Biologically Plausible Deep Learning through Neuroevolution

Master Thesis

in Computational Neuroscience

by Johannes Rieke

(johannes.rieke@gmail.com)

30 January 2020

Technical University of Berlin

Bernstein Center for Computational Neuroscience Berlin

Supervisors:

Prof. Dr. Matthew Larkum (HU Berlin)

Prof. Dr. Benjamin Grewe (ETH Zürich)

Table of contents

Statutory declaration

Die selbständige und eigenhändige Anfertigung versichert an Eides statt

Berlin, den

……
Unterschrift

Abstract

Recently, deep learning has been increasingly used as a framework to understand information processing in the brain. Most existing research in this area has focused on finding biologically plausible learning algorithms. In this thesis, we investigate how the architecture of a neural network interacts with such learning algorithms. Inspired by the intricate wiring of biological brains, we use an evolutionary algorithm to develop network architectures by mutating neurons and connections. Within the evolution loop, the synaptic weights of the networks are trained with one of four learning algorithms, which vary in complexity and biological plausibility (backpropagation, feedback alignment, Hebbian learning, last-layer learning). We investigate how the evolved network architectures differ between learning algorithms, both in terms of their performance on image classification with MNIST as well as their topologies. We show that for all learning algorithms, the evolved architectures learn much better than random networks – i.e. evolution and learning interact. Also, we find that more complex learning algorithms (e.g. backpropagation) can make use of larger, deeper, and more densely connected networks, which achieve higher performances. For simpler and biologically more plausible learning algorithms (e.g. Hebbian learning), however, the evolution process prefers smaller networks. Our results demonstrate that the architecture of a neural network plays an important role in its ability to learn. Using a range of learning algorithms, we show that evolutionary processes can find architectures of artificial neural networks that are particularly suited for learning – similar to how biological evolution potentially shaped the complex wiring of neurons in the brain.

Zusammenfassung

Deep Learning wurde in letzter Zeit vermehrt als Inspiration genutzt, um die Informationsverarbeitung im Gehirn zu verstehen. Die bestehende Forschung in diesem Bereich konzentriert sich meist darauf, biologisch plausible Lernalgorithmen zu finden. In dieser Arbeit untersuchen wir, wie die Architektur eines neuronalen Netzwerks mit solchen Lernalgorithmen interagiert. Inspiriert von der komplexen Verschaltung biologischer Gehirne verwenden wir einen evolutionären Algorithmus, um Netzwerkarchitekturen durch Mutation von Neuronen und Verbindungen zu entwickeln. Innerhalb des Evolutionsalgorithmus werden die synaptischen Gewichte der Netzwerke mit einem von vier Lernalgorithmen trainiert, die in Komplexität und biologischer Plausibilität variieren (Backpropagation, Feedback Alignment, Hebbian Learning, Last-Layer-Learning). Wir untersuchen, wie sich die entwickelten Netzwerkarchitekturen zwischen den Lernalgorithmen unterscheiden, sowohl hinsichtlich ihrer Leistung bei der Klassifizierung von Bildern mit MNIST als auch hinsichtlich ihrer Topologie. Wir zeigen, dass die per Evolution entwickelten Architekturen für alle Lernalgorithmen besser lernen als zufällige Netzwerke – Evolutions- und Lernprozesse interagieren also. Wir stellen weiterhin fest, dass komplexere Lernalgorithmen (z. B. Backpropagation) von größeren, tieferen und dichter verschalteten Netzwerke profitieren, die bessere Resultate erzielen. Für einfachere und biologisch plausiblere Lernalgorithmen (z. B. Hebbian Learning) bevorzugt der Evolutionsprozess jedoch kleinere Netzwerke. Unsere Ergebnisse zeigen, dass die Architektur eines neuronalen Netzwerks seine Fähigkeit zu lernen maßgeblich beeinflusst. Unter Verwendung verschiedener Lernalgorithmen zeigen wir, dass evolutionäre Prozesse Architekturen für künstliche neuronale Netzwerke finden können, die sich besonders zum Lernen eignen – ähnlich wie die biologische Evolution möglicherweise die komplexe Verschaltung von Neuronen im Gehirn entwickelte.

1. Introduction

1.1 Deep learning-inspired neuroscience

Over the past decade, the fields of artificial intelligence and machine learning have seen tremendous progress. Most of these advances were caused by deep learning, i.e. training deep neural networks on large datasets with the backpropagation algorithm (LeCun et al. 2015). Today, neural networks can easily recognize the content of an image (Krizhevsky et al. 2012), translate natural language at scale (Wu et al. 2016), or play complex board and computer games (Silver et al. 2016, Vinyals et al. 2019).

Deep learning has always been closely related to neuroscience: Major foundations of deep learning were directly inspired by information processing in the brain – from the early perceptron introduced by Rosenblatt (1958) to the use of convolution operations for computer vision problems (LeCun et al. 1999). Even today, deep learning regularly borrows ideas from neuroscience research, e.g. neuronal replay (Mnih et al. 2015) or short-term memory (Graves et al. 2014, Weston et al. 2015). While modern artificial neural networks can deviate quite far from their biological counterparts, the fundamental computational principles (simultaneous processing of information, multiple layers of abstraction, learning by altering connection strengths, ...) are still similar. Also, it was recently shown that representations observed in deep neural networks map quite well to neuronal representations in visual and auditory cortex (Yamins & DiCarlo 2016, Kell et al. 2018).

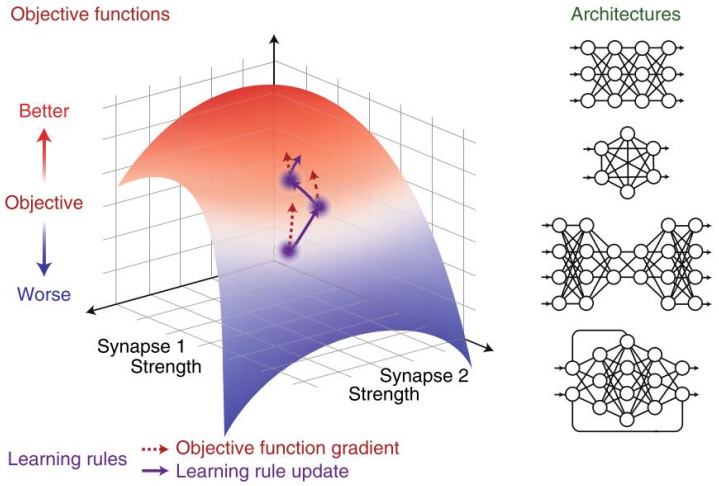

Figure 1: Deep learning as a framework for neuroscience.
Richards et al. (2019) argue that neuroscience research should take inspiration from three core components of deep learning: Objective functions (top left), learning rules (bottom left), and network architectures (right). From: Richards et al. (2019)

These similarities and the success of deep learning in artificial intelligence have recently led some researchers to propose deep learning as a guideline for future neuroscience research (Richards et al. 2019, Marblestone et al. 2016). Comparing artificial neural networks to the cortex might lead to a new understanding of processes in the brain and fill an important gap: Current neuroscience research usually focuses on computations at the level of small circuits (due to experimental limitations), or on the large-scale global activity of the brain (e.g. in cognitive neuroscience). Deep learning may offer a framework “in-between” and explain how large networks in the brain compute and learn. Richards et al. (2019) recently proposed three areas where neuroscience can take inspiration from deep learning (fig. 1): Objective functions (which quantify the performance of a network on a given task and determine what is actually learnt); learning rules (which specify how the synaptic weights of the network change during learning); and network architectures.

So far, most research in this area has focused on developing new learning rules. The backpropagation algorithm (Rumelhart et al. 1986), the workhorse of basically all recent deep learning research, poses some challenging problems for an implementation in the brain (chapter 2.2.2). Various recent papers have attempted to find biologically plausible alternatives, ranging from simply lifting some constraints of backpropagation (Lillicrap et al. 2016, Akrout et al. 2019), over finding alternative methods that avoid gradient backpropagation (Lee et al. 2015, Scellier & Bengio 2017), to proposing models based on computational primitives in the cortex (Sacramento et al. 2018, Guerguiev et al. 2017).

1.2 Network architecture as an inductive bias for learning

In this thesis, we investigate how such learning algorithms interact with network architectures. As pointed out by Richards et al. (2019), the architecture of a neural network (i.e. the configuration of connections and neurons) plays an important role in its ability to learn and process information.

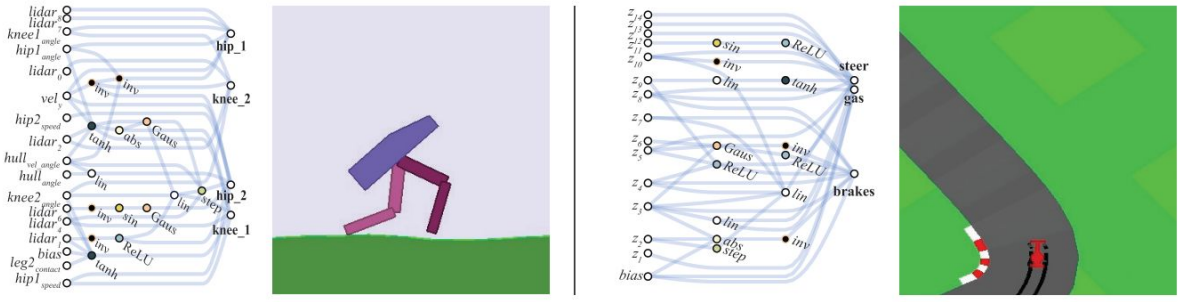

Figure 2: Weight agnostic neural networks for two reinforcement learning tasks.
Left: This network maneuvers a two-legged walker through a difficult terrain. **Right:** This network steers a car on a racing track. Both network architectures were found through neuroevolution, without any form of weight training. Instead, all connections use a single shared weight value. From: Gaier & Ha (2019)

In nature, we find two indicators for the importance of network architectures: First, the wiring in intelligent brains is usually very intricate. The human brain, e.g., consists of close to 100 million neurons (Herculano-Houzel 2009). Its largest structure, the neocortex, appears to be made up of repeated columns of neurons, which have a specific connectivity pattern (da Costa & Martin 2010). This intricate architecture is very different from a random or feedforward neural network. It was developed and fine-tuned during millions of years of evolution. As such, the architecture of the brain most likely already stores a lot of information in itself and can serve as a strong inductive bias for learning (in the sense of optimizing synaptic weights; Richards et al. 2019, Zador 2019). Second, many animals can perform complex behaviors right after birth: Ducks are able to swim and eat shortly after hatching, newborn mice react to dangerous stimuli, and human infants innately recognize faces (Zador 2019, Marcus 2018). These innate behaviors cannot rely on synaptic plasticity and learning from data (as in deep learning). It is likely that they are to some part encoded in the wiring of neural networks instead of synaptic weight configurations.

How powerful can artificial neural networks be by relying on a complex architecture? Inspired by innate behaviors of animals, Gaier & Ha (2019) investigated this question with so-called weight agnostic neural networks. Instead of training synaptic weights, these networks use a single, fixed weight value for all connections. Therefore, the output of the network depends mostly on its architecture, not on the weight configuration or a learning algorithm. Gaier & Ha then use an evolutionary algorithm to develop the architectures of these weight agnostic networks. Surprisingly, they find that a fine-tuned architecture alone can already master fairly complex tasks, such as several tasks for reinforcement learning (fig. 2) or image classification on the MNIST dataset. We want to emphasize again that these networks have never seen any weight training; all of their capabilities are encoded within their architecture.

1.3 Our approach: Combining evolution and learning

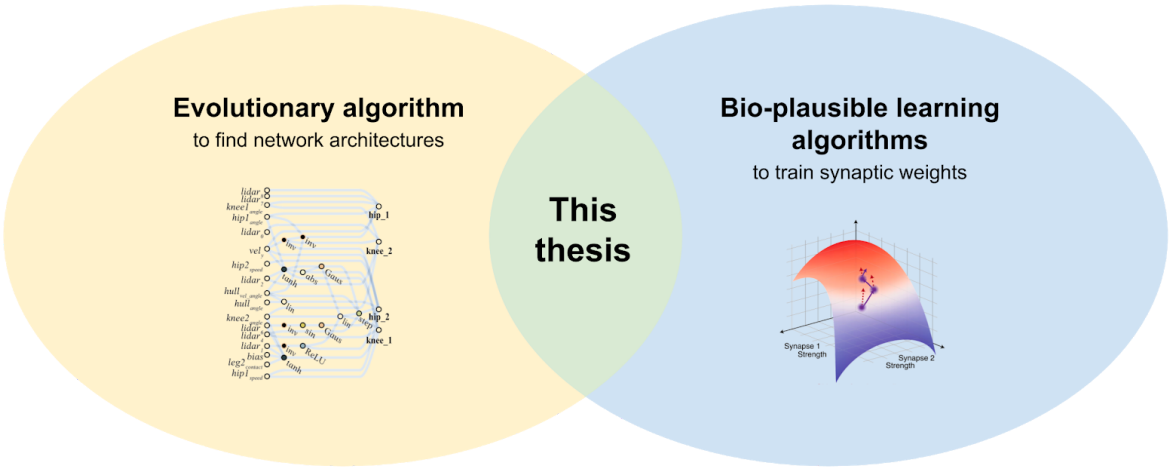

Figure 3: Approach of this thesis.
Small figures from Gaier & Ha (2019) and Richards et al. (2019).

In this thesis, we combine evolved network architectures with learning algorithms (fig. 3): Just like in Gaier & Ha (2019), we use an evolutionary approach to find network architectures through selection and mutation. Within this outer evolutionary loop, we train the synaptic weights of the evolved networks using one of four learning algorithms, which differ in their complexity and biological plausibility. Namely, we use backpropagation (Rumelhart et al. 1986), feedback alignment (Lillicrap et al. 2016), Hebbian learning (Hebb 1949), and last-layer learning. Our approach makes the evolutionary algorithm select for network topologies that are well-suited for learning. We analyze how the evolved networks differ between learning algorithms, both in terms of architecture as well as image classification performance on the MNIST dataset.

In effect, our approach is similar to what actually happens in biology: While the basic building plan of a biological brain is encoded in the genome and therefore subject to biological evolution, synaptic weights are highly plastic and change within an animal's lifetime. Also, our approach is closely related to a field of machine learning research called neural architecture search (NAS). The goal of NAS is to find neural network architectures that are optimal for a specific task. Similar to this thesis, some approaches combine evolutionary algorithms with a learning algorithm. However, they usually use only the backpropagation algorithm and have different goals and methods (chapter 2.3).

The thesis is structured as follows: First, we give an overview of the theoretical background and literature. As this thesis relates to a lot of different areas in neuroscience and machine learning, chapter 2 introduces a few select topics and shows how they relate to our approach. This includes some basic knowledge of the cortex (chapter 2.1), recent advances on biologically plausible learning algorithms (chapter 2.2), and related methods in neuroevolution (chapter 2.3). Chapter 3 presents our approach and discusses both the evolutionary algorithm and the learning algorithms we use. In chapter 4, we present the results of the thesis – first, we perform some experiments to replicate the results of Gaier & Ha (2019) on weight agnostic networks, then, we combine evolution and learning and investigate how they interact. We conclude with some general insights and a discussion of future research (chapter 5).

Our implementation is available at https://github.com/jrieke/evolution-learning.

2. Background

2.1 Architecture and learning in the cortex

The ultimate goal of biologically plausible deep learning is to understand the brain. Therefore, we give a short overview of information processing and learning in the brain. Specifically, we look at its basic architecture and building blocks, as well as some computational primitives and mechanisms for learning. This knowledge is also important for some of the learning algorithms in chapter 2.2. We assume the reader to be familiar with the basics of neurons, synapses, and the nervous system.

The analysis here focuses on the mammalian (neo-)cortex. For a discussion of cortical structures in other animals, see Nieuwenhuys (1994). The neocortex is the largest brain area in highly developed mammals. In humans, it makes up 80 % of brain volume and covers almost the entire surface area (Kaas 2011). It is the most versatile and flexible brain tissue and the seat of most “higher” functions of intelligence, such as language, planning or reasoning. Fig. 4 shows its anatomy and basic functional areas. Unlike more primitive brain regions, its architecture and wiring are not highly specialized to a single function. These properties make the neocortex the ideal analogy for artificial neural networks, which can learn diverse, highly complex functions from scratch.

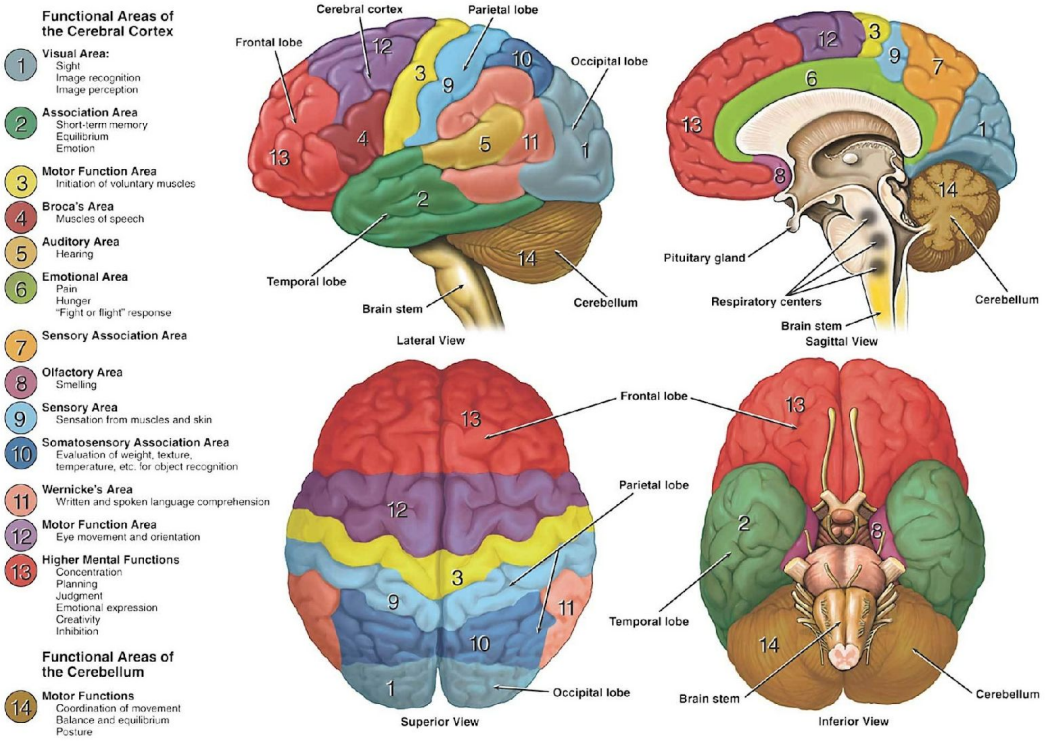

Figure 4: Anatomy and functions of the human cortex. From: Sukel (2019)

2.1.1 Cortical columns and circuits

A popular idea to describe the architecture of the cortex is the columnar hypothesis. It claims that the cortex is organized in vertical “columns”. These cortical columns span all six cortical layers and repeat across the entire cortical surface. They serve as the primary unit of computation and present a general-purpose architecture to process incoming signals. The columnar hypothesis is primarily based on the observation by Mountcastle (1957) that neurons of somatosensory cortex with similar functional properties are arranged in radial, vertical columns. Since Mountcastle’s original idea, many functional and anatomical entities in the cortex have been associated with a columnar organization (e.g. orientation-selective columns and ocular dominance columns in primary visual cortex, Hubel & Wiesel 1972). While the exact identity of cortical columns may be debatable (Horton & Adams 2005), most research today agrees that the cortex is at least on some level organized into small, vertical, modular units of computation (da Costa & Martin 2010, Douglas & Martin 2007).

If the cortex consists of modular computational units, what does such a unit look like? Each column is composed of two types of neurons: Pyramidal neurons and interneurons (fig. 5). Pyramidal neurons are the primary neurons of the neocortex and make up around 70 % of its neurons (Nieuwenhuys 1994). They are always excitatory and usually project long axons to other cortical columns or brain areas (see red axons in fig. 5 left). Pyramidal neurons constitute the column’s largest input and sole output system. The anatomy and function of interneurons are much more diverse: A single column contains many different types of interneurons – they can be inhibitory or excitatory and vary in shape and size (Markram et al. 2004, Nieuwenhuys 1994). Fig. 5 (right) shows some examples. Usually, interneurons project only within a cortical column and play specific roles in regulating neuronal activity.

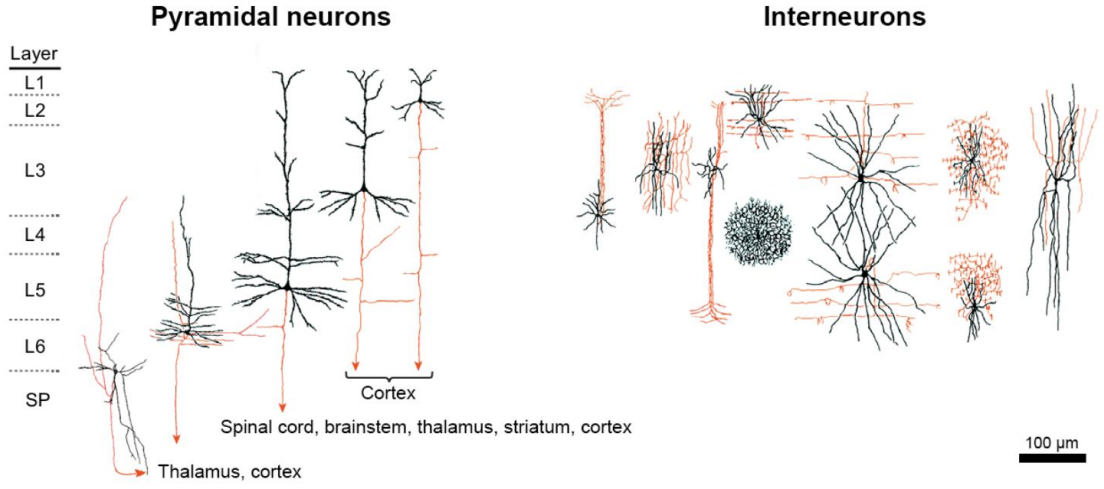

Figure 5: Neuron types in the cortex. Dendrites are shown in black, axons are shown in red. **Left**: Pyramidal neurons of different layers and their projection targets. **Right:** Different interneuron types (from left to right: stellate neuron, cell with axonal arcades, double bouquet cell, basket cell (top) and neurogliaform cell (bottom), basket cells, chandelier cells, bitufted cell). Adapted from: Kwan et al. (2012)

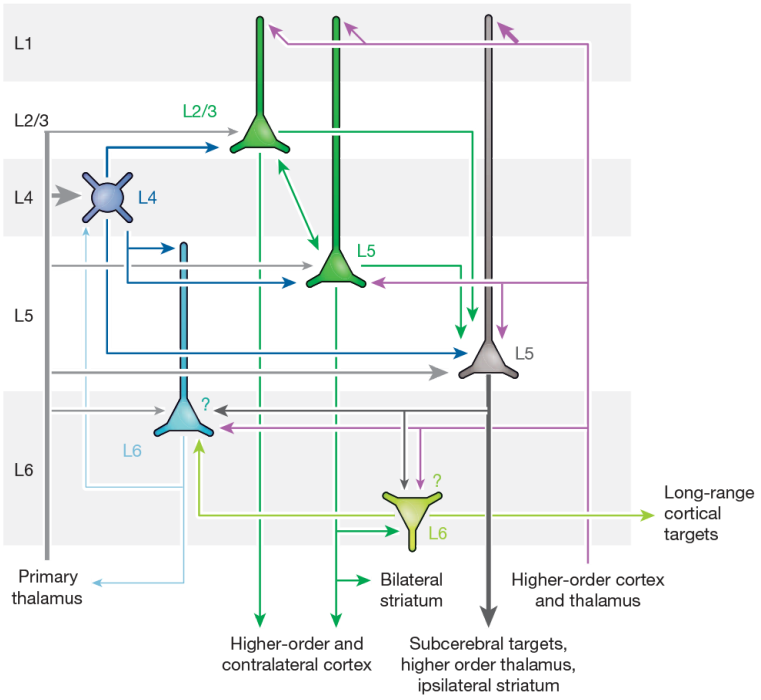

Figure 6: Connectivity of the canonical circuit. Questions marks indicate connections that are not conclusively verified. Adapted from: Harris & Mrsic-Flogel (2013)

Based on the idea of cortical columns, Douglas et al. (1989) and Douglas & Martin (2004) suggested a canonical circuit for the cortex (see also reviews in Roelfsema & Holtmat 2018 and Harris & Mrsic-Flogel 2013). In their model, each column integrates a combination of feedforward signals (from hierarchically lower cortical areas, e.g. sensory cortex) and feedback signals (from hierarchically higher cortical areas). Fig. 6 shows a schematic of the connectivity. Input signals from hierarchically lower areas (or sensory information relayed by the thalamus) arrive at all layers but primarily layer 4 (L4). From here, the signal is propagated throughout the column – most strongly to L2/3 and then to L5. In both layers, it terminates on basal dendrites of pyramidal cells. Feedback connections from hierarchically higher areas arrive primarily in L1 and target apical tuft dendrites of L2/3 and L5 pyramidal neurons. These pyramidal neurons integrate feedforward and feedback signals and send outputs to other columns or brain areas.

2.1.2 Computation in pyramidal neurons

As described above, pyramidal neurons play the main role in signal processing in the cortex. They integrate feedforward and feedback signals from other columns and areas and send out new signals. In the deep learning analogy, it is pretty clear that pyramidal neurons are the cells that should be associated with neurons in artificial neural networks (Richards & Lillicrap 2018; note that interneurons play vital computational roles as well). Therefore, we briefly discuss the computational principles they use to integrate signals.

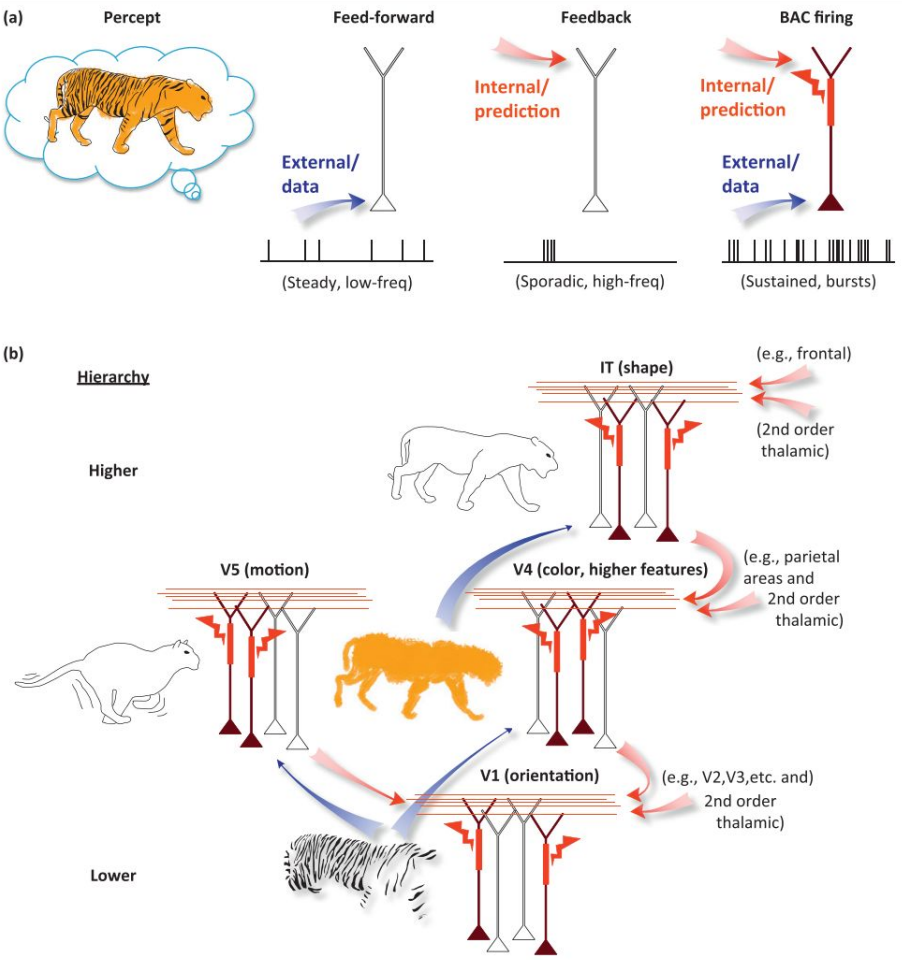

Figure 7: Integration of feedforward and feedback information in cortical pyramidal neurons.
A: Single pyramidal neuron receiving feedforward input at basal dendrites and feedback input at apical dendrites. Both signals are integrated via BAC firing and bursting. **B:** Proposed role of BAC firing in the information flow across different hierarchical areas of the visual cortex. From: Larkum (2013)

In pyramidal neurons of L2/3 and L5, feedforward and feedback signals arrive at different locations (fig. 7A): Feedforward information from hierarchically lower areas usually targets the basal dendrites, which surround the soma. In contrast, feedback information from hierarchically higher areas preferentially targets the distal apical dendrites. While the effect of feedback information in the cortex is not thoroughly understood, it was shown to play a role in contextual modulation (Gilbert & Li 2013), stimulus perception (Takahashi et al. 2016, Manita et al. 2015), and many other scenarios. Also, it was recently proposed to act as an error signal, similar to backpropagated error signals in deep learning (Whittington & Bogacz 2019, Richards & Lillicrap 2018).

Naively, feedback signals seem to play a minor role in the activity of the neuron: Their target sites, the apical dendrites, are electrotonically separated from the soma. Any incoming feedback signals are strongly attenuated before they can reach the soma and influence spiking (fig. 7A center). However, electrical activity in the apical tuft can cause broad calcium potentials, which last up to 50 ms *in vitro*. These calcium spikes can influence neuron activity

by lowering the firing threshold at the soma. Combined with feedforward input at the basal dendrites, they can lead to bursting (fig. 7A right). In turn, feedforward input at the basal dendrites can lower the threshold for calcium spikes in the apical tuft. This process is termed “backpropagation activated calcium spike firing”, or short BAC firing (Larkum 2013).

BAC firing presents a plausible mechanism for pyramidal neurons to integrate feedforward and feedback signals. Applied to a hierarchy of brain areas (e.g. in visual cortex), it suggests a framework for information processing across cortical areas (fig. 7B): At any point in time, streams of signals run up and down the hierarchy. The feedforward stream extracts features of the sensory input, while the feedback stream provides predictions, modulates the feedforward stream, and potentially mediates error signals. Both streams are combined and matched in pyramidal cells through calcium spikes and BAC firing.

2.1.3 Synaptic plasticity

Given the wiring described above, how does the cortex learn new things? A long-standing belief in neuroscience is that the main mechanism is altering synaptic connections between neurons (Citri & Malenka 2008). This is analogous to learning in artificial neural networks, where connection weights are changed to reach the desired input-output mapping. Of course, synaptic changes in biology are much more complex than shifting a single weight value: They can have short or long-lasting effects and lead to potentiation or depression of synaptic efficacy. Most mechanisms work by either modifying neurotransmitter release at the pre-synaptic site or changing the amount of neurotransmitter-gated ion channels at the post-synaptic site.

Short-term plasticity can last for milliseconds up to a few minutes. It usually affects individual spikes or spike trains. Short-term plasticity plays a role in adaptation to sensory inputs, short-term memory, and transient behavior changes. However, it is hardly involved in long-term learning of new skills. The effect on synaptic efficacy can be increasing (short-term potentiation; STP) or decreasing (short-term depression; STD).

Long-term plasticity is involved in forming long-term memories and learning new skills. It was already proposed by Cajal and first investigated in detail by Hebb (1949; see chapter 2.2.8 for a description of Hebbian learning rules). Since then, long-term plasticity has been found in many areas of the brain (Maffei 2011). Similar to short-term plasticity, it can lead to potentiation (long-term potentiation; LTP) or depression (long-term depression; LTD) of synaptic efficacy. Chemically, long-term potentiation mostly affects NMDA receptors but sometimes also various other receptor types (Collingridge & Bliss 1987).

2.2 Biologically plausible learning algorithms

Richards et al. (2019) suggest three areas where deep learning can inspire neuroscience: Learning algorithms, objective functions, and architectures (chapter 1.1 and fig. 1). Most existing research in this field focuses on learning algorithms and finding biologically plausible variants. Here, we present a few exemplary approaches. We will use some of these learning algorithms in the practical part of this thesis to train the networks we find via evolution. For further reviews, see Whittington & Bogacz (2019) and Richards & Lillicrap (2018).

First, let us answer a seemingly simple question: Why do we even need learning algorithms in deep learning? Artificial neural networks learn by adapting the weights of their synaptic connections, which changes the computation of the network. Naively, one could alter the weights randomly and see what works best. Obviously, this process is highly inefficient. In order to adapt the weights efficiently, we need to solve the credit assignment problem: How does a single synaptic weight contribute to the network output and how should it be changed in order to make the network better on a given task? This problem is non-trivial for deep networks where weights in lower layers contribute to the output in a highly non-linear fashion. Sophisticated learning algorithms try to solve this problem – most notably, the famous backpropagation algorithm (chapter 2.2.2).

In the cortex, the situation looks surprisingly similar: Just like artificial neural networks, the cortex learns primarily by changing the strength of synaptic connections, even though it uses more complex mechanisms (Citri & Malenka 2008, chapter 2.1.3). Also, the cortex is organized in a hierarchy of brain areas, similar to layers in an artificial neural network. Therefore, it also needs to solve the multi-layer credit assignment problem in order to adapt synapses in lower layers (fig. 8). It seems likely that on a global scale, the cortex uses a sophisticated learning algorithm – even though known learning rules for biological neurons are rather simple and local (e.g. Hebbian learning, chapter 2.2.8). Or to put it differently: It seems unlikely that the brain can reach its high level of intelligence without any kind of efficient learning algorithm that optimizes a global objective function (Marblestone et al. 2016, Richards & Lillicrap 2018).

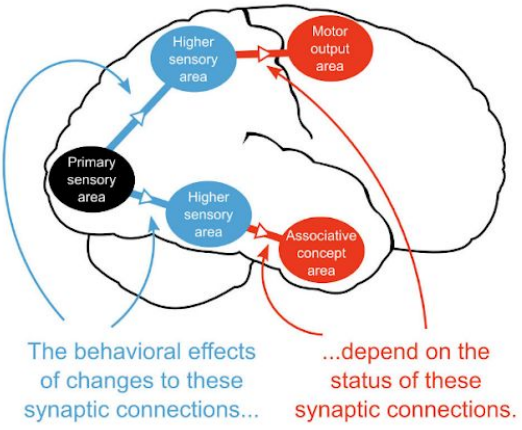

Figure 8: The credit assignment problem in the brain. From: Guerguiev et al. (2017)

2.2.1 Basics: Training deep neural networks

Before we delve into the learning algorithms, we want to give a brief introduction to artificial neural networks and how they are trained. We explain all the terms that are required to understand the learning algorithms and show how artificial neural networks map to biological ones. For a thorough review, see Nielsen (2015) and LeCun et al. (2015).

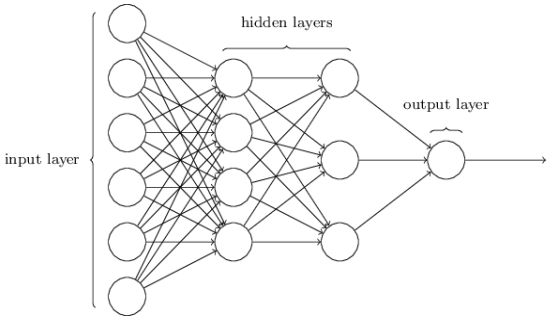

Figure 9: Neural network with two hidden layers.
Circles depict neurons, arrows depict connections. From: Nielsen (2015).

Artificial neural networks consist of neurons and weighted synaptic connections. The neurons are typically arranged in layers (fig. 9): The first layer (input layer) represents the input data (e.g. the pixel values of an image). This input data is transferred via synaptic connections to the next (hidden) layer, where it is integrated and modified in artificial neurons. In this fashion, the signals are processed throughout the layers of the network, yielding more and more abstract representations at each layer. Eventually, the last layer (output layer) generates an output (e.g. classification probabilities for the image content). This layered approach to information processing is similar to the hierarchical organization of brain areas in the cortex, e.g. in the visual system. In this view, artificial neurons are usually associated with pyramidal neurons or entire cortical columns (chapters 2.1.1 and 2.1.2).

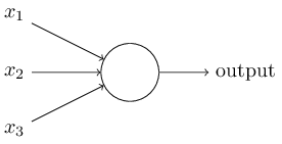

Figure 10: An artificial neuron.
The neuron has three incoming connections and one outgoing connection. From: Nielsen (2015)

How do artificial neurons process information? In short, each neuron integrates the signals from its incoming connections and emits a new signal to its outgoing connections. In a biological context, these signals are usually interpreted as firing rates (even though they can have negative values). For a single neuron j with multiple inputs x_i (fig. 10), the output can be written as:

$$a_j = \sigma\Big(\sum_i w_{ji} x_i + b_j\Big) \tag{1}$$

Here, the inputs x_i are multiplied by the weights w_{ji} and summed. The bias b_j of the neuron serves as a threshold for incoming signals. Finally, a nonlinear activation function σ is applied (e.g. sigmoid or ReLU).

Using vector-notation, we can write the activities of all neurons in a layer l as:

$$\boldsymbol{a}^l = \sigma(\underbrace{\mathbf{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l}_{=:\boldsymbol{z}^l}) \tag{2}$$

$\mathbf{W}^l$ is the weight matrix of layer l. Each element w_{ji} is the weight from neuron i in layer $l-1$ to neuron j in layer l.

The power of an artificial neural network arises from tuning its weights and biases until it produces the desired output. This process is called training or learning. The remarkable point here is that neural networks can learn from raw data and do not require feature-engineering.

There are three major ways to train a neural network: Supervised, unsupervised, and reinforcement learning. In supervised learning, the “correct” output for each data sample is known (often called target or label). The network can compare its own output to this desired solution and update its parameters accordingly. In unsupervised learning, the desired output is not known, and the neural network has to make sense of the input data itself. In reinforcement learning, the network does not get labels but sparse rewards from an environment. Here, we only deal with supervised learning.

How are the weights updated during supervised learning? First, the neural network has to evaluate how well it already works. This is described by the objective function C (also called loss or cost function), which in the simplest case measures the difference between the actual and desired output. Popular examples of objective functions are mean squared error and cross-entropy. The goal of training is to minimize the objective function. This is usually done via gradient descent: After evaluating the loss on some data, we calculate its gradient with respect to the weights and biases. The gradient is a vector that points in the direction of increasing loss. Therefore, we update the parameters of the network in the direction of the negative gradient to decrease the loss. For a weight w_{ji}, this update step can be described by:

$$\Delta w_{ji} = -\eta \frac{\partial C}{\partial w_{ji}} \tag{3}$$

Here, $\partial C / \partial w_{ji}$ is the partial derivative of the loss function with respect to the weight. This partial derivative represents one component of the gradient vector. The parameter η (learning rate) controls the magnitude of the update. Typical neural networks contain thousands or millions of weights and biases in multiple layers. Therefore, calculating all partial derivatives of the gradient is not straightforward. The popular backpropagation algorithm offers a computationally efficient solution to this problem.

2.2.2 Backpropagation

The backpropagation algorithm (Rumelhart et al. 1986, Werbos 1974) is the de-facto standard learning algorithm in deep learning. It offers an efficient way to compute the gradient of the loss function w.r.t. the parameters of the network. This gradient is required for learning with gradient descent (chapter 2.2.1).

Method. Backpropagation makes use of the chain rule to compute intermediate "error" signals for each layer. Given the objective function C, the error for the output layer L can be calculated as:

$$\boldsymbol{\delta}^L = \nabla_{\boldsymbol{z}^L} C = \nabla_{\boldsymbol{a}^L} C \odot \sigma'(\boldsymbol{z}^L) \tag{4}$$

Here, $\boldsymbol{a}^L$ is the vector of output neuron activities (equal to the network output), σ is the nonlinearity used in the forward pass, and $\boldsymbol{z}^L$ is the vector of summed inputs for all output neurons (see equation 2). $\odot$ is the Hadamard product. Note that $\boldsymbol{\delta}^L$ is a vector with one error value per output neuron.

This error vector is then propagated to lower layers via:

$$\boldsymbol{\delta}^l = \nabla_{\boldsymbol{z}^l} C = ((\mathbf{W}^{l+1})^\top \boldsymbol{\delta}^{l+1}) \odot \sigma'(\boldsymbol{z}^l) \tag{5}$$

Again, $\boldsymbol{z}^l$ is the vector of summed inputs to layer l. $\mathbf{W}^{l+1}$ is the weight matrix of layer $l+1$. Each entry w_{ji} gives the weight for the connection from neuron i in layer l to neuron j in layer $l+1$.

Using the backpropagated error terms, we can calculate the gradients of the objective function w.r.t. to all weights and biases:

$$\frac{\partial C}{\partial w^l_{ji}} = a^{l-1}_i \delta^l_j \tag{6}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{7}$$

Biological plausibility. Could the brain implement backpropagation? While the algorithm offers a simple solution to the credit assignment problem, it has long been criticized for being biologically not plausible (Crick 1989, Bengio et al. 2015). We discuss four issues:

1. Weight sharing problem: As equation 5 shows, backpropagating the errors through the networks requires the exact transpose of the feedforward weights. Translated to a circuit model (e.g. for synaptic connections between neurons or cortical columns), this implies two things: 1) For each forward connection, there needs to be a

reciprocal feedback connection between the same neurons (or cortical columns). 2) This feedback connection needs to share the weight of the forward connection, i.e. both synapses need to have exactly the same strength. Even though the cortex contains bidirectional connections (Song et al. 2005) and cortico-cortical loops (Young et al. 2019), it seems implausible that such a specific wiring exists throughout the cortex.

2. Separation problem: In backpropagation, the forward and backward passes are separate computations. First, the forward pass computes neuron activities. Then, the backward pass computes gradients. In the brain, however, forward and backward connections seem to be active at all times. There is no apparent time signal that would control the two phases. Not only are the computations in backpropagation separate in time, but also in space: For each neuron, the forward and backward passes store separate signals (activity in the forward pass and error term in the backward pass). To implement this in biology, either a neuron would need to store both values at the same time, or the circuit would require separate error neurons. Alternatively, Naud & Sprekeler (2018) recently suggested that pyramidal neurons could multiplex forward and backward signals.

3. Spike problem: Artificial neural networks compute with real-valued signals, which are usually interpreted as firing rates. Biological neurons, however, communicate via discrete, all-or-none spikes. This poses an issue for a biological implementation of backpropagation because discrete spikes are not differentiable. To get around this problem, Zenke & Ganguli (2018) recently proposed a way to compute gradients for spike trains.

4. Supervision problem: Supervised learning via gradient descent and backpropagation requires an objective function and output labels. In the brain, there is no clear way to express these features. The brain might rely on other forms of learning (unsupervised, self-supervised, or reinforcement learning) but it is questionable if these methods are enough to achieve high performance. Also, no matter if the brain uses supervised learning or other methods, there should still be some form of credit assignment similar to backpropagation. Marblestone et al. (2016) discuss some ways in which the brain could implement objective functions.

Due to these issues, it is unlikely that the brain uses backpropagation in the same way as deep learning. Still, it seems likely that it implements some way of performing credit assignment across multiple layers. In the following, we present several approaches to tackle the issues discussed above.

2.2.3 Feedback alignment

Feedback alignment (Lillicrap et al. 2016) is a simple modification of backpropagation that aims to solve the weight sharing problem.

Method. The algorithm is equivalent to backpropagation. However, instead of using the transpose of the weight matrix to propagate the error terms to lower layers, feedback alignment uses a matrix $\mathbf{B}^l$ of fixed random feedback weights for each layer l (fig. 11). This eliminates the need for weight sharing. Equation 5 of backpropagation becomes:

$$\boldsymbol{\delta}^l = ((\mathbf{B}^{l+1})^\top \boldsymbol{\delta}^{l+1}) \odot \sigma'(\boldsymbol{z}^l) \tag{8}$$

The elements of each $\mathbf{B}^l$ are sampled from a normal distribution and stay constant during learning.

Results. Even though feedback alignment seems quite simple and arbitrary, it works surprisingly well: On MNIST, Lillicrap et al. (2016) report a test error of 2.1 % for a 3-layer network, which is similar to the backpropagation error for their network (2.4 %). How is it possible that random feedback weights propagate useful error signals? Lillicrap et al. find that the forward weights of the network align to the fixed feedback weights during learning. After some time, forward and feedback weights approximate the pseudoinverse of each other. For error propagation, the pseudoinverse is almost as effective as the transpose in equation 5. While feedback alignment works well on MNIST, it has bad scaling properties: Performance on CIFAR10 (a more complex image classification dataset) is worse and on ImageNet (today's standard benchmark for image classification) far worse than backpropagation (Liao et al. 2016, Bartunov et al. 2018, Xiao et al. 2018).

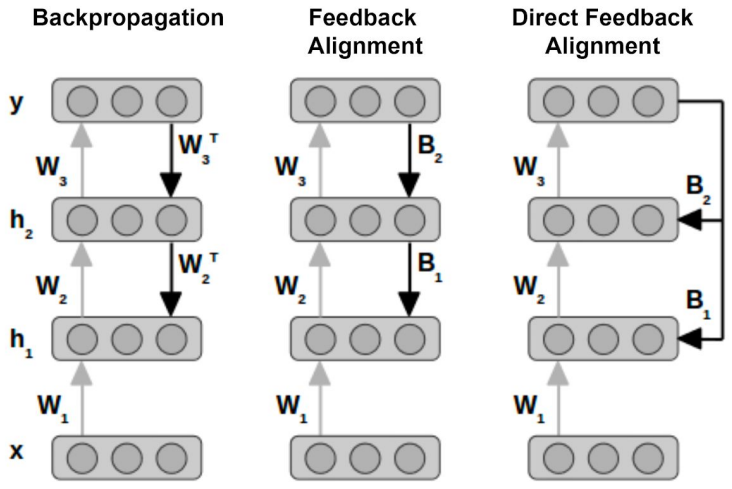

Figure 11: Backpropagation vs. feedback alignment variants. Grey arrows indicate the forward pass (computation of neuron activities), black arrows indicate the backward pass (computation of gradients). Note that we use a instead of h for the activity in the main text. Adapted from: Nøkland (2016)

Biological plausibility. Feedback alignment shows that forward and backward connections do not necessarily need to have symmetric weights in order to propagate useful error signals. However, the algorithm is quite limited to small datasets and networks. Apparently, deeper networks cannot simply rely on the alignment between forward and backward weights. This makes it questionable if the cortex can get around the weight sharing problem so easily. Also, while feedback alignment successfully questions the necessity for weight sharing, it does obviously not present a biologically plausible solution either, as synaptic weights in the brain are not constant.

Variants. Direct feedback alignment (Nøkland et al. 2016) uses fixed random feedback weights but propagates error signals directly from the output layer to each hidden layer, not layer by layer through the network (fig. 11). Using the same network, it performs worse than feedback alignment on MNIST and CIFAR-10 (Bartunov et al. 2018). Sign-symmetry (Liao et al. 2016) is similar to feedback alignment but each feedback weight only shares its sign with the corresponding forward weight, not the magnitude. This implements a milder form of weight sharing. It scales better than feedback alignment and even achieves good results on the ImageNet benchmark (Xiao et al. 2018, Akrout et al. 2019).

2.2.4 Weight mirroring

Akrout et al. (2019) present an extension of feedback alignment, which they call weight mirror. Just like in feedback alignment, the feedback weights are initialized randomly but they are adapted during training.

Method. Weight mirroring initializes the network with random feedback weights and uses the same forward/backward pass as feedback alignment. It introduces an additional "mirroring phase", where all neurons fire randomly with zero mean. Based on this random firing, the feedback weights $\mathbf{B}^l$ are updated with a Hebbian learning rule:

$$\Delta \mathbf{B}^l = \eta_B \boldsymbol{x}^l (\boldsymbol{y}^l)^\top \tag{9}$$

Here, $\boldsymbol{x}^l$ is the input for layer l during the mirroring phase and $\boldsymbol{y}^l$ is its output.

Results. Akrout et al. (2019) show that the Hebbian update pushes the feedback weights in the direction of the transpose of the forward weights. This makes the algorithm scale much better than pure feedback alignment. Even on the challenging ImageNet dataset, it achieves results comparable to backpropagation.

Biological plausibility. Weight mirroring shows that symmetric feedback weights are not required *a priori*. Instead, they can be made symmetric during learning, using a simple update rule. Notably, the mirroring phase does not have to occur right after training the forward weights. In the brain, such a phase could be handled during idle times (e.g. sleep). No matter if the brain implements a similar mechanism or not, weight mirroring shows that weight sharing is not a serious restriction of learning algorithms that propagate gradients.

Note that while weight mirroring tackles weight sharing, it still requires feedback connections in the exact same places as forward connections.

2.2.5 Target propagation

Target propagation (LeCun 1986, Bengio 2014, Lee et al. 2015) is a gradient-free alternative to backpropagation. It mitigates the weight sharing problem by using explicit feedback connections and tackles the separation problem by propagating activity targets instead of gradients.

Method. Target propagation has two major differences to backpropagation: 1) It uses explicit feedback weights that are independent of the forward weights (instead of weight sharing). 2) It propagates an activity target to each hidden neuron (instead of the objective function gradient). In a first phase, it adapts the forward weights. The activity target $\hat{\boldsymbol{a}}^L$ for the final layer is set to directly minimize the global loss function:

$$\hat{\boldsymbol{a}}^L = \boldsymbol{a}^L - \alpha \nabla_{\boldsymbol{a}^L} C \tag{10}$$

Using the feedback weights, targets are then propagated to all lower layers via:

$$\hat{\boldsymbol{a}}^{l-1} = g^l(\hat{\boldsymbol{a}}^l) \tag{11}$$

Here and in the following, we use the function h^l to denote the application of the forward weights similar to equation 2, and g^l to denote the application of the feedback weights. If there is a mismatch between neuron activities and targets in the final layer (i.e. the network does not function perfectly), this mismatch is propagated back through the network. Then, target propagation adapts the forward weights via gradient descent on the layer-local loss:

$$C^l = \|\boldsymbol{a}^l - \hat{\boldsymbol{a}}^l\|^2 \tag{12}$$

This loss is local to a layer because all other parameters except the weights of the current layer are assumed to be fixed.

In a second phase, target propagation alters the feedback weights. This step uses the following insight: If there is no mismatch between activity and target, the backward function g^l should approximate the inverse of the forward function h^l. Therefore, the feedback weights are adapted by gradient descent on the layer-local reconstruction loss:

$$C^l_{rec} = \|\boldsymbol{a}^{l-1} - g^l(\boldsymbol{a}^l)\|^2 \tag{13}$$

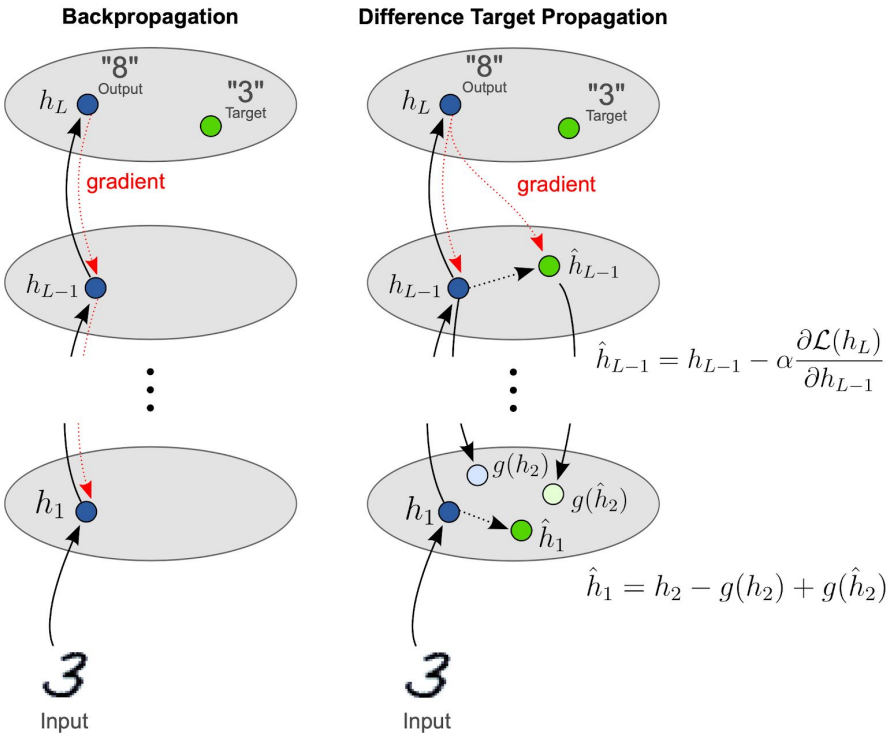

Figure 12: Backpropagation vs. difference target propagation.
Note that we use a instead of h for the activity in the main text. Also, the equations here are for difference target propagation (Lee et al. 2015), while in the main text we only explain the equations for normal target propagation. From: Bartunov et al. (2018)

Results. We present the results of difference target propagation (Lee et al. 2015) because it is the only variant investigated empirically (Bartunov et al. 2018). Vanilla target propagation (as described above) suffers from the problem that all samples for a class label produce similar targets (Bartunov et al. 2018). Difference target propagation solves this problem by adding a stabilizing term (fig. 12). The algorithm works well on MNIST, even though its results are slightly worse than backpropagation and feedback alignment. Also, it scales poorly to larger datasets. On CIFAR10 and ImageNet, its results are even worse than feedback alignment and far away from the results of backpropagation (Bartunov et al. 2018).

Biological plausibility. Target propagation solves the weight sharing problem by introducing explicit feedback connections. Also, it tackles the separation problem: The activity targets lie in the same space as the neuron activities themselves – in contrast to the gradients in backpropagation or feedback alignment. This makes it easier to imagine a mechanism by which neurons could represent both signals at once. Target propagation requires a secondary phase to train the feedback weights. Neurons and connections in the cortex, however, seem to be performing the same operations at all times.

Variants. In the original implementation of difference target propagation, targets for the penultimate layer were computed using the global loss, which is biologically not plausible. In Simplified Difference Target Propagation (SDTP; Bartunov et al. 2018), they introduce a different way to compute these targets. The algorithm performs slightly worse than difference target propagation on MNIST, CIFAR10, and ImageNet.

2.2.6 Equilibrium propagation

Equilibrium propagation (Scellier & Bengio 2017) is an energy-based framework to train artificial neural networks, which avoids propagating gradient signals.

Method. Equilibrium propagation operates on recurrent networks in continuous time. It consists of two phases: First, in the free phase, only the input is clamped to the network. The states of all neurons are updated by gradient descent on an energy function (Hopfield energy in the original paper), until they reach a fixed point. At this point, we collect the gradient of the energy w.r.t. the weights and biases. Second, in the weakly clamped phase, both input and label are clamped to the network. The total energy of the network increases by the objective function. Again, the network settles to a fixed point by gradient descent on the new total energy. We collect the gradient of the total energy w.r.t. the parameters. Finally, we update the weights and biases based on the difference between the two collected gradients. Importantly, all gradients are local to neurons, i.e. they do not require any gradient backpropagation.

Results. Scelier & Bengio (2017) achieve a test error of 2-3 % on MNIST (for networks with up to three hidden layers), which is a bit worse than backpropagation. Equilibrium propagation was not tested on more complex datasets yet.

Biological plausibility. Similar to the forward and backward pass in backpropagation, equilibrium propagation requires two separate phases (free and weakly-nudged phase). However, both phases use the same circuitry and perform the same computation. This presents a partial solution to the separation problem. Note that Scellier & Bengio (2017) use symmetric weights for their network (as in backpropagation), even though equilibrium propagation does not necessarily require it (see also Scellier et al., 2018, for a relaxation of this requirement). Equilibrium propagation can be related to spike-timing dependent plasticity (STDP), a known learning mechanism of biological neurons (chapter 2.2.8).

2.2.7 Dendritic computation

Most models above are inspired by the backpropagation algorithm and aim to solve some of its issues. Recently, Guerguiev et al. (2017) and Sacramento et al. (2018) proposed two models that take inspiration from computational principles in the cortex – specifically, the integration of feedforward and feedback signals in the dendrites of pyramidal neurons (chapter 2.1.2). See also Richards & Lillicrap (2018) for a review on this topic.

Model. Both models use networks with multi-compartmental neurons and are continuous in time (instead of point neurons and discrete time as for standard artificial neural networks). Inspired by pyramidal cells, a neuron contains three compartments, each with its own voltage: A basal compartment that integrates feedforward signals, an apical compartment that integrates feedback signals, and a somatic compartment connecting them (fig. 13). As in feedback alignment, the feedback weights are fixed and random (even though Sacramento et al. briefly investigate an update rule for feedback weights). In Guerguiev et al. (2017), the

learning step requires two phases, similar to equilibrium propagation (chapter 2.2.6): In the first phase, only the input is clamped to the network. In the second phase, the output neurons receive an additional training current based on the label. After each phase, a plateau potential is calculated in the apical compartments (similar to calcium-driven potentials in pyramidal neurons; chapter 2.1.2). The differences between these two plateau potentials drive plasticity and learning. In Sacramento et al. (2018), the learning step requires only one phase. The network contains additional interneurons (fig. 13B), which are modeled after SST interneurons in the cortex. These cells cancel the feedback signals in apical compartments. If a label is present, they cannot cancel the feedback anymore. This mismatch serves as an error signal and drives the plasticity of the feedforward weights.

Results. Both models perform well on MNIST. Sacramento et al. (2018) are on par with feedback alignment (1.96 % test error), while Guerguiev et al. (2017) are a bit worse (3.2 %). The model of Guerguiev et al. (2017) does not improve beyond a single hidden layer – potentially because they use feedback connections from the output layer directly to each hidden layer, similar to direct feedback alignment (chapter 2.2.3). The models were not tested on more complex datasets yet.

Biological plausibility. Both models tackle the separation problem of backpropagation. Using separate compartments for feedforward and feedback signals alleviates the need for separate circuitry during the forward and backward pass. Both signals can be integrated within the same neuron. While Guerguiev et al. (2017) still need two separate phases with different computations, the model in Sacramento et al. (2018) uses the same computation at all times. Both models use fixed and random feedback weights. While this solves the weight sharing problem for a simple problem like MNIST, it is potentially problematic for larger datasets (chapter 2.2.3). Guerguiev et al. (2017) use plateau potentials to drive synaptic plasticity. While this is backed by experiments (Bittner et al. 2017), the exact role of plateau potentials in pyramidal neurons is still up for debate. Also, the one-to-one mapping of interneurons and pyramidal neurons in Sacramento et al. (2018) is questionable from a biological point of view. Regardless of these issues, both models show how computational principles found in the cortex could mediate learning on complex datasets.

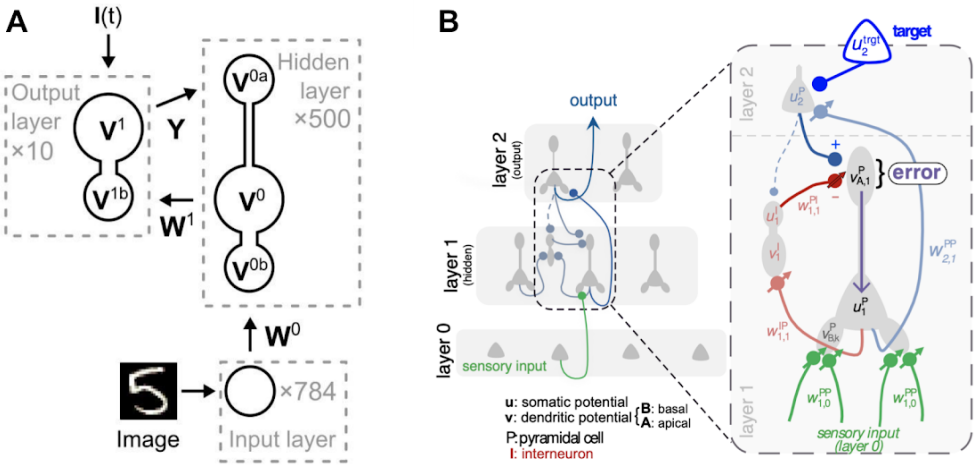

Figure 13: Networks using dendritic computation.
A: The network in Guerguiev et al. (2017) uses neurons with three compartments, each with their own voltage. Feedforward connections terminate on the basal compartment, feedback connections terminate on the apical compartment. From: Guerguiev et al. (2017). **B:** The network in Sacramento et al. (2018) contains additional interneurons (red), which play a vital role to generate error signals. From: Sacramento et al. (2018).

2.2.8 Hebbian learning and other local learning rules

Unlike the learning algorithms presented above, traditional neuroscience research has focused mostly on local learning rules. Local means that the rate of change of a synaptic weight depends exclusively on the pre- and post-synaptic neurons. Arguably, these learning rules are easier to investigate experimentally because they involve only two neurons and no complete network with feedforward and feedback connections. However, they do not minimize a global objective function, which makes them less powerful at solving complex problems.

One of the earliest local learning rules is Hebbian learning, which was proposed by Hebb (1949). It is based on the observation that a connection between two neurons is strengthened if the pre-synaptic neuron repeatedly excites the post-synaptic neuron above its firing threshold. Colloquially, it is often expressed as “cells that fire together, wire together”. A range of experimental evidence supports the idea of Hebbian learning (Martin et al. 2000). Formally, the weight change is often expressed with firing rates as:

$$\Delta w_{ji} = \eta f(r_i) g(r_j) \tag{14}$$

Here, r_i and r_j are the firing rates of pre- and postsynaptic neurons, f and g are arbitrary functions, and η is the learning rate. As the rule shows, Hebbian learning is a very generic framework that can be adapted to different specific rules. Two popular related learning rules are the BCM rule (Bienenstock et al. 1982) and Clopath rule (Clopath et al. 2010). See also Gerstner & Kistler (2002) for more details on Hebbian learning rules.

Spike-timing dependent plasticity (STDP; Markram et al. 1997, Feldmann 2012) presents a mechanism how Hebbian learning can be used for long-term potentiation and depression. It adjusts the synaptic weight based on the relative timing of pre- and post-synaptic spikes (fig. 14): If the pre-synaptic spike occurs before the post-synaptic spike, the weight is increased (long-term potentiation). If the pre-synaptic spike occurs after the post-synaptic spike, the weight is decreased (long-term depression).

Local learning rules can easily be extended to incorporate global feedback signals. These rules are often called three-factor or neo-Hebbian rules. In addition to Hebbian learning, a third factor M_j influences the weight change:

$$\Delta w_{ji} = \eta f(r_i) g(r_j) M_j \tag{15}$$

This third factor can be equal for all neurons (e.g. a neuromodulator such as dopamine) or it can be neuron-specific. It can influence both the speed of learning (via its magnitude) as well as the direction of the weight change (via its sign). Note that backpropagation can be easily re-written as a three-factor rule, where the third factor depends on the error term propagated through the network.

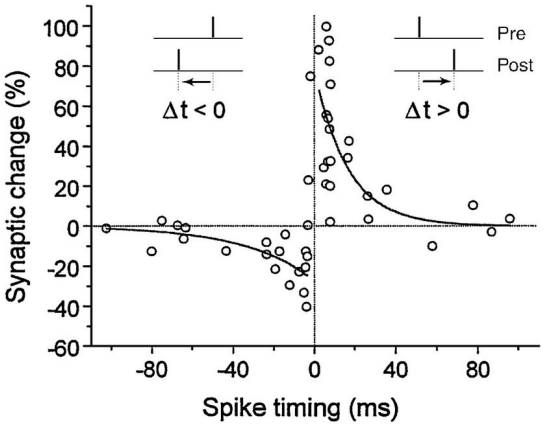

Figure 14: Synaptic changes in STDP. On the left side, the pre-synaptic neuron fires before the post-synaptic neuron and the connection is strengthened. On the right side, the post-synaptic neuron fires first and the connection is weakened. From: Asl (2018)

2.3 Neuroevolution

The learning algorithms described above are usually applied to simple, feedforward neural networks. In this thesis, we investigate how they work together with more complex network architectures. Our inspiration comes from the intricate wiring of connections in the brain, which likely provides an inductive bias for learning (chapter 2.1.1). Similar to the biological evolution of the brain, we use an evolutionary approach to find neural network architectures – a method commonly known as neuroevolution. In this chapter, we want to give an overview of existing research in this area. First, we introduce the basics of evolutionary and genetic algorithms. Then, we describe different approaches to neuroevolution and how they relate to our work. Table 1 presents a summary of all the methods we discuss.

2.3.1 Evolutionary and genetic algorithms

Evolutionary algorithms solve optimization problems, using concepts from biological evolution. They create a set of candidate solutions and evaluate them on a given objective function, yielding a fitness score for each solution. Based on these fitness scores, the next generation of candidate solutions is created, which likely perform better on the objective function. This process of evaluating and creating new candidate solutions is repeated until the desired performance is reached (Ha 2017). In contrast to gradient-based optimization, evolutionary approaches can handle non-differentiable problems (e.g. optimizing neural network architectures, as in this thesis).

Genetic algorithms are a specific class of evolutionary algorithms, which store information about candidate solutions in a genome. A genome is a set of variables (genes), which can assume different values. Similar to biological genomes, new variants are created by recombination of two parent genomes and random mutation of genes. In neuroevolution, the genome usually contains information about the network topology and/or its weights.

Table 1: Comparison of neuroevolution methods.
The columns indicate how network architectures and weights are developed.

	Architecture	**Weights**
NEAT (Stanley & Miikkulainen, 2002)	evolution	evolution
Neural architecture search	evolution/reinforcement learning/Bayesian optimization/...	backpropagation
Weight agnostic neural networks (Gaier & Ha, 2019)	evolution	fixed (with single shared weight value)
Ours	evolution	backpropagation/feedback alignment/Hebbian learning/last-layer learning

In general, the steps for a basic genetic algorithm are:

1. Initialize: Create a population of N candidate solutions with different genomes
2. Selection: Evaluate the fitness of each candidate solution on an objective function (e.g. by running a simulation)
3. Reproduction: Repeat N times:
    - Pick two parents (usually based on fitness values, e.g. the best performing solutions)
    - Crossover: Create a child by mixing the genomes of both parents (for each gene, use the variant of one parent)
    - Mutation: Change the genome of the child randomly
    - Add the child to the new population
4. Replace old population with new population
5. Repeat step 2-4 until the desired performance is reached

2.3.2 Neuroevolution and NEAT

In neuroevolution, evolutionary or genetic algorithms are used to evolve neural networks. The most popular genetic algorithm for neuroevolution is called NEAT (NeuroEvolution of Augmenting Topologies; Stanley & Miikkulainen 2002). NEAT optimizes the architecture and the weights of a neural network at the same time (table 1). Our approach in this thesis is loosely based on it.

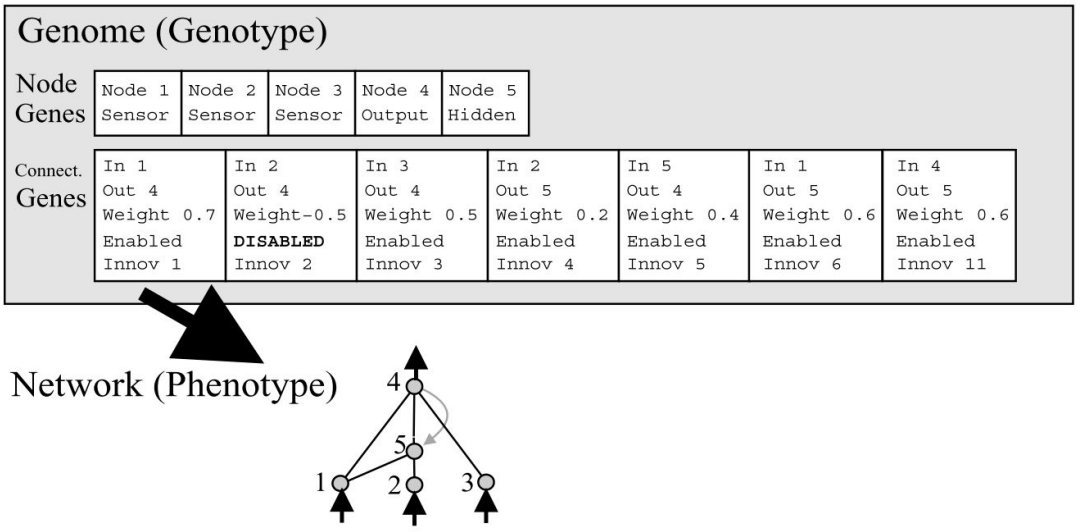

Figure 15: Genomic representation of networks in NEAT. Each candidate solution has node and connection genes (top), which map to a unique neural network (bottom). From: Stanley & Miikkulainen (2002)

NEAT follows the basic outline of a genetic algorithm described above. Each candidate solution is a neural network. Its architecture and weights are stored in the genome, as shown in fig. 15. Additionally, NEAT uses the following methods:

- Mutations: Mutations in NEAT can modify both the network architecture and weights. In every generation, each connection weight is changed randomly with a given probability. Topologies are altered by two random mutations: 1) Adding a new connection between two existing nodes, or 2) adding a new node by splitting an existing connection.

- Historical marking: Due to the topological mutations in NEAT, different networks can have very different architectures and genome lengths. This is problematic for crossover (the combination of two parent genomes into a single offspring genome): It might not be possible to combine two different topologies into a fully functional network. This problem is called competing conventions. To tackle it, NEAT introduces historical markings. If a mutation adds a new gene (e.g. a new connection), it is labeled with an innovation number, which is globally incremented after every mutation. At crossover, the parent genomes are lined up according to their innovation numbers. If genes of the two parents match in their innovation number, one of them is chosen randomly. For all genes without matching counterparts, the genes of the fitter parent are passed on to the offspring genome.

- Speciation: To protect topological innovations, NEAT divides its population into different species. At each generation, networks are sorted into these species based on the similarity of their topologies (using the historical markings to calculate a distance measure). When the best networks are chosen for reproduction, networks only compete with other networks within their species, not the entire population.

The evolutionary approach in this thesis is inspired by NEAT but differs in some key aspects. We only perform topological mutations to change connections and neurons. Weights, however, are trained using one of four learning algorithms. Also, we leave out historical markings and speciation to simplify the algorithm.

2.3.3 Neural architecture search

Neural architecture search (NAS) is a relatively recent field of machine learning research. The goal of NAS is to find a network architecture that achieves maximum performance on a given dataset, using an automated search algorithm. Usually, neural network architectures are hand-designed by the developer. This process requires lots of experience and time and can lead to suboptimal results. NAS aims to automate this step. For a comprehensive review of the field, we refer to Elsken et al. (2019).

In contrast to the NEAT algorithm presented above, NAS algorithms only search for network architectures. Weights, however, are trained with the standard backpropagation algorithm. Therefore, it can be framed as meta-learning: The outer learning loop finds network architectures, whereas the inner loop finds the weights themselves (table 1). Some early

papers on NAS date back to the 1990s, but most modern approaches were developed since 2017 (Elsken et al. 2019). NAS is very related to our work – strictly speaking, we are doing NAS for different learning algorithms, some of which are biologically plausible. However, there are some important differences beyond the learning algorithm.

First, in contrast to our work, NAS is not limited to evolutionary algorithms. Instead, a range of methods is used to find architectures. The two most popular approaches are reinforcement learning and evolution. Fig. 16 shows one example for each method. Other papers use Bayesian optimization, Monte Carlo tree search, or direct gradient-based optimization. Real et al. (2019) indicate that reinforcement learning and evolutionary approaches perform similarly well.

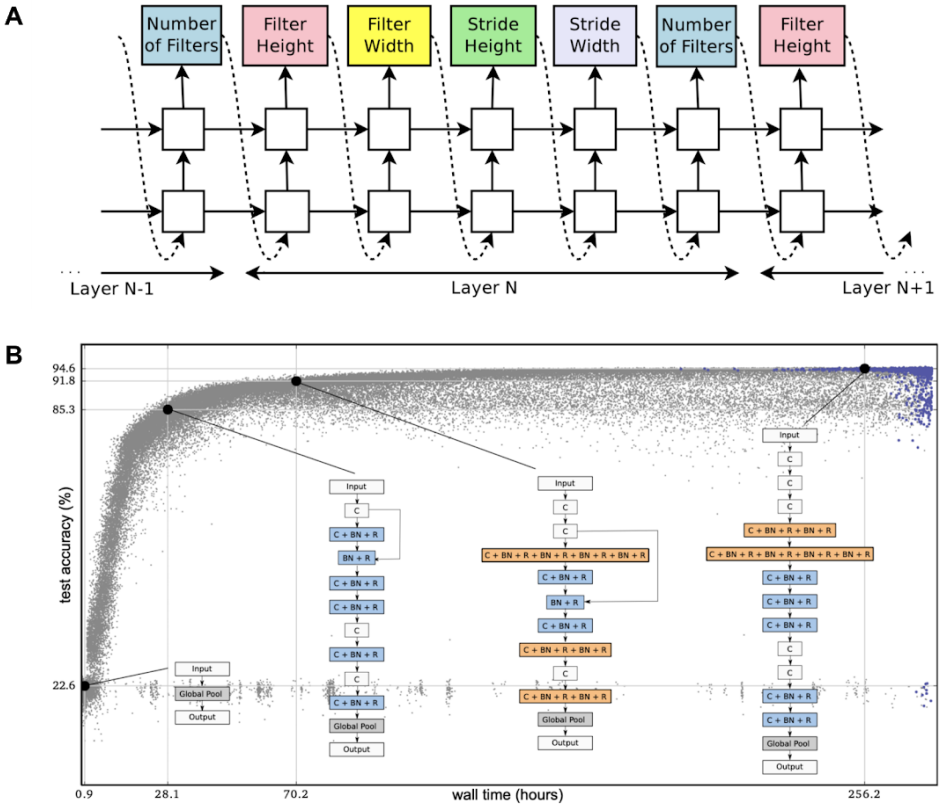

Figure 16: Neural architecture search (NAS) via reinforcement learning and evolution.
A: Zoph & Le (2017) use reinforcement learning to find architectures. A recurrent neural network (bottom) suggests each parameter of the candidate (convolutional) network (top). The recurrent network is trained with the REINFORCE algorithm, using the performances of the candidate networks as rewards. From: Zoph & Le (2017)
B: Real et al. (2017) use an evolutionary algorithm, which inserts, removes or modifies network layers. The plot shows test set accuracy on CIFAR-10 during evolution. Each dot is one individual from the population. The graphs show four exemplary networks from different stages of evolution. From: Real et al. (2017)

Second, most recent approaches in NAS manipulate complete network layers, not single neurons or connections as in our work (Elsken et al. 2019). In the simplest case, network layers are stacked in a chain-like structure. The composition of this chain and the parameters of the layers (e.g. stride or filter size for convolutional layers) are then found via NAS. Evolutionary approaches often use mutations that add, remove or modify complete layers (e.g. Real et al. 2017). In this thesis, however, we evolve the wiring of individual neurons, in analogy to the detailed and specific wiring of the cortex (chapter 2.1.1).

Finally, we do not aim to solve the engineering problem of finding the best possible performance on a given dataset. We are fully aware that even slightly more sophisticated networks (e.g. with convolutions) can outperform the networks we evolve in this thesis. Instead, we want to investigate how different learning algorithms work together with neuroevolution – just like learning algorithms in the brain had to work together with biological evolution to produce intelligence. In short, we are more interested in the evolved architectures themselves than in reaching maximum performance.

2.3.4 Weight agnostic neural networks

Next, we want to present a recent paper by Gaier & Ha (2019). It inspired the topic of this thesis and uses similar methods for the evolutionary algorithm.

The paper aims to find neural network architectures that are inherently capable of solving tasks – without any form of weight training and regardless of the specific weight configuration. These networks are termed weight agnostic neural networks (WANN). The objective is motivated by innate behaviors in animals: Many animals can perform difficult tasks right after birth, without any form of learning from data (i.e. without adapting synaptic connections; Zador 2019, Marcus 2018).

To find such architectures, Gaier & Ha employ an evolutionary algorithm with similar topological mutations as in NEAT. The weights, however, are not found via evolution (as in NEAT) or gradient-based learning (as in NAS). Instead, the networks use a fixed weight value that is shared by all connections. At every generation, each network is evaluated on the task using a set of such shared weight values. This gives an estimate of the capabilities of the architecture regardless of the weight configuration – the network becomes “weight agnostic”. Consequently, the evolution loop searches for network architectures that encode the solution to the task directly within their architecture.

This approach differs from NAS: It skips the inner loop of learning via backpropagation and replaces it with an evaluation on shared weight values (table 1). While NAS searches for architectures that provide a good substrate for learning, the WANN approach searches for architectures that can themselves solve the task. Also, in contrast to most modern approaches to NAS, the networks are not mutated on the level of complete network layers but by adding individual neurons and connections (comparable to NEAT and our work).

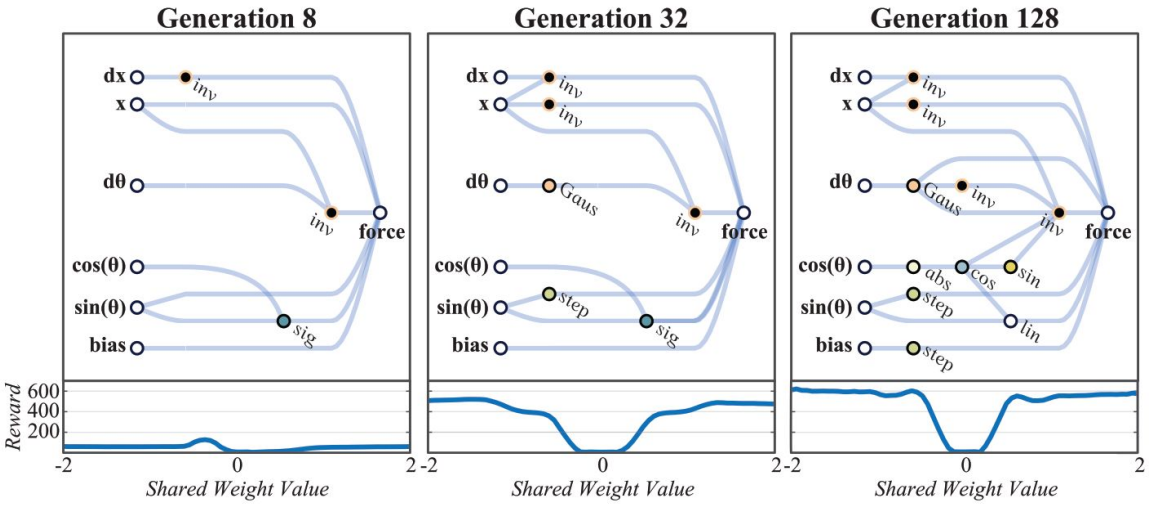

Figure 17: Weight agnostic network architectures at different stages of evolution. These networks were evolved on the CartPoleSwingUp task (see chapter 4.1 for details). The plots on the bottom show the cumulative reward for different shared weight values. From: Gaier & Ha (2019)

Gaier & Ha evaluate WANNs on four tasks: Three environments for reinforcement learning (RL) and classification of the MNIST dataset of handwritten digits (see chapter 4.1 for details). Fig. 17 shows some exemplary architectures found during evolution. On all tasks, the evolved weight agnostic networks achieve competitive results. The networks are able to master the RL tasks (even though they do not achieve the same scores as networks with weight training). They easily beat hand-designed architectures that are initialized with shared weight values. On MNIST, weight agnostic networks achieve 92 % test accuracy (for the best performing shared weight value). This is on par with linear regression (i.e. a shallow neural network), which requires the training of thousands of weights.

Obviously, these results cannot compete with sophisticated approaches using weight training. Still, they are quite remarkable considering that not a single weight in these networks is trained and all information is encoded in the network architecture. The paper shows that fine-tuned architectures can have a huge impact on the performance of a neural network.

3. Methods

The practical part of this thesis combines neuroevolution (chapter 2.3) with several learning algorithms, which differ in their complexity and biological plausibility (chapter 2.2). By evolving and mutating network topologies, we aim to find architectures that are particularly suited to each learning algorithm. We analyze the differences between evolved networks, both in terms of task performance as well as topology.

The approach consists of two elements, which are shown in fig. 18: An outer loop, which tunes network architectures (i.e. neurons and connections) via an evolutionary algorithm, and an inner loop, which tunes the network weights using one of four learning algorithms. This approach is inspired by Gaier & Ha (2019). In contrast to their work, we add the inner training loop whereas they evaluate networks using fixed weights (chapter 2.3.4). Also, our method is related to neural architecture search but uses multiple learning algorithms and has different goals (chapter 2.3.3). We run experiments on a set of reinforcement learning (RL) environments and on image classification with MNIST. The implementation is available at https://github.com/jrieke/evolution-learning.

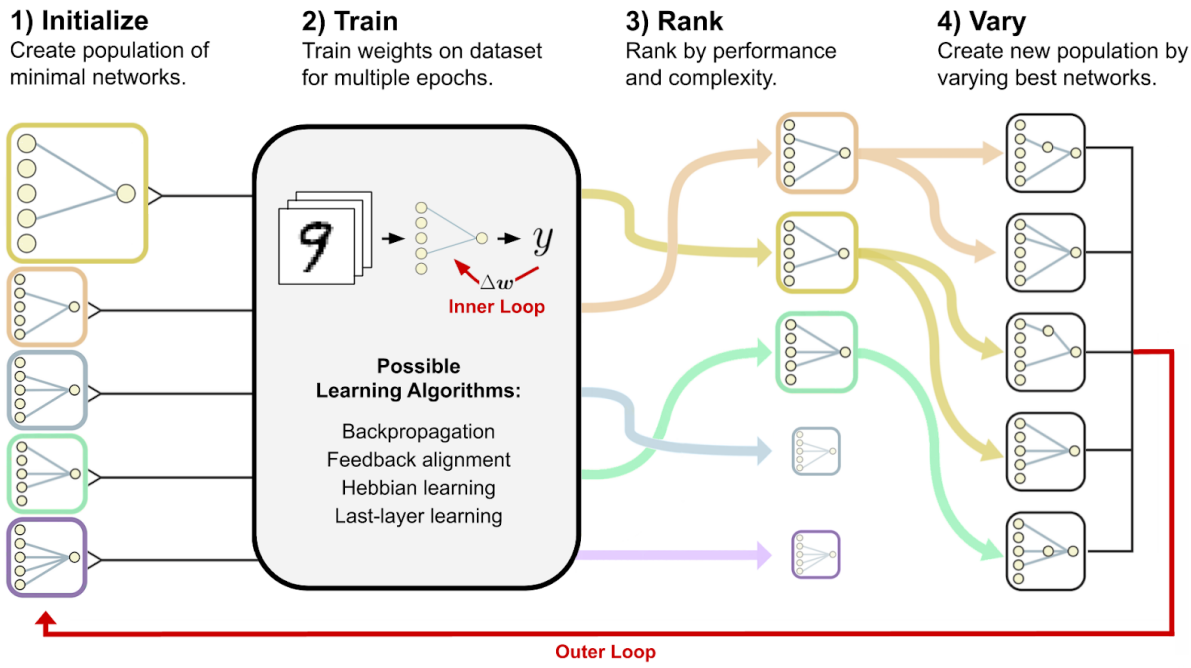

Figure 18: Schematic of our approach. Adapted from: Gaier & Ha (2019).

3.1 Outer loop: Evolutionary Algorithm

The evolutionary algorithm to find network architectures (outer loop) is loosely based on NEAT, a popular algorithm for neuroevolution (chapter 2.3.2; Stanley & Miikkulainen 2002). We use it in a similar setup to Gaier & Ha (2019). However, we use only a single fixed activation function (whereas Gaier & Ha use multiple activation functions and introduce an additional mutation to change them) and alter some hyperparameters (see chapter 3.3 for details). The outer loop consists of the following four steps:

Initialization. We start by initializing a population of simple networks. Each network has a fixed number of input neurons and output neurons (i.e. no hidden layer). The connections between input and output neurons are initialized with a given probability, so that each network contains some random connections. The neurons have no bias, use the ReLU activation function at hidden layers and a softmax at the output layer for the classification task.

Training and evaluation. After initialization, all networks are trained on the given task or dataset. This is the inner loop of our algorithm. We use one of four learning algorithms for training, which we explain below (chapter 3.2). Regardless of the method, this step yields a performance measure for each network in the population, which we call its reward (cumulative reward for RL environments, negative cross-entropy loss for classification).

Ranking. Next, we rank all networks in the population, so that the best ones can be selected for reproduction. Two objectives are used for ranking: 1) The reward on the task or dataset as evaluated above, 2) the complexity of the network (specifically, we use the reciprocal of the number of connections in order to penalize complex networks; Clune et al. 2013). This ensures that the simpler network is prefered if two networks perform equally well. We use the multi-objective function NSGA-II, which ranks according to both objectives at the same time, based on dominance relations (Deb et al. 2002). While this intentionally encourages simple networks, we still want to give complex network structures a chance to develop. Therefore, we only use both objectives in 20 % of all generations; in all other generations, we rank solely based on the reward.

Reproduction. Now that the networks are ranked, we want to take the best ones and reproduce them for the next generation. We first apply elitism (i.e. the 20 % best networks are directly advanced to the next generation) and culling (i.e. the 20 % worst networks are excluded from the breeding pool). The remaining networks are subject to tournament selection: We pick a fixed number of random networks, choose the best one, perform several mutations, add it to the new population, and repeat. In comparison to picking the best networks from the entire population, tournament selection increases the variety of architectures. We perform one of two mutations with given probabilities: Either, we add a connection between two existing neurons, or we add a (hidden) neuron by splitting an existing connection in two. We do not combine parent networks via crossover.

After reproduction, the new population of networks is trained, evaluated, ranked and reproduced. We repeat this process for a fixed number of generations.

3.2 Inner loop: Learning algorithms

In the inner loop, we train and evaluate all networks in the population. First, the weights of all connections are initialized from a normal distribution (standard deviation 0.1). Then, we train for multiple epochs on the complete training set, using the Adam optimizer (Kingma & Ba 2014). We use one of four learning algorithms for training, which are outlined below. They differ in their level of complexity and biological plausibility. See also chapter 2.2 for a detailed review of learning algorithms.

Backpropagation. In this setting, we use the standard backpropagation of errors method to train the network (Rumelhart et al. 1986; chapter 2.2.2). With backpropagation, our approach becomes similar to neural architecture search, even though the specific methodology and goals differ (chapter 2.3.3).

Feedback alignment. In this setting, we use feedback alignment, a variant of backpropagation which tackles the weight sharing problem (Lillicrap et al. 2016; chapter 2.2.3). Specifically, we use similar rules as backpropagation to calculate the gradients (equation 5) but replace the transposed forward weight with random, fixed feedback weights (equation 8), sampled from the normal distribution. As the forward connections in our networks are sparse, we use feedback connections only in those places where forward connections exist.

Hebbian learning. In this setting, we train the final classification layer of the network (i.e. all weights connecting to output neurons) with normal gradient descent (equation 4 and equation 6). The weights of hidden connections are trained with Oja's rule, a learning rule based on Hebbian learning (Hebb 1949; chapter 2.2.8). For a given neuron, the vector $\boldsymbol{w}$ of its incoming connections changes according to:

$$\Delta \boldsymbol{w} = \eta y(\boldsymbol{x} - y\boldsymbol{w}) \tag{16}$$

Here, $\boldsymbol{x}$ is the input vector to the neuron and y is its output. In comparison to the simple pre-times-post form of Hebbian learning (equation 14), Oja's rule adds a normalization term. For a given neuron, this term normalizes the vector of incoming weights so that its Euclidean norm is one, and stabilizes the weight update. To keep the algorithm as simple as possible, we do not incorporate additional modifications that were shown to increase the performance of Hebbian learning (Wadhwa & Madhow 2016). Note that all weight updates in this setting are local (i.e. they only use information adjacent to a synapse) and therefore biologically plausible. Training the output weights with gradient descent does not impair biological plausibility, as no gradients are propagated through the network.

Last-layer learning. In this setting, we train only the final classification layer of the network, using gradient descent (equation 4 and equation 6). All weights below the final layer are not trained but just initialized at random and kept fixed.

After training, the inner loop has to assign a reward to each network in the population. This reward determines how well the (trained) network performs on the given task and is used for ranking and reproduction in the evolutionary algorithm. For the RL tasks, this reward is the cumulative reward from the environment, averaged over multiple trials. For classification, we use the negative cross-entropy loss on the train set. To save up on computation, we do not re-evaluate the network after training but simply use the reward during the last epoch of training (averaged over batches). Note that this metric is recorded while the network is still improving, so it may not represent the “true” reward on the training set.

3.3 Hyperparameters

Table 2 lists all hyperparameters for the MNIST dataset. These values are compared against the hyperparameters for the MNIST experiment in Gaier & Ha (2019), who use almost the same evolutionary algorithm. In contrast to their parameters, we introduce three changes:

1. Smaller population and tournament size, fewer generations. This decreases the runtime of our experiments, which is already quite long due to the inner training loop.
2. A single fixed activation function (ReLU). Gaier & Ha evaluate networks with fixed weights. In order to increase the computational flexibility of these weight agnostic networks, they use multiple activation functions. Because our weights are trained, we do not need this additional flexibility.
3. More mutations per generation, higher probability of adding connections compared to neurons, using the complexity objective for ranking less often. These changes ensure that the networks can grow quickly during evolution. In preliminary experiments, we found that many connections are required to make proper use of the learning algorithms.

3.4 Parallelization

We implement our approach in Python 3, making use of the PyTorch library for neural networks (Paszke 2019). To speed up experiments, we parallelize with the joblib library (Varoquaux 2020): At each step of the evolution loop, we have to train hundreds of networks. These training procedures are independent, so they can be easily parallelized. For all experiments in this thesis, we used 36-core CPU instances from Amazon Web Services, and distributed networks equally across all cores. This way, a single experiment can be run within a few days. We did not use GPUs because we found that for our configuration, the massive parallelization possible on multi-core CPUs typically outweighs the speed benefits of GPUs.

Table 2: Hyperparameters and comparison to Gaier & Ha (2019).
* = Gaier & Ha (2019) report a tournament size of 32 in their paper but use 64 in their code. We notified the authors about this bug, so the numbers in the online version of their paper might be adapted after the completion of this thesis.

Hyperparameter	Ours	Weigh agnostic neural networks in Gaier & Ha (2019) – MNIST experiment
Generations	1000	4096
Population size	180	960
Tournament size	32	64*
Elite ratio (%)	20	20
Cull ratio (%)	20	20
Probability to use complexity objective (%)	20	80
Probability to add connection (%)	80	25
Probability to add node (%)	20	25
Probability to change activation (%)	-	50
Activation function(s)	ReLU	Linear, Step, Sine, Cosine, Gaussian, Tanh, Sigmoid, Inverse, Absolute, ReLU
Initial active connections (%)	5	5
Mutations per generation	5	1
Standard deviation for weight initialization	0.1	-
Optimizer	Adam	-
Learning rate	0.01	-
Batch size	256	-
Epochs per generation	5	-

4. Results

4.1 Replicating the results from Gaier & Ha (2019)

As a first step, we replicated the results from Gaier & Ha (2019) on weight agnostic neural networks (WANNs; chapter 2.3.4). This serves both as a validation of their paper as well as our own implementation.

In order to make our networks equal to WANNs, we change three things:

1. We replace the inner loop of our algorithm (chapter 3.2) with a simple weight agnostic evaluation. Instead of training the network, we set a fixed weight value that is shared across all connections. Each network is evaluated on 16 trials (RL environments) or 1000 images (classification). In order to test the performance of the architecture (and not just the specific weight value), we evaluate for each shared weight value out of {-2, -1, -0.5, 0.5, 1, 2}. Based on the results, we calculate two reward metrics: 1) The average reward across all weight values, 2) the reward for the best performing weight value. The objectives for ranking are adapted accordingly: In those generations, in which we use the complexity objective (chapter 3.1), we rank according to mean reward and complexity now. In those generations, in which we do not use the complexity objective, we use both reward metrics.

2. We use multiple activation functions (see table 2 for a list). This gives the network additional computational flexibility (which is vital because due to the shared weights, it does not have the flexibility that usually comes along with different weight values).

3. We use the same hyperparameters as in Gaier & Ha (2019) (see also table 2), but only run each experiment three times (instead of nine times). We picked the best run based on the mean reward across weight values observed during evolution.

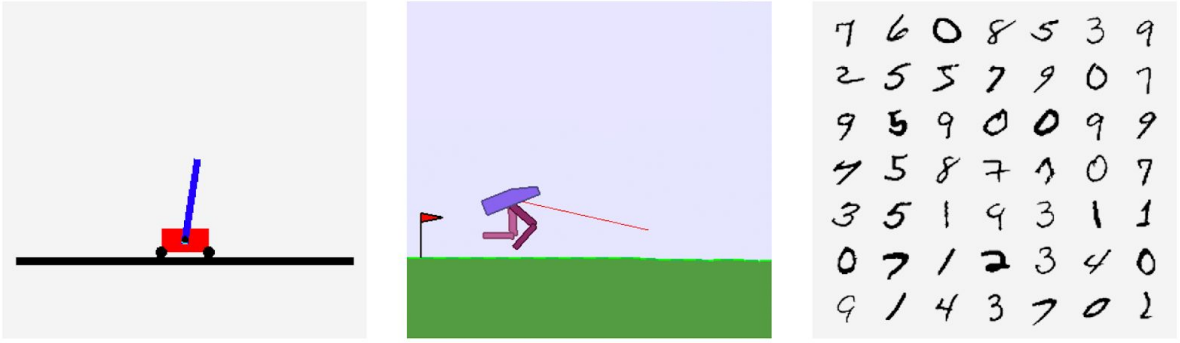

Figure 19: The three tasks used to evolve weight agnostic networks.
From left to right: CartPoleSwingUp, BipedalWalker-v2, MNIST.

We ran experiments for three tasks that were also used in the WANN paper (fig. 19): CartPoleSwingUp is a simple reinforcement learning (RL) environment that requires the agent to steer a cart in order to balance a pole (the pole is hanging down at the beginning). BipedalWalker-v2 is another RL environment with a two-legged robot that needs to be maneuvered across a terrain. In addition to the RL environments, we evolve networks on a classification task, the well-known MNIST dataset of handwritten digits (LeCun et al. 2020). The input is a grayscale image of a digit, which needs to be classified into ten different categories (digits 0-9). The dataset consists of 60 000 training and 10 000 test images. As in Gaier & Ha (2019), we preprocess these images by rescaling to 16 x 16 and deskewing (even though the latter did not seem to have a major impact on performance in preliminary experiments). See Gaier & Ha (2019) for a more detailed description of these tasks.

The results of our implementation are shown in table 3. To get these final numbers, we used the same evaluation methods as in the code of Gaier & Ha (2019): For the RL tasks, we ran 100 trials for each of the weight values used during evolution (-2, -1, -0.5, 0.5, 1, 2). For MNIST, we evaluated on the MNIST test set for ten linearly spaced weight values in the range [-2, 2]. The setting “Random shared weight” in table 3 is the mean across all tested weight values, while “Tuned shared weight” is the result for the best performing weight value. We did not evaluate networks in the settings “Random weights” and “Tuned weights” from the original paper.

Our results are in agreement with the original paper (table 3). Slight differences on the RL environments could simply be an effect of random fluctuations between runs (especially considering that the original paper picked the best results out of nine runs, while we only ran three times). For MNIST, we found a slightly higher accuracy in the “Random shared weight” setting, combined with a lower standard deviation. The comparable results make us confident that our implementation works correctly. They also validate the results of the original paper.

Table 3: Results of best evolved weight agnostic networks on different tasks.
CartPoleSwingUp and Bipedal-Walker-v2 show cumulative rewards averaged over 100 trials, while MNIST shows test set accuracies. Note that the standard deviation for CartPoleSwingUp and BipedalWalker-v2 is across different trials on the same environment, while the standard deviation on MNIST is across weight values.

Task	Setting	Gaier & Ha (2019)	Ours
CartPoleSwingUp	Random shared weight	515 ± 58	**519 ± 39**
	Tuned shared weight	**723 ± 16**	608 ± 46
BipedalWalker-v2	Random shared weight	**51 ± 108**	42 ± 37
	Tuned shared weight	**261 ± 58**	183 ± 77
MNIST	Random shared weight	82.0 % ± 18.7 %	**88.3 % ± 5.9 %**
	Tuned shared weight	**91.9 %**	91.8 %

4.2 Combining evolution and learning

For the main experiments, we combined an evolutionary algorithm (outer loop) with four different learning algorithms (inner loop), as described in chapter 3. We ran all experiments on image classification with the MNIST dataset of handwritten digits. As before, all images were scaled to 16 x 16 pixels and then deskewed.

In preliminary experiments, we found that our evolutionary approach suffers from a serious flaw: It mostly adds direct connections from the input to the output layer, neglecting potential new connections between intermediate network layers. This is because direct input-output connections improve the accuracy of the network the most. This behavior is problematic for our analysis because connections in the final output layer are trained in the same way with all four learning algorithms (by direct gradient descent). Therefore, we could hardly observe any differences between the evolved architectures for those four experiment settings (both in terms of performance and architecture).

To get around this problem, we decided to introduce an additional constraint: We use a fixed, small number of output connections in all networks and do not alter this configuration during evolution. Specifically, we initialize the networks with a fixed number of connections between input and output neurons (around 5 % of all possible connections) and add some random hidden neurons and connections (by splitting around 50 % of existing connections into two connections and a hidden neuron). Note that this can result in multiple hidden layers because we allow the algorithm to split connections that were inserted previously. Fig. 20 shows an exemplary network after initialization. During evolution, we do not add any new connections to output neurons, only to hidden neurons. Therefore, the sparse output layer stays fixed and the evolutionary algorithm has to adapt to it. This constraint makes the task harder to solve and forces the networks to rely more on hidden neurons and connections. This is just what we want because these hidden connections are trained differently by the four learning algorithms.

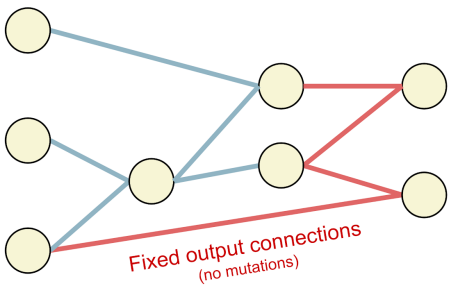

Figure 20: Example network after initialization.
The network is initialized with a fixed, sparse output layer (red). Then, some hidden neurons and connections (blue) are added, which are later modified through evolution. This network is only for illustration; the networks used on MNIST are obviously larger.

4.2.1 Performance of evolved networks

Table 4 lists the best networks that were found through evolution, each evaluated on the test set of MNIST. We find that more complex learning algorithms achieve better results: The networks evolved with backpropagation and feedback alignment have the highest accuracies, Hebbian learning is considerably lower, and training only the last layer of the network reaches the lowest accuracy.

Fig. 21 shows the evolution process in more detail: During the first few generations, the networks improve similarly for all learning algorithms. However, the accuracies for Hebbian learning and last-layer learning stagnate quite early (after 50-100 generations) and improve only slightly thereafter. The networks for backpropagation and feedback alignment, however, keep improving almost throughout the experiment. We find a similar picture for the number of connections in the networks (fig. 21 bottom): Hebbian learning and last-layer learning add connections slowly after an initial ramp-up. Apparently, these learning algorithms do not benefit from larger network sizes after a certain point. For backpropagation and feedback alignment, however, evolution keeps adding connections, which improves performance.

Does the evolutionary algorithm create capable architectures, or do the networks just get better the more connections they have – regardless of their exact layout? To analyze this question, we compare against randomly mutated networks. We trained and tested these networks with the same hyperparameters as the evolved networks. Table 4 lists their accuracies on MNIST, averaged across 20 different random networks. To get a fair comparison, we used the same number of connections as the evolved networks for each setting. Fig. 22 shows a more detailed comparison for different network sizes. For all learning algorithms, the evolved networks clearly outperform randomly mutated networks. Especially for Hebbian learning and last-layer learning, this difference is quite pronounced: E.g., for Hebbian learning, the best evolved network (table 4) has an accuracy more than four standard deviations higher than the mean of randomly mutated networks. This makes it practically impossible to find such a high-performing network architecture by chance.

Table 4: Accuracies of evolved network architectures on the MNIST test set.
We compare with randomly mutated networks (values are mean ± standard deviation over 20 independent networks). To get a fair comparison, the randomly mutated networks have the same number of connections as the best evolved network for each learning algorithm.

	Best evolved network	**Randomly mutated networks (with same number of connections)**
Backpropagation	92.9 %	82 % ± 4 % (best of 20: 87.7 %)
Feedback alignment	91.9 %	80 % ± 4 % (best of 20: 85.5 %)
Hebbian learning	74.6 %	49 % ± 6 % (best of 20: 60.1 %)
Last-layer learning	67.8 %	43 % ± 6 % (best of 20: 57.1 %)

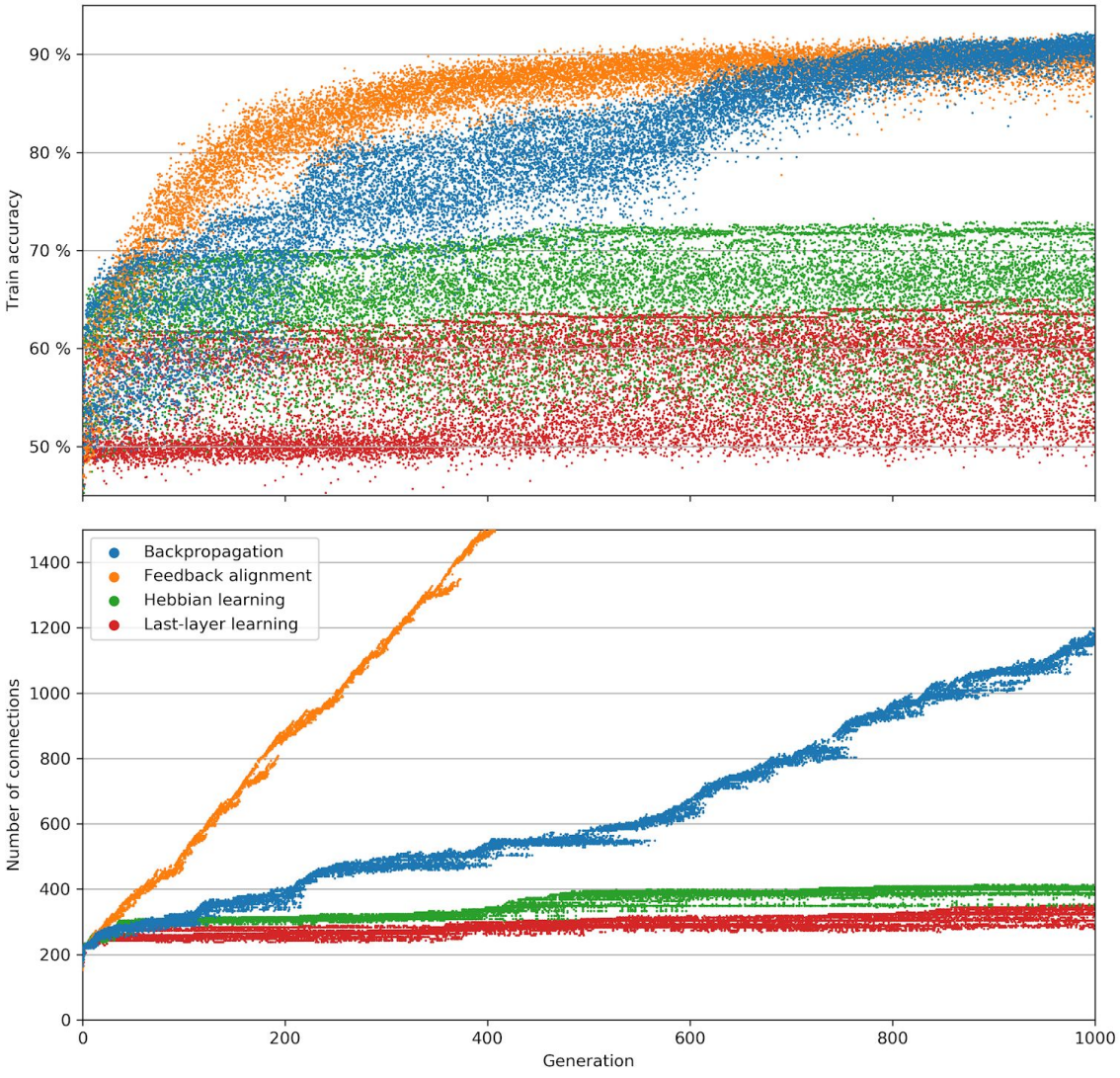

Figure 21: Development of the population of networks during evolution.
Each dot represents one network in the population. To increase visibility, we show only 10 % of all networks. **Top:** Train accuracy on MNIST. Note that these values are averaged over the last epoch of training, so they differ from the test set accuracies in table 4. **Bottom:** Number of connections in each network.

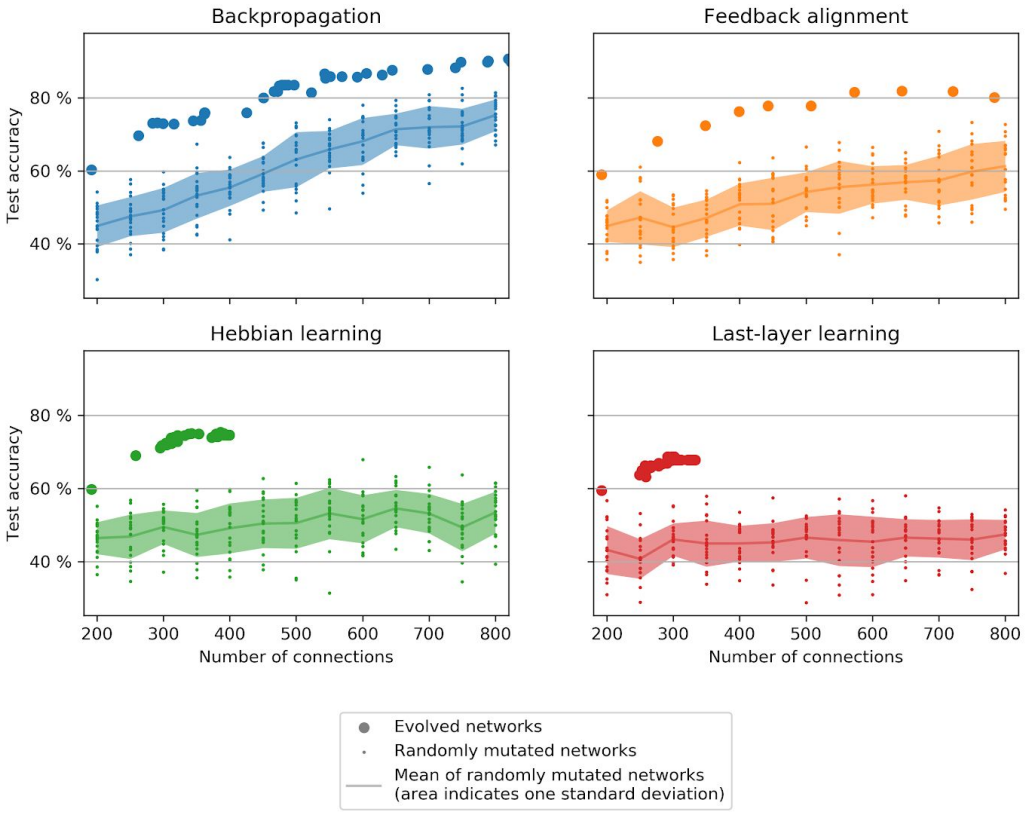

Figure 22: Evolved vs. randomly mutated networks. We show the best networks (large dots) at different stages of evolution and compare with randomly mutated networks of the same size (small dots; 20 independent networks per evaluation). We also show the mean (line) and standard deviation (area) over these randomly mutated networks.

4.2.2 Learning within a generation

Next, we analyze the learning process of networks within a generation. Fig. 23 shows learning curves at different points of evolution. We find that the accuracy usually plateaus within a few generations. This is important because the evolutionary algorithm needs to evaluate the network performance after convergence – otherwise, the benefit of small changes to the network (e.g. a single added connection) might not show up. In the experiments, we trained for five epochs (red line in fig. 23), after which most networks seem to have reached their final performance. As expected, we see that learning progresses rather slowly at the beginning of evolution but improves in later generations. Also, the learning curves show more variation early on because the network architectures are not optimized yet.

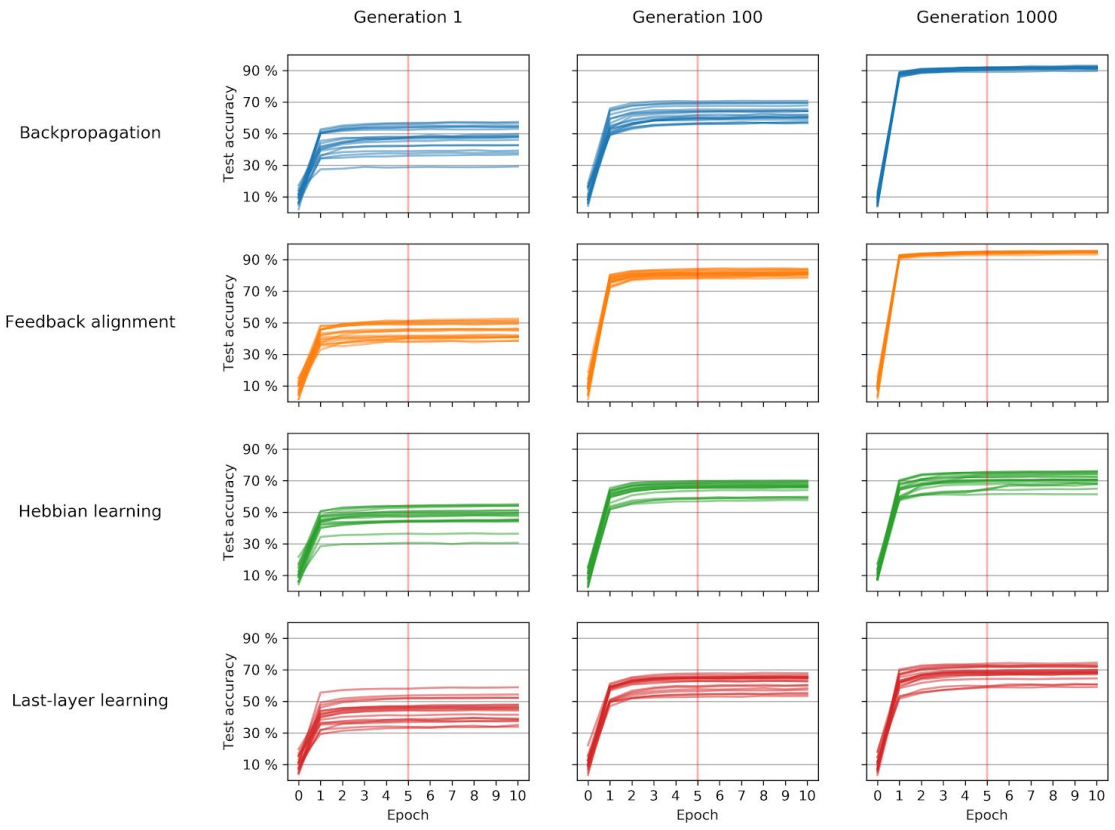

Figure 23: Learning curves on MNIST at different stages of evolution.
Each line represents one network in the population. To increase visibility, we show only 10 % of all networks. Epoch 0 shows performance before training. Note that we stop training during the evolution loop after five epochs (indicated by red vertical line).

4.2.3 Network topologies

Finally, we analyze the architectures of the evolved networks. Specifically, we are interested in seeing whether the optimal topologies differ between the four learning algorithms. Table 5 lists the numbers of connections, neurons, and layers in a network, averaged across the population. We find strong differences: The networks evolved for backpropagation and feedback alignment have many neurons and connections, which are arranged in deep architectures. The networks evolved for Hebbian learning and last-layer learning have considerably fewer connections, neurons, and layers. Apparently, they did not benefit from the added computational flexibility that comes along with more connections.

Table 5: Topological features of networks after 1000 generations.
Mean ± standard deviation across the population. Note that "layers" in our networks are not densely connected layers (as is usual in artificial neural networks). Instead, we sort the sparsely connected neurons in our networks into layers based on their connectivity.

	Connections	Neurons	Layers
Backpropagation	1167 ± 13	471 ± 2	11.2 ± 0.4
Feedback alignment	3018 ± 6	790 ± 3	24.8 ± 0.4
Hebbian learning	400 ± 13	356 ± 2	7.1 ± 0.3
Last-layer learning	336 ± 16	358 ± 4	6.2 ± 0.4

Besides these basic topological features, we calculated node degrees in the evolved networks. The node degree is the number of connections to or from a given neuron. It is an important measure in graph theory: The higher the node degrees in a network are, the more densely it is connected. Fig. 24 shows degree distributions in the evolved networks, averaged across the population. We find that backpropagation and feedback alignment have many neurons with high node degree, i.e. these networks are connected quite densely. In contrast, the networks for Hebbian learning and especially last-layer learning contain mostly neurons with low degree (< 5), i.e. few connections to other neurons. Apparently, these simple learning algorithms could benefit neither from large networks nor dense connections.

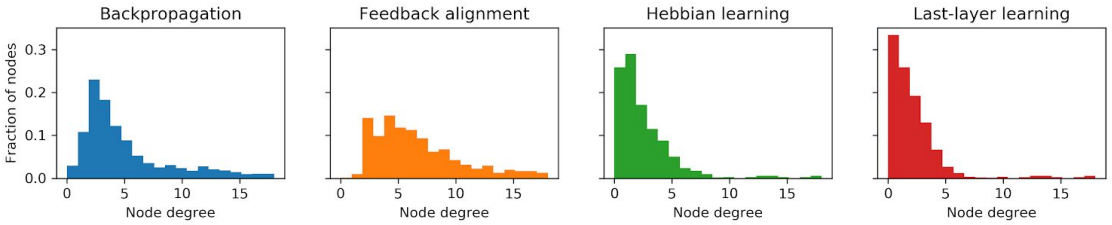

Figure 24: Degree distribution of networks after 1000 generations (averaged across the population).
The node degree is the number of connections to or from a given neuron.

5. Conclusion

How does the human brain achieve its intelligence? We can certainly say that two processes are involved: Evolution across generations and learning during an individual's lifetime. While evolution seems to primarily shape the basic architecture of the brain, learning adapts synaptic connections in order to tune the computations within neural circuits. From machine learning research, we know that both mechanisms can have a major impact on cognitive abilities: Most research in deep learning uses sophisticated learning algorithms to learn complex tasks in blank-slate networks, which shows the capabilities of learning through synaptic weight changes. On the other end of the spectrum, Gaier & Ha (2019) recently demonstrated that fine-tuned network architectures can solve certain tasks without any weight training at all.

In this thesis, we combine evolution and learning, using image classification on MNIST as a test case. We evolved the architectures of artificial neural networks with an evolutionary algorithm and used a number of different learning algorithms to train the synaptic weights. We show that both mechanisms interact with each other: The evolutionary algorithm successfully found network architectures that are particularly suited to each learning algorithm. We also show that the resulting networks differ between learning algorithms – both in terms of performance as well as topologies. Generally, we find that more sophisticated learning algorithms (e.g. backpropagation) achieve better results on image classification and can make use of larger, deeper, and more densely connected networks. Simpler and more biologically plausible learning algorithms (e.g. Hebbian learning), however, do not seem to benefit as much from increased network sizes.

Our work relates to a central question in deep learning and biology alike: How much innate structure should a neural network have? Is it helpful if the network architecture is heavily specialized and contains inductive biases (as the intricate wiring in biological brains seems to advocate)? Or does the network learn better if it starts from a blank slate and picks up all important knowledge from data (as mostly done in deep learning)? This nature vs. nurture debate has recently been discussed in a number of papers (Zador 2019, Marcus 2018, Marblestone et al. 2016). Based on the results of this thesis, we argue that some prior structure in network architectures (as we find through evolution) can definitely help the learning process.

This study presents a first attempt at exploring the combination of neuroevolution and biologically plausible learning algorithms. It was limited by strong computing and runtime requirements: In each experiment, we trained hundreds of networks across hundreds of generations. Even though we parallelized our implementation, these requirements were a major bottleneck for more sophisticated experimentation. As one consequence, we could only perform experiments on the relatively small MNIST dataset. Compared to modern image classification datasets, MNIST is quite easy to master, even without sophisticated networks (Illing et al. 2019). Therefore, we had to introduce an additional constraint on the output layer in order to see the effects of evolution and any differences between learning algorithms.

Partially based on these limitations, we propose four ideas for future research:

1. Using our approach on more complex image classification datasets, e.g. CIFAR10 (Krizhevsky 2009) or ImageNet (Deng et al. 2009). As mentioned, this would require more computing resources and optimizations. With more complex datasets, it should be easier to analyze the effects of evolution. However, it is not clear how our approach scales to such datasets, as it was recently shown that biologically plausible learning algorithms can easily fail on complex datasets (Bartunov et al. 2018).

2. Running the evolutionary algorithm on multiple datasets in parallel. Right now, the evolved network architectures obviously overfit to the dataset we use. In the brain, however, there seems to be a general circuit architecture, which can adapt to multiple functions (chapter 2.1.1). Inspired by this, one could potentially evolve network architectures that can adapt to multiple tasks through weight training.

3. Adding modularity to network architectures. The mammalian cortex seems to consist of repeatable elements, which serve as computational units (chapter 2.1.1). Similarly, our evolutionary algorithm could be adapted to evolve modules of neurons that are repeated to form complete networks. Note that this idea is similar to modular designs in neural architecture search (Reisinger et al. 2004). Alternatively, one could evolve rules for how networks are built instead of manipulating specific connections (Stanley et al. 2009). Adding modularity might also have a biological motivation, considering that biological genomes do not have the capacity to store the exact wiring of the entire brain (Zador et al. 2019).

4. Evolving feedback connections. Gaier & Ha (2019) showed that evolved feedforward architectures can solve complex tasks without weight training. Similarly, it should be possible to evolve the feedback connections in networks to produce more meaningful gradients. With such feedback connections, it might be possible to train networks efficiently with relatively simple learning algorithms (e.g. Hebbian learning).

In the implementation, we made sure to build a flexible framework to combine evolutionary and learning algorithms, which can be run efficiently through parallelization. We publish our code at https://github.com/jrieke/evolution-learning to encourage further experimentation.

Acknowledgments

I want to thank many people who have been involved in the creation of this thesis:

Prof. Matthew Larkum and Prof. Benjamin Grewe for supervising my thesis, providing me with insight into their work, and teaching me all kinds of neuroscience research that I had never heard of before.

João Sacramento, Maria Cervera, Johannes von Oswald, Christian Henning, and all other members of the Grewe lab in Zürich for giving me the idea for this thesis, their help in every aspect of its creation, endless discussions and the good work environment in Zürich.

My brother Christoph for helpful discussions and feedback.

The Swiss European Mobility Program for providing funding and the admin staff at BCCN Berlin and INI Zürich for taking care of all organizational hurdles.

References

Akrout, M., Wilson, C., Humphreys, P. C., Lillicrap, T., & Tweed, D. (2019). Deep Learning without Weight Transport. NeurIPS. Retrieved from http://arxiv.org/abs/1904.05391

Asl, M. M. (2018). Propagation Delays Determine the Effects of Synaptic Plasticity on the Structure and Dynamics of Neuronal Networks. Institute for Advanced Studies in Basic Sciences. https://doi.org/10.13140/RG.2.2.28601.88166

Bartunov, S., Santoro, A., Richards, B. A., Marris, L., Hinton, G. E., & Lillicrap, T. (2018). Assessing the Scalability of Biologically-Motivated Deep Learning Algorithms and Architectures. NeurIPS. https://doi.org/arXiv:1807.04587v1

Bengio, Y., Lee, D.-H., Bornschein, J., Mesnard, T., & Lin, Z. (2015). Towards Biologically Plausible Deep Learning. ArXiv Preprint. Retrieved from http://arxiv.org/abs/1502.04156

Bienenstock, E. L., Cooper, L. N., & Munro, P. W. (1982). Theory for the Development of Neuron Selectivity: Orientation Specificity and Binocular Interaction in Visual Cortex. The Journal of Neuroscience, 2(1), 32–48. https://doi.org/10.1093/aje/kwx073

Bittner, K. C., Milstein, A. D., Grienberger, C., Romani, S., & Magee, J. C. (2017). Behavioral time scale synaptic plasticity underlies CA1 place fields. Science, 357(6355), 1033–1036. https://doi.org/10.1126/science.aan3846

Citri, A., & Malenka, R. C. (2008). Synaptic plasticity: Multiple forms, functions, and mechanisms. Neuropsychopharmacology, 33(1), 18–41. https://doi.org/10.1038/sj.npp.1301559

Clopath, C., Büsing, L., Vasilaki, E., & Gerstner, W. (2010). Connectivity reflects coding: A model of voltage-based STDP with homeostasis. Nature Neuroscience, 13(3), 344–352. https://doi.org/10.1038/nn.2479

Clune, J., Mouret, J.-B., & Lipson, H. (2013). The Evolutionary Origins of Modularity. Proceedings of the Royal Society B, 280, 41–42. https://doi.org/10.7551/978-0-262-32621-6-ch007

Collingridge, G. L., & Bliss, T. V. P. (1987). NMDA receptors - their role in long-term potentiation. Trends in Neurosciences, 10(7), 288–293. https://doi.org/10.1016/0166-2236(87)90175-5

Crick, F. (1989). The recent excitement about neural networks. Nature. https://doi.org/10.1038/337129a0

da Costa, N. M., & Martin, K. A. C. (2010). Whose cortical column would that be? Frontiers in Neuroanatomy, 4. https://doi.org/10.3389/fnana.2010.00016

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, 6(2), 182–197. https://doi.org/10.1109/4235.996017

Deng, J., Dong, W., Socher, R., Li, L.-J., Kai Li, & Li Fei-Fei. (2009). ImageNet: A large-scale hierarchical image database. CVPR. https://doi.org/10.1109/CVPR.2009.5206848

Douglas, R. J., Martin, K. A. C., & Whitteridge, D. (1989). A Canonical Microcircuit for Neocortex. Neural Computation, 1, 480–488. https://doi.org/10.1162/neco.1989.1.4.480

Douglas, R. J., & Martin, K. A. C. (2004). Neuronal Circuits of the Neocortex. Annual Review of Neuroscience, 27, 419–451. https://doi.org/10.1016/j.cub.2007.04.024

Douglas, R. J., & Martin, K. A. C. (2007). Mapping the matrix: the ways of neocortex. Neuron, 56(2), 226–238. https://doi.org/10.1016/j.neuron.2007.10.017

Elsken, T., Metzen, J. H., & Hutter, F. (2019). Neural Architecture Search: A Survey. Journal of Machine Learning Research, 20, 1–21. Retrieved from http://arxiv.org/abs/1808.05377

Feldman, D. E. (2012). The spike timing dependence of plasticity. Neuron, 75(4), 556–571. https://doi.org/10.1016/j.neuron.2012.08.001.The

Gaier, A., & Ha, D. (2019). Weight Agnostic Neural Networks. NeurIPS. Retrieved from http://arxiv.org/abs/1906.04358

Gerstner, W., & Kistler, W. M. (2002). Spiking Neuron Models: Single Neurons, Populations, Plasticity. Cambridge: Cambridge University Press.

Gilbert, C. D., & Li, W. (2013). Top-down influences on visual processing. Nature Reviews Neuroscience, 14(5), 350–363. https://doi.org/10.1038/nrn3476

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural Turing Machines. ArXiv Preprint. Retrieved from http://arxiv.org/abs/1410.5401

Guerguiev, J., Lillicrap, T. P., & Richards, B. A. (2017). Towards deep learning with segregated dendrites. ELife, 6. https://doi.org/10.7554/eLife.22901

Ha, D. (2017). A Visual Guide to Evolution Strategies. Retrieved January 21, 2020, from http://blog.otoro.net/2017/10/29/visual-evolution-strategies/

Harris, K. D., & Mrsic-Flogel, T. D. (2013). Cortical connectivity and sensory coding. Nature, 503(7474), 51–58. https://doi.org/10.1038/nature12654

Hebb, D. O. (1949). The organization of behavior: A neuropsychological theory. New York: Wiley.

Herculano-Houzel, S. (2009). The human brain in numbers: A linearly scaled-up primate brain. Frontiers in Human Neuroscience, 3. https://doi.org/10.3389/neuro.09.031.2009

Horton, J. C., & Adams, D. L. (2005). The cortical column: a structure without a function. Philosophical Transactions of the Royal Society B: Biological Sciences, 360(1456), 837–862. https://doi.org/10.1098/rstb.2005.1623

Hubel, D. H., & Wiesel, T. N. (1972). Laminar and columnar distribution of geniculo-cortical fibers in the macaque monkey. Journal of Comparative Neurology, 146(4), 421–450. https://doi.org/10.1002/cne.901460402

Illing, B., Gerstner, W., & Brea, J. (2019). Biologically plausible deep learning - But how far can we go with shallow networks? Neural Networks, 118, 90–101. https://doi.org/10.1016/j.neunet.2019.06.001

Kaas, J. H. (2011). Neocortex in early mammals and its subsequent variations. Annals of the New York Academy of Sciences, 1225(1), 28–36. https://doi.org/10.1111/j.1749-6632.2011.05981.x

Kell, A. J. E., Yamins, D. L. K., Shook, E. N., Norman-Haignere, S. V., & McDermott, J. H. (2018). A Task-Optimized Neural Network Replicates Human Auditory Behavior, Predicts Brain Responses, and Reveals a Cortical Processing Hierarchy. Neuron, 98(3), 630–644. https://doi.org/10.1016/j.neuron.2018.03.044

Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. ICLR. Retrieved from http://arxiv.org/abs/1412.6980

Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. Retrieved from https://www.cs.toronto.edu/~kriz/cifar.html

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. NeurIPS. Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Kwan, K. Y., Šestan, N., & Anton, E. S. (2012). Transcriptional co-regulation of neuronal migration and laminar identity in the neocortex. Development, 139(9), 1535–1546. https://doi.org/10.1242/dev.069963

Larkum, M. (2013). A cellular mechanism for cortical associations: An organizing principle for the cerebral cortex. Trends in Neurosciences, 36(3), 141–151. https://doi.org/10.1016/j.tins.2012.11.006

Le Cun, Y. (1986). Learning Process in an Asymmetric Threshold Network. Disordered Systems and Biological Organization, 233–240. https://doi.org/10.1007/978-3-642-82657-3_24

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436–444. https://doi.org/10.1038/nature14539

LeCun, Y., Cortes, C., & Burges, C. J. (2020). The MNIST database of handwritten digits. Retrieved January 21, 2020, from http://yann.lecun.com/exdb/mnist/

LeCun, Y., Haffner, P., Bottou, L., & Bengio, Y. (1999). Object Recognition with Gradient-Based Learning. Shape, Contour and Grouping in Computer Vision, 319–345.

Lee, D.-H., Zhang, S., Fischer, A., & Bengio, Y. (2015). Difference Target Propagation. ECML/PKDD.

Liao, Q., Leibo, J. Z., & Poggio, T. (2016). How Important Is Weight Symmetry in Backpropagation? AAAI. Retrieved from https://arxiv.org/abs/1510.05067

Lillicrap, T. P., Cownden, D., Tweed, D. B., & Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. Nature Communications, 7(13276). https://doi.org/10.1038/ncomms13276

Maffei, A. (2011). The many forms and functions of long term plasticity at GABAergic synapses. Neural Plasticity. https://doi.org/10.1155/2011/254724

Manita, S., Suzuki, T., Homma, C., Matsumoto, T., Odagawa, M., Yamada, K., … Murayama, M. (2015). A Top-Down Cortical Circuit for Accurate Sensory Article A Top-Down Cortical Circuit for Accurate Sensory Perception. Neuron, 86(5), 1304–1316. https://doi.org/10.1016/j.neuron.2015.05.006

Marblestone, A. H., Wayne, G., & Kording, K. P. (2016). Towards an integration of deep learning and neuroscience. Frontiers in Computational Neuroscience. https://doi.org/10.3389/fncom.2016.00094

Marcus, G. (2018). Innateness, AlphaZero, and Artificial Intelligence. ArXiv Preprint. Retrieved from http://arxiv.org/abs/1801.05667

Markram, H., Lübke, J., Frotscher, M., & Sakmann, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. Science, 275(5297), 213–215. https://doi.org/10.1126/science.275.5297.213

Markram, H., Toledo-Rodriguez, M., Wang, Y., Gupta, A., Silberberg, G., & Wu, C. (2004). Interneurons of the neocortical inhibitory system. Nature Reviews Neuroscience, 5(10), 793–807. https://doi.org/10.1038/nrn1519

Martin, S. J., Grimwood, P. D., & Morris, R. G. M. (2000). Synaptic Plasticity and Memory: An Evaluation of the Hypothesis. Annual Review of Neuroscience, 23, 649–711.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. a, Veness, J., Bellemare, M. G., … Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529–533. https://doi.org/10.1038/nature14236

Mountcastle, V. B. (1957). Modality and Topographic Properties of Single Neurons of Cat's Somatic Sensory Cortex. Journal of Neurophysiology, 20(4), 408–434.

Naud, R., & Sprekeler, H. (2018). Sparse bursts optimize information transmission in a multiplexed neural code. Proceedings of the National Academy of Sciences of the United States of America, 115(27), E6329–E6338. https://doi.org/10.1073/pnas.1720995115

Nielsen, M. (2015). Neural Networks and Deep Learning. Determination Press. Retrieved from http://neuralnetworksanddeeplearning.com/

Nieuwenhuys, R. (1994). The neocortex. An overview of its evolutionary development, structural organization and synaptology. Anatomy and Embryology, 190(4), 307–337.

Nøkland, A. (2016). Direct Feedback Alignment Provides Learning in Deep Neural Networks. NeurIPS. Retrieved from https://arxiv.org/abs/1609.01596

Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. (2019). Regularized Evolution for Image Classifier Architecture Search. AAAI. https://doi.org/10.1609/aaai.v33i01.33014780

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., … Kurakin, A. (2017). Large-scale evolution of image classifiers. ICML. Retrieved from https://arxiv.org/abs/1703.01041

Reisinger, J., Stanley, K. O., & Miikkulainen, R. (2004). Evolving reusable neural modules. GECCO. https://doi.org/10.1007/978-3-540-24855-2_7

Richards, B. A., & Lillicrap, T. P. (2018). Dendritic solutions to the credit assignment problem. Current Opinion in Neurobiology, 54, 28–36. https://doi.org/10.1016/j.conb.2018.08.003

Richards, B. A., Lillicrap, T. P., Beaudoin, P., Bengio, Y., Bogacz, R., Christensen, A., … Kording, K. P. (2019). A deep learning framework for neuroscience. Nature Neuroscience, 22(11), 1761–1770. https://doi.org/10.1038/s41593-019-0520-2

Roelfsema, P. R., & Holtmaat, A. (2018). Control of synaptic plasticity in deep cortical networks. Nature Reviews Neuroscience, 19(3), 166–180. https://doi.org/10.1038/nrn.2018.6

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533–536. https://doi.org/10.1038/323533a0

Sacramento, J., Bengio, Y., Costa, R. P., & Senn, W. (2018). Dendritic cortical microcircuits approximate the backpropagation algorithm. NeurIPS. Retrieved from https://arxiv.org/abs/1810.11393

Scellier, B., & Bengio, Y. (2017). Equilibrium Propagation: Bridging the Gap Between Energy-Based Models and Backpropagation. Frontiers in Computational Neuroscience. https://doi.org/10.3389/fncom.2017.00024

Scellier, B., Goyal, A., Binas, J., Mesnard, T., & Bengio, Y. (2018). Generalization of Equilibrium Propagation to Vector Field Dynamics. ArXiv Preprint. Retrieved from http://arxiv.org/abs/1808.04873

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484–489. https://doi.org/10.1038/nature16961

Song, S., Sjöström, P. J., Reigl, M., Nelson, S., & Chklovskii, D. B. (2005). Highly nonrandom features of synaptic connectivity in local cortical circuits. PLoS Biology, 3(3), 0507–0519. https://doi.org/10.1371/journal.pbio.0030068

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. Evolutionary Computation, 10(2), 99–127. Retrieved from http://nn.cs.utexas.edu/?stanley:ec02

Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. Artificial Life, 15(2), 185–212. https://doi.org/10.1162/artl.2009.15.2.15202

Steiner, B., Devito, Z., Chintala, S., Gross, S., Paszke, A., Massa, F., … Bai, J. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS. Retrieved from https://arxiv.org/abs/1912.01703

Sukel, K. (2019). Neuroanatomy: The Basics. Retrieved January 21, 2020, from https://www.dana.org/article/neuroanatomy-the-basics/

Varoquaux, G. (2020). joblib. GitHub. Retrieved from https://github.com/joblib/joblib

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., … Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature, 575(7782), 350–354. https://doi.org/10.1038/s41586-019-1724-z

Wadhwa, A., & Madhow, U. (2016). Learning Sparse, Distributed Representations using the Hebbian Principle. ArXiv Preprint. Retrieved from https://arxiv.org/abs/1611.04228

Werbos, P. (1974). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. Harvard University.

Weston, J., Chopra, S., & Bordes, A. (2015). Memory Networks. ICLR. Retrieved from http://arxiv.org/abs/1410.3916

Whittington, J. C. R., & Bogacz, R. (2019). Theories of Error Back-Propagation in the Brain. Trends in Cognitive Sciences, 23(3), 235–250. https://doi.org/10.1016/j.tics.2018.12.005

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., … Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. ArXiv Preprint. Retrieved from http://arxiv.org/abs/1609.08144

Xiao, W., Chen, H., Liao, Q., & Poggio, T. (2018). Biologically-Plausible Learning Algorithms Can Scale to Large Datasets. ArXiv Preprint. Retrieved from https://arxiv.org/abs/1811.03567

Yamins, D. L. K., & DiCarlo, J. J. (2016). Using goal-driven deep learning models to understand sensory cortex. Nature Neuroscience, 19(3), 356–365. https://doi.org/10.1038/nn.4244

Young, H., Belbut, B., Baeta, M., & Petreanu, L. (2019). Laminar-specific cortico-cortical loops in mouse visual cortex. BioRxiv Preprint. https://doi.org/10.1101/773085

Zador, A. M. (2019). A critique of pure learning and what artificial neural networks can learn from animal brains. Nature Communications, 10(1). https://doi.org/10.1038/s41467-019-11786-6

Zenke, F., & Ganguli, S. (2018). SuperSpike: Supervised learning in multilayer spiking neural networks. Neural Computation, 30(6), 1514–1541. https://doi.org/10.1162/neco_a_01086

Zoph, B., & Le, Q. V. (2017). Neural architecture search with reinforcement learning. ICLR. Retrieved from https://arxiv.org/abs/1611.01578