Self-Replication in Neural Networks

Thomas Gabor*
LMU Munich
thomas.gabor@ifi.lmu.de

Steffen Illium
LMU Munich

Maximilian Zorn
LMU Munich

Cristian Lenta
LMU Munich

Andy Mattausch
LMU Munich

Lenz Belzner
Technische Hochschule Ingolstadt

Claudia Linnhoff-Popien
LMU Munich

Abstract A key element of biological structures is self-replication. Neural networks are the prime structure used for the emergent construction of complex behavior in computers. We analyze how various network types lend themselves to self-replication. Backpropagation turns out to be the natural way to navigate the space of network weights and allows non-trivial self-replicators to arise naturally. We perform an in-depth analysis to show the self-replicators' robustness to noise. We then introduce artificial chemistry environments consisting of several neural networks and examine their emergent behavior. In extension to this work's previous version (Gabor et al., 2019), we provide an extensive analysis of the occurrence of fixpoint weight configurations within the weight space and an approximation of their respective attractor basins.

Keywords
Neural network, self-replication, artificial chemistry system, soup, weight space

1 Introduction

Dawkins (1976) stressed the importance of self-replication to the origin of life. He argued that proto-RNA was able to copy its molecule structure within a soup of randomly interacting elements. This allowed it to reach a stability in concentration that could not be maintained by any other kind of structure. As the story goes, life evolved more or less as an elaborate means to maintain the copying of structural information.

Since the early days of computing, the recreation of biological structures has been a target of research, starting from the early formulation of an evolutionary process by Turing (1950) and including famous examples like Box (1957), Gardner (1970), or Dorigo and Di Caro (1999). Also see the overviews given by Koza (1994) or Bäck et al. (1997). Although also conceived very early, (Minsky & Papert, 1972; Rosenblatt, 1958), neural networks have only rather recently found broad practical application for advanced tasks like image recognition (Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012), or strategic game playing (Silver et al., 2017). The variety of uses shows that neural networks are a powerful tool of abstraction for various domains. However, in

* Corresponding author.

all these cases neural networks are used with a certain intent, i.e., equipped with an extrinsic goal function. Through backpropagation, the distance of the network's output to the goal function can be systematically minimized, making the network match the goal to an increasing extent.

The wide variety of application domains shows the power of neural networks as a functional abstraction. For other functional abstractions, such as expressions in the λ-calculus (Church, 1932) or a variety of assembler-like instruction sets and automata (Dittrich et al., 2001; Görnerup & Crutchfield, 2008), it is known that, when a population of random instances of said functional abstractions is set up and allowed to interact, self-replicators arise naturally (see Fontana & Buss, 1996, or Dittrich & Banzhaf, 1998, respectively). For neural networks, Chang and Lipson (2018) have shown that self-application (i.e., constructing new neural networks by applying neural networks to other neural networks) may lead to the formation of a self-replicating structure, albeit a rather trivial instance of one. In this article, we (a) repeat these results for a broader range of neural network architectures and (b) extend the interaction model by the notion of self-training, which yields lots of non-trivial fixpoints. We then (c) subject these fixpoints to various degrees of noise and analyze their behavior, shedding light on how fixpoints occur within the network weight space. The examined setup allows us to (d) construct an artificial chemistry setup using neural networks as individuals that (under certain circumstances, of course) reliably produces a variety of non-trivial self-replicators.

This article is an extended version of the article originally published by Gabor et al. (2019) with the substantial addition of contribution (c) as described above. The additional content can be found mainly in the new section 3.3, Weight Space Analysis, including Figures 7–10, and the final experiment described in section 3.5, including Figure 14. This article is structured as follows: We first describe all formal definitions of our approach and then provide a series of experiments examining the behavior of self-replicating networks; among the latter, we first discuss experiments on single independent neural networks and then continue with experiments on soups consisting of multiple interacting networks. After that, we briefly discuss related work and provide a conclusion.

2 Approach

We provide a brief introduction to how neural networks function, then we proceed to discuss how to apply neural networks to other neural networks and how to train neural networks using other neural networks.

2.1 Basics

Neural networks are most commonly made from layers of neurons that are connected to adjacent layers of neurons. What all neurons have in common is the base functionality of receiving input values (in the form of a matrix or vector), applying weights and biases (given as the network's parameters), and computing output values via a specific activation function (given as part of the network's architecture). Note that while neural networks originated as a model of biological neurons, they cannot accurately fulfill that role with respect to modern knowledge about biological neurons and instead serve as general function approximators in machine learning.

The most basic form of a feedforward network is a single-layer perceptron, consisting of many fully connected cells that provide a transformation of the input based on their learned parameters. Mathematically, each cell in such a network is described by a function

$$y = f(\sum_i w_i x_i + b)$$

where x_i is the value produced by the i-th input cell, w_i is the weight assigned to that connection, b is a cell-specific bias, f is the activation function, and y is the cell's output.

The *recurrent neural network* (RNN) structure allows cells to retain some information throughout multiple executions: The result of the evaluation step t is passed to the evaluation at step $t+1$ as vector h_{t+1}. A recurrent cell's activation at every timestep t is $h_t = f(Wx_t + Wh_{t-1})$ where W are

the network weights (Chung et al., 2014). This allows for more powerful models when processing sequential inputs.

A neural network thus defines a function $\mathcal{N} : \mathbb{R}^p \to \mathbb{R}^q$ for input length p and output length q. For an input vector $x \in \mathbb{R}^p$ we write the computation of the corresponding output vector $y \in \mathbb{R}^q$ as $y = \mathcal{N}(x)$. A neural network $\mathcal{N}$ is usually given by (a) its architecture, i.e., the types of neurons used, their activation function, and their topology and connections, as well as (b) its parameters, i.e., the weights assigned to the connections. Whenever the architecture of a neural network is fixed, we can define a neural network by its parameters $\overline{\mathcal{N}} \in \mathbb{R}^r$. Note that $r =_{def} |\overline{\mathcal{N}}|$ depends on the amount of internal connection and hidden layers, but as all inputs and all outputs must be connected somehow to other cells in the network it always holds that $r > p$ and $r > q$.

2.2 Application

In the course of this work, we are interested in having neural networks that can be applied to other neural networks (and can output other neural networks). It is evident that if we want neural networks to self-replicate, we need to enable them to output an encoding of a neural network containing at least as many weights as they contain themselves. We discuss multiple approaches to do so but first introduce a general notation covering all the approaches: We write $\mathcal{O} = \mathcal{N} \triangleleft \mathcal{M}$ to mean that $\mathcal{O}$ is the neural network that is generated as the output of the neural network $\mathcal{N}$ when given the neural network $\mathcal{M}$ as input. When $\mathcal{M}$ and $\mathcal{O}$ are sufficiently smaller than $\mathcal{N}$, i.e., if $|\overline{\mathcal{M}}| \ll |\overline{\mathcal{N}}|$ and $|\overline{\mathcal{O}}| \ll |\overline{\mathcal{N}}|$, then we can simply define the output network $\mathcal{O}$ via its weights $\overline{\mathcal{O}} = \mathcal{N}(\overline{\mathcal{M}})$. However, these conditions obviously do not allow for self-replication. Thus, we introduce several *reductions* that allow us to define the application operator $\triangleleft$ differently and open it up for self-replication. Note that for these definitions, we assume that $\mathcal{M}$ and $\mathcal{O}$ have the same architecture and that the application of $\mathcal{N}$ keeps the size of the input network, i.e., $\mathcal{M} : \mathbb{R}^p \to \mathbb{R}^p$ for some p and $|\overline{\mathcal{M}}| = |\overline{\mathcal{O}}| = p$.

Reduction 1 (Weightwise Application). *We define* $\mathcal{N} : \mathbb{R}^4 \to \mathbb{R}$ *fixed. Let* $\overline{\mathcal{M}} = \langle v_i \rangle_{0 \le i < |\overline{\mathcal{M}}|}$. *We then set*

$$\overline{\mathcal{O}} = \langle w_i \rangle_{0 \le i < |\overline{\mathcal{O}}|}$$

where $w_i = \mathcal{N}(v_i, l(i), c(i), p(i))$

and $l(i)$ *is the number of the layer of the weight* i, $c(i)$ *is the number of the cell that the weight* i *leads into, and* $p(i)$ *is the positional number of weight* i *among the weights of its cell. We use* $\overline{\mathcal{O}}$ *to define* $\mathcal{O} = \mathcal{N} \triangleleft_{ww} \mathcal{M}$.

Note that l, c, p depend purely on the networks' architectures and the index of the weight i, not on the value of the weight v_i. Theoretically, we could pass on i to the network directly, but it seemed more reasonable to provide the network with the most semantically rich information we have. Also note that we normalize $l, c, p : \mathbb{N} \to [0; 1] \subset \mathbb{R}$, i.e., the positional values are encoded by real numbers between 0 and 1 as is common for inputs to neural networks.

Intuitively, weightwise application calls $\mathcal{N}$ on every single weight of $\mathcal{M}$ and provides the weight's value and some information on where in the network the weight lives. $\mathcal{N}$ then outputs a new value for that respective weight. After calling $\mathcal{N}$ for $|\overline{\mathcal{M}}| = |\overline{\mathcal{O}}|$ times, we have a new output net $\mathcal{O}$. This approach is most similar to the one used by Chang and Lipson (2018).

Reduction 2 (Aggregating Application). *Let* $\mathrm{agg}_a : \mathbb{R}^a \to \mathbb{R}$ *be an aggregator function taking in an arbitrary number of parameters* a. *Let* $\mathrm{deagg}_a : \mathbb{R} \to \mathbb{R}^a$ *be a de-aggregating function returning an arbitrary number of outputs* a. *Let* $\overline{\mathcal{M}} = \langle v_i \rangle_{0 \le i < |\overline{\mathcal{M}}|}$. *Let*

$$\mathcal{M} \downarrow_b^{\mathrm{agg}} = \langle \mathrm{agg}_{a_i}(v_i, \ldots, v_{i+a_i-1}) \rangle_{0 \le i < b}$$

where $a_i = \lfloor \frac{|\overline{\mathcal{M}}|}{b} \rfloor$ *for* $i < b - 1$ *and* $a_i = \lfloor \frac{|\overline{\mathcal{M}}|}{b} \rfloor + (|\overline{\mathcal{M}}| \mod b)$ *for* $i = b - 1$. *Let*

$$\langle w_i \rangle_{0 \leq i < b} \uparrow_b^{\text{deagg}} = \text{deagg}_{a_0}(w_0) +\!\!+ \ldots +\!\!+ \text{deagg}_{a_{b-1}}(w_{b-1})$$

where a_i *is defined as above and* $+\!\!+$ *is tuple concatenation. We define* $\mathcal{N} : \mathbb{R}^b \to \mathbb{R}^b$ *for a fixed* b. *We then set:*

$$\overline{\mathcal{O}} = \mathcal{N}(\mathcal{M} \downarrow_b^{\text{agg}}) \uparrow_b^{\text{deagg}}$$

We use $\overline{\mathcal{O}}$ *to define* $\mathcal{O} = \mathcal{N} \lhd_{\text{agg}} \mathcal{M}$ *given fixed functions* agg, deagg.

For our experiments, we use the average for aggregation

$$\text{agg}_a(v_0, \ldots, v_{a-1}) = \sum_{i=0}^{a-1} \frac{v_i}{a}$$

and use a trivial de-aggregation function as defined by:

$$\text{deagg}_a(w) = \underbrace{(w, \ldots, w)}_{a \text{ times}}$$

Intuitively, the aggregating application simply reduces the number of weight parameters to a fixed number b by aggregating sub-lists of the weight list into single values. Those single values are then passed to the network and its ouput values are copied to all previously aggregated weights. A lot of different aggregation and de-aggregation functions could be thought of; however, early tests with variants introducing more randomness or different topologies showed no differences in results. Thus, we focus on the simple instantiation of the aggregation application as given above.

Reduction 3 (Recurrent Application). *We define* $\mathcal{N} : \mathbb{R} \times \mathbb{R}^H \to \mathbb{R} \times \mathbb{R}^H$ *as a recurrent neural network with a hidden state* $h \in \mathbb{R}^H$ *for some* $H \in \mathbb{N}$. *Let* $\overline{\mathcal{M}} = \langle v_i \rangle_{0 \leq i < |\overline{\mathcal{M}}|}$. *We then set*

$$\overline{\mathcal{O}} = \langle w_i \rangle_{0 \leq i < |\overline{\mathcal{O}}|}$$

where w_i *is given via*

$$(w_i, h_{i+1}) = \mathcal{N}(v_i, h_i)$$

where $h_0 = \mathbf{0}$. *We use* $\overline{\mathcal{O}}$ *to define* $\mathcal{O} = \mathcal{N} \lhd_{rnn} \mathcal{M}$.

Since RNNs are able to process input sequences of arbitrary length, the recurrent application technically just needs to define $\mathcal{N}$ as an RNN and simply apply it to the weights of another network. Even though this reduction appears most simple and natural, the explosion of gradients within larger RNNs means that after a series of applications they are very prone to diverge to very large output values if not sufficiently controlled, which we will show later in the experiment section of this article. We reckon that an extension to RNNs (making them accessible to self-replication) should be possible; however, since vanilla RNNs are not so fit for self-replication, we leave this extension to future work.

We can use these several types of reductions for application to build a mathematical model of self-replication in neural networks.

Definition 1 (Self-Application). *Given a neural network $\mathcal{N}$. Let $\triangleleft$ be a suitable application reduction. We call the neural network $\mathcal{N}' = \mathcal{N} \triangleleft \mathcal{N}$ the self-application of $\mathcal{N}$.*

Definition 2 (Fixpoint, Self-Replication). *Given a neural network $\mathcal{N}$. Let $\triangleleft$ be a suitable application reduction. We call $\mathcal{N}$ a fixpoint with respect to $\triangleleft$ iff $\mathcal{N} = \mathcal{N} \triangleleft \mathcal{N}$, i.e., iff $\mathcal{N}$ is its own self-application. We also say that $\mathcal{N}$ is able to self-replicate.*

Since network weights are real-valued and are the result of many computations, checking for the equality $\mathcal{N} = \mathcal{N} \triangleleft \mathcal{N}$ is not entirely trivial. We thus relax the fixpoint property a bit.

Definition 3 (ε-Fixpoint). *Given a neural network $\mathcal{N}$* with weights $\overline{\mathcal{N}} = \langle v_i \rangle_{0 \leq i < |\overline{\mathcal{N}}|}$. *Let $\triangleleft$ be a suitable application reduction. Let $\varepsilon \in \mathbb{R}$ be the error margin of the fixpoint property. Let $\mathcal{N}' = \mathcal{N} \triangleleft \mathcal{N}$ be the self-application of $\mathcal{N}$ with weights $\overline{\mathcal{N}'} = \langle w_i \rangle_{0 \leq i < |\overline{\mathcal{N}'}|}$. We call $\mathcal{N}$ an* ε-fixpoint *or a fixpoint up to ε iff for all i it holds that $|w_i - v_i| < \varepsilon$.*

2.3 Training

As stated above, neural networks are commonly used in conjunction with backpropagation, which can adjust their weights to a desired configuration. We assume that we have a set of input vectors $\mathbf{x}_0, \ldots, \mathbf{x}_n$ and a corresponding set of desired output vectors $\mathbf{y}_0, \ldots, \mathbf{y}_n$. We want our neural network $\mathcal{N}$ to represent the relation between these sets. Thus, the loss for a single sample $(\mathbf{x}_i, \mathbf{y}_i)$ is defined as $|\mathcal{N}(\mathbf{x}_i) - \mathbf{y}_i|$. Minimizing the loss of a neural network is called *training*. We use the stochastic gradient descent (SGD) optimizer to apply gradient updates or rather weight changes to minimize the loss for a given sample $(\mathbf{x}_i, \mathbf{y}_i)$, which results in an updated network $\mathcal{N}' = \mathcal{N} \rightsquigarrow (\mathbf{x}_i, \mathbf{y}_i)$. We call $\rightsquigarrow$ the training operator. For sets of sample points $\mathbf{x} = \mathbf{x}_0, \ldots, \mathbf{x}_n$ and $\mathbf{y} = \mathbf{y}_0, \ldots, \mathbf{y}_n$, we also write $\mathcal{N} \rightsquigarrow \mathbf{x}, \mathbf{y}$ as shorthand for $\mathcal{N} \rightsquigarrow (\mathbf{x}_0, \mathbf{y}_0) \rightsquigarrow \ldots \rightsquigarrow (\mathbf{x}_n, \mathbf{y}_n)$.

We argue that training neural networks is another natural way of evolving them (as is application). Thus, we also want to train a neural network with another neural network as input and output data.[1] Of course, we again need to use reduction on said other neural networks. In short we write:

Reduction 4 (Weightwise Training). *Given neural networks $\mathcal{M}, \mathcal{N}$ with $\overline{\mathcal{M}} = \langle v_i \rangle_{0 \leq i \leq n}$ for some n. We write $\mathcal{N}' = \mathcal{N} \rightsquigarrow_{ww} \mathcal{M}$ iff*

$$\mathcal{N}' = \mathcal{N} \rightsquigarrow \langle (v_i, l(i), c(i), p(i)) \rangle_{0 \leq i \leq n}, \langle (v_i) \rangle_{0 \leq i \leq n}$$

where l, c, p are defined as in Reduction 1.

Reduction 5 (Aggregating Training). *Given neural networks $\mathcal{M}, \mathcal{N}$. Given a suitable aggregator function* agg *and aggregated size b. We write $\mathcal{N}' = \mathcal{N} \rightsquigarrow_{agg} \mathcal{M}$ iff*

$$\mathcal{N}' = \mathcal{N} \rightsquigarrow (\mathcal{M} \downarrow_b^{\text{agg}}, \mathcal{M} \downarrow_b^{\text{agg}})$$

where the $\downarrow$ operation is defined as in Reduction 2.

Reduction 6 (Recurrent Training). *Given neural networks $\mathcal{M}, \mathcal{N}$. We write $\mathcal{N}' = \mathcal{N} \rightsquigarrow_{rnn} \mathcal{M}$ iff*

$$\mathcal{N}' = \mathcal{N} \rightsquigarrow (\overline{\mathcal{M}}, \overline{\mathcal{M}})$$

where $\mathcal{N}$ is trained on a sequence $\overline{\mathcal{M}}$ by being applied one by one recurrently.

1 For the scope of this article, we train neural networks only with (a reduction of) themselves as input. We extend that approach to a "train on other" operator in Gabor, Illium, et al. (2021).

Intuitively, these training reductions transform the input network $\mathcal{M}$ to a smaller representation (as do the application reductions, cf. Reductions 1–3) and then train the network $\mathcal{N}$ to accurately reproduce that representation.

Note that usually, when training a neural network, we derive training samples from a large data set or generate them automatically. In this case, we can use these training reductions to define the notion of self-training:

Definition 4 (Self-Training). *Given a neural network $\mathcal{N}$. Let $\leftsquigarrow$ be a suitable training reduction. We call the network $\mathcal{N}' = \mathcal{N} \leftsquigarrow \mathcal{N}$ the result of self-training $\mathcal{N}$.*

We can apply self-training for many consecutive steps; however, in contrast to usual training in neural networks, the samples made available for training only depend on the network's own weights and introduce no randomness or additional coverage of the search space beyond their own (mostly pre-determined) evolution via self-training.

3 Experiments

We define three types of experiments: First, we test the two distinct approaches to self-replication based on application of neural networks to other neural networks and training using backpropagation on self-generated limited training points, respectively. Lastly, we show a strong connection between both approaches.

Note that for the sake of simplicity, we fixed all network architectures in the following experiments to only include two hidden layers with two cells each. Although evaluations were run with various activation functions, all plots show linear activation since we observed no qualitative difference between various activations. Similarly, bias was set to 0 in all plotted instances.

3.1 Self-Application

When subjecting a randomly initialized neural network $\mathcal{N}$ to repeated self-application with respect to the weightwise application $\triangleleft_{ww}$, the weight vector $\overline{\mathcal{N}}$ tends to converge to the all-zero vector $\mathbf{0} = \langle 0 \rangle_{|\overline{\mathcal{M}}|}$. This was already indicated by Chang and Lipson (2018) for a very similar reduction approach. This effect probably stems from a phenomenon observed by Schoenholz et al. (2017): Randomly initialized neural networks tend to map their inputs to output values closer to $\mathbf{0}$. Figure 1 shows that the same effect also occurs for the aggregating reduction $\triangleleft_{agg}$ as it depicts the journey of several neural networks through the space of weight vectors.[2] Very few steps of self-application suffice to draw all neural networks to the principle component analysis (PCA) coordinates (X = 0, Y = 0), which in fact correspond to the weight vector $\mathbf{0}$.

The same plot for the weightwise application $\triangleleft_{ww}$ looks rather similar. Figure 2 shows the resulting networks after several steps of self-application for all types of application reductions. Here, we discern five observations: A neural network $\mathcal{N}$ is (a) *divergent* iff at any point in time any of its weights assumed the value ∞ or $-\infty$. Once this has happened, there is no returning from it. If the network assumes (b) the ε-fixpoint given by the weight vector $\mathbf{0}$, i.e., all its weights are sufficiently close to 0, we call the network a *zero fixpoint*. Note that for all experiments we set $\varepsilon = 10^{-5}$. If the network's weights resemble (c) any other ε-fixpoint, we call it a non-zero, non-trivial, or simply *other fixpoint*. At this stage, we also checked for (d) *second-order fixpoints*, i.e., networks $\mathcal{N}$ fulfilling the weaker property $\mathcal{N} = \mathcal{N} \triangleleft \mathcal{N} \triangleleft \mathcal{N}$. However, we never found any such networks.[3] Anything else falls into the category (e) *other*. Note that Figure 2 shows that no non-zero fixpoints are found

2 To be able to plot highly dimensional weight vectors on paper, we derive the two principle components of the observed weight vectors using standard PCA and plot the weight vector as a point in that two-dimensional space. We use this technique for all such figures.

3 This remains true throughout all experiments presented in this article and by Gabor, Illium, et al. (2021). We discuss this again as an important target for future work.

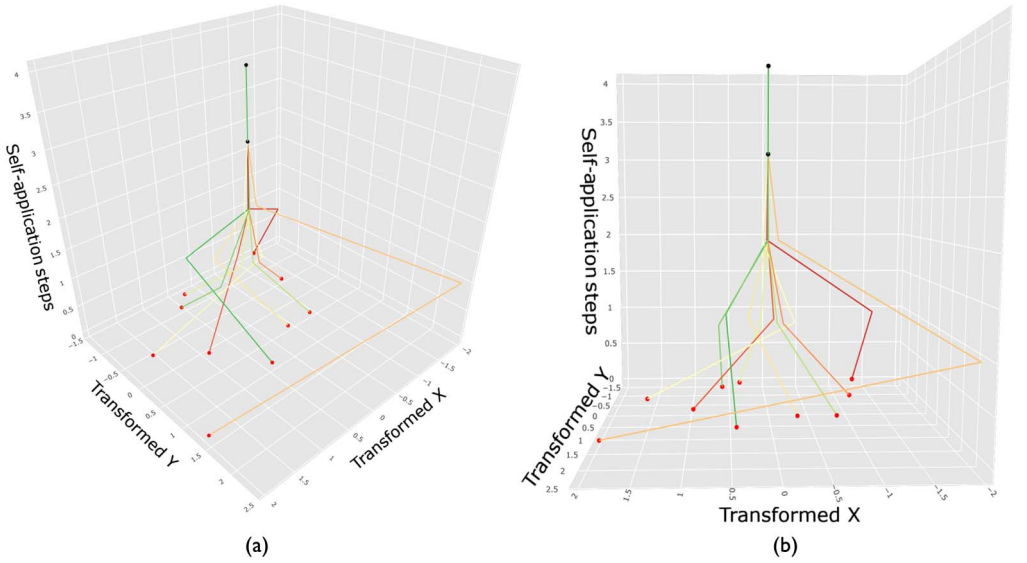


Figure 1. 10 independent runs of *self-application* with respect to the aggregating reduction $\triangleleft_{agg}$. 10 neural networks $\mathcal{N}$: $\mathbb{R}^4 \rightarrow \mathbb{R}^4$ with two hidden layers with two cells each were initialized randomly and then subjected to 4 self-applications each. The figure shows two perspectives on the same three-dimensional graph. The 20 weights in total per network were visualized in a two-dimensional space based on the transformed bases X and Y derived via principle component analysis (PCA). All networks converge on (X = 0, Y = 0), which corresponds to the weight vector **0**.

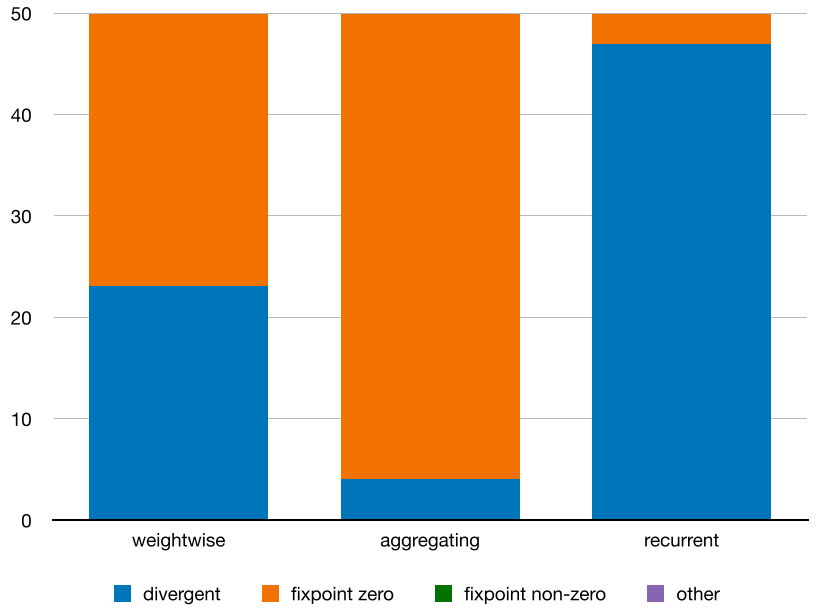


Figure 2. 50 independent runs of *self-application* each with respect to all three different types of reduction. We show an analysis of the networks (which were initialized randomly) after 100 steps of self-application.

for any reduction and that recurrent neural networks are most prone to diverge during repeated self-application.

We also checked for the chance to just randomly generate a neural network which happens to be a fixpoint. However, among 100,000 randomly generated nets for each type of reduction, we did not find a single fixpoint. Thus, we can clearly attribute the attraction towards **0** to self-application.

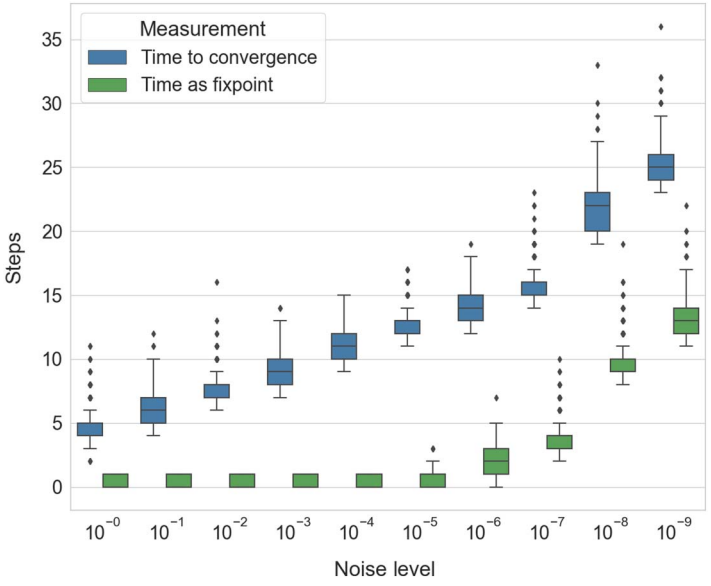


Figure 3. The robustness of *known, perfectly constructed* fixpoint $\mathcal{I}$ with respect to the weightwise application $\triangleleft_{ww}$. The *x* axis shows the range of noise that the known fixpoint's weights were subjected to. The *y* axis shows for how many steps of self-application the network was still regarded as a 10^{-5}-fixpoint (green) and after how many steps of self-application the network was regarded as diverged (blue).

For the weightwise application $\triangleleft_{ww}$, it is rather easy to construct a non-zero fixpoint by hand: For a network $\mathcal{I}$, we set all leftmost weights per layer to 1 and all other weights to 0, thus implementing the identity function on the first value within the inputs of $\mathcal{I}$, which is the original weight. $\mathcal{I}$ thus clearly fulfills the fixpoint property. This allows us to test if the non-zero fixpoints form an attractor in the weight space like the zero fixpoint does: We added small amounts of noise to all weights of $\mathcal{I}$ and then subjected the resulting network $\mathcal{J}$ to several steps of self-application, checking if $\overline{\mathcal{J}}$ would remain stable around $\overline{\mathcal{I}}$ or *verge*, i.e., either converge towards $\mathbf{0}$ or diverge towards infinite weights. However, even adding just a maximum of 10^{-9} noise to each weight eventually caused all networks to verge. Figure 3 shows the experiment for various amounts of noise. Adding less noise unsurprisingly causes the network to fulfill the ε-fixpoint property longer and to verge later. There still is a possibility that the network fulfills the fixpoint property again when converging to $\mathbf{0}$ (but we did not count that as remaining robust in any way).

Thus, while self-application on its own shows a stable intent to approach the fixpoint $\mathbf{0}$, it does not seem capable of creating any other fixpoints.

3.2 Self-Training

Subjecting randomly generated neural networks to self-training with respect to the weightwise training reduction $\leftsquigarrow_{ww}$ yields results as shown in Figure 4. All networks evolve for a few steps of self-training, then their weights remain constant. Note that each network approaches a different point in the weight space. Most interestingly, these points are fixpoints, even though we only apply self-training, and fixpoints are defined using self-application. Moreover, all of these fixpoints are non-zero.

Figure 5 shows an analysis for all types of training reductions: While RNNs still tend to diverge a lot, aggregating networks converge towards weights that do not represent a fixpoint. However, the weightwise networks converge to non-trivial fixpoints with utmost reliability.

Now that we are able to easily "grow natural fixpoints" using weightwise training $\leftsquigarrow_{ww}$, we can examine these trained (not constructed) fixpoints as well. We repeat the experiment presented in Figure 3 by taking such a trained fixpoint and subjecting it to noise to check its robustness.

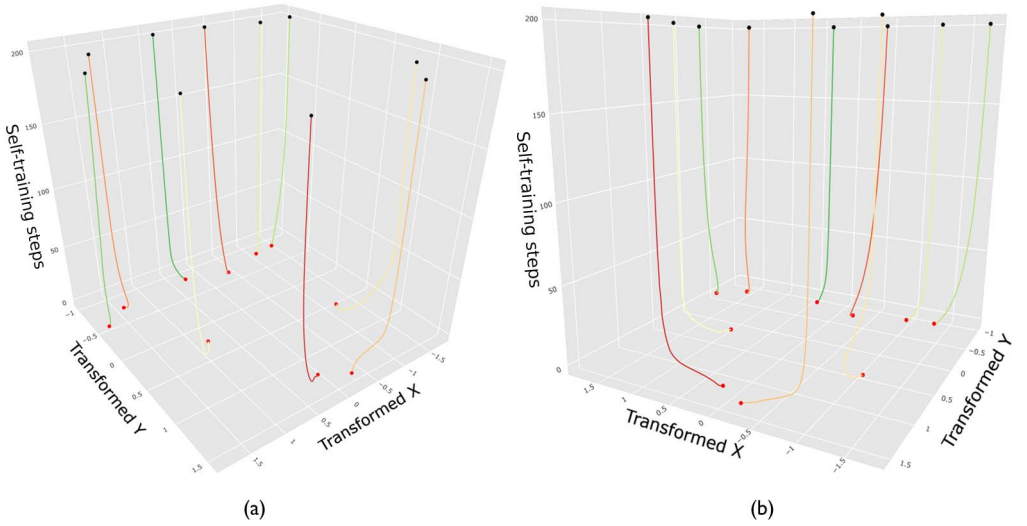


Figure 4. 10 independent runs of *self-training* with respect to the weightwise training reduction $\leftrightsquigarrow_{ww}$. 10 neural networks $\mathcal{N} : \mathbb{R}^4 \to \mathbb{R}$ with two hidden layers with two cells each were initialized randomly and then subjected to 200 steps of self-training each. The figure shows two perspectives on the same three-dimensional graph. The 20 weights in total per network were visualized in a two-dimensional space based on the transformed bases X and Y derived via PCA. All networks converge to different fixpoints with non-zero weights.

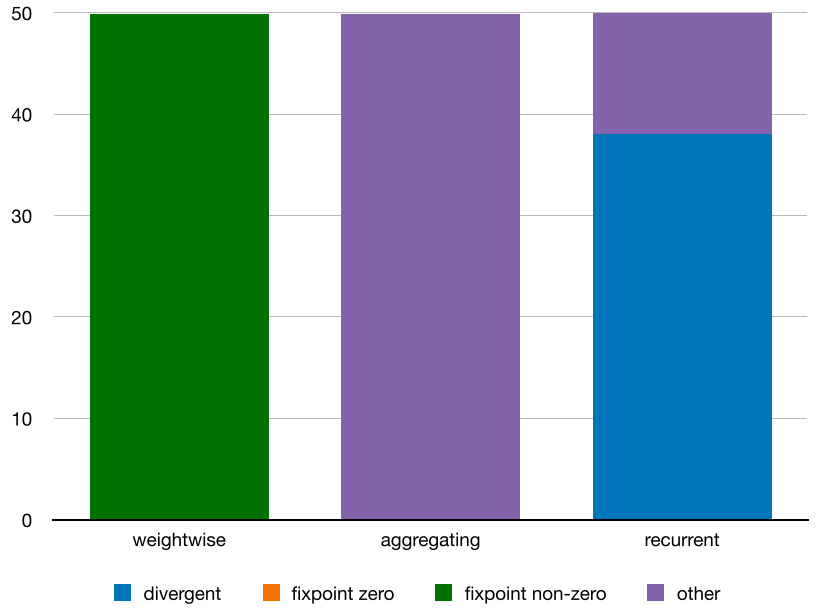


Figure 5. 50 independent runs of *self-training* each with respect to all three different types of training reduction. We show an analysis of the networks (which were initialized randomly) after 1,000 steps of self-training.

The results are shown in Figure 6 and look somewhat similar to Figure 3. However, even when subjected to only very small amounts of noise, the network can uphold its fixpoint property for only a much more limited number of steps. This can be explained by the fact that we stop training the original network once it reaches fixpoint property and thus even without noise it will not work perfectly but lose its fixpoint property after a certain number of applications. However, it will still

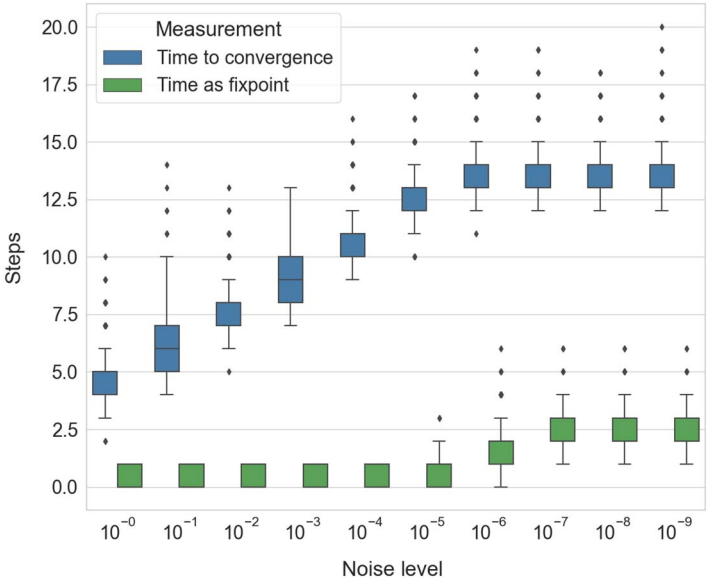


Figure 6. The robustness of a 10^{-5}-fixpoint generated via *self-training* with respect to the weightwise application $\triangleleft_{ww}$. The *x* axis shows the range of noise that the trained fixpoint's weights were subjected to. The *y* axis shows for how many steps of self-application the network was still regarded as a 10^{-5}-fixpoint (green) and after how many steps of self-application the network was regarded as diverged (blue).

reliably remain a fixpoint for at least a few applications when subjected to noise levels below the $\varepsilon = 10^{-5}$ threshold.

3.3 Weight Space Analysis

Experiments as shown in Figure 4 also tell us that by using weightwise training $\leftsquigarrow_{ww}$ we can train a fixpoint by starting with basically any randomly initialized neural network if we apply enough iterations of $\leftsquigarrow_{ww}$. However, it is far from the case that every weight configuration fulfills the fixpoint property. Instead, the trajectories as they are shown in Figure 4 appear to be headed to a specific point within the (PCA-transformed) weight space. However, fixpoints are not especially rare either since no two (fully) randomly generated networks in our experiments ended up evolving (via self-training) into the same fixpoint.

It remains an open question how exactly fixpoints are placed within the weight space and what influences their occurrence. However, we can give a preliminary analysis. Instead of generating random initial weight configurations, we implement a "noisy cloning" mechanism: We start with a fixpoint that was trained for 2,500 steps of weightwise training $\leftsquigarrow_{ww}$ and subject the fixpoint to random noise at various levels,[4] thus generating new networks to work with that remain (according to the noise level) somewhat close to the original network within the weight space. We call the original network the *parent* and the networks generated from its random variation its *children*. So far, this setup is (bar the longer training times) identical to the setup used to generate Figure 6. Now, however, we give the children networks a chance to evolve on their own by training them again for 2,500 steps of weightwise training $\leftsquigarrow_{ww}$. Figure 7 shows the respective trajectories for one parent: We can see that the noisy cloning process scatters the children throughout the vicinity of the parent within the (PCA-transformed) weight space, as shown by the sharp changes in the trajectories at

4 To be more exact, given a noise level ζ we substitute each weight w_i within the neural network $\mathcal{N}$, $0 \leq i < |\overline{\mathcal{N}}|$, with $w_i' := w_i + z \cdot \zeta$ where z is chosen randomly and uniformly from the interval $[-1.0; 1.0]$.

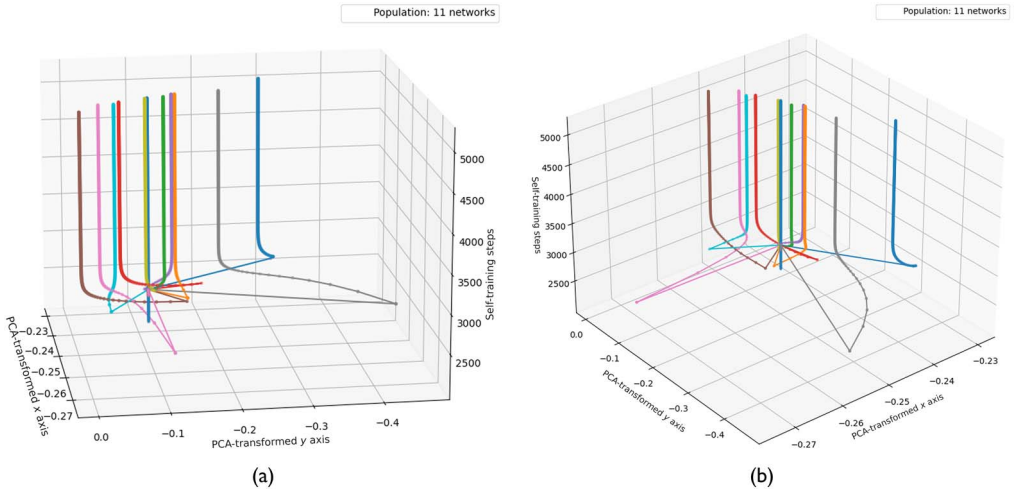


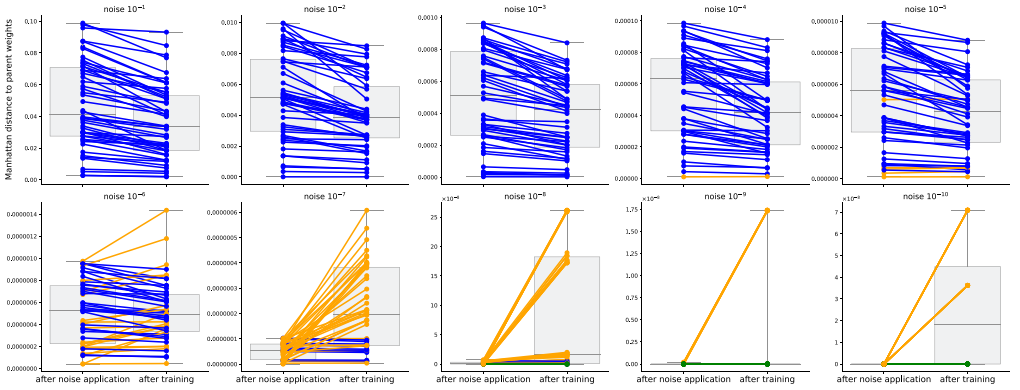
Figure 7. 10 children that are generated via *noisy cloning* with a noise level of 10^{-2} from a single ε-fixpoint parent (blue line starting from the plot's base) after 2,500 training steps using $\leftsquigarrow ww$. Children are then trained again for 2,500 steps using $\leftsquigarrow ww$. The figure shows two perspectives on the same three-dimensional graph. The 20 weights in total per network were visualized in a two-dimensional space based on the transformed bases X and Y derived via PCA. All children reach ε-fixpoint status again.

2,500 training steps. When self-training kicks in again, the children lock in to a new fixpoint. While all of these children regain ε-fixpoint status again, none of them evolve back to the parent's position. Some, however, travel remarkable distances towards the parent's original position.

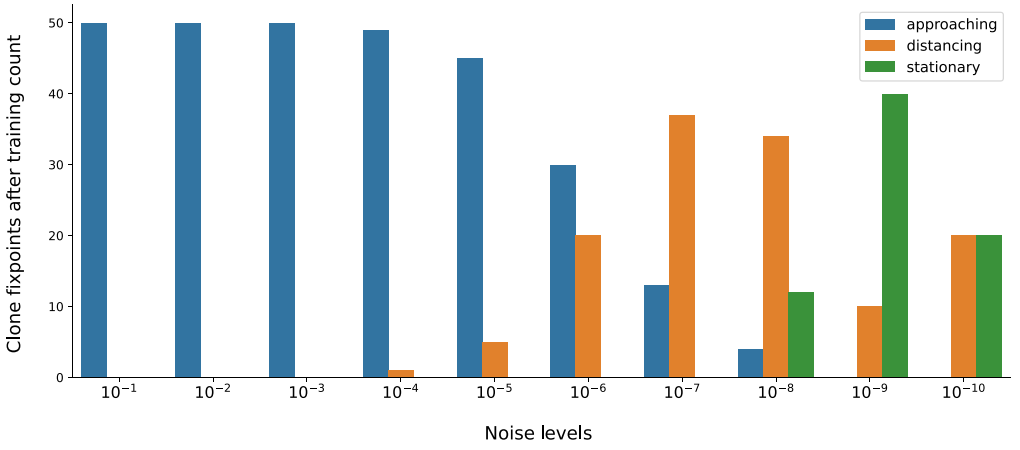
This intuition gained from just looking at the PCA-transformed trajectories can be verified. To do so, we conduct the experiment shown in Figure 7 for more parents (and thus more children) and various different noise levels. The results are shown in Figure 8. We can see that for high levels of noise, most notably noise greater than 10^{-6}, every single child trains towards a fixpoint that is closer to its parent's position than the child's original position was. Thus, despite the application of random noise via noisy cloning, children retain some attraction to the parent's position in weight space. It should be noted, however, that basically every child also stops at some fixpoint between its starting position (after noise application) and its parent's position and does not "go all the way" back to its parent's position in weight space.

This behavior changes drastically at noise level 10^{-7} and below. At very low noise levels, children are much more prone to evolve to fixpoint in a direction that actually points away from their parents. It should be noted that the distance they travel to do so remains rather constant even when the noise level decreases drastically. Figure 8 shows that even when only disturbed by noise at a noise level of 10^{-11}, a child can still travel at least a distance of $4 \cdot 10^{-8}$ or more than 40 times the distance of the initial disturbance. Such distances are never traveled at higher noise levels.

From the experiments shown in Figure 8 we can conclude that at small scales new fixpoints are more likely found by traveling comparatively large distances, while at large scales a certain attraction to a fixpoint position (that is somehow retained through noisy cloning) leads to finding lots of intermediate fixpoints easily. But where does the obvious border between small and large scales come from? To provide a preliminary answer, we analyze the *fixpoint quality* of the parents and children, i.e., the error margin ε up to which they still count as an ε-fixpoint. Figure 9 shows that after 2,500 steps of self-training, almost all networks could be counted as 10^{-6}-fixpoints, some even as 10^{-7}-fixpoints, reaching much lower error margins than our usual $\varepsilon = 10^{-5}$ threshold. As a side note, we verify that children after another 2,500 steps of training are able to reach basically the same noise level their parents did, regardless of how much noise we applied when cloning them.



(a) Children and their evolution



(b) Occurences of categories of behavior

Figure 8. The effect of self-training using $\leftsquigarrow_{ww}$ after *noisy cloning*. For various noise levels ranging from 10^{-1} to 10^{-10} we train 5 parents for 2,500 steps (making sure all of them reach ε-fixpoint status) and produce 10 children per parent via noisy cloning with the respective noise level. We leave out all diverging children. (a) Left-side boxplots and points show the children's distance to their respective parent (i.e., their weights' Manhattan distance to the parent's weights in weight space) directly after being generated via noisy cloning. Right-side boxplots and points show the children's distance to their respective parent after being trained for 2,500 timesteps. We categorize the children's evolution between these two points as either *approaching* when the distance decreases (blue), *distancing* when the distance increases (orange), or *stationary* when its stays exactly the same up to common floating point precision (green). (b) This plot shows the same data and sums up the occurrence of each of these three categories per noise level.

As now we know how good our networks actually are, we may suspect that the 10^{-7} noise level they reach in training (cf. Figure 9) may be the border at which we see them change their behavior when finding fixpoints from positions nearby to a parent fixpoint (cf. Figure 8). Future work will have to examine how that border shifts for training step amounts changing from the 2,500 we applied here. We suspect that these plots might show a certain compartmentalization of the weight space with regard to fixpoints at lower scales. When there are no more fixpoints in between child and original parent, this would explain why training for fixpoint status may pull the child away from its parent and towards a far out fixpoint at low scales. High scales might then be so filled with fixpoints that there always is a fixpoint to find that is closer to the original parent.

To gain a little bit more insight into that phenomenon, we construct another experiment. We generate parents via self-training and explore the lines in 20-dimensional weight space between them by generating children and looking at the fixpoint they tend to train to. Figure 10 agrees with

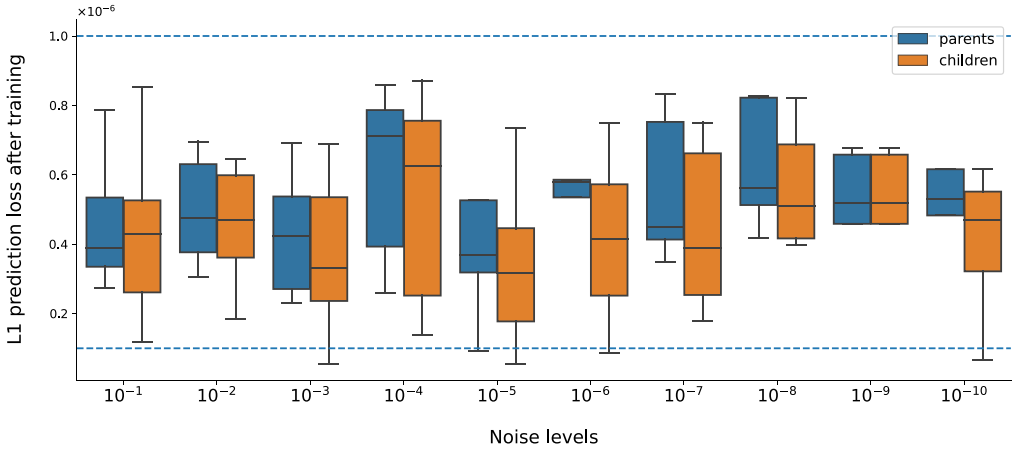


Figure 9. Prediction loss for the fixpoint networks of the experiment shown in Figure 8. Parents are trained from random initial weights in 2,500 steps of self-training using $\curvearrowleft_{ww}$. Children are generated using *noisy cloning* with the respective noise level from a parent and then trained for 2,500 steps of self-training using $\curvearrowleft_{ww}$. A prediction loss of δ means that children would still count as an ε-fixpoint for all $\varepsilon \geq \delta$. The lower blue dashed line shows $\delta = 10^{-7}$, while the upper blue dashed line shows $\delta = 10^{-6}$.

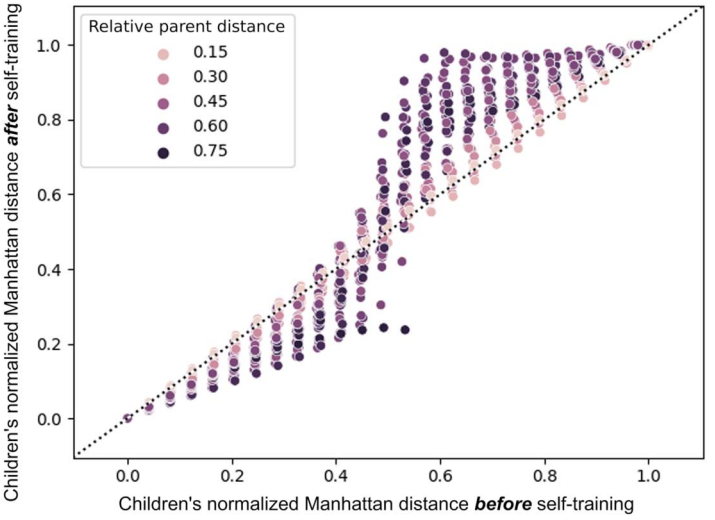


Figure 10. Weight space between two fixpoints. For 10 parent fixpoints generated via self-training with $\curvearrowleft_{ww}$ for 2,500 steps from randomly initialized weight configurations, we explore the 45 lines between them. For each line, we generate 25 children at equal distance to each other across the line. We then train the children to fixpoint status using $\curvearrowleft_{ww}$. The *x* axis shows each child's normalized distance to the starting point of the line, with a distance of 0.0 meaning that it is at the same position as the first parent and a distance of 1.0 meaning that it is at the same position as the second parent. The *y* axis shows the child's normalized distance to the original starting point of the line after the child has been trained to fixpoint status again. The color of the points shows the length of the line, i.e., the absolute distance between the two parents compared to the longest line length occurring in our experiment. The dotted black line in the plot illustrates the diagonal for orientation.

our previous results here: Children of parents who are far apart (dark colors) approach their first parent when they are on the first half of the line and distance themselves from their first parent (as they approach their second parent) when they are on the second half of the line. Children whose parents were closer together (light colors) show a slight tendency for the inverse effect, i.e., they (on some occasions) slightly increase their distance to the parent they were generated closer to. As that effect is much less pronounced, they form an almost straight line in Figure 10.

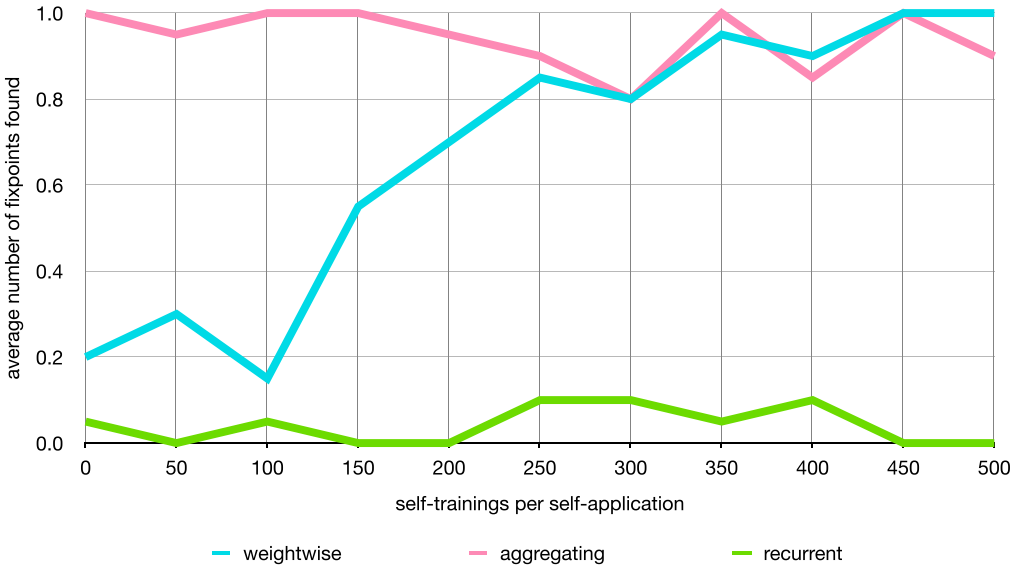


Figure 11. Evaluation of a *mixed* setting of self-application and self-training. For each type of reduction, 20 neural networks were generated at random and then subjected to 4 steps of self-application. In between those steps, $0, 50, \ldots, 500$ steps of self-training were executed (see the x axis). The y axis shows the average ratio of fixpoints (both zero and non-zero) found out of all runs, where a value of 1 means that all runs resulted in a fixpoint.

3.4 Mixed Setting

In order to elaborate on the opportunities of interaction between self-application and self-training, we construct an experiment where the two appear in alternation. The results are shown in Figure 11: While aggregating networks reach the zero fixpoint so fast via self-application that self-training is not able to add anything to that, weightwise networks need at least 200–300 steps of self-training between each self-application to converge to fixpoints as reliably.

3.5 Soups

As we have discussed several means of neural networks interacting with themselves, it seems a reasonable next step to open up these interactions and build a population of mutually interacting networks. A suitable combination of a population of individuals and various interactions is called *soup* and works like an artificial chemistry system (cf. Dittrich et al., 2001). This means that a soup evolves over a fixed number of epochs. At every epoch, several different interaction operators can be applied to networks in the population with a certain chance, resulting in new networks and thus a changed population.

Interaction 1 (Self-Train). *Applied to every single network* $\mathcal{N}$ *for a number of steps* A, *self-training substitutes its weights with* $\mathcal{N}' = \mathcal{N} \underbrace{\rightsquigarrow \mathcal{N} \ldots \rightsquigarrow \mathcal{N}}_{A \text{ times}}$.

Interaction 2 (Attack). *Applied to two random networks* $\mathcal{M}, \mathcal{N}$ *at a chance* α, *attacking substitutes the weights of the attacked network* $\mathcal{M}$ *with the weights given via* $\mathcal{M}' = \mathcal{N} \triangleleft \mathcal{M}$.

Intuitively, attacking applies the function represented by the network $\mathcal{N}$ to another network $\mathcal{M}$. Self-training remains basically unchanged from the non-soup scenario and provides a background evolution to every network in the population, even when it is not involved in any attack.

Figure 12 shows the evolution of a soup employing self-training and attacking. The networks start out randomly placed in the weight space and self-train towards fixpoints in the beginning. The big jumps in the networks' trajectories stem from being attacked by other networks; self-training then leads them to new fixpoints. Note that as self-training causes the networks to converge towards

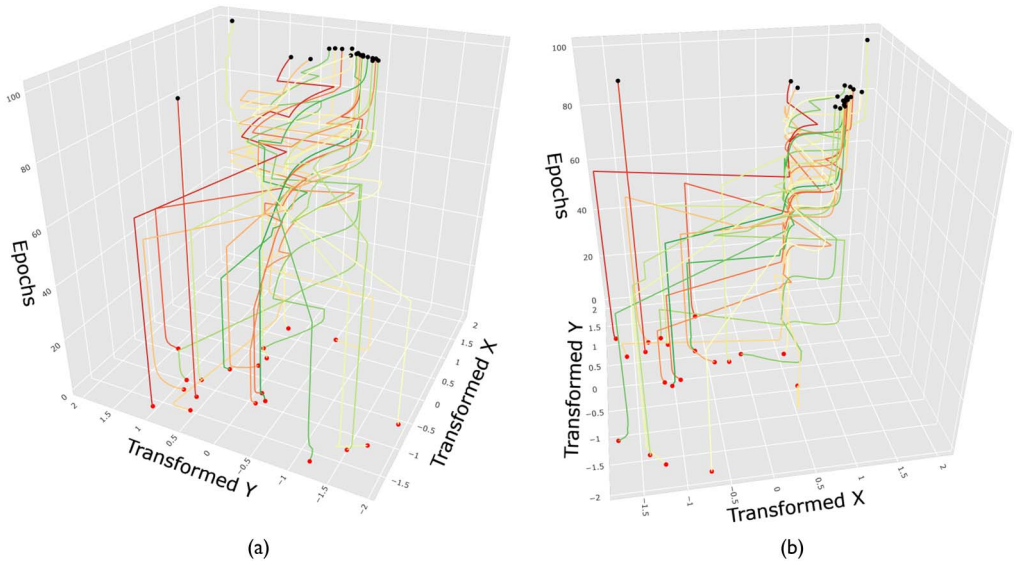


Figure 12. Run of one soup consisting of 20 neural networks using the weightwise reductions $\triangleleft_{ww}$ and $\leftsquigarrow_{ww}$. The 20 neural networks $\mathcal{N} : \mathbb{R}^4 \to \mathbb{R}$ with two hidden layers with two cells each were initialized randomly and then evolved for 100 epochs. Per epoch, every network had a chance of 0.1 to attack another network and was subjected to 30 steps of self-training. This setup allowed for emergent behavior of the network forming a cluster at a region of all non-zero fixpoints. The figure shows two perspectives on the same three-dimensional graph. The 20 weights in total per network were visualized in a two-dimensional space based on the transformed bases X and Y derived via PCA.

fixpoints, the impact of near-fixpoint networks' attacks becomes less and less prominent. Most interestingly though, almost all attacks seem to drive the attacked networks towards the main cluster of the soup, where most networks gather in the end. This not only shows emergent behavior as the networks form a group as a cluster of fixpoints somewhere in the weight space (neither at the center of mass from the initial population nor anywhere near $\mathbf{0}$), but also can be interpreted as a clear instance of (self-)replication within the networks of this soup.

In Figure 13, we further evaluate the impact of parameter A in Interaction 1 for both weightwise and aggregating neural networks. (As recurrent networks already did not show sufficient compatibility with application, we omit these results.) More self-training manages to stabilize the weightwise networks' ability to find non-zero fixpoints. Still, even in a soup setting, aggregating networks converge to $\mathbf{0}$ to a strong degree.

Last, we repeat our experiments with noisy cloning (cf. Figure 7) in a setting with a soup, i.e., we take several parents, generate children in their vicinity using noisy cloning as described above, and then let all the parents and children evolve as soup (instead of training them individually). Figure 14 shows the soup's evolution. What we can notice here is that even though we open up the networks to attacking, all children tend to stick together and we see no real influence between the soups. This, of course, is also due to the fact that (as their parents are fixpoints) the children are also already pretty close to fixpoint status and their attacks probably do not do real damage, even when paired with networks from the other cluster.

4 Related Work

There is some research on generating neural networks using other neural networks (cf., e.g., Deutsch, 2018; Schmidhuber, 1992; Stanley et al., 2009). However, without any suitable reduction operations, these approaches cannot be used to produce self-replicating structures.

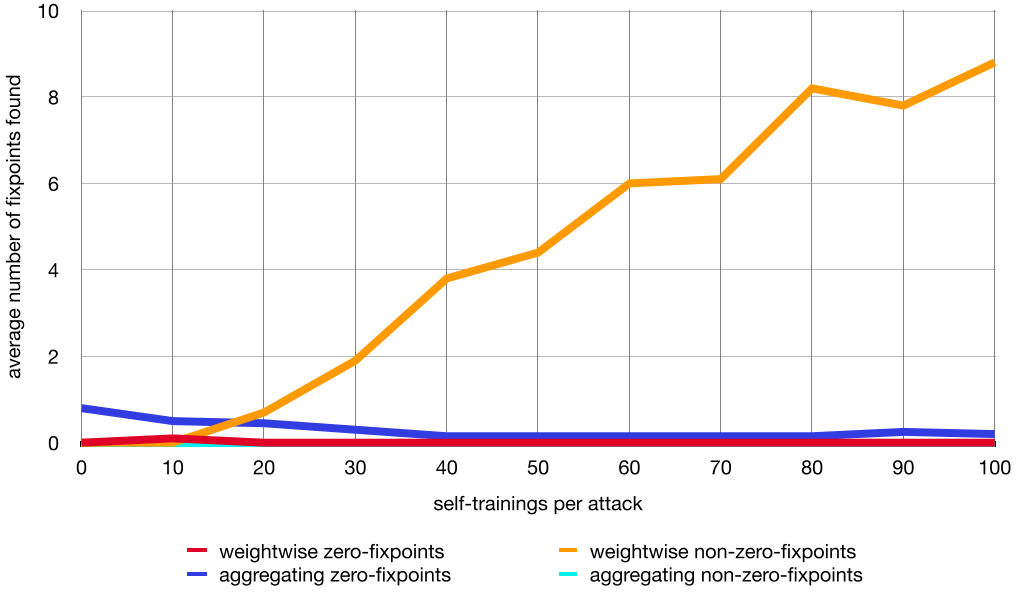


Figure 13. Evaluation of the impact of the number of training steps per epoch on a soup consisting of 10 neural networks using the weightwise application $\triangleleft_{ww}$ or the aggregating application $\triangleleft_{agg}$. Averaged over 10 runs. Per epoch, every network had a chance of 0.1 to attack another network and was subjected to a fixed number of steps of self-training (see the *x* axis). The *y* axis shows the average number of (zero or non-zero) fixpoints present in the final population of the soup.

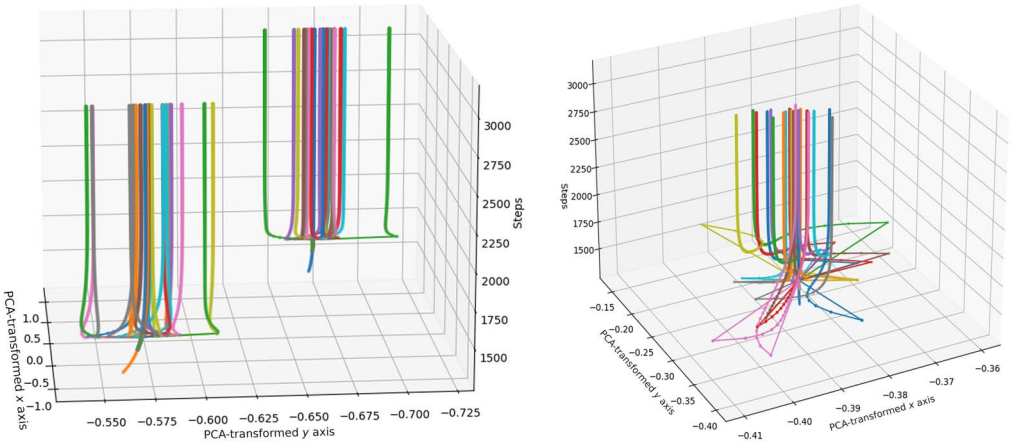


Figure 14. Two parent networks are trained to fixpoint status using 1,500 steps of self-training with $\leftrightsquigarrow_{ww}$. Then they generate 25 children each via *noisy cloning*. All 52 networks are then evolved for 3,000 timesteps of self-training (90% chance) or attacking (10% chance). The left plot shows all networks' evolution; the right plot zooms in on one of the clusters to give a better intuition of its internal structure. The 20 weights in total per network were visualized in a two-dimensional space based on the transformed bases X and Y derived via PCA.

Our results on self-application agree with Chang and Lipson (2018) on the weightwise reduction. We extended the experiments with several means of reduction and managed to find non-trivial, non-zero fixpoints up to a very low error ε by introducing our weightwise reduction in combination with our notion of self-training. We augmented the approach by studying the combination of self-application and self-training. However, the inclusion of auxiliary fitness functions has not been

considered in this article and we refer to Gabor, Illium, et al. (2021) for auxiliary fitness in a soup context.

The idea to generate fixpoints via repeated self-application is based on Fontana and Buss (1996), who showed the emergence of fixpoints from having random expressions in the λ-calculus interact. They, too, construct an artificial chemistry system based on their functional abstraction and see complex structures of fixpoints arise. Unfortunately, we did not observe higher-order fixpoints as they did for λ-expressions, which should be considered an important direction of future research on neural network soups. Possible connections between λ-fixpoints or larger organizational structures in general and fixpoints in neural networks may still be explored (Larkin & Stocks, 2004).

Görnerup and Crutchfield (2008) provide additional insight into the evolution of complexity within soups: They choose so called ϵ-machines, i.e., finite-memory communication channels (Crutchfield & Görnerup, 2006), for function approximators because there exists a well-defined metric for their individual complexity (which cannot exist for λ-expressions, for example). Since neural networks are likewise finite-memory structures, one could imagine applying a similar metric in our case. However, providing such a metric on single neural networks is still a task for future research. Furthermore, various differences in the construction of the soups between this article and Görnerup and Crutchfield (2008) (most notably the constant addition of newly generated particles in Görnerup & Crutchfield, 2008) require further analysis before more parallels can be drawn.

In general, a vast amount of research is dedicated to artificial chemistry systems, utilizing very different representations for the particles in the soup: Dittrich et al. (2001) and Matsumaru et al. (2005) provide excellent overviews, to which we refer for the sake of brevity.

5 Conclusion

We have presented various reduction operations without any claim of completeness. Interesting reduction possibilities like extracting the main frequencies of the weight vector using a Fourier transformation or fine-tuning RNNs for this scenario are still to be tested to the full extent. Most importantly, all settings, architectures, and parameters of the neural networks we constructed still allow for more thorough exploration and evaluation in future work.

We have also performed some exploration of the distribution of fixpoints within the weight space by generating lots of non-trivial fixpoints using our setup of self-training. In this extended version of the article, we added experiments that analyze the neighborhood of given fixpoints by adding noise at various scales. We have gained some insight into a compartmentalization of the weight space with respect to fixpoints. Future work still needs to examine how this compartment effect arises and can be influenced by original training accuracy or other factors.

Having multiple neural networks interact directly with each other's weights within a soup has been one of the more outlandish ideas of this article, as pointed out by Chang (2021) referring to the original (non-extended) version. Accordingly, observing these soups exert some emergent behavior (cf. Figure 12) has been one of the most fascinating aspects of this line of research. While we evaluated some parameters, there exist many different ways to evolve such a soup and many different interactions whose effects are yet to be explored. We introduce new interactions like *learn*, which substitutes the weights of the learning network $\mathcal{M}$ with the weights given via $\mathcal{M}' = \mathcal{M} \leftsquigarrow \mathcal{N}$, in Gabor, Illium, et al. (2021), where we also discuss more involved patterns of soup behavior.

It should also be noted that for now, we do not track any relationships between particles beyond their spontaneous, random, and rather short-lived pairing for some interactions. Future work might introduce a topology between particles (that might or might not be influenced by the particles' position within the weight space) or might keep track of genealogical relationships (similarly to Gabor, Phan, & Linnhoff-Popien, 2021) and might thus discern between horizontal and vertical inheritance of information, for example.

Eventually, we think that the dynamics of a soup might open up neural networks to a new kind of learning by not (only) applying a goal function (and its respective loss) directly but by simply

guiding a soup a certain way, perhaps achieving more diversity and robustness in the solutions reached (cf., e.g., Gabor et al., 2018; Prokopenko, 2013). Future work will show if methods learned from these settings can be integrated into common machine learning. To this end, soups (or self-replicating particles in general) must be integrated with the standard machine learning framework (including extrinsic reward functions) and most importantly must be able to learn complex tasks as a group.

References

Bäck, T., Hammel, U., & Schwefel, H.-P. (1997). Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, *1*(1), 3–17. https://doi.org/10.1109/4235.585888

Box, G. E. (1957). Evolutionary operation: A method for increasing industrial productivity. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, *6*(2), 81–101. https://doi.org/10.2307/2985505

Chang, O. (2021). *Autogenerative networks* [Unpublished doctoral dissertation]. Columbia University.

Chang, O., & Lipson, H. (2018). Neural network quine. In *Proceedings of the ALIFE 2018: The 2018 conference on Artificial Life* (pp. 234–241). MIT Press. https://doi.org/10.1162/isal_a_00049

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). *Empirical evaluation of gated recurrent neural networks on sequence modeling*. ArXiv. https://doi.org/10.48550/arXiv.1412.3555

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics*, *33*(2), 346–366. https://doi.org/10.2307/1968337

Crutchfield, J. P., & Görnerup, O. (2006). Objects that make objects: The population dynamics of structural complexity. *Journal of The Royal Society Interface*, *3*(7), 345–349. https://doi.org/10.1098/rsif.2006.0114, PubMed: 16849243

Dawkins, R. (1976). *The selfish gene*. Oxford University Press.

Deutsch, L. (2018). *Generating neural networks with neural networks*. ArXiv. https://doi.org/10.48550/arXiv.1801.01952

Dittrich, P., & Banzhaf, W. (1998). Self-evolution in a constructive binary string system. *Artificial Life*, *4*(2), 203–220. https://doi.org/10.1162/106454698568521, PubMed: 9847424

Dittrich, P., Ziegler, J., & Banzhaf, W. (2001). Artificial chemistries: A review. *Artificial Life*, *7*(3), 225–275. https://doi.org/10.1162/106454601753238636, PubMed: 11712956

Dorigo, M., & Di Caro, G. (1999). Ant colony optimization: A new meta-heuristic. In *Proceedings of the 1999 congress on Evolutionary Computation: CEC99* (pp. 1470–1477). IEEE. https://doi.org/10.1109/CEC.1999.782657

Fontana, W., & Buss, L. W. (1996). The barrier of objects: From dynamical systems to bounded organizations. In J. Casti & A. Karlqvist (Eds.), *Boundaries and barriers* (pp. 56–116). Addison-Wesley.

Gabor, T., Belzner, L., Phan, T., & Schmid, K. (2018). Preparing for the unexpected: Diversity improves planning resilience in evolutionary algorithms. In *2018 IEEE international conference on autonomic computing (ICAC)* (pp. 131–140). IEEE. https://doi.org/10.1109/ICAC.2018.00023

Gabor, T., Illium, S., Mattausch, A., Belzner, L., & Linnhoff-Popien, C. (2019). Self-replication in neural networks. In *Proceedings of the ALIFE 2019: The 2019 conference on Artificial Life* (pp. 424–431). MIT Press. https://doi.org/10.1162/isal_a_00197

Gabor, T., Illium, S., Zorn, M., & Linnhoff-Popien, C. (2021). Goals for self-replicating neural networks. In *Proceedings of the ALIFE 2021: The 2021 conference on Artificial Life* (pp. 101–110). MIT Press. https://doi.org/10.1162/isal_a_00439

Gabor, T., Phan, T., & Linnhoff-Popien, C. (2021). Productive fitness in diversity-aware evolutionary algorithms. *Natural Computing*, *20*(3), 363–376. https://doi.org/10.1007/s11047-021-09853-3

Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American*, *223*(4), 120–123. https://doi.org/10.1038/scientificamerican1070-120 (Also available at https://www.jstor.org/stable/24927642)

Görnerup, O., & Crutchfield, J. P. (2008). Hierarchical self-organization in the finitary process soup. *Artificial Life*, *14*(3), 245–254. https://doi.org/10.1162/artl.2008.14.3.14301, PubMed: 18489247

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., & Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, *29*(6), 82–97. https://doi.org/10.1109/MSP.2012.2205597

Koza, J. R. (1994). Artificial life: Spontaneous emergence of self-replicating and evolutionarily self-improving computer programs. In C. G. Langton (Ed.), *Artificial Life III* (pp. 225–262). Reading, MA: Addison-Wesley.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. https://doi.org/10.1145/3065386

Larkin, J., & Stocks, P. (2004). Self-replicating expressions in the lambda calculus. In *Proceedings of the 27th Australasian conference on Computer Science: ACSC2004* (pp. 167–173). Australian Computer Society.

Matsumaru, N., Centler, F., di Fenizio, P. S., & Dittrich, P. (2005). Chemical organization theory as a theoretical base for chemical computing. In *Proceedings of the 2005 workshop on Unconventional Computing: From cellular automata to wetware* (pp. 75–88). Luniver Press.

Minsky, M. L., & Papert, S. (1972). *Perceptrons: An introduction to Computational Geometry*. MIT Press.

Prokopenko, M. (Ed.) (2013). *Guided self-organization: Inception*. Springer. https://doi.org/10.1007/978-3-642-53734-9

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, PubMed: 13602029

Schmidhuber, J. (1992). Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, *4*(1), 131–139. https://doi.org/10.1162/neco.1992.4.1.131

Schoenholz, S. S., Pennington, J., & Sohl-Dickstein, J. (2017). *A correspondence between random neural networks and statistical field theory*. ArXiv. https://doi.org/10.48550/arXiv.1710.06570

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270, PubMed: 29052630

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, PubMed: 19199382

Turing, A. (1950). Computing machinery and intelligence. *Mind: A Quarterly Review of Psychology and Philosophy*, *59*(236), 433–460. https://doi.org/10.1093/mind/LIX.236.433