple generations the optimized network exhibits a high accuracy.

5.4.1 MNIST Classification with a Liquid State Machine

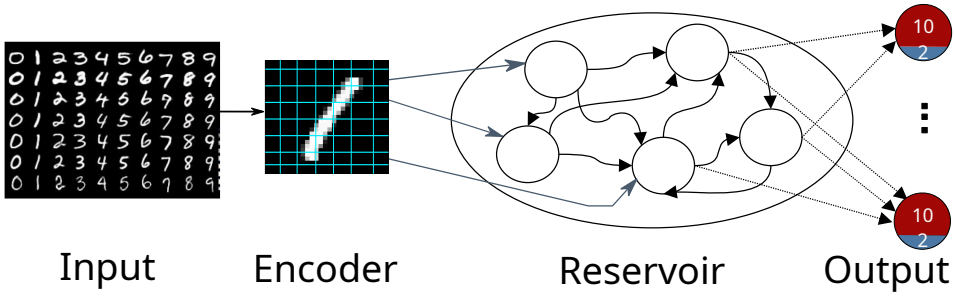


Figure 5.2: An illustration of a reservoir network classifying the MNIST dataset. The input is encoded into firing rates and fed to the reservoir. The output consists of a cluster of 10 excitatory neurons colored in red and 2 inhibitory neurons colored in blue. The output with the highest activity indicates the digit presented in the input.

The reservoir is a liquid state machine implemented in the NEST simulator. As shown in Figure 5.2 the network consists of an input encoding layer, a recurrent reservoir and an output layer. The weights between the reservoir and the output layer are optimized to increase the classification accuracy.

The network consists of three populations of leaky integrate-and-fire (LIF) neurons (Section 2.2.2), the encoder, the reservoir and the output. The input to the network are MNIST digits, encoded into firing rates. Rate coding is a popular choice to encode image pixels to spikes (Guo et al. 2021). This scheme encodes each pixel into a firing rate, which can be utilized as the firing rate for a poisson generator to create spike trains. The firing rate coding is the frequency with which a neuron fires proportional to the gray value intensity of the pixel:

$$\nu_{m,n} = I(m,n)\frac{\nu_{\max} - \nu_{\min}}{p_{\max}} + v_{\min} \tag{5.4}$$

where $\nu_{m,n}$ is the mean firing rate for the poisson generator in NEST for a pixel (m,n) of the image I, with $p_{\max}$ the highest pixel intensity value and $\nu_{\max}$ and $\nu_{\min}$ are the maximum and minimum defined rates. The pixel intensities range from 0 to 255 and are mapped between $[1, 100]$ Hz. The input pixels are connected to a total of 784 excitatory neurons in an one-to-one connection.

The reservoir has 1600 excitatory and 400 inhibitory neurons, while the output has a population of 12 neurons (see Figure 5.2, 10 excitatory (red), 2 inhibitory (blue)) per digit. The connections in the reservoir are random. They are limited to a maximal outdegree of 6% and 8% for each excitatory and inhibitory neuron. In this setting, three digits of the dataset (0 to 2) are classified, thus there are three output clusters. Each excitatory neuron receives a maximal indegree of 640 connections and each inhibitory neuron receives an indegree of maximal 460 connections from the reservoir, thus resulting in 28800 $(= 800 \times 12 \times 3)$ connections in total. The neurons within the output cluster are recurrently connected and have no connections to neurons in the other clusters. The network exhibits low spiking activity in all three parts if no input is presented. The network is constructed in the optimizee's `create_individual` function. The connection weights are sampled from a normal distribution with $\mu = 70$ and $\sigma = 50$ for the excitatory neurons and $\mu = -90$ and $\sigma = 50$ for the inhibitory neurons.

When simulating the network (`simulate` function), a small batch of 12 different numbers are presented to network for 500 ms per image as spike trains. Furthermore, each neuron receives background Poissonian noise with a mean firing rate of ≈ 5 spikes/s in order to maintain a low activity within the reservoir.

A warming up simulation phase lasting for 100 ms is run before any image is presented. This

decays all neuron parameters to their resting values. Similarly, between every image there is a cooling period of 200 ms where no input is shown. After the simulation, the output with the highest spike activity indicates the presented digit.

Over the generations, the value of the optimized weights may rise. To counteract this, the weights are clipped in the `bounding_func` function if they are greater than or less than 1000 or -1000. Additionally, in the same fashion as creating new individuals, random values are sampled from a Gaussian distribution with $\mu = 0$ and $\sigma = 200$ and added to the clipped weights, so that the optimization process does not prematurely converge. This helps the network to have a high classification performance in later generations. Otherwise the high weights would result in an unstable dynamical regime (e.g. too much stimulation received on the outputs) and thus degrade the performance.

5.4.2 Fitness function of the Reservoir Network

In the output the firing rates of all clusters are acquired and then the softmax function is applied:

$$\sigma(\mathbf{x})_j = \frac{e^{x_j}}{\sum_k e^{x_k}},$$

where $\sigma : \mathbb{R}^k \to [0,1]^k$ and $\mathbf{x} = (x_0, x_1, \dots x_k) \in \mathbb{R}^k$, $j = 1, \dots, k$ is the vector of firing rates. The highest value indicates the classified digit. Since every image in the dataset has a label the mean squared error or loss $\mathcal{L}$ between the prediction and the corresponding label can be calculated:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{5.5}$$

with y_i the label and $\hat{y}_i$ the predicted output encoded as one-hot vectors with a non-zero entry corresponding to the position of the label. The optimizer used in the outer loop is the ensemble Kalman filter, which optimizes the weights to minimize the distance between the model output and the training label. Thus, the fitness function is defined as $f = 1 - \mathcal{L}$ and is used to rank individuals (see next Section). After the presentation of the image batch, the optimizer receives the fitness and the softmax model output for each input.

5.4.3 Optimizing the Reservoir with the Ensemble Kalman Filter

The ensemble Kalman filter is the optimizer in the outer loop to update the weights between the reservoir and the output, this is similar to the optimization procedure described in Section 4.2. Before the optimization, the weights are normalized to be in the range of $[0, 1]$. Additionally, the weights from the reservoir to the output are concatenated to construct one

individual. In total, 98 individuals are optimized and each individual has 28800 weights. In terms of the EnKF setting, the ensemble members are the network weights, the observations are the labels and the the model output is the result of the softmax operation. In Section 4.4.2 it was shown with around 100 ensembles it is possible to reach at least chance level on the MNIST dataset. However, the experiments were conducted using convolutional neural networks and tested with harsh conditions such as poor weight initialization as well as different activation functions. The simulation time takes a long time to finish, since in every generation the network has to be reconstructed, thus the number of ensemble members is limited in this case.

The EnKF optimization is slightly modified to replace poorly performing individuals with the best individuals. The individuals are ranked according to their fitness and the worst n individuals are replaced with m best ones. Furthermore, random noise drawn from a normal distribution is added to the replacing individuals, this increases the search space for the parameters to find different and possibly better solutions. n and m are set to be 10% of the corresponding individuals. One hyper-parameter of the EnKF is γ (here set to $\gamma = 0.5$), it has a similar effect as the learning rate in stochastic gradient descent. A lower γ may lead to a faster convergence but also has the risk of overshooting minima which results in longer training runs. In contrast a higher γ is slower to converge, or can get trapped in minima and thus hindering the exploration of further parameter spaces. In this setting, the EnKF with the added extensions is a suitable optimizer, because it is able to quickly converge to minima and provide satisfactory results. Unfortunately, it is not possible to train the whole dataset, since the simulations take a long time to finish.

5.4.4 Classification Performance of the Reservoir

Figure 5.3 depicts the performance of the reservoir up to 400 generations. The fitness is acquired over a subset of the MNIST test set in every tenth generation. Before the training, the test set of 10000 images is separated from the training set of 60000 images in order to contain digits which are not going to be presented during the training phase.

While the mean fitness steadily increases over the generations, the best individual fitness exceeds 0.8 at generation 50 and improves to a fitness close to 1.0 at the end of the generations. Towards the end of training, the mean fitness increases, however, the standard deviation also increases slightly. The highest standard deviation of 0.12 occurs in generation 37, it reaches a minimum value of 0.04 at generation 180 and remains around 0.09 at the end of the training. It is important to note that the green curve depicts the performance of the best, i.e. highest performing, individual in each generation, which is not necessarily always the same individual.

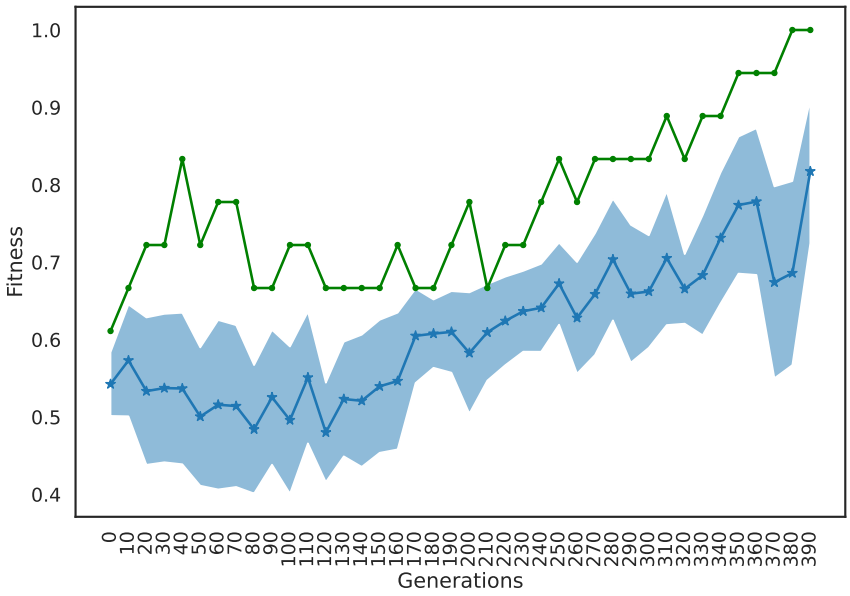


Figure 5.3: Every tenth generation the reservoir is evaluated on a subset of the MNIST test data. The L2L simulations run over 390 generations. The blue star-dotted line depicts the mean fitness, while the shaded area is the standard deviation of all individuals. The green line shows the best fitness in every tenth generation.

In every training and testing phase 12 images are presented for 500 ms on each generation, resulting in long simulation times. Thus, the whole dataset cannot be processed and the total number of used images is limited to 4800 (4320 training, 480 testing). To process an entire generation of 98 individuals, including the optimization phase, the run on an HPC[2] takes less than 3 minutes. In comparison, a grid search on 28000 parameters exploring a range of 20 values for each weight would require the evaluation of 20^{28000} combinations.

The fast convergence behaviour of the EnKF makes it possible to reach an optimal solution within a few generations. The added modifications to sample new individuals from well performing ones and perturbing them enables the optimization to find better solutions by exploring other parameter ranges. Section 5.6 describes the optimization process when the EnKF is moved into the inner loop to update the weights, while the hyper-parameters of the optimizer are adapted by a genetic algorithm in the outer loop.

[2]The simulations are run on the JUSUF HPC, with 2 × AMD EPYC 7742 CPUs, 2 × 64 cores, 2.25 GHz, for HPC details see https://apps.fz-juelich.de/jsc/hps/jusuf/index.html

5.5 Analysis of the Parameters

In the following I will analyze parameters of the evolved reservoir previously described in Section 5.4. This will help to understand how the optimization process evolved the network and can provide insight for future improvements regarding the presented methods, in order to employ them on SNNs in a more efficient manner. In particular, I will focus on the connection weights and the corresponding covariance matrix values of the EnKF obtained before every 10th generation, i.e. the iteration in which the test set is evaluated and none of the parameters are updated. Thus, there are 40 parameter sets of weights and covariance matrices, out of 400 generations.

5.5.1 Weights Analysis

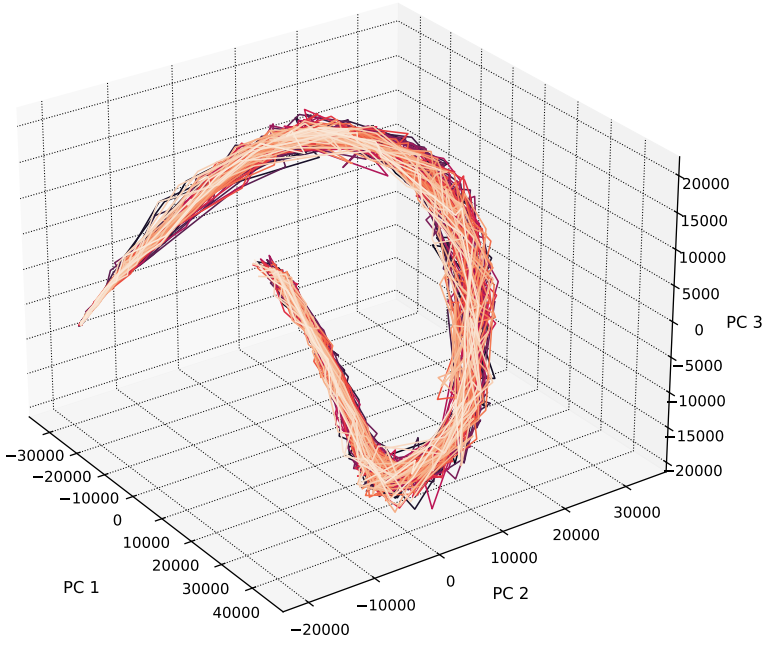


Figure 5.4: PCA of all individual weights of every 10th generation. Each color corresponds to one individual. All weights lie on the same manifold. This is due to the EnKF optimization which quickly converges the weights to an optima.

As an initial analysis and to see if patterns within the weights are emerging, principal com-

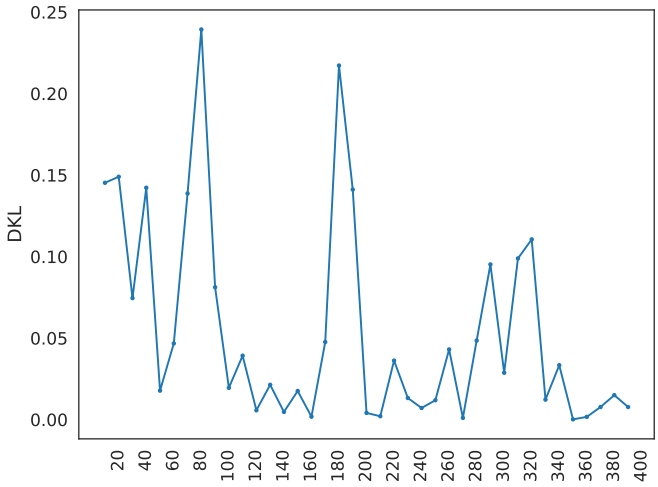


Figure 5.5: DKL between the connection weights over the generations. In later generations the DKL does not vary much due to the convergence of the optimization.

ponent analysis (PCA) is applied. The function is from the Python library scikit-learn [3] (Pedregosa et al. 2011). The PCA function in scikit-learn first learns the model parameters and in a second step transforms and reduces the dimensionality of the data, the weights in this case. The weights are organized in an array of a shape of $40 \times 98 \times 28800$ (every 10th generation $\times$ individuals $\times$ weights), which is reduced to $40 \times 98 \times 3$ by selecting 3 principal components. The result is depicted in Figure 5.4. Every color displays all weights of one individual lying on an arch-shaped manifold in a three dimensional space. If only early generations are plotted (e.g. up to 200th generation) the shape is not as smooth as in Figure 5.4 and much more scattered in space. The smooth form is due to the quick convergence behaviour of the EnKF.

In later generations the weights do not vary much, as shown in Figure 5.5 which depicts the Kullback-Leibler divergence (DKL) between the connection weights. The DKL is obtained as described in Section 4.4.5. Up to generation 180 the divergence is high, especially between generations 60 and 100 as well as 160 and 200. Between generation 280 and 340 slight increases are visible, but the DKL stays low afterwards. In the fitness plot (Figure 5.3) the performance changes in those ranges as well. In contrast to earlier generations, the performance drop which occurs, for example, between generation 360 and 370 (see Figure 5.3) is less captured by the DKL.

[3] Version 1.0.2

5.5.2 Covariance Matrix Analysis

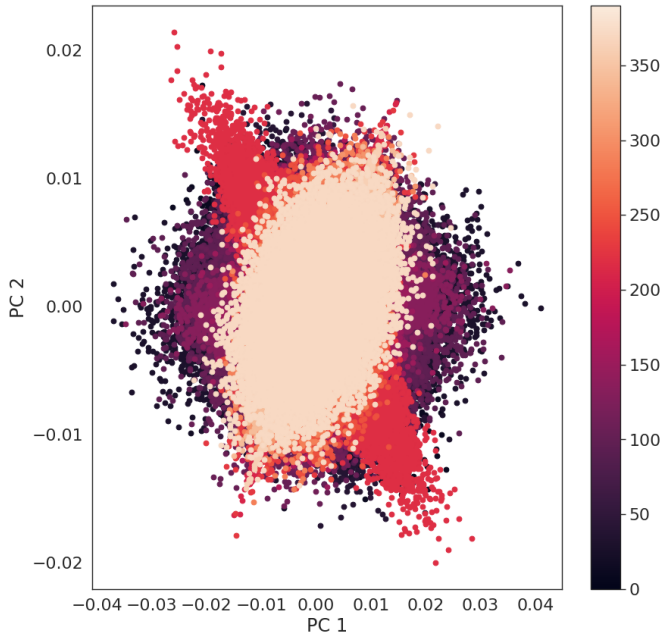


Figure 5.6: The covariance matrix $\mathbf{C}(\mathbf{U})$ values over the generations are depicted in the PC space. Bright colors indicate later generations. While in earlier generations (black and dark purple colors) the high variance of the values is visibly reflected as a spread over the parameter space, in later generations (red and yellow colors) the value spread is less and more contracted.

Besides the weight updates, the EnKF has two covariance matrices $\mathbf{C}(\mathbf{U})$ and $\mathbf{D}(\mathbf{U})$ (see Section 3.3.3 for details), which are interesting to analyze as well. Similar to the PCA analysis in the previous section, it is also possible to observe the covariance matrices in the principle component (PC) space. In this work, matrix $\mathbf{C}(\mathbf{U})$ is of interest, since it relates the ensemble members to the predicted model output. As described in Section 4.2.1 the ensemble members are the optimized weights for the new individuals. In contrast, the covariance matrix $\mathbf{D}(\mathbf{U})$ (which has dimensions of 3×3) provides less values as it correlates only the model outputs.

Figure 5.6 shows a scatter plot of the covariance matrix $\mathbf{C}(\mathbf{U})$ in a two dimensional PC space. In this step, the covariance matrix values over every tenth (test) generations are fit using the PCA function, thus 40 parameter sets are available. The matrix $\mathbf{C}(\mathbf{U})$ has a shape of $40 \times 12 \times 28800 \times 3$. The last dimension corresponds to the dimension of the output

cluster. The 12 dimensions in the middle denote the 12 images from the mini-batch, as for every image the covariance matrix is calculated. Unless otherwise stated, figures presented in this section show the first dimension, i.e. the covariance matrix of the first image of the mini-batch. The parameters for the other images are similar. While bright colors such as red, orange (generation $200 - 300$) and yellow (or cream colored) indicate later generations (generation $300 - 400$), black and dark purple colors denote earlier generations ($0 - 200$). In earlier generations the covariance matrix values are scattered over the space, in contrast the spread in later generations is less and more contracted around zero. To highlight later generations Figure 5.6 is drawn in two dimensions. If drawn in a three dimensional space, points of later generations are not visible, they lie in the center of the point cloud, covered by points of earlier generations. In contrast, the point clouds of the later generations are more elliptic and rotated around their horizontal axis. Especially, the red colored cloud (around generation 200) has a very narrow, elliptic form, which is due to the distribution of its values. In general, the covariance matrix has small values, with a mean close to zero and a standard variation of 0.003, which is captured in the PC space as well.

As mentioned above, the last three dimensions of matrix $\mathbf{C}(\mathbf{U})$ correspond to the three outputs of the reservoir network. Thus, a detailed look into the last dimension within the PC space may shed light on further details. The following listing describes the analysis in three steps:

1. Obtain for each of the three dimensions the PCA components.

2. Create a boolean mask to bound the components. The respective ranges for the three outputs are values ≤ -0.013 and ≥ 0.014. These values are obtained after calculating the minimum and maximum of the components and adding/subtracting a small value (0.01, 0.009) to the minimum or maximum. This step decreases the number of components by excluding values close to zero.

3. Plot the result over the generations.

Figure 5.7 shows the result for each of the outputs. The y-axis depicts the covariance matrix values, the x-axis are the number of obtained points, i.e. the mask consists up to 600 entries. The color gradient indicates the generation number as previously described in Figure 5.6. In all three outputs the covariance matrix values of later generations are narrowing down towards 0 and stay between the range $[-0.01, 0.01]$. Earlier generations (e.g. generation 50, dark purple) range between $[-0.02, 0.02]$. Furthermore, for output 2 generation 380 is depicted as a thin line and oscillates around 0, i.e. the covariance matrix values are small (in the order of $1e^{-7}$). This occurs, if the weights are heavily adapted and is also visible in the DKL of Figure 5.5, where a small spike can be observed at generation 380. In contrast, the last generation of output 1 is not as distinct and narrowed, it still has values over 0.01. This

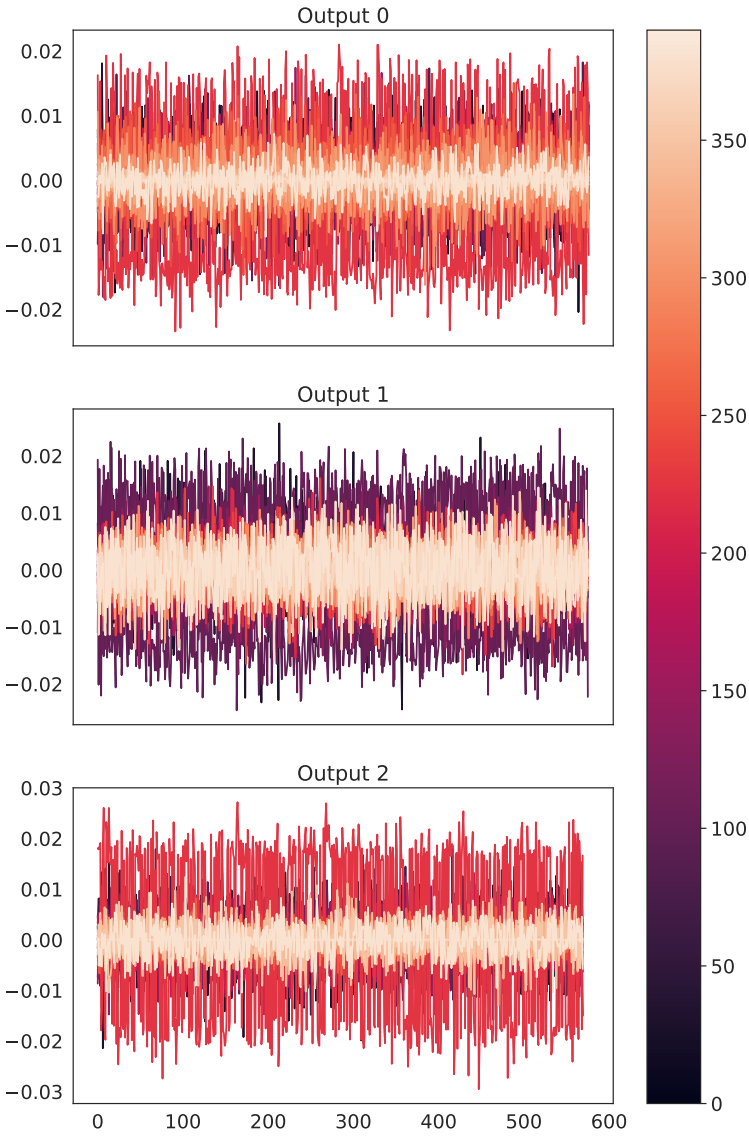


Figure 5.7: Covariance matrix values obtained from masked PCA components. The brighter the color the higher the generation number.

may be related to the convergence behaviour of the optimization procedure. While for output 0 and output 2 the weight parameters are already optimized, the parameters for output 1 may need further optimization, i.e. the network may need more training runs.

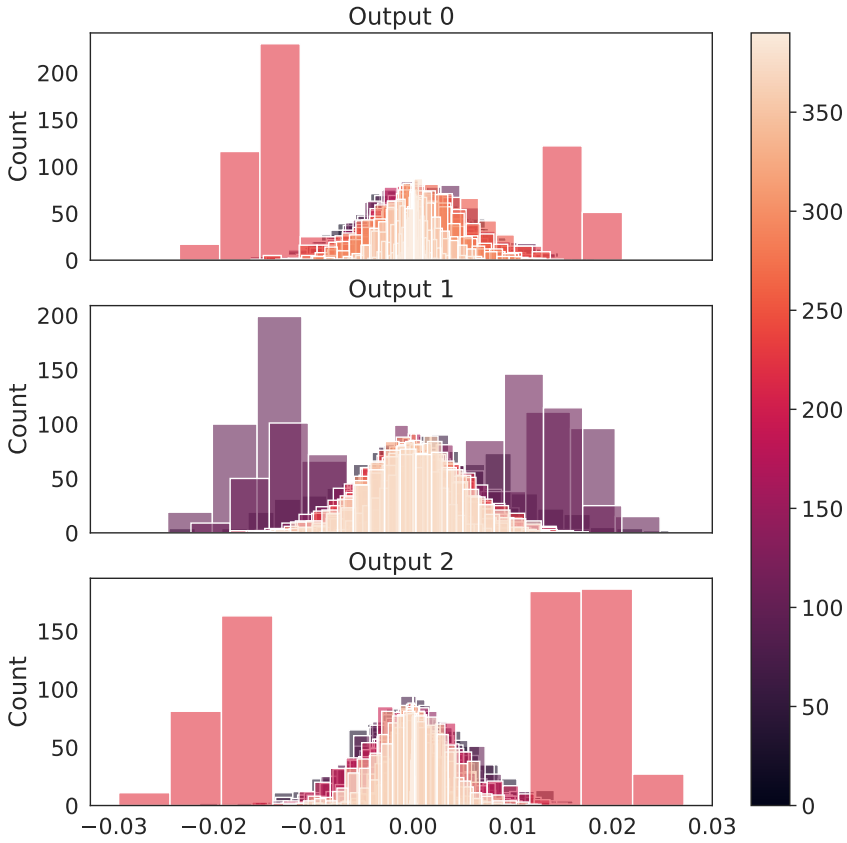


Figure 5.8: Histogram of the masked covariance matrix $\mathbf{C}(\mathbf{U})$.

Another noticeable aspect are the red colored lines, i.e. points from generations ≥ 200. The red colored points were also prominent in the form of a narrow shaped ellipse in Figure 5.4. A different point of view gives the histogram in Figure 5.8. This plot represents Figure 5.7 as a histogram and helps to see if the red colored values prominently occur in all outputs. As visible, for all outputs the distribution has a mean around 0 and throughout the generations the standard deviation is shrinking towards the mean as well. The red bars correspond to the red lines and the red point cloud in the previous figures. For output 0 and output 2 the red bars are in the range of $[-0.02, -0.02]$ and $[-0.03, 0.03]$. Because values close to zero are excluded, they seem to have a bimodal distribution, which is not the case and they are normal distributed, i.e. unimodal. This behaviour occurs after the weights are heavily updated and is often followed by a performance increase. Before the parameter update, the covariance matrix has a narrow distribution, which indicates a fitness decrease, afterwards

the distribution is broadened to adapt the weights. A weight change is also visible in the DKL of Figure 5.5, where it has a peak between generations 180 and 200. Additionally, whenever the performance drops, the weights are sampled from the best individuals of the previous generation, which also influences the weight change. Similarly, in early generations, values for output 1 are in the range of $[-0.03, 0.03]$ (see purple bars). In this case, a high variance between these values is more likely to occur, since at this point, the optimization has not converged yet.

5.6 Hyper-Parameter Optimization with L2L

L2L is a hyper-parameter optimization library. Since the framework is agnostic to the model and its parameters it was possible to optimize the connection weights. Hyper-parameters such as the EnKF's γ were manually set and tuned.

In this section, the ensemble Kalman filter is applied within the inner loop to optimize the reservoir weights. In the outer loop hyper-parameters of the EnKF are optimized using L2L's genetic algorithm. The hyper-parameters are:

- $\boldsymbol{\gamma}$ controls the learning rate. A lower γ lets the optimizer explore the parameter space by updating the weights heavily, i.e. a small γ increases the weights when multiplied with the covariance matrices which have small values. This may lead to a faster convergence but also has the risk to overshoot and oscillate around minima (for an example illustrating this behaviour see Figure 3.2). In contrast, a higher value is updating the weights less and is more robust in terms of convergence, however, it increases the optimization iterations required to converge towards an optimal solution.
- **Ensemble size** is the number of optimized individuals. As discussed in Section 4.4.2 and Section 5.4.3 the size can vary and is task dependent.
- **Repetitions** define how often the same batch of images is shown. While repetitions helped to increase the performance of the CNN, this approach was not necessary when optimizing the reservoir. Note, only repetitions of size ≥ 2 are regarded as repetitions, since in Python a for loop of range 1 is executed once, i.e. not repeated.

5.6.1 Hyper-Parameter Optimization Workflow

While the optimizee follows the same setup described in Section 5.4.3, the EnKF optimizer is now additionally applied in the inner loop. However, this change requires an adaptation of the workflow, which will be described in the following. The new workflow consists of two parts, the *creation* and *simulation* phase. In contrast to the previous setup only 20 individuals are initialized. These individuals spawn up to 50 ensemble members in parallel on an HPC node. The optimizer adapts the number of ensembles, which executes additional steps necessary to reserve compute resources.

The workflow starts with the *creation* step, which is displayed as a simplified flowchart in Figure 5.9. It mainly constructs the reservoir network and creates the individuals. The creation phase is executed once in generation 0 and initializes the hyper-parameters in the

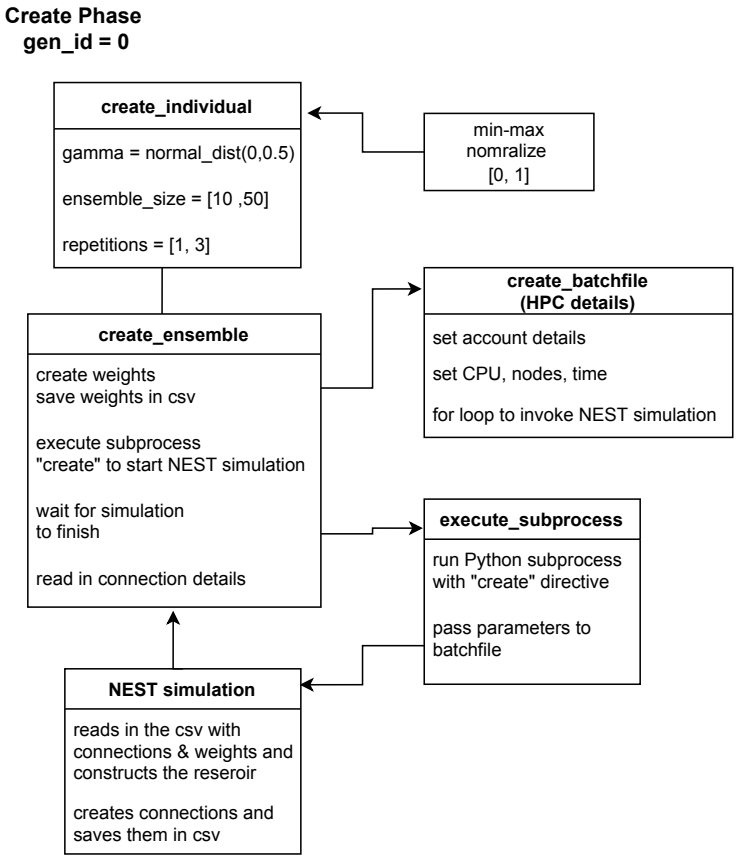


Figure 5.9: The create phase initializes the hyper-parameters and parameters such as weights and invokes the reservoir network. The constructed network reads in the weights and saves the connection details on the disk as csv files.

`create_individual` function. All parameters are set randomly: γ is drawn from a normal distribution with $\mu = 0$ and $\sigma = 0.5$ and is strictly positive, the ensemble size is uniformly distributed between 10 and 50, similarly the repetitions are uniformly distributed between 1 and 3. `create_individual` induces a cascade of follow-up methods. First, the hyper-parameters are min-max normalized to be in the range of $[0, 1]$. This is necessary for the mutation step within the GA, since noise drawn from the same distribution is added onto the individuals. Thus, it is important that the parameters are in the same range. Additionally, the `bounding_func` function ensures that the hyper-parameters do not exceed the range of $[0.001, 1]$ before starting with the inner loop. Later, in the simulate phase, the parameters are re-normalized, i.e. transformed to their original range. Then `create_ensemble` creates the

weights for the reservoir and saves them on the disk as csv files, so that they can be read from within the NEST simulation. Additionally, `create_ensemble` calls `create_batchfile` to write a small batch script including details for the HPC run such as the account details or the number of CPUs utilized in the simulation. It also reserves a node to execute the simulations and sets the time the simulation needs to finish. 50 simulations are able to run in parallel on a single node. This number is defined by the ensemble size. Furthermore, the function prepares a for loop inside the batch file to invoke the NEST simulations. As a next step, it calls the `execute_subprocess` function, which invokes a Python subprocess to pass parameters, such as the path, to the csv files and the "create" directive to indicate that the NEST simulation should read in the path, construct only the network and save the connection structure without invoking its internal simulation call (`simulate()`). NEST saves the network architecture as a table describing how the neurons are connected. This means that all individuals share the same architecture even if the weights will change as the optimization process takes place. Note, only the architecture is constructed, no further simulation is run at this point. Here, the construction process creates four tables, corresponding to the connections between the reservoir and outputs, i.e. the excitatory to excitatory, excitatory to inhibitory, inhibitory to excitatory and inhibitory to excitatory neurons. Finally, `create_ensemble` waits until the network construction is finished and reads in the connection details. This step is needed to correctly reassign the weights to the network for the next simulation phase, since the structure is changed to a two dimensional matrix when passing the weights to the EnKF optimizer.

After the creation phase, the simulation phase continues. The inner loop has $100/\textit{repetitions}$ iterations and runs for 8 generations in total. L2L's `simulate` function starts the simulation phase as shown in Figure 5.10. First, the function initializes the inner loop optimizer, the ensemble Kalman filter. Then it reads in the MNIST data and corresponding labels for the training or testing step and saves the data for the NEST simulation. Additionally, the function obtains the optimized hyper-parameters from the trajectory and transforms them back from the normalized to the original ranges, which is for simplicity called re-normalization [4]. Afterwards, the `load_weights` function is called to load and check the weights. If the number of ensemble members increases, new weights have to be created. New ensemble members are created by sampling from already optimized weights and perturbing them. Here, the aforementioned connection structure is maintained, e.g. weights for excitatory to excitatory connections are only sampled from corresponding connection weights.

The next step in the workflow executes the Python subprocess with the "simulate" directive. The `execute_subprocess` function passes parameters to the NEST simulation, which reconstructs the reservoir by reading in the connection structure and the weights. Since multiple

[4] The method described here should not be confused with the renormalization technique from quantum field theory and statistical mechanics.

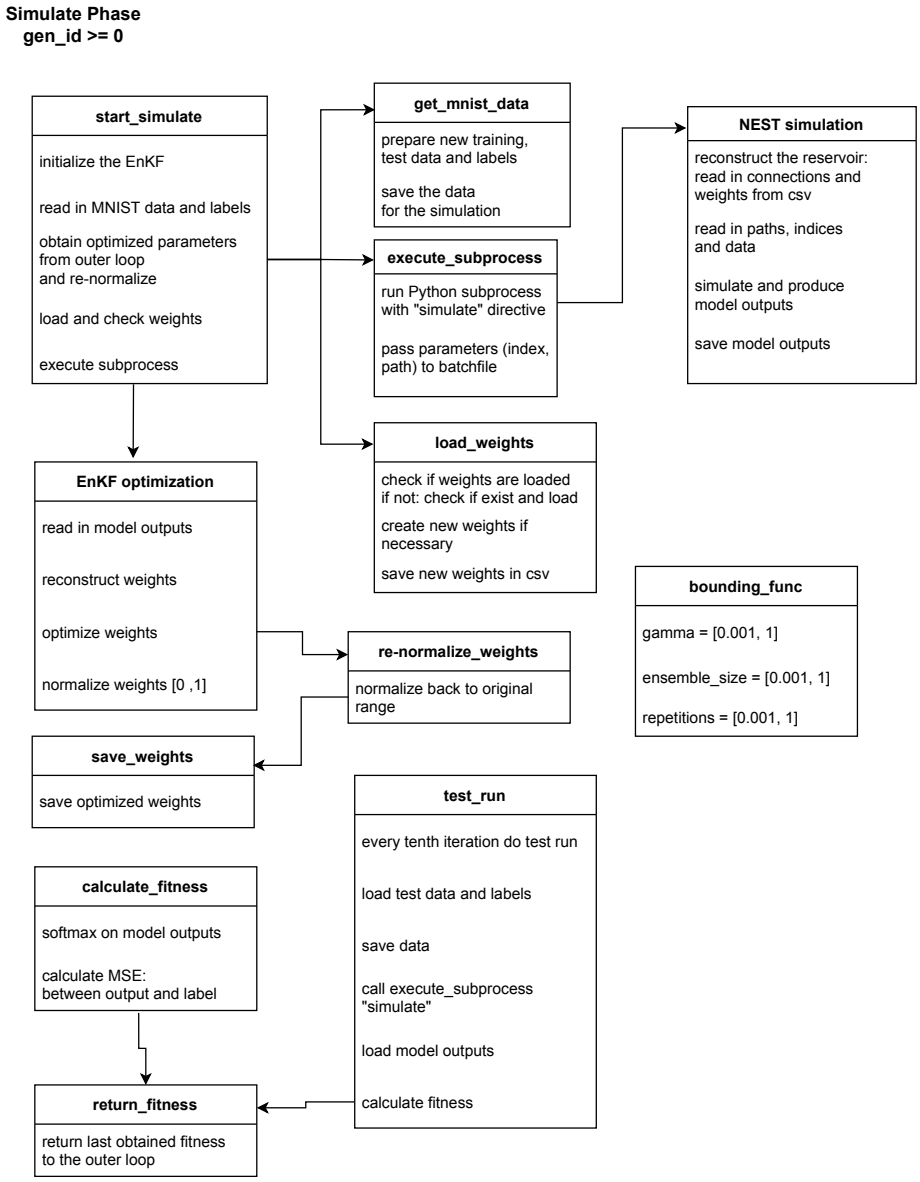


Figure 5.10: The simulation phase simulates the optimizee – the reservoir –, executes the training and test runs and returns the fitness with the hyper-parameters to the outer loop.

individuals spawn in parallel, the generation index, the individual number, and the ensemble member number are transferred as well, in order to identify the correct individual when loading back the model outputs. Then, NEST simulates the reservoir and the data is presented to the network in order to obtain the model output activity. After the NEST simulation the optimization starts. The inner loop optimizer class reads the saved model outputs and reconstructs the weights from the table format to a matrix and normalizes the weights to be in the range of $[0, 1]$. The EnKF optimizes the weights, which are then re-normalized and saved. Finally, in every tenth iteration a test run is initialized to obtain a fitness value. The test run is equivalent to the training, but skips the optimization part and uses the test set and test labels as input for the reservoir. Both the training and test runs use the `calculate_fitness` function to obtain the fitness. Then, the softmax function applied on the model outputs squashes the values between $[0, 1]$. The highest value, which indicates the model prediction, and the MNIST label are then used to calculate the MSE. However, only at the end of the overall inner loop iteration the fitness and the hyper-parameters of the training phase are returned to the outer loop.

5.6.2 Hyper-Parameter Optimization Results

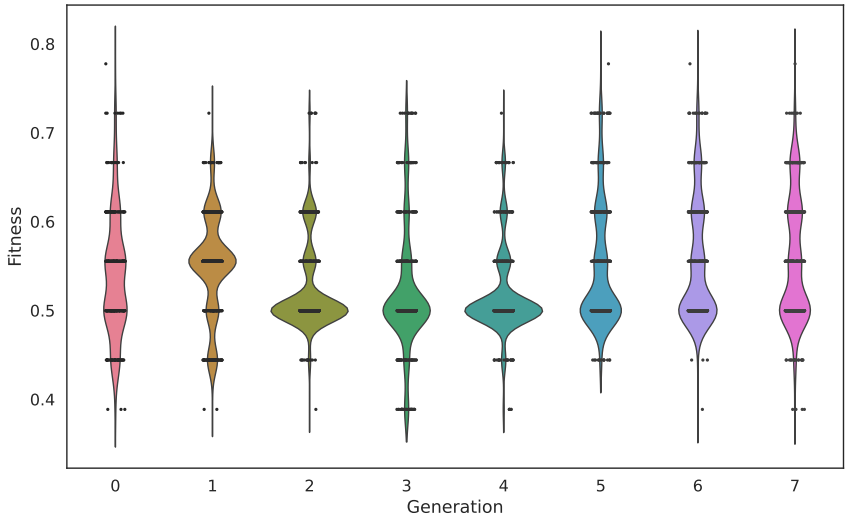


Figure 5.11: Optimized hyper-parameters increase the fitness over generations. Individual points denote the fitness of an ensemble member. The violin plots depict the distribution of the fitness points.

The hyper-parameters of the reservoir in Section 5.4.3 are manually tuned. By setting $\gamma = 0.5$,

the ensemble member size to 98 and no repetitions the network reaches a good performance. However, setting parameters in a manual fashion is tedious and prone to errors. Fortunately, L2L automates this process.

The two-loop optimization workflow consists of a total of 8 generations and 100/*repetitions* inner loop iterations. It is important to note, that in order to better compare how the hyper-parameters evolve in every generation the weights are reset to their initial values, i.e. the weight training progress is not transferred to the next generation. Nonetheless, the found parameter configuration should be similar to the manual setting and reach a comparable performance within 100 iterations. Figure 5.11 depicts the fitness over 8 generations. The figure consists of two plots, a scatter plot, where points indicate the fitness over the mini-batch from an ensemble member and a violin plot, which represents the underlying fitness distribution. The horizontal placement of the points within a generation is the order the ensemble member have in the data and has no real value. In total, there are 4356 points displayed on the figure. However, due to the adapting ensemble size, every generation has a different number of points which ranges from 300 to over 600. The fitness values range from 0.4 to 0.78. While in most of the generations (e.g. generations 2 to 4) the mean fitness is around 0.5, it slightly increases in later generations towards 0.55. The density estimation of the violin plots shows this by a narrowed density at 0.5, especially visible in generations 5 to 7. Moreover, the number of points increases towards a higher fitness, with the highest fitness at 0.78. This can be seen by observing the endpoints of the violin plots, the density thins out at lower fitness ranges and the plot is longer than earlier generations. For example, generation 5 has the overall highest performance, the lowest part of the violin plot is at 0.45 while the highest point is close to 0.8. All these characteristics indicate a performance increase induced by optimizing the hyper-parameters.

Figure 5.12 shows how the hyper-parameters evolve over the generations. The y-axis depicts the corresponding value for the parameter and the color indicates the fitness. The brighter the color the higher the fitness is. Each point corresponds to an ensemble member (of an individual), as mentioned earlier. The top left panel of the figure shows a clear trend for γ. The more generations pass the closer the points get between 0.4 and 0.5, with the brightest points around 0.5. This observation overlaps with the manual setting, where it is set to $\gamma = 0.5$. In the first generation the values range from ≤ 0.1 to 1.2 but are quickly pushed towards 0.5. A few points are close to 1.2, however, from the second generation on there are no values greater than 0.7. This exhibits the effect of γ, in early generations a low value may be beneficial to explore different local optima, but may not converge into a suitable optimum. On the other hand, a high value, e.g. close to 1, is useful to exploit the feature space around minima, but may prevent further exploration.

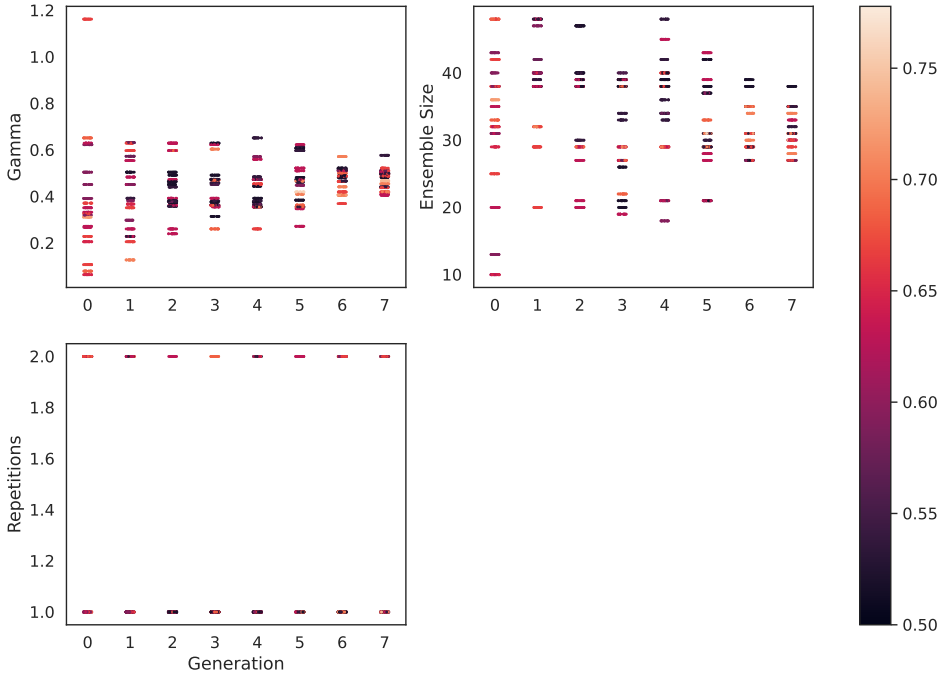


Figure 5.12: The fitness is related to the optimized hyper-parameters over the generations. While γ and the ensemble member size show a clear trend towards certain values, the repetition parameter seems to be not as relevant. Each individual point depicts the fitness of an ensemble member. The brighter the color the higher the fitness.

A similar behaviour can be observed for the ensemble member size, which is initially in the range from 10 to 50 members and converges towards 30 members. Observing the fitness, a high number of ensemble members does not lead to a higher fitness. In general, an ensemble with less members than 40 achieves a high performance.

The parameter for repetitions seems not to be relevant as the other parameters. Interestingly, the repetition never reaches the value of 3 (i.e. 2 repetitions of the same mini-batch). This value neither occurs at the initialization nor is it reached afterwards. It follows that repeating a mini-batch in this setup does not increase the performance. In generation 7 the best fitness is reached without any repetitions and may indicate that training with new data leads to a better performance than presenting the same mini-batch.

5.7 Discussion

Adapting hyper-parameters in a manual way is a tedious, time consuming, usually unsystematic and error prone task. Often, the whole workflow has to be executed from scratch to see the effects of adapting the parameters. Learning to learn is a concept to optimize (hyper-)parameters via experience. The idea is to learn on a family of tasks and generalize well enough to perform more efficiently on new tasks than the performance of a fresh training run. To achieve this, learning to learn is composed as a two loop structure, the inner and outer loop. In the inner loop an optimizee or an algorithm with learning capabilities, such as a neural networks, learns from the family of tasks. A fitness function evaluates the performance of the trained optimizee. In most cases the outer loop optimizer is from the family of metaheuristics, as these types of optimizers only require a fitness vector and parameters to optimize, thus, they are applicable for a variety of problems. To generate the set of parameters, called an individual, instances of the model are created. It is possible to execute the individuals in parallel in order to decrease the computation time and load. After receiving the parameters and the fitness the outer loop optimizer ranks the individuals according to their fitness and applies the optimization. The optimizer recombines and/or perturbs the individuals' parameters for the next generation and returns them to the inner loop.

L2L is a Python framework which implements the concept of learning to learn. It is able to automatically optimize (hyper-)parameters of a given model. The framework is agnostic to the model and can optimize all kind of parameters as long they are in the correct form and provide a fitness. The framework enables a simple interface to parallelize the execution of optimizees on HPCs.

Optimizing a spiking neural network to do a specific task is not trivial. In contrast to an artificial neural network, gradient descent is not directly applicable (see Section 3.2.4), thus, other optimization procedures are required. Metaheuristics, such as the EnKF, are alternative solutions to circumvent this problem. The ensemble Kalman filter is able to optimize the connection weights without the need of any approximative method. The task described in this work is to classify the MNIST dataset with a reservoir network. One interesting property of the reservoir is the capability to map the input to a space where the output is able to classify it. To achieve the classification only the weights from the reservoir to the output need to be updated, thus, decreasing the number of parameters to optimize. Due to the length of the simulation the network is trained on a subset of the dataset. Nonetheless, the model is successfully able to classify three digits and achieve a high performance.

To understand how the network learns, the connection weights and the covariance matrix $\mathbf{C}(\mathbf{U})$ of the EnKF are analyzed in Section 5.5. The weight analysis in PCA space shows

that the weights plotted over generations lie on an arch like shaped manifold. A possible explanation is the fast convergence due to the EnKF optimizer, which shifts the parameters to a mean close to zero with a decreasing standard deviation. A performance drop due to misclasification of the network results in a change of weights as the DKL in Figure 5.5 shows. High peaks indicate a change in the weights. The covariance matrix analysis in PC space displays the converging behaviour of the optimizer. Additionally, the covariance matrix is responsible for the direction and the speed of the convergence. A performance drop in the fitness results in small covariance values, expressed by a narrow distribution, which in successive the generation is expanded to correct the direction the optimization should take (see Figure 5.6 and Figure 5.7). In later generations the covariance matrix values converge towards a zero mean and also show a decreasing standard deviation. To summarize, by changing its values, the covariance matrix defines when to explore the parameter space and when to exploit it: While small values close to zero followed by high values in subsequent generations indicate an exploration step, slow changing values with a decreasing standard deviation point to an exploitation or to a convergence of the optimization.

A higher network performance can be achieved by manually tuning the hyper-parameters of the optimizer. L2L automates this process. In Section 5.6 the EnKF optimization is also applied in the inner loop and three hyper-parameters of the optimizer are adapted in the outer loop. In 8 generations this two-loop optimization process shows a clear trend towards certain parameter ranges and is comparable to the manual configuration. Figure 5.12 displays the fitness in relation to the parameters. For parameter γ an increase of small values and a decrease of high values towards 0.5 is observable. Similarly, the ensemble size converges towards a value of 30. A repetitive presentation of the same mini-batch does not have a big effect on the performance. Given the long runs and restricted time on the HPC, more training runs with new data is preferable in order to increase the performance.

6 Optimizing Spiking Neural Networks enhances Foraging Behaviour of Swarm-Agents

The previous chapter (Chapter 5) described the optimization of a reservoir network to classify MNIST digits using L2L as an optimization library. In this chapter SNNs controlling swarm-agents, in particular ants, are optimized via a genetic algorithm to enhance the foraging behaviour. The ants explore the simulated environment and forage food, their task is to return the food to their nest as much as possible while the simulation is ongoing (Section 6.1.1). The ants get rewarded or punished solely based on their actions (Section 6.2.2). The ant colony simulation and foraging task is well described in the literature, however, most approaches define action rules which guide and characterize the behaviour of the ants. The optimization process in this work is not constrained by any rules. Actions such as pheromone depositing are not manually defined, instead, the learning of such a behaviour is achieved through the evolutionary algorithm.

In Section 6.1 I will motivate the ant colony modeling problem and relate it to other works. Then, I will continue with the setup of the environment, the model and the optimization process (Section 6.2). Over the generations the ants learn to deposit the pheromone which leads to self-organization and self-coordination, thus enhancing the food foraging. In order to better understand the foraging behaviour, I will employ different analysis techniques (Section 6.3). I will compare the foraging performance of the evolved SNN-driven model to a simple, rule based system. The latter serves as a base line regarding the foraging performance. An additional test of the SNN-driven model assess the effect of the pheromone usage on the foraging behaviour when the pheromone sensor of the ants is disabled. Furthermore, I compare the performance of an evolved colony driven by an another SNN-driven model, where the ants have their pheromone pathways disabled, i.e. the ants are not able to perceive and deposit the pheromone. Finally, to understand if there is a relation between the ants' perceptions and their actions, I will correlate the input spike activity to the output spike activity of the network.

6.1 Modeling the Ant Colony

Swarm colonies use chemical signals as an efficient approach to interact and collaborate. Social insects, such as ants and termites, use pheromones to guide their cohorts while foraging for food. The coordination is self-organized, without following a central lead. Colonies are able to solve complex tasks, such as finding the shortest paths to the next food source. This behaviour is shaped by evolutionary processes over millions of generations. The modeling of ant colonies is a well studied field. The models are inspired by biological observations and implement (probabilistic) action rules to define the behaviour of the agents to interact with their cohorts (see also Section 3.3.2). By depositing chemical signals on the environment the agents are able to self-organize and self-coordinate. Although simple rules may help the system to solve specific tasks, it may not be an efficient solution and may not reflect the natural foraging behaviour. Furthermore, manually defining action rules for complex tasks, is difficult endeavour. To understand if an emerging self-organization and self-coordination within the swarm can be related to physiological properties and whether this self-organization can improve the foraging behaviour, it is useful to simulate the swarm behaviour within an environment to solve a complex task. In this work, the simulation encompasses the task of a swarm of agents to forage for food and return it to the nest. An SNN steers an agent, in particular an ant from an ant colony, in a 2D environment.

In the literature, there are many computational models which replicate, within certain boundaries, the dynamics observed in biological ant colonies. Vittori et al. (2004) and similarly Bandeira de Melo et al. (2008) build their probabilistic models based on the foraging behaviour of the Argentine ants Linepithema. Due to the pheromone usage their models exhibit self-organization and are able to find the shortest route to a food source in a maze. Wilensky (1997) implements a simple, rule based model to steer the local decisions of the ants. By utilizing pheromones the ants exhibit self-coordination and collaboration.

In Hecker et al. (2015), the authors implement a rule-based probabilistic model mimicking the foraging behaviour of seed-harvester ants and apply the model on real robots in environments where the food distribution changes. The model is evolved using genetic algorithms, so that the robots are successfully able to navigate within the environment and collect the food.

Similarly, other authors have explored models of different types of swarm agents solving problems in other application fields. H. Duan et al. (2014) proposes a framework to control agents, called shills, which are able to find strategies and provide those to other agents. Their model implements the iterated prisoner's dilemma game, where the agents can update their strategies based on pre-designed rules. The updates are adapted by a particle swarm optimization algorithm.

A combination of swarms and an SNN is presented by Chevallier et al. (2010). The authors propose a spatio-temporal model called "SpikeAnts" and show the emergence of a synchronized behaviour between agents of an ant colony. In their work, each ant is represented by two interconnected neurons. The network receives no external stimuli and does not include any learning rules. The authors conclude that the swarm is able to self-organize due to the emergent distributed learning process the ants display.

The papers cited above observe and analyze emergent communication and self-coordination as a result of the swarm organization, however, these approaches are probabilistic state models or rule-based systems that enforce collaborative behaviour. It is a difficult approach to rely only on the physiology of the agents to achieve a high performance in specific tasks. When including a collaborative or adaptive behaviour within a dynamic environment the complexity raises even more and requires to incorporate many complex rules. Thus, finding different rules scales up with the complexity of the task and number of agents, an optimal solution is often intractable. Optimizing for an emergent behaviour could provide a reasonable and adaptable approach to tackle this scenario.

In this work, the behaviour of the agents is not based on pre-defined rules. While navigating, the agents perceive their environment through multimodal sensory stimuli from various sources such as visual, mechanoreceptory, or chemical (pheromone) sensors. The sensory information is received by input neurons, which are connected to a second layer of all-to-all connected neurons. The second layer is connected to the output neurons which are responsible for the agent's actions such as movement and pheromone depositing. The all-to-all connection avoids bounding the network to a pre-defined topology, i.e. the type of connection is not fixed and can be adapted during the optimization process. There is no explicit mapping between the sensory input and the ants' behaviour, such as perception of visual information and pheromone depositing. A genetic algorithm optimizes the SNN's connection weights and spike time delays, so that the network is able to steer the ants to efficiently forage for food. The 2D environment, is based on the ant-colony model by Wilensky (1997). In this model, the pheromone diffusion and evaporation is simulated in a grid world.

6.1.1 Environment and Task

Figure 6.1 shows a screenshot from the NetLogo (Tisue et al. 2004) environment of the ant colony foraging for food. The ants deposit the pheromone on their way to the food sources (green leaves) and around their nest (black, brown hexagon). The blue patches indicate the pheromone concentration; the brighter the color the higher the concentration. The concentration exponentially evaporates over time. Other agents can smell the pheromone

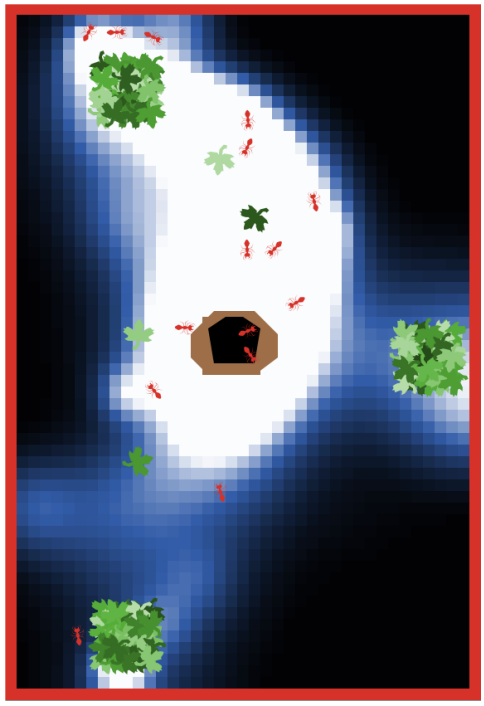


Figure 6.1: Screenshot from the NetLogo environment depicting the ant colony foraging for food. Blue patches indicate the pheromone concentration. The brighter the color the higher the concentration. Green patches depict food piles. The nest is the black hexagon in the middle. The red wall around the environment is impassable, the ants cannot cross the world from one side to the other.

and react to it. Some ants are following the pheromone trail to the food source or the nest, other ants are exploring the environment. The ants can only collect one food patch at a time and must return it to the nest before they are able to transport another food patch again. In every generation the position of the nest is changed randomly, in order to avoid overfitting the SNNs to the position of the food sources.

6.1.2 Network Architecture

An exemplary SNN steering an ant is depicted in Figure 6.2. The network has twelve input neurons. The first three neurons are receptors which react to the position of the pheromone. The next three neurons are able to locate the nest. The queen receptor indicates the middle of the nest. The reward receptors and nociceptors determine the reward and punishment.

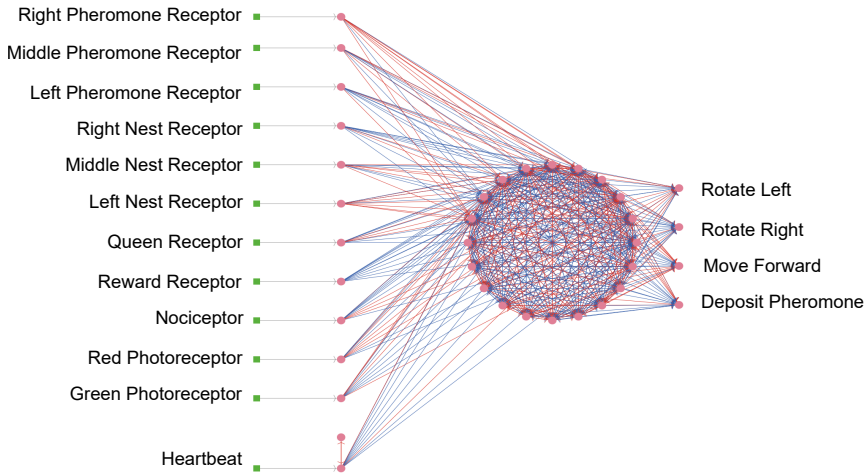


Figure 6.2: Spiking network to control an ant of the colony. The network consist of 12 neurons for the input, 20 neurons in the middle and 4 neurons in the output.

The green and red photoreceptors are activated whenever food or the wall is seen by the ant. The heartbeat neuron stimulates the network in every timestep by supplying a small direct current in order to maintain activity in the network. Twenty neurons are in the middle layer and are connected in an all-to-all manner. The four output neurons are fully connected with the middle layer and control the movement (rotate left, right, move forward) and the activation to deposit the pheromone.

6.2 Optimizing the Ant's Network with L2L

The simulations are run in the inner loop of an L2L experiment[1]. The optimization workflow follows the same two loop structure as described in Section 5.2 and is schematically depicted in Figure 6.3. In every generation there are 32 individuals and each individual is constituted by an instance of the SNN; note that the individual is not a simulated ant. In each simulation, 15 ants are foraging for food, meaning there are 15 copies of the same individual, i.e. the same network parameter configuration. Since there are 32 individuals, 32 simulations are executed in parallel, thus, $32 * 15 = 480$ networks are run in total per generation. The simulation of the SNN utilizes NEST as the back-end.

[1]The simulations are conducted on a workstation with an AMD Ryzen Threadripper CPU, 32 cores (3.7Ghz), 64GB RAM and Ubuntu OS 21.04.

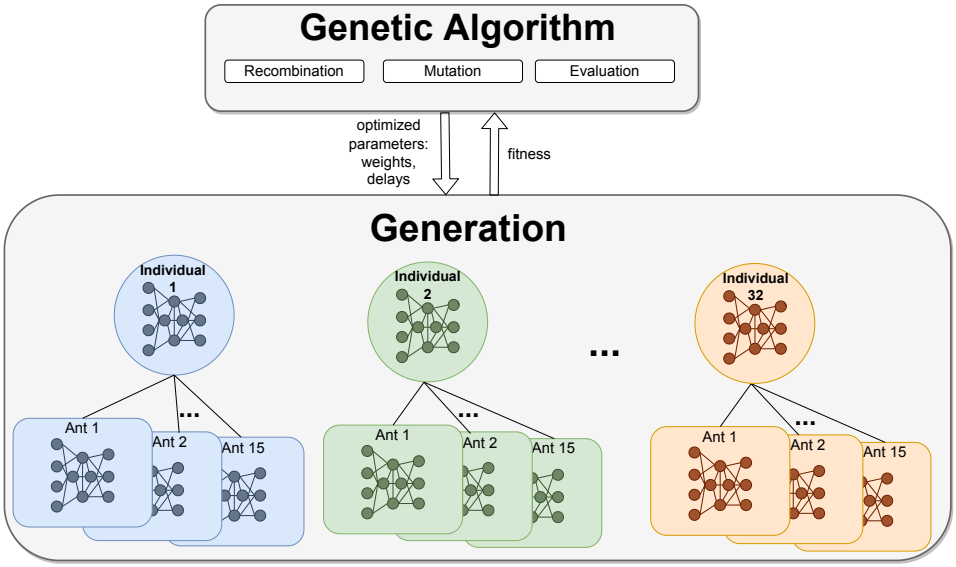


Figure 6.3: Optimization workflow for the ant foraging task. In every generation 32 SNNs, which constitute the individuals in the inner loop of L2L, are used to steer the behaviour of ants in a colony during the simulation of the foraging task. Each ant in the colony is steered by a copy of the SNN. The genetic algorithm optimizes the connection weights and delays of the SNN.

6.2.1 L2L Simulation Details

In the `create_individual` function, the connection weights are uniformly distributed in the range of $[-25, 25]$ and delays are in $[1, \ldots, 7)_{\mathbb{N}^+}$. Additionally, the parameters are min-max normalized and saved in a csv file. They are read in when constructing the SNN in NEST. The `simulate` function calls the NEST simulation by executing a Python subprocess. The environment is based on the model of Wilensky (1997) implemented in NetLogo. NetLogo is a multi-agent simulator and modeling environment. Every object in a NetLogo simulation is an agent and agents can communicate with each other. With NetLogo it is easy to replicate the same simulation and agent. Blue colored patches represent the pheromone which is deposited by the ants on the environment (see Figure 6.1).

The inner loop waits until the simulation finishes and collects the fitness which is calculated based on the amount of food collected by the colony and returned to the nest, the extend of movement and pheromone depositing (see Section 6.2.2). The parameters of the network and the fitness values are then sent to the GA in the outer loop. In every generation, the weights are clipped in the `bounding_func` to stay in the range of $[-20, 20]$ and delays to $[1, 5]$.

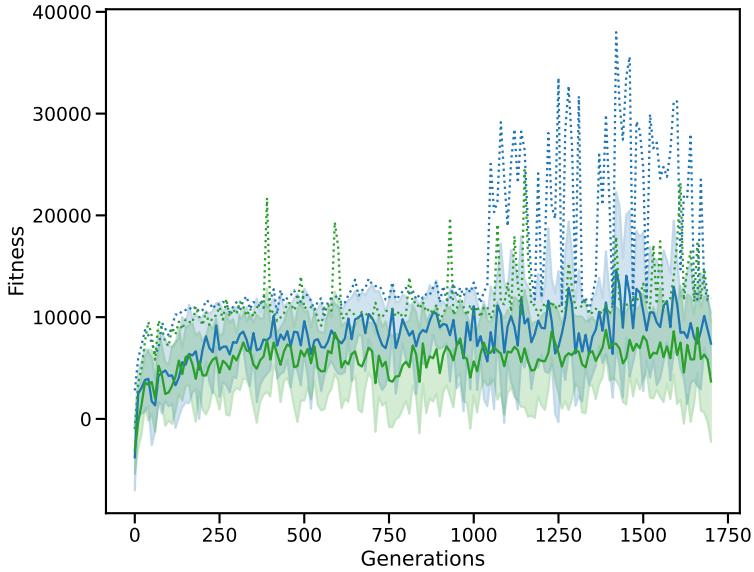


Figure 6.4: Evolution of the fitness of two different ant colony simulations over 1700 generations. In the first SNN-driven model the pheromone pathway is activated. However, the ants of the second SNN-driven model are not able to sense and deposit pheromones. The blue data series illustrates the fitness evolution of the first model. The green data series shows it for the second model. The dotted data series depicts the best individual of every generation. The blue/green solid line is the mean fitness of all individuals and the blue/green shaded area is the standard deviation.

Restricting the parameters to these values resulted in an enhanced swarm performance.

6.2.2 Ant Foraging Performance

The fitness of the ant colony optimization is the summation of the rewards and punishments received by each of the ants in the colony during the simulation. An ant receives a small positive reward for touching a food patch and a large reward for returning with food to the nest. This induces the ants to quickly return to their nest, whenever they find food. A small punishment is given at every time step. This encourages the colony to quickly complete the task. At the same time, the ants receive a small punishment for every action in order to avoid excessive reactions, such as depositing too much pheromone or exhibiting too much movement. In this section, two different SNN-driven models are compared. The first model is SNN-driven (SNN model 1) as described in Section 6.1.2. The second model is SNN-driven

(SNN model 2) as well, however, the pheromone pathways are disabled and the ants do not have the capability to deposit or perceive pheromones. Note, that both SNN-driven models are evolved over 1700 generations, the optimization workflow follows the scheme explained above.

Behaviour	Cost
Resting	-0.5
Dropping Pheromone	-0.05
Rotation	-0.02
Movement	-0.25
Return nest	220
Touch food	1.5
η	30.0

Table 6.1: Cost values to calculate the fitness of the colony. Every movement, rotation and pheromone deposit of each ant results in a small punishment. Returning to the nest with food and touching the food is positively rewarded.

The simulation takes $T = 2000$ steps to finish, a step size corresponds to 20 ms simulation time in NEST. The ants additionally receive a reward if they are able to collect all the food before the simulation ends. This reward is the result of the difference between the total simulation time and the time T_s the ants require to collect the food and is multiplied by a scalar η. This can be written as:

$$f_i = \sum_{t=1}^{T_s} \left(\sum_{j=1}^{J} \mathcal{N}_{i,j}^{(t)} + \mathcal{F}_{i,j}^{(t)} + \mathcal{C}_{i,j}^{(t)} \right) + \eta \left(T - T_s \right), \tag{6.1}$$

where $t = 1, \ldots, T_s$ is the simulation step with $T_s \leq T$ and T the total simulation time, η is a scalar to weight the speed of the ants to collect the food. i is a specific individual, $j = 1, \ldots, J$ is the ant index and J is the total number of ants in a colony. $\mathcal{N}$ is the positive reward value for coming back to the nest with food, $\mathcal{F}$ is a positive value for touching the food and $\mathcal{C}$ is the punishment cost. The specific values are listed in Table 6.1.

Figure 6.4 shows the evolution of the fitness over 1700 generations for two different simulations. As mentioned above, two different ant colonies are trained, in the first SNN-driven model the ants have the capability to perceive and deposit pheromones, however, in the second SNN-driven model the ant's pheromone pathway is disabled, i.e. the ants are only using their visual receptors to perceive the food. For both runs the mean fitness is increasing over the generations. After 1000 generations the performance increase of the first model slows down and oscillates around a mean of 8100 with a standard deviation of 3970. The highest fitness is at 38004 in generation 14020, specific individuals reach a fitness higher than 30000. The

mean of the second model converges after 700 generations with a value of 5800 and a standard deviation of 3950. The highest fitness is 24450 in generation 1150. In contrast to the first model, specific individuals reach a high fitness in early generations, e.g. in generation 390 a fitness of 21716 is observable. Compared to a randomly initialized SNN, these results show that the SNN exhibits an increasing performance due to the optimization with a genetic algorithm. Furthermore, colonies capable of utilizing the pheromone efficiently (SNN model 1) display a higher performance than the colonies which rely only on their visual perception (SNN model 2).

6.3 Analysis of the Ant Foraging Behaviour

After evolving the SNN, the ant colony is able to efficiently forage for food. This section investigates how the swarm develops and adapt its behaviour over multiple generations. First, the performance of the SNN-driven model is compared to a rule based system, in which each ant follows a set of pre-defined rules. It is additionally compared to a second SNN-driven model in which the agents lack the ability to perceive and deposit pheromones. Second, the behaviour of the ants is analyzed by correlating the input and output spike activity of the corresponding SNN.

6.3.1 Comparing the Performance to a rule-based Model

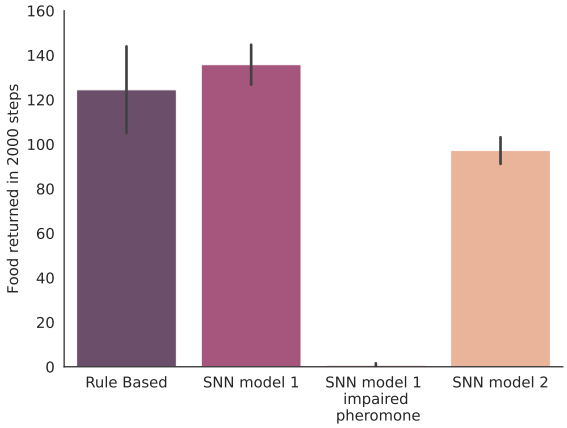


Figure 6.5: Ant model performance comparison. The boxes show the average number of food patches collected within 2000 simulation steps over 100 trials. The vertical black line is the standard deviation. The position of the nest is changed in every trial.

To assess the performance of the ant colony the average amount of food foraged by the colony and returned to the nest is calculated within the simulation time of 2000 steps and over 100 trials. Three piles of food are distributed in different locations of the 2D-environment as depicted in Figure 6.1. Each pile contains 50 units of food making a total of 150 units, which is the maximum score that a colony can reach in this setup.

Figure 6.5 shows the foraging performance of the two models. The boxes depict the mean number of collected food over 100 trials and the black vertical line is the standard deviation (SD). The first model is a rule based system implemented in NetLogo (Wilensky 1997). Here, the ant foraging behaviour is determined by pre-defined action rules. The self-organized behaviour of the ants emerges from the interactions of the individual members. The rule based system is the baseline for the comparison. In this setting, the colony reaches a mean of 124.45 of foraged food units with a standard deviation of 19.552. In contrast to the rule-based system, the SNN model 1 reaches a higher mean score (135.74) and a lower standard deviation (9.007). The effect of the pheromone on the colony coordination is measured by disabling the pheromone sensing in the evolved SNN model (SNN model 1 impaired pheromone). The foraging performance drops significantly, the ants are not able to collect the food (mean: 0.67, SD: 0.936). The ants of SNN model 2 do not have the ability to utilize the pheromone, the performance is lower than the rule based system and SNN model 1 (mean: 97.1, SD: 60.0).

Based on these results, the evolved model with a self-coordinated system (SNN model 1) is more performant than the rule based system and the SNN model 2, which relies only on the visual sensory pathways to navigate and forage for food. Through the optimization procedure, the ants are able to learn to communicate via pheromones, which increases the foraging performance. The behaviour is not manually encoded and the communication within the colony emerges as a swarm strategy through evolutionary optimization.

6.3.2 Correlating the Input and Output Activity

To better comprehend how the ant behaviour evolves over the generations, all input spike trains are correlated with all output spike trains for all ants of the best individual of specific generations. First, the input and output spike trains are binned in a histogram with a bin number of 2000, which corresponds to the simulation step size of 20 ms. Second, the mean Pearson product-moment correlation coefficients between all binned spike trains are obtained. Figure 6.6 depicts a correlation heatmap for the input (x-axis) and the output (y-axis) activity (see Figure 6.2 for the input/output description of the network) over 4 different generations. A bright color indicates a high correlation, a dark color denotes a negative correlation. The nomenclature follows the naming scheme of Figure 6.2, i.e. the output activity *Left* is rotate

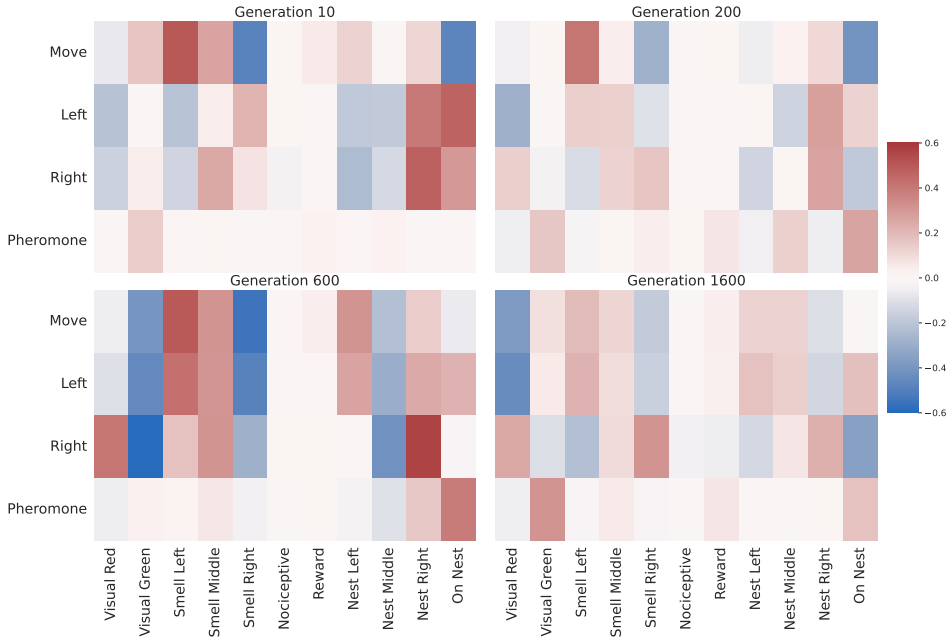


Figure 6.6: Heatmap showing the correlation between the network input and output spike activity. Each heatmap depicts the mean Pearson product-moment correlation coefficients between the input and output spike trains of all ants from the best individual in that generation.

left, *Move* means move forward.

Across the generations it can be observed that *Smell Middle* is positively correlated with *Move*, with a value of 0.265, which decreases to 0.116 in generation 1600. The Figure also shows that *Smell Right* is negatively correlated with *Move*, in generation 10 it has a value of 0.487 but at generation 1600 the strength of this anti-correlation reduces to 0.182. Interestingly, in early generations *Smell Left* is negatively correlated with *Left*, but the correlation gets positive after 200 generations. A negative correlation between *Move* and *On Nest* (0.220 in generation 10) can be observed in early generations, while for later generations it becomes uncorrelated (0.002 in generation 1600). In generation 10, *Nest Middle* and *Left* are anti-correlated (0.187) and positively correlated in generation 1600 (0.135). The correlation *Visual Green* and *Pheromone* is slightly positive in early generations (0.136 in generation 10) and increases in later generations (0.313 in generation 1600). Extensive pheromone depositing is punished by the fitness function and thus, leads to a rather restrained usage. A slight increase in the correlation between *Pheromone* and *On Nest* over the generations is also observable.

Due to the optimization with the evolutionary algorithm the input-output relationship exhibits a variable behaviour across the generations, the relationship between input and output gets weakened or enhanced. Some of these mappings fit to observations made in nature, e.g. the increasing correlation between *Pheromone* and *Visual Green* reflects the positive reinforcement when an ant perceives food. Additionally, the pheromone serves as an attractor and as a guiding signal for the colony to food patches or to identify the direction of the nest. This is expressed by the correlation between smelling pheromones and the ant movement.

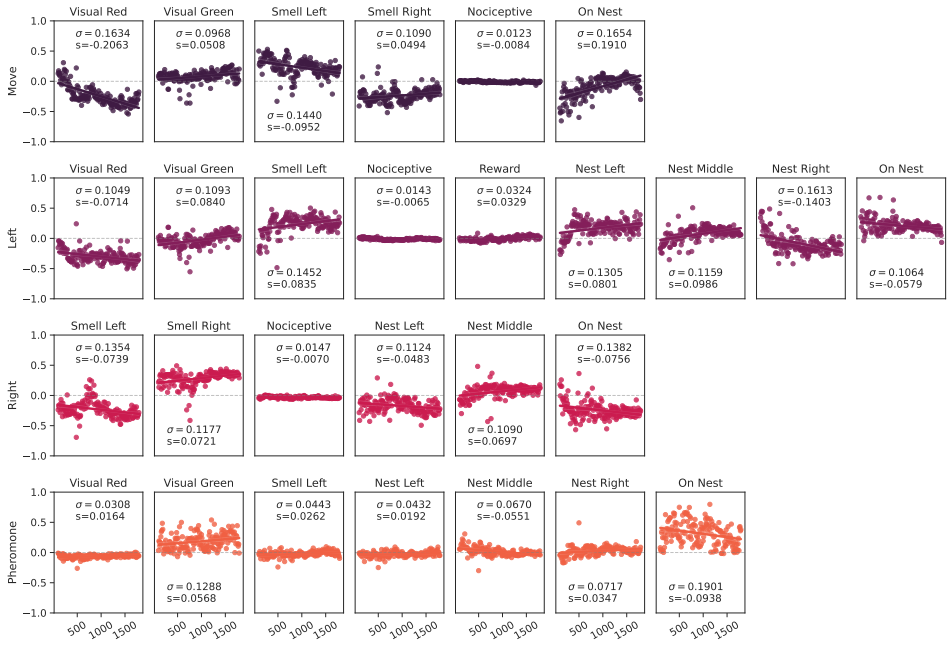


Figure 6.7: Regression plots of the input-to-output correlation. In every tenth generation the correlations between the input and output spike trains of the first ant of the best individual are obtained. A regression (solid lines) is fit on the resulting correlation values. The s-value indicates the slope of the curve. The null-hypothesis states that the slope of the curve is zero. Only plots with a p-value below 0.005 are displayed. σ is the standard deviation of the points.

The pheromone heatmap already provides an interesting interpretation of the ant behaviour for specific generations. To see how the behaviour evolves over all generations and to determine trends for certain actions, the above obtained input-to-output correlation values are fit using a linear regression for the same ant of the best individual. The regression, depicted in Figure 6.7, returns a p- and s-value. A low p-value rejects the null-hypothesis that the slope of the curve is zero (no correlation) and the s-value is the slope of the curve. Figure 6.7 displays

plots with a $p < 0.005$. This visualization depicts trends in the ant behaviour over generations. For example, *Visual Red* is initially positively correlated to *Move*, but shows a trend towards anti-correlation over the generations. In contrast, *Visual Green* and *Pheromone* show an increasing trend, which aligns with the observation made above. Similarly, *Smell Right* has an increasing positive correlation with *Right* and an increasing negative correlation with *Left*. *Smell Left* is positively correlated to *Move* in early generations, but tends to zero later on. Furthermore, *Smell Left* shows an increasing correlation with *Nest Left* and it decreases with *Nest Right*. *Smell Middle* displays an increasing positive correlation with *Move* over the generations. *Nest Middle* has an increasing positive correlation with *Left*, while *Nest Right* shows an increasing negative correlation with *Left*. Initially, pheromone is dropped in the vicinity of the nest. However, the pheromone output activity decreases over the generations, indicated by a negative trend for *On Nest*.

Pheromone is used as an attractor and trails with higher pheromone concentration mark shortest paths to food sources. This results from a combination of optimized movements while sensing the pheromone, the evaporation rate of the pheromone, and the evolutionary pressure of the fitness metric towards the pattern explore, get the food and return it to the nest, which resembles the evolutionary shaped behaviour of natural ant colonies.

6.4 Discussion

This chapter presented the successful optimization and evolution of SNNs controlling agents foraging for food. Due to the emerging self-coordination and self-organization via pheromones the ants are able to collaborate and increase their performance. In this setting, the ants learn to communicate which leads to an efficient performance during the foraging task. Disabling the ant's pheromone sensing drastically reduces the performance of the swarm. This highlights the importance of the (pheromone) communication, which emerges over hundreds of generations, for the effective collaboration between the agents. The performance comparison shows that the evolved colony steered by an SNN achieves a higher performance than the multi-agent rule-based colony. It is important to note, that the release of pheromone does not obey any pre-defined rules or a manual synaptic pre-configuration of the network. Instead, it is an emergent behaviour that is triggered under certain conditions and established during the evolutionary process. Interestingly, neither the mechanism to trigger the release of pheromone nor the interpretation of these signals is engineered. The optimization algorithm efficiently exploits the simulated physiological properties of the ants and the characteristics of the environment.

7 Conclusions and Outlook

Optimizing spiking and artificial neural networks to solve complex tasks is not an easy procedure. Optimization requires to embed the task in a suitable framework consisting of the mathematical problem formulation, an objective function, and a fitness function to evaluate the performance of the algorithm (Richards et al. 2019). Furthermore, adapting the network parameters, such as connection weights, is not the only configuration that leads to a suitable performance. Hyper-parameters influence and control the learning process heavily. In my thesis I present ways to leverage gradient-free optimization techniques to be applicable on spiking neural networks (SNNs) and artificial neural networks (ANNs) without the need to change the network structure or employ complex approximations to calculate gradients.

7.1 Conclusion and Discussion

In the beginning of Chapter 2, I introduce the basic concepts of artificial and biological networks. It is important to understand the differences between those types of networks. While ANNs are more task specific, e.g. they are used for classification or language processing, SNNs are utilized to understand brain activity and related processes. In recent years the interest in the artificial intelligence (AI) community to combine biological processes with artificial networks increased (Marblestone et al. 2016; Lake et al. 2017; Hassabis et al. 2017; Zador 2019). Many building blocks of ANNs incorporate biological, neuronal processes in an abstract manner and draw inspiration from neuroscience. This interconnection increases the learning efficiency and network performance on many machine learning tasks. In neuroscience the interconnection is still emerging and subject to extensive research (Bartunov et al. 2018; Richards et al. 2019; Tavanaei et al. 2019; Lobo et al. 2020). Since the brain is able to easily solve learning tasks, SNNs should have the potential to do the same. However, training and optimizing SNNs to solve tasks is still challenging, but promising.

In my thesis I propose ways to apply optimization on SNNs which also work on ANNs. I explore gradient-free optimization techniques, which can be seamlessly applied on both artificial and spiking networks. In machine learning, gradient descent and backpropagation are popular methods to optimize neural networks. However, depending on the initialization and

other properties, such as the selection of activation functions, the problem of vanishing or exploding gradients can occur. This problem hinders networks from learning since the gradients either are extremely small or too high. Furthermore, the application of backpropagation and gradient descent on SNNs is biologically implausible or requires complex approximations. A challenging point is the non-differentiability of the spiking non-linearity. The all-or-nothing behaviour of the neuron to emit a spike can be understood as a Heaviside step function and makes the calculation of a gradient challenging or impossible without approximations. Instead, in this thesis I investigate metaheuristics, such as genetic algorithms or the ensemble Kalman filter. Metaheuristics are black-box optimizers based on population decisions and are a powerful, alternative solution to gradient descent. Every metaheuristic requires several randomly initialized instances of the algorithm and a fitness function to evaluate the performance. Metaheuristics are model agnostic and can be employed on a wide range of problems as long as the algorithm or model can be represented as an optimization problem.

In Chapter 4, I explicate the problem of vanishing gradients in a convolutional neural network (CNN) and describe an optimization technique which is able to overcome this problem. Here, I design a setting based on the selection of network weight initialization and activation functions that leads to the vanishing gradient problem when the network is trained with gradient descent and backpropagation. The task is to classify digits from the MNIST dataset. In the initialization the weights are sampled from a normal distribution and the activation functions are sigmoidal functions, such as the logistic or the tanh function. To see the effects of the vanishing gradient problem, I analyze the gradients and activation values. When trained with stochastic gradient descent, the network is not able to classify the digits. A detailed look into the histogram of the activation values and gradients show that they do not change throughout the training. A similar picture can be seen for the optimization with Adam, although it is slightly more robust. It takes over 30 epochs before the network is able to classify, however the performance is not satisfactory. As an alternative to gradient descent I propose the ensemble Kalman filter (EnKF) as a gradient-free optimization technique. Due to the fast convergence on non-convex problems with several optima, the ensemble Kalman filter is suitable for problems where calculation of the gradient is not possible or requires complex approximations. I apply the EnKF on the same network setting and within one epoch the network exhibits a high classification performance. In contrast to the gradient descent methods, the activation values do vary over the iterations, thus, the network is able to learn. Additionally, I analyze the effects on the network performance when changing the hyper-parameters of the EnKF, such as the number of ensemble members, repetitions of the same mini-batch and different activation functions. A simple algorithm based on previous loss and the actual one adapts the hyper-parameters of the EnKF. However, it becomes clear that an automated hyper-parameter optimization method is preferable, especially if the number of parameters increases. Although the EnKF provides a suitable optimization solution the

run time of the algorithm is long. Surprisingly, benchmarking the compute time reveals that accessing the model output takes the longest time to finish in contrast to e.g. the calculation of the covariance matrices. Changing the array object, in which model outputs are stored, may decrease the indexing time. Optimizing the run time of the EnKF is a topic for future investigation.

The simple algorithm to adapt the parameters of the EnKF showed the need to have an automated way of optimizing hyper-parameters. L2L is a framework to optimize hyper-parameters and parameters. L2L consists of a two loop structure, the inner and outer loop. In the inner loop an algorithm or a model is trained or simulated on a task of a family of tasks. The outer loop optimizes the (hyper-) parameters of the algorithm. A fitness function evaluates the performance of the inner loop algorithm. The naming scheme in L2L is inspired by evolutionary algorithms, thus, the performance is called fitness and the optimized parameters are named individuals. In L2L the optimizers are gradient-free metaheuristics and are agnostic to the model or algorithm.

In Chapter 5, I describe hyper-parameter and parameter optimization utilizing learning to learn. I explain the advantages of automated hyper-parameter optimization and how it can improve parameter exploration in contrast to a manual approach. I start the chapter with introducing the concept of learning to learn, I use the EnKF as an outer loop optimizer to adapt the connection weights of a spiking reservoir network to classify digits. The reservoir is implemented in the spiking neural network simulator NEST and each individual is distributed on an high performance computing system (HPC) and runs in parallel. Due to the multi-threading implementation of NEST, the individuals scale well on the HPC. Additionally, L2L enables further parallelization by running multiple instances of the network at the same time on different compute nodes. Although the simulation takes time and only 3 digits are presented to the network, after 400 generations the reservoir exhibits a high accuracy. Due to the fast convergence behaviour of the EnKF the optimization can prematurely converge to a local optimum. In order to circumvent this problem I implemented a method which ranks the individuals according to their fitness and replaces the poorly performing individuals with the best ones. Additionally, I perturb the new individuals by adding random values drawn from a normal distribution in order to increase the parameter search space and to find different or possibly better solutions. In order to understand the optimization of the evolving reservoir network I explore and analyze the connection weights and the covariance matrix of the EnKF over the generations in the principle components (PC) space. The weights over the generations lay on the same arch-shaped manifold in 3D space and show a similar convergence behaviour which is due to EnKF convergence. Similarly, the Kullback-Leibler divergence of the weights varies within 200 generations and stabilizes afterwards with minor oscillations around generation 300. The covariance matrices of the EnKF determine the optimization di-

rection and convergence, thus the outcome of the optimization. The covariance matrix values for the three different outputs of the reservoir in PC space decrease over generations, which is in line with the converge of the weights.

The learning progress of the algorithm is influenced by its hyper-parameters. In order to see how hyper-parameters affect the learning performance of the reservoir network, I design a workflow which includes the EnKF optimization in the inner loop and adapt parameters of the optimizer in the outer loop using a genetic algorithm (GA). In this setting, I optimize three hyper-parameters: 1. γ, 2. ensemble member size and 3. repetitions of the same mini-batch. While the GA finds a similar configuration to the manually tuned setting for γ ($\gamma \approx 0.5$), the optimizer proposes a lower number of ensemble members (≈ 30) According to the optimization results, the repetitions are not required. Instead a higher number of training samples is preferable. These results show the benefits of automated hyper-parameter searches: a manual configuration is error prone and requires to run the simulation several times, which is tedious. Instead, hyper-parameter optimization automatically explores and finds suitable configurations, which could be missed otherwise. It also makes better usage of computational resources as the exploration is guided by a meaningful metric and not just by random exploration or subjective experience from the user.

The final chapter (Chapter 6) addresses the application of optimization techniques for spiking neural networks applied to a multi-agent systems. In this setting, the problem deals with the optimization of SNNs and the emergence of self-coordination and self-organization among the agents. Here, SNNs are steering agents or ants of an ant colony, which collectively forage for food. The connection weights and delays of the network are optimized by a GA. The ants are able to deposit chemical signals, called pheromones. There are no pre-defined rules to control the ant behaviour, i.e. there is no explicit mapping of the ant's sensory input and its actions. The optimization process has to find a solution to enhance the foraging performance. The simulation is run within L2L and only the fitness and the connection weights are sent to the optimizer. During the optimization, self-coordination and self-organization within the colony emerges. The ants learn to deposit the pheromone, which enhances the foraging behaviour. This highlights the importance of communication by using the pheromone and results in an effective collaborative between the ants.

I compare the foraging performance using an SNN with a ruled-based model. The performance of the SNN is higher than the ruled-based model. However, turning off the pheromone sensing has as a consequence that the ants are not able to fulfill their task anymore and the performance drops significantly. Furthermore, to have a better understanding of the foraging behaviour, I correlate the input and output spiking activity of the network over multiple generations. The analysis shows that pheromone is utilized as an attractor and high concen-

tration pheromone trails indicate the shortest paths to the food source. This resembles the actual behaviour of ant colonies observed in nature.

In terms of optimization, a challenge is to create the fitness function used to evaluate the ant foraging performance. The fitness function is problem specific and finding an efficient function is a complex task. To illustrate this point, the foraging task can be extended so that the ants are punished whenever they collide. However, just adding a simple cost value for the collision increases the complexity of the training and optimization. The ants erratically spin around or stop moving after a few steps. This behaviour might resolve with further training runs, but it is more likely that the fitness function has to be adapted. It is important to find a balance between punishment and reward cost, which leads to several trials and manual adjustments. This exploratory and exploitative behaviour is influenced by the fitness function. A strict fitness function, i.e. every action in the simulation generates a reward or a punishment, may lead to exploitation of local optima. However, it may also restrict the exploration of different, better optima. Conversely, a lax fitness function may result in an overly exploratory behaviour, that does not exhibit any exploitation.

7.2 Outlook

In my work I illustrated the commonalities and differences between SNNs and ANNs and how optimization can be applied on both network types. I investigated meta-learning and metaheuristics to optimize hyper-parameters and practicable solutions to efficiently scale algorithms in a parallel fashion using the L2L framework. The framework is applicable on SNNs and ANNs and provides an easy way to optimize parameters following the learning to learn two loop structure. The concept of learning to learn incorporates learning by generalization, i.e. given a family of tasks, the learning algorithm can utilize previously learned knowledge to enhance it's performance on the actual task. The main idea comprises that with this scheme the algorithm is able to generalize and perform better on a variety of tasks. Therefore, I aim to test the reservoir network additionally on a second dataset, such as the letters dataset (see Chapter 4). After reaching a high accuracy on the MNIST dataset, the network will be trained on the letters dataset to achieve a higher performance in a quicker fashion in comparison to a single training scheme.

One problem when training networks on multiple datasets is catastrophic forgetting (McCloskey et al. 1989, see also Section 2.5). Learning to learn in combination with bio-inspired techniques is able to overcome this issue (Kirkpatrick et al. 2017; Beaulieu et al. 2020; Nicholas et al. 2021). Another approach is multi-objective optimization, which separates the objectives from a single fitness function and optimizes the algorithm for several independent objectives

in an interchangeable manner. It is possible to weight specific objectives with a pre-factor to find the optimal pareto front. The L2L framework is already able to process several fitness values, and thus, allows for multi-objective optimization.

Sampling and perturbing parameters by adding noise were effective techniques to increase the performance when training the network. By applying those techniques, exploration and exploitation of optima was possible. However, the balance between exploration and exploitation is hard to achieve and requires a careful configuration of parameters. Future work includes the adaption of the noise parameters or even the noise itself via L2L. For example, when sampling from a Gaussian distribution the parameter σ, which controls the standard variation, can be adjusted with L2L. Another parameter to optimize is the number of replaced individuals. In this setting, 10% of the best individuals replaced 10% of the worst individuals. Depending on the network performance these numbers could be adapted automatically by L2L.

To further gain insight for a follow-up parameter analysis and leverage optimization, a visualization through generations may prove to be useful. A visualization tool which can plot the evolution of the parameters using simple diagrams such as histograms, correlations and similar statistics can be helpful to create a better understanding of the optimization. As demonstrated by Tensorboard[1], a desirable feature would be to interact with the plot while the simulation is ongoing.

An interesting approach is to extend the capabilities of the optimization by using neural networks as optimizers. This method was proposed by Andrychowicz et al. (2016), where the outer loop recurrent neural network (RNN) was learning the gradients of the inner loop RNN. In the setting of my work, the outer loop network could learn the distribution of the parameter space of the inner loop reservoir and predict the next set of parameters. For example, generative models, such as autoencoders, are able to approximate the high-dimensional distribution of the parameter space and create new samples by estimating the likelihood of each observation or input (Ruthotto et al. 2021). Such a model could be the optimizer in the outer loop. One other interesting direction is to search for an optimal network architecture for a specific task or for several tasks. This concept, known as neural architecture search (NAS, Zoph and Q. Le 2017) or neuroevolution (Stanley 2017), uses genetic algorithms to optimize the number of layers and neurons of a neural network. In the case of the reservoir network, the number of excitatory and inhibitory neurons inside the reservoir and output clusters, as well as the connections to the output can be optimized in order to enhance the separation and approximation properties of the network.

In conclusion, I presented an optimization scheme applied on spiking and artificial networks within the framework of meta-learning using metaheuristics. With this work I leverage op-

[1] https://www.tensorflow.org/tensorboard/

timization applied on different types of networks for scientists working in the intersection of neuroscience and AI. Finally, I have displayed the positive effects of cross-fertilization between both fields regarding gradient-free optimization.

List of Figures

Bibliography

Andrychowicz, Marcin, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas (2016). "Learning to learn by gradient descent by gradient descent". In: *Advances in neural information processing systems*, pp. 3981–3989 (cit. on pp. 74, 120).

Azulay, Aharon and Yair Weiss (2018). "Why do deep convolutional networks generalize so poorly to small image transformations?" In: *arXiv preprint arXiv:1805.12177* (cit. on p. 22).

Baddeley, Alan (2003). "Working memory: looking back and looking forward". In: *Nature reviews neuroscience* 4.10, pp. 829–839 (cit. on p. 34).

Bandeira de Melo, Elton Bernardo and Aluízio Fausto Ribeiro Araújo (2008). "Modeling ant colony foraging in dynamic and confined environment". In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pp. 169–176 (cit. on p. 102).

Bartunov, Sergey, Adam Santoro, Blake Richards, Luke Marris, Geoffrey E Hinton, and Timothy Lillicrap (2018). "Assessing the scalability of biologically-motivated deep learning algorithms and architectures". In: *Advances in neural information processing systems* 31 (cit. on pp. 44, 115).

Beaulieu, Shawn, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O Stanley, Jeff Clune, and Nick Cheney (2020). "Learning to continually learn". In: *arXiv preprint arXiv:2002.09571* (cit. on p. 119).

Bellec, Guillaume, Franz Scherr, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass (2019). "Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets". In: *arXiv preprint arXiv:1901.09049* (cit. on p. 45).

Bengio, Samy, Yoshua Bengio, and Jocelyn Cloutier (1995). "On the search for new learning rules for ANNs". In: *Neural Processing Letters* 2.4, pp. 26–30 (cit. on p. 74).

Bengio, Y., P. Frasconi, and P. Simard (Mar. 1993). "The problem of learning long-term dependencies in recurrent networks". In: *IEEE International Conference on Neural Networks*, 1183–1188 vol.3. DOI: 10.1109/ICNN.1993.298725 (cit. on p. 54).

Bengio, Y., P. Simard, and P. Frasconi (Mar. 1994). "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166. ISSN: 1941-0093. DOI: 10.1109/72.279181 (cit. on p. 54).

Bengio, Yoshua, Samy Bengio, and Jocelyn Cloutier (1990). *Learning a synaptic learning rule*. Citeseer (cit. on p. 74).

Bengio, Yoshua, Dong-Hyun Lee, Jorg Bornschein, Thomas Mesnard, and Zhouhan Lin (2015). "Towards biologically plausible deep learning". In: *arXiv preprint arXiv:1502.04156* (cit. on pp. 11, 33).

Bengio, Yoshua, Thomas Mesnard, Asja Fischer, Saizheng Zhang, and Yuhuai Wu (2017). "STDP-compatible approximation of backpropagation in an energy-based model". In: *Neural computation* 29.3, pp. 555–577 (cit. on p. 33).

Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.Feb, pp. 281–305. ISSN: ISSN 1533-7928 (cit. on p. 75).

Bishop, Christopher M. (2007). *Pattern Recognition and Machine Learning, 5th Edition*. Information science and statistics. Springer, pp. I–XX, 1–738. ISBN: 9780387310732 (cit. on pp. 19, 22, 40).

Blum, Christian and Andrea Roli (2003). "Metaheuristics in combinatorial optimization: Overview and conceptual comparison". In: *ACM computing surveys (CSUR)* 35.3, pp. 268–308 (cit. on p. 46).

Brazdil, Pavel, Jan N van Rijn, Carlos Soares, and Joaquin Vanschoren (2022). *Metalearning: Applications to automated machine learning and data mining*. Springer Nature (cit. on p. 77).

Bulatov, Yaroslav (Feb. 2018). *notMNIST*. Kaggle dataset. URL: https://www.kaggle.com/jwjohnson314/notmnist#notMNIST_large (cit. on p. 57).

Buonomano, Dean V and Wolfgang Maass (2009). "State-dependent computations: spatiotemporal processing in cortical networks". In: *Nature Reviews Neuroscience* 10.2, pp. 113–125 (cit. on p. 29).

Cao, Yue, Tianlong Chen, Zhangyang Wang, and Yang Shen (2019). "Learning to Optimize in Swarms". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2019/file/ec04e8ebba7e132043e5b4832e54f070-Paper.pdf (cit. on p. 75).

Chevallier, Sylvain, Hélène Paugam-Moisy, and Michèle Sebag (2010). "SpikeAnts, a spiking neuron network modelling the emergence of organization in a complex system". In: *NIPS'2010*, pp. 379–387 (cit. on p. 103).

Chowdhery, Aakanksha et al. (2022). "Palm: Scaling language modeling with pathways". In: *arXiv preprint arXiv:2204.02311* (cit. on p. 12).

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2015). "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (cit. on p. 21).

Crick, Francis (1989). "The recent excitement about neural networks." In: *Nature* 337.6203, pp. 129–132 (cit. on p. 44).

Dayan, Peter and Laurence F Abbott (2005). *Theoretical neuroscience: computational and mathematical modeling of neural systems*. Vol. 11. MIT press, pp. v–vi. DOI: 10.1016/j.cobeha.2016.07.008 (cit. on p. 26).

DiCarlo, James J, Davide Zoccolan, and Nicole C Rust (2012). "How does the brain solve visual object recognition?" In: *Neuron* 73.3, pp. 415–434 (cit. on pp. 11, 32).

Doumas, Leonidas AA, John E Hummel, and Catherine M Sandhofer (2008). "A theory of the discovery and predication of relational concepts." In: *Psychological review* 115.1, p. 1 (cit. on p. 34).

Duan, Haibin and Changhao Sun (2014). "Swarm intelligence inspired shills and the evolution of cooperation". In: *Scientific reports* 4.1, pp. 1–8 (cit. on p. 102).

Duan, Yan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel (Nov. 2016). "RL2: Fast Reinforcement Learning via Slow Reinforcement Learning". In: *arXiv:1611.02779 [cs, stat]*. arXiv: 1611.02779 (cit. on p. 74).

Elman, Jeffrey L (1990). "Finding structure in time". In: *Cognitive science* 14.2, pp. 179–211 (cit. on p. 26).

Erhan, Dumitru, Yoshua Bengio, Aaron Courville, and Pascal Vincent (2009). "Visualizing higher-layer features of a deep network". In: *University of Montreal* 1341.3, p. 1 (cit. on p. 22).

Evensen, Geir (1994). "Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics". In: *Journal of Geophysical Research: Oceans* 99.C5, pp. 10143–10162 (cit. on p. 49).

Finn, Chelsea, Pieter Abbeel, and Sergey Levine (2017). "Model-agnostic meta-learning for fast adaptation of deep networks". In: *International Conference on Machine Learning*. PMLR, pp. 1126–1135 (cit. on p. 74).

Finn, Chelsea and Sergey Levine (Oct. 2017). "Meta-Learning and Universality: Deep Representations and Gradient Descent Can Approximate Any Learning Algorithm". In: *arXiv:1710.11622 [cs]*. arXiv: 1710.11622 (cit. on p. 74).

Finn, Chelsea, Aravind Rajeswaran, Sham Kakade, and Sergey Levine (2019). "Online meta-learning". In: *International Conference on Machine Learning*. PMLR, pp. 1920–1930 (cit. on p. 75).

Finn, Chelsea, Kelvin Xu, and Sergey Levine (2018). "Probabilistic model-agnostic meta-learning". In: *Advances in neural information processing systems* 31 (cit. on p. 75).

Gil Fernández, Vanessa, Sara Nocentini, and José Antonio del Río Fernández (2014). "Historical first descriptions of Cajal-Retzius cells: from pioneer studies to current knowledge". In: *Frontiers In Neuroanatomy, 2014, vol. 8, num. 32* (cit. on p. 23).

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics* (cit. on p. 44).

Goldman-Rakic, Patricia S (1991). "Cellular and circuit basis of working memory in prefrontal cortex of nonhuman primates". In: *Progress in brain research* 85, pp. 325–336 (cit. on p. 34).

Gonon, Lukas and Juan-Pablo Ortega (2019). "Reservoir computing universality with stochastic inputs". In: *IEEE transactions on neural networks and learning systems* 31.1, pp. 100–112 (cit. on p. 29).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press (cit. on pp. 21, 22, 43).

Grossberg, Stephen (1987). "Competitive learning: From interactive activation to adaptive resonance". In: *Cognitive science* 11.1, pp. 23–63 (cit. on p. 44).

Guerguiev, Jordan, Timothy P Lillicrap, and Blake A Richards (2017). "Towards deep learning with segregated dendrites". In: *ELife* 6, e22901 (cit. on p. 11).

Guo, Wenzhe, Mohammed E Fouda, Ahmed M Eltawil, and Khaled Nabil Salama (2021). "Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems". In: *Frontiers in Neuroscience* 15, p. 638474 (cit. on p. 81).

Harlow, Harry F (1949). "The formation of learning sets." In: *Psychological review* 56.1, p. 51 (cit. on p. 34).

Hassabis, Demis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick (2017). "Neuroscience-inspired artificial intelligence". In: *Neuron* 95.2, pp. 245–258 (cit. on pp. 11, 30, 32, 115).

Hasson, Uri, Samuel A Nastase, and Ariel Goldstein (2020). "Direct fit to nature: an evolutionary perspective on biological and artificial neural networks". In: *Neuron* 105.3, pp. 416–434 (cit. on p. 11).

Hayou, Soufiane, Arnaud Doucet, and Judith Rousseau (2019). "On the impact of the activation function on deep neural networks training". In: *International conference on machine learning*. PMLR, pp. 2672–2680 (cit. on p. 54).

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034 (cit. on p. 44).

He, Xin, Kaiyong Zhao, and Xiaowen Chu (2021). "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212, p. 106622 (cit. on p. 75).

Hecker, Joshua P and Melanie E Moses (2015). "Beyond pheromones: evolving error-tolerant, flexible, and scalable ant-inspired robot swarms". In: *Swarm Intelligence* 9.1, pp. 43–70 (cit. on p. 102).

Hermans, Michiel and Benjamin Schrauwen (2012). "Recurrent kernel machines: Computing with infinite echo state networks". In: *Neural Computation* 24.1, pp. 104–133 (cit. on p. 28).

Herty, Michael and Giuseppe Visconti (2019). "Kinetic methods for inverse problems". In: *Kinetic & Related Models* 12.19375093_2019_5_1109, p. 1109. ISSN: 19375093 (cit. on p. 49).

Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. (2001). *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies* (cit. on p. 54).

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780 (cit. on p. 34).

Hochreiter, Sepp, A. Steven Younger, and Peter R. Conwell (Aug. 2001). "Learning to Learn Using Gradient Descent". In: *Artificial Neural Networks — ICANN 2001*. Springer, Berlin, Heidelberg, pp. 87–94. DOI: 10.1007/3-540-44668-0_13 (cit. on p. 74).

Holland, John H (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press (cit. on p. 47).

Huh, Dongsung and Terrence J Sejnowski (2018). "Gradient descent for spiking neural networks". In: *Advances in neural information processing systems* 31 (cit. on p. 45).

Hutter, Frank, Lars Kotthoff, and Joaquin Vanschoren, eds. (2019). *Automated Machine Learning - Methods, Systems, Challenges*. Springer (cit. on p. 75).

Iba, Hitoshi and Nasimul Noman (2020). *Deep Neural Evolution*. Springer (cit. on p. 49).

Iglesias, Marco A, Kody J H Law, and Andrew M Stuart (Mar. 2013). "Ensemble Kalman methods for inverse problems". In: *Inverse Problems* 29.4, p. 045001. DOI: 10.1088/0266-5611/29/4/045001. URL: https://doi.org/10.1088/0266-5611/29/4/045001 (cit. on pp. 49, 50).

Jaderberg, Max et al. (2017). "Population based training of neural networks". In: *arXiv preprint arXiv:1711.09846* (cit. on p. 75).

Jaeger, Herbert (2001). "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note". In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148.34, p. 13 (cit. on p. 27).

Jäkel, Sarah and Leda Dimou (2017). "Glial cells and their function in the adult brain: a journey through the history of their ablation". In: *Frontiers in cellular neuroscience* 11, p. 24 (cit. on p. 24).

Janjic, T., D. McLaughlin, S. E. Cohn, and M. Verlaan (2014). "Conservation of Mass and Preservation of Positivity with Ensemble-Type Kalman Filter Algorithms". In: *Monthly Weather Review* 142.2, pp. 755–773 (cit. on p. 49).

Jarrett, Kevin, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun (2009). "What is the best multi-stage architecture for object recognition?" In: *2009 IEEE 12th International Conference on Computer Vision*, pp. 2146–2153. DOI: 10.1109/ICCV.2009.5459469 (cit. on p. 21).

Jordan, Jakob, Tammo Ippen, Moritz Helias, Itaru Kitayama, Mitsuhisa Sato, Jun Igarashi, Markus Diesmann, and Susanne Kunkel (2018). "Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers". In: *Frontiers in neuroinformatics* 12, p. 2 (cit. on pp. 12, 29).

Jordan, Michael I (1997). "Serial order: A parallel distributed processing approach". In: *Advances in psychology*. Vol. 121. Elsevier, pp. 471–495 (cit. on p. 26).

Katzfuss, Matthias, Jonathan R Stroud, and Christopher K Wikle (2016). "Understanding the ensemble Kalman filter". In: *The American Statistician* 70.4, pp. 350–357 (cit. on pp. 49, 50).

Kennedy, James and Russell Eberhart (1995). "Particle swarm optimization". In: *Proceedings of ICNN'95-international conference on neural networks*. Vol. 4. IEEE, pp. 1942–1948 (cit. on p. 75).

Keskar, Nitish Shirish and Richard Socher (2017). "Improving generalization performance by switching from adam to sgd". In: *arXiv preprint arXiv:1712.07628* (cit. on p. 42).

Khaligh-Razavi, Seyed-Mahdi and Nikolaus Kriegeskorte (2014). "Deep supervised, but not unsupervised, models may explain IT cortical representation". In: *PLoS computational biology* 10.11, e1003915 (cit. on p. 11).

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (cit. on pp. 41, 42).

Kirkpatrick, James et al. (2017). "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the national academy of sciences* 114.13, pp. 3521–3526 (cit. on p. 119).

Kovachki, Nikola B. and Andrew M. Stuart (Aug. 10, 2018). "Ensemble Kalman Inversion: A Derivative-Free Technique For Machine Learning Tasks". In: arXiv: http://arxiv.org/abs/1808.03620v1 [cs.LG] (cit. on pp. 51, 55, 56, 68, 71).

Lake, Brenden M, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman (2017). "Building machines that learn and think like people". In: *Behavioral and brain sciences* 40 (cit. on pp. 11, 115).

Larochelle, Hugo and Geoffrey E Hinton (2010). "Learning to combine foveal glimpses with a third-order Boltzmann machine". In: *Advances in neural information processing systems* 23 (cit. on p. 33).

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998). "Gradient-based learning applied to document recognition". In: 86, pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791 (cit. on pp. 43, 64).

LeCun, Yann, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4, pp. 541–551 (cit. on pp. 21, 39).

LeCun, Yann, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller (2012). "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, pp. 9–48 (cit. on pp. 20, 43).

LeCun, Yann, Corinna Cortes, and CJ Burges (2010). "MNIST handwritten digit database". In: *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2, p. 18 (cit. on pp. 21, 54).

Lillicrap, Timothy P, Daniel Cownden, Douglas B Tweed, and Colin J Akerman (2016). "Random synaptic feedback weights support error backpropagation for deep learning". In: *Nature communications* 7.1, pp. 1–10 (cit. on p. 44).

Lobo, Jesus L., Javier Del Ser, Albert Bifet, and Nikola Kasabov (2020). "Spiking Neural Networks and online learning: An overview and perspectives". In: 121, pp. 88–100. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2019.09.004 (cit. on pp. 25, 115).

Maass, Wolfgang (2011). "Liquid state machines: motivation, theory, and applications". In: *Computability in context: computation and logic in the real world*, pp. 275–296 (cit. on p. 29).

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 pp. 27–29).

MacKay, David J. C. (2003). *Information Theory, Inference, and Learning Algorithms*. Copyright Cambridge University Press (cit. on p. 68).

Malik, Hasmat, Atif Iqbal, Puneet Joshi, Sanjay Agrawal, and Farhad Ilahi Bakhsh (2021). *Metaheuristic and evolutionary computation: algorithms and applications*. Springer (cit. on pp. 38, 39).

Marblestone, Adam H, Greg Wayne, and Konrad P Kording (2016). "Toward an integration of deep learning and neuroscience". In: *Frontiers in computational neuroscience*, p. 94 (cit. on pp. 12, 115).

Marr, D. and T. Poggio (1976). *From Understanding Computation to Understanding Neural Circuitry*. Tech. rep. USA (cit. on p. 30).

Martín Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/ (cit. on p. 41).

McCloskey, Michael and Neal J Cohen (1989). "Catastrophic interference in connectionist networks: The sequential learning problem". In: *Psychology of learning and motivation*. Vol. 24. Elsevier, pp. 109–165 (cit. on pp. 33, 119).

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133 (cit. on p. 17).

Mirikitani, Derrick T. and Nikolay Nikolaev (2008). "Dynamic Modeling with Ensemble Kalman Filter Trained Recurrent Neural Networks". In: *2008 Seventh International Conference on Machine Learning and Applications*. IEEE. DOI: 10.1109/icmla.2008.79 (cit. on p. 55).

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *nature* 518.7540, pp. 529–533 (cit. on p. 33).

Myburgh, Johannes C, Coenraad Mouton, and Marelie H Davel (2021). "Tracking translation invariance in CNNs". In: *Southern African Conference for Artificial Intelligence Research*. Springer, pp. 282–295 (cit. on p. 22).

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *Icml* (cit. on p. 21).

Neftci, Emre O, Hesham Mostafa, and Friedemann Zenke (2019). "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks". In: *IEEE Signal Processing Magazine* 36.6, pp. 51–63 (cit. on p. 45).

Nicholas, I, Hsien Kuo, Mehrtash Harandi, Nicolas Fourrier, Christian Walder, Gabriela Ferraro, and Hanna Suominen (2021). "Learning to continually learn rapidly from few and noisy data". In: *AAAI Workshop on Meta-Learning and MetaDL Challenge*. PMLR, pp. 65–76 (cit. on p. 119).

Nikolić, Danko, Stefan Häusler, Wolf Singer, and Wolfgang Maass (2009). "Distributed fading memory for stimulus properties in the primary visual cortex". In: *PLoS biology* 7.12, e1000260 (cit. on p. 29).

Nøkland, Arild (2016). "Direct feedback alignment provides learning in deep neural networks". In: *Advances in neural information processing systems* 29 (cit. on p. 44).

O'Neill, Joseph, Barty Pleydell-Bouverie, David Dupret, and Jozsef Csicsvari (2010). "Play it again: reactivation of waking experience and memory". In: *Trends in neurosciences* 33.5, pp. 220–229 (cit. on p. 34).

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (June 2013). "On the difficulty of training recurrent neural networks". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, pp. 1310–1318. URL: https://proceedings.mlr.press/v28/pascanu13.html (cit. on p. 43).

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on pp. 29, 41).

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830 (cit. on p. 86).

Pennington, Jeffrey, Samuel Schoenholz, and Surya Ganguli (2017). "Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice". In: *Advances in neural information processing systems*, pp. 4785–4795 (cit. on p. 43).

Ratcliff, Roger (1990). "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions." In: *Psychological review* 97.2, p. 285 (cit. on p. 33).

Ravi, Sachin and Hugo Larochelle (2017). "Optimization as a model for few-shot learning". In: *In International Conference on Learning Representations (ICLR)* (cit. on p. 74).

Rendell, Larry, Raj Seshu, and David Tcheng (1987). "More Robust Concept Learning Using Dynamically–Variable Bias". In: *Proceedings of the Fourth International Workshop on Machine Learning*. Elsevier, pp. 66–78 (cit. on p. 74).

Rice, John R (1976). "The algorithm selection problem". In: *Advances in computers*. Vol. 15. Elsevier, pp. 65–118 (cit. on p. 74).

Richards, Blake A et al. (2019). "A deep learning framework for neuroscience". In: *Nature neuroscience* 22.11, pp. 1761–1770 (cit. on pp. 11, 12, 14, 115).

Rosenblatt, Frank (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory (cit. on p. 17).

Rumelhart, David E, Richard Durbin, Richard Golden, and Yves Chauvin (1995). "Backpropagation: The basic theory". In: *Backpropagation: Theory, architectures and applications*, pp. 1–34 (cit. on pp. 18, 39).

Russell, Alexander, Garrick Orchard, Yi Dong, Ştefan Mihalaş, Ernst Niebur, Jonathan Tapson, and Ralph Etienne-Cummings (2010). "Optimization methods for spiking neurons and networks". In: *IEEE transactions on neural networks* 21.12, pp. 1950–1962. DOI: 10.1109/TNN.2010.2083685 (cit. on p. 12).

Ruthotto, Lars and Eldad Haber (2021). "An introduction to deep generative modeling". In: *GAMM-Mitteilungen* 44.2, e202100008. DOI: https://doi.org/10.1002/gamm.202100008. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/gamm.202100008. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.202100008 (cit. on p. 120).

Sacramento, João, Rui Ponte Costa, Yoshua Bengio, and Walter Senn (2018). "Dendritic Cortical Microcircuits Approximate the Backpropagation Algorithm". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., pp. 8735–8746 (cit. on p. 45).

Salimans, Tim, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever (2017). "Evolution strategies as a scalable alternative to reinforcement learning". In: *arXiv preprint arXiv:1703.03864* (cit. on pp. 48, 80).

Samadi, Arash, Timothy P Lillicrap, and Douglas B Tweed (2017). "Deep learning with dynamic spiking neurons and fixed feedback weights". In: *Neural computation* 29.3, pp. 578–602 (cit. on p. 44).

Schillings, C. and A. M. Stuart (2018). "Convergence analysis of ensemble Kalman inversion: the linear, noisy case". In: *Appl. Anal.* 97.1, pp. 107–123. ISSN: 0003-6811. DOI: 10.1080/00036811.2017.1386784. URL: https://doi.org/10.1080/00036811.2017.1386784 (cit. on p. 49).

Schmidhuber, Jürgen (1987). "Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook". PhD thesis. Technische Universität München (cit. on p. 74).

Schultz, Wolfram (1998). "Predictive reward signal of dopamine neurons". In: *Journal of neurophysiology* 80.1, pp. 1–27 (cit. on p. 33).

Schwenzer, Max, Sebastian Stemmler, Muzaffer Ay, Thomas Bergs, and Dirk Abel (2019). "Ensemble Kalman filtering for force model identification in milling". In: *Procedia CIRP* 82, pp. 296–301 (cit. on p. 49).

Shi, Zhiwei and Min Han (2007). "Support vector echo-state machine for chaotic time-series prediction". In: *IEEE transactions on neural networks* 18.2, pp. 359–372 (cit. on p. 28).

Simon, Dan (2013). *Evolutionary optimization algorithms*. John Wiley & Sons (cit. on p. 47).

Speck, Robert, Michael Knobloch, Sebastian Lührs, and Andreas Gocht (June 8, 2020). "Using Performance Analysis Tools for a Parallel-in-Time Integrator". In: *Parallel-in-Time Integration Methods*. Vol. 356. Springer Proceedings in Mathematics & Statistics. 9th Workshop on Parallel-in-Time Integration, online (online), 8 Jun 2020 - 12 Jun 2020. Cham: Springer International Publishing, pp. 51–80. ISBN: 978-3-030-75932-2 (print). DOI: 10.1007/978-3-030-75933-9_3. URL: https://juser.fz-juelich.de/record/901885 (cit. on p. 79).

Stanley, Kenneth O (2017). "Neuroevolution: A different kind of deep learning". In: *O'Reilly Media*. https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/ (cit. on p. 120).

Stanley, Kenneth O and Risto Miikkulainen (2002). "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2, pp. 99–127 (cit. on p. 75).

Surace, Simone Carlo, Jean-Pascal Pfister, Wulfram Gerstner, and Johanni Brea (2020). "On the choice of metric in gradient-based theories of brain function". In: *PLoS computational biology* 16.4, e1007640 (cit. on p. 12).

Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton (2013). "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning*, pp. 1139–1147 (cit. on p. 54).

Tarantola, Albert (2004). *Inverse Problem Theory and Methods for Model Parameter Estimation*. USA: Society for Industrial and Applied Mathematics. ISBN: 0898715725 (cit. on p. 55).

Tavanaei, Amirhossein, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida (2019). "Deep learning in spiking neural networks". In: *Neural networks* 111, pp. 47–63 (cit. on p. 115).

Thrun, Sebastian and Lorien Pratt (2012). *Learning to learn*. Springer Science & Business Media (cit. on pp. 73, 75).

– (1998). "Learning to learn: Introduction and overview". In: *Learning to learn*. Springer, pp. 3–17 (cit. on p. 74).

Tisue, Seth and Uri Wilensky (2004). "Netlogo: A simple environment for modeling complexity". In: *International conference on complex systems*. Vol. 21. Boston, MA, pp. 16–21 (cit. on p. 103).

Van Gerven, Marcel (2017). "Computational foundations of natural intelligence". In: *Frontiers in computational neuroscience*, p. 112 (cit. on p. 30).

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30 (cit. on p. 33).

Vittori, Karla, Jacques Gautrais, Aluizio FR Araújo, Vincent Fourcassié, and Guy Theraulaz (2004). "Modeling ant behavior under a variable environment". In: *International Workshop on Ant Colony Optimization and Swarm Intelligence*. Springer, pp. 190–201 (cit. on p. 102).

Von Bartheld, Christopher S, Jami Bahney, and Suzana Herculano-Houzel (2016). "The search for true numbers of neurons and glial cells in the human brain: A review of 150 years of cell counting". In: *Journal of Comparative Neurology* 524.18, pp. 3865–3895 (cit. on p. 23).

Wang, Jane X., Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick (Nov. 2016). "Learning to Reinforcement Learn". In: *arXiv:1611.05763 [cs, stat]*. arXiv: 1611.05763 (cit. on p. 74).

Whittington, James CR and Rafal Bogacz (2019). "Theories of error back-propagation in the brain". In: *Trends in cognitive sciences* 23.3, pp. 235–250 (cit. on p. 12).

Widrow, Bernard and Marcian E Hoff (1960). *Adaptive switching circuits*. Tech. rep. Stanford Univ Ca Stanford Electronics Labs (cit. on p. 17).

Wierstra, Daan, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber (2014). "Natural evolution strategies". In: *The Journal of Machine Learning Research* 15.1, pp. 949–980 (cit. on pp. 48, 80).

Wilensky, Uri (1997). "Netlogo ants model". In: *Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL* (cit. on pp. 102, 103, 106, 110).

Woźniak, Stanisław, Angeliki Pantazi, Thomas Bohnstingl, and Evangelos Eleftheriou (2020). "Deep learning incorporating biologically inspired neural dynamics and in-memory computing". In: *Nature Machine Intelligence* 2.6, pp. 325–336 (cit. on p. 45).

Xie, Di, Jiang Xiong, and Shiliang Pu (2017). "All you need is beyond a good init: Exploring better solution for training extremely deep convolutional neural networks with orthonormality and modulation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 6176–6185 (cit. on p. 54).

Yamazaki, Kashu, Viet-Khoa Vo-Ho, Darshan Bulsara, and Ngan Le (2022). "Spiking neural networks and their applications: A Review". In: *Brain Sciences* 12.7, p. 863 (cit. on p. 12).

Yamins, Daniel LK and James J DiCarlo (2016). "Using goal-driven deep learning models to understand sensory cortex". In: *Nature neuroscience* 19.3, pp. 356–365 (cit. on pp. 11, 31).

Yoo, Andy B, Morris A Jette, and Mark Grondona (2003). "Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing*. Springer, pp. 44–60 (cit. on p. 79).

Zador, Anthony M (2019). "A critique of pure learning and what artificial neural networks can learn from animal brains". In: *Nature communications* 10.1, pp. 1–7 (cit. on p. 115).

Zoph, Barret and Quoc Le (2017). "Neural Architecture Search with Reinforcement Learning". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=r1Ue8Hcxg (cit. on p. 120).

Zoph, Barret and Quoc V. Le (Nov. 2016). "Neural Architecture Search with Reinforcement Learning". In: *arXiv:1611.01578 [cs]*. arXiv: 1611.01578 (cit. on p. 75).

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 43
Algorithms for massively parallel generic *hp*-adaptive finite element methods
M. Fehling (2020), vii, 78 pp
ISBN: 978-3-95806-486-7
URN: urn:nbn:de:0001-2020071402

Band / Volume 44
The method of fundamental solutions for computing interior transmission eigenvalues
L. Pieronek (2020), 115 pp
ISBN: 978-3-95806-504-8

Band / Volume 45
Supercomputer simulations of transmon quantum computers
D. Willsch (2020), IX, 237 pp
ISBN: 978-3-95806-505-5

Band / Volume 46
The Influence of Individual Characteristics on Crowd Dynamics
P. Geoerg (2021), xiv, 212 pp
ISBN: 978-3-95806-561-1

Band / Volume 47
Structural plasticity as a connectivity generation and optimization algorithm in neural networks
S. Diaz Pier (2021), 167 pp
ISBN: 978-3-95806-577-2

Band / Volume 48
Porting applications to a Modular Supercomputer
Experiences from the DEEP-EST project
A. Kreuzer, E. Suarez, N. Eicker, Th. Lippert (Eds.) (2021), 209 pp
ISBN: 978-3-95806-590-1

Band / Volume 49
Operational Navigation of Agents and Self-organization Phenomena in Velocity-based Models for Pedestrian Dynamics
Q. Xu (2022), xii, 112 pp
ISBN: 978-3-95806-620-5

Band / Volume 50
Utilizing Inertial Sensors as an Extension of a Camera Tracking System for Gathering Movement Data in Dense Crowds
J. Schumann (2022), xii, 155 pp
ISBN: 978-3-95806-624-3

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 51
Final report of the DeepRain project
Abschlußbericht des DeepRain Projektes
(2022), ca. 70 pp
ISBN: 978-3-95806-675-5

Band / Volume 52
JSC Guest Student Programme Proceedings 2021
I. Kabadshow (Ed.) (2023), ii, 82 pp
ISBN: 978-3-95806-684-7

Band / Volume 53
Applications of variational methods for quantum computers
M. S. Jattana (2023), vii, 160 pp
ISBN: 978-3-95806-700-4

Band / Volume 54
Crowd Management at Train Stations in Case of Large-Scale Emergency Events
A. L. Braun (2023), vii, 120 pp
ISBN: 978-3-95806-706-6

Band / Volume 55
Gradient-Free Optimization of Artificial and Biological Networks using Learning to Learn
A. Yeğenoğlu (2023), II, 136 pp
ISBN: 978-3-95806-719-6

Weitere ***Schriften des Verlags im Forschungszentrum Jülich*** unter
http://wwwzb1.fz-juelich.de/verlagextern1/index.asp

IAS Series
Band / Volume 55
ISBN 978-3-95806-719-6

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum