Self-Replication in Neural Networks

Thomas Gabor¹, Steffen Illium¹, Andy Mattausch¹, Lenz Belzner², Claudia Linnhoff-Popien¹

¹LMU Munich
²MaibornWolff
thomas.gabor@ifi.lmu.de

Abstract
The foundation 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. We argue that backpropagation is the natural way to navigate the space of network weights and show how it allows non-trivial self-replicators to arise naturally. We then extend the setting to construct an artificial chemistry environment of several neural networks.

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. Eventually, 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), Conway (1970) or Dorigo and Di Caro (1999). Also see the overviews given by Koza (1994) or Bäck et al. (1997). Although conceived very early as well (Rosenblatt (1958) and Minsky and Papert (1972)), neural networks have only 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 all these cases neural networks are used with a certain intend, i.e., equipped with a goal function. Through backpropagation, the distance of the network’s output to the goal function can be systematically minimized.

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)), it is known that, when a population of random instances of said functional abstractions are set up and allowed to interact, self-replicators arise naturally (see Fontana and Buss (1996) or Dittrich and Banzhaf (1998), respectively). For neural networks, Chang and Lipson (2018) have shown that self-application may lead to the formation of a self-replicating structure, albeit a rather trivial instance of one. In this paper, 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. This allows us to (c) 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.

Foundations
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.

Basics
Neural networks are most commonly made from layers of neurons that are connected to the next layers of neurons and so on. As there are many kinds of neurons (fully connected, recurrent, convolutional) there are also many kinds of layers. Variations of this scheme go up to well established structures within such layers, consisting not only of single functional cells (LSTM, attention mechanism). What they have in common is the base functionality of accepting values (in form of a matrix or vector), application of weights or bias (a matrix of a similar shape, also known as the network’s parameters), followed by a specific activation function (linear, sigmoid, relu, tanh, e.g.) that transforms the outputs. Note that while neural networks originated as a model of biological neurons, they cannot accurately fulfill that role anymore and instead serve as general function approximators.

The most basic form of a feed-forward network is a single-layer perceptron, consisting of many fully connected
cells that provide a transformation of the input on basis of its 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 and \( f \) is the activation function.

The recurrent neural network (RNN) structure allows to pass an additional vector \( h \) to the current calculation. This improves the performance when processing sequential inputs. 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 time step \( t \) is \( h_t = f(W x_t + W h_{t-1}) \) where \( w \) are the network weights (Chung et al. (2014)).

A neural network thus defines a function \( \mathcal{N} : \mathbb{R}^p \rightarrow \mathbb{R}^q \) for input length \( p \) and output length \( q \). 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 net is fixed, we can define a neural network by its parameters \( \mathcal{N} \in \mathbb{R}^r \). Note that \( |\mathcal{N}| = \text{def} r \) 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 \( r > q \).

**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 net containing at least as many weights as 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 \( |\mathcal{M}| < |\mathcal{N}| \) and \( |\mathcal{O}| < |\mathcal{N}| \), we can simply define the output network \( \mathcal{O} \) via its weights \( \mathcal{O} = \mathcal{N}(\mathcal{M}) \). However, these conditions obviously do not allow for self-replication. Thus, we introduce several reductions that allow to define the 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., \( \mathcal{M} : \mathbb{R}^p \rightarrow \mathbb{R}^p \) for some \( p \) and \( |\mathcal{M}| = |\mathcal{O}| = p \).

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

\[ \mathcal{O} = \langle v_i \rangle_{0 \leq i < |\mathcal{O}|} \]

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

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

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 \( l, c, p : \mathbb{N} \rightarrow [0;1] \subset \mathbb{R} \), i.e., the positional values are encoded by reals between 0 and 1 as is common for inputs to neural networks.

Intuitively, the weightwise reduction 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 \( |\mathcal{M}| = |\mathcal{O}| \) 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).** Let \( \text{agg} : \mathbb{R}^a \rightarrow \mathbb{R} \) be an aggregator function taking in an arbitrary amount of parameters \( a \). Let \( \text{deagg} : \mathbb{R} \rightarrow \mathbb{R} \) be a de-aggregating function returning an arbitrary amount of outputs \( a \). Let \( \mathcal{M} = \langle v_i \rangle_{0 \leq i < |\mathcal{M}|} \). Let

\[ \mathcal{M} \downarrow_{\text{agg}} = (\text{agg}_{a}(v_i, \ldots, v_{i+a-1}))_{0 \leq i < b} \]

where \( a_i = \lceil \frac{|\mathcal{M}|}{b} \rceil \) for \( i < b - 1 \) and \( a_i = \lfloor \frac{|\mathcal{M}|}{b} \rfloor \) for \( i = b - 1 \). Let

\[ \langle w_i \rangle_{0 \leq i < b} \downarrow_{\text{deagg}} = \text{deagg}_{b\text{eagg}}(w_0) \oplus \ldots \oplus \text{deagg}_{b\text{eagg}_{b-1}}(w_{b-1}) \]

where \( a_i \) is defined as above and \( \oplus \) is concatenation. We define \( \mathcal{N} : \mathbb{R}^b \rightarrow \mathbb{R}^b \) for a fixed \( b \). We then set:

\[ \mathcal{O} = \mathcal{N}(\mathcal{M} \downarrow_{\text{agg}}) \downarrow_{\text{deagg}} \]

We use \( \mathcal{O} \) to define \( \mathcal{O} = \mathcal{N} \downarrow_{\text{agg}} \mathcal{M} \) given fixed functions \( \text{agg}, \text{deagg} \).

For our experiments, we use the average for aggregation

\[ \text{agg}_{\text{av}}(v_0, \ldots, v_{a-1}) = \frac{1}{a} \sum_{i=0}^{a-1} v_i \]

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

\[ \text{deagg}_{\text{av}}(w) = \begin{cases} w_{i} & \text{a times} \\ i \end{cases} \]

Intuitively, the aggregating reduction simply reduces the amount of weight parameters to a fixed amount \( b \) by aggregating sub-lists of the weight list into single values. Those single values are then passed to the network and its output 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 reduction as given above.
Reduction 3 (RNN). We define $\mathcal{N} : \mathbb{R} \times \mathbb{R}^H \rightarrow \mathbb{R} \times \mathbb{R}^H$ as a recurrent neural network with a hidden state $h \in \mathbb{R}^H$ for some $H \in \mathbb{N}$. Let $\mathcal{M} = \langle v_i \rangle_{0 \leq i < |\mathcal{M}|}$. We then set
$$\mathcal{O} = \langle w_i \rangle_{0 \leq i < |\mathcal{O}|}$$
where $w_i$ is given via
$$\mathcal{N}(v_i, h_i) = (w_i, h_{i+1})$$
where $h_0 = 0$. We use $\mathcal{O}$ to define $\mathcal{O} = \mathcal{N} \triangleleft_{mn} \mathcal{M}$.

Since recurrent neural networks are able to process input sequences of arbitrary length, the RNN reduction technically just needs to define $\mathcal{N}$ as a recurrent neural network 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 recurrent neural networks means that they are very prone to diverge to very large output values if not sufficiently controlled. We reckon that an extension to recurrent neural networks (making them accessible to self-replication) should be possible, however, since vanilla recurrent neural networks are not so fit for self-replication, we refer this extension to future work.

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

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

Definition 2 (Fixpoint, Self-Replication). Given a neural network $\mathcal{N}$. Let $\lhd$ be a suitable reduction. We call $\mathcal{N}$ a fixpoint with respect to $\lhd$ iff $\mathcal{N} = \mathcal{N} \lhd \mathcal{N}$, 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 $\mathcal{N} = \mathcal{N} \lhd \mathcal{N}$ is not entirely trivial. We thus relax the fixpoint property a bit.

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

Training
As stated above, neural networks are commonly used in conjunction with backpropagation to adjust their weights to a desired configuration. We assume that we have a set of input vectors $x_0, ..., x_n$ and a corresponding set of desired output vectors $y_0, ..., y_n$. We want our neural network $\mathcal{N}$ to represent the relation between these sets. The loss for a single sample $(x_i, y_i)$ is defined as $|\mathcal{N}(x_i) - y_i|$. Minimizing the loss of a neural network is called training. We use the SGD optimizer to apply gradient updates or rather weight changes to minimize the loss for given a given sample $(x_i, y_i)$, which results in an updated network $\mathcal{N}' = \mathcal{N} \lhd (x_i, y_i)$. We call $\lhd$ the training operator. For sets of sample points $x = x_0, ..., x_n$ and $y = y_0, ..., y_n$, we also write $\mathcal{N} \lhd x, y$ as shorthand for $\mathcal{N} \lhd (x_0, y_0) \lhd ... \lhd (x_n, y_n)$.

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 other neural networks as input and output data. 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 $\mathcal{M} = \langle v_i \rangle_{0 \leq i \leq n}$ for some $n$. We write $\mathcal{N}' = \mathcal{N} \lhd_{\text{wn}} \mathcal{M}$ iff
$$\mathcal{N}' = \mathcal{N} \lhd ((v_i, l(i, c(i), p(i)))_{0 \leq i \leq n}, ((v_i))_{0 \leq i \leq n})$$
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 $\text{agg}$ and aggregated size $b$. We write $\mathcal{N}' = \mathcal{N} \lhd_{\text{agg}} \mathcal{M}$ iff
$$\mathcal{N}' = \mathcal{N} \lhd (M \downarrow^\text{agg}_b, \mathcal{M} \uparrow^\text{agg}_b)$$
where the $\downarrow$ operation is defined as in Reduction 2.

Reduction 6 (Recurrent Training). Given neural networks $\mathcal{M}, \mathcal{N}$. We write $\mathcal{N}' = \mathcal{N} \lhd_{\text{rm}} \mathcal{M}$ iff
$$\mathcal{N}' = \mathcal{N} \lhd (\mathcal{M}, \mathcal{M})$$
where $\mathcal{N}$ is trained on a sequence $\mathcal{M}$ by being applied one by one recurrently.

Intuitively, these training reductions transform the input set $\mathcal{M}$ 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. However, we can use these training reductions to define the notion of self-training:

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

Experiments
We define three types of experiments, which 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
Figure 1: 10 independent runs of self-application with respect to the aggregating reduction $\triangleleft_{\text{agg}}$. 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 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.

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.

**Self-Application**

When subjecting a randomly initialized neural network $\mathcal{N}$ to repeated self-application with respect to the weightwise reduction $\triangleleft_{\text{w}}$, the weight vector $\mathcal{N}$ tends to converge to the all-zero vector $0 = (0)_{|\mathcal{N}|}$. 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 $0$. Figure 1 shows that the same effect also occurs for the aggregating reduction $\triangleleft_{\text{agg}}$. It shows the journey of several neural networks through the space of weight vectors. Very few steps of self-application suffice to draw all neural networks to the coordinates $(X=0, Y=0)$, which in fact correspond to the weight vector $0$.

The same plot for the weightwise reduction $\triangleleft_{\text{w}}$ looks rather similar. Figure 2 shows the resulting networks after several steps of self-application. 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 $\infty$ or $-\infty$. Once this has happened, there is no returning from it. If the network assumes (b) the $\varepsilon$-fixpoint given by the weight vector $0$, 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 $\varepsilon$-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. Anything else falls into the category (e) other.

---

1To be able to plot highly-dimensional weight vectors on paper, we derive the two principle components of the observed weight vectors using standard principle component analysis (PCA) and plot the weight vector as a point in that two-dimensional space. We use this technique for all such figures.
Self-Training

Subjecting randomly generated neural networks to self-training with respect to the aggregating training reduction \( \omega_{agg} \) 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 a detailed analysis for all types of reduction: While recurrent neural networks 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.

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 6: 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 about 200–300 steps of self-training between each self-application to converge to fixpoints as reliably.

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 amount 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 \( N \) for an amount of steps \( A \), self-training substitutes its weights with \( N' = N \mapsto N_a' \mapsto N' \).

**Interaction 2** (Attack). Applied to two random networks \( M, N \) at a chance \( a \), attacking substitutes the weights of the attacked network \( M \) with the weights given via \( M' = N_a' \mapsto M' \).

Intuitively, attacking applies the function represented by the network \( N \) to another network \( M \). 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 7 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...
Figure 4: 10 independent runs of self-training with respect to the aggregating training reduction $\leftrightarrow_{ww}$. 10 neural networks $\mathcal{N} : \mathbb{R}^d \rightarrow \mathbb{R}$ 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 types of reduction. We show an analysis of the networks (which were initialized randomly) after 1000 steps of self-training.

Figure 6: 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, ..., 500 steps of self-training were executed (see 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.

In Figure 8, 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 0 to a distinctive degree.
Figure 7: Run of one soup consisting of 20 neural networks using the weightwise reductions $\triangleright_{ww}$ and $\triangleleft_{ww}$. The 20 neural networks $N: \mathbb{R}^4 \rightarrow \mathbb{R}$ with two hidden layers with two cells each were initialized randomly and then evolved for 100 epoch. 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 for 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.

Related Work

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

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 $\varepsilon$ 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 our work.

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 $\lambda$-calculus interact. They, too, construct an artificial chemistry system based on their functional abstraction and see complex structures of fixpoints arise. Sadly, we did not observe higher-order fixpoints as they did for $\lambda$-expressions. Possible connections between $\lambda$-fixpoints or larger organizational structures in general and fixpoints in neural networks may still be explored (Larkin and Stocks (2004)).

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.

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 are still to be tested to 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. Especially discovering some kind of measurement of how rare fixpoints actually are and if they can occur in all regions of the weight space would be helpful.

We think that perhaps the most interesting contributions are the distinct behaviors observed in the soup made of neural networks (Figure 7). 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. Early results on an interaction called learn, which
substitutes the weights of the learning network \( M \) with the weights given via \( M' = M \sim N' \) look most promising but were left out for brevity.

Eventually, we think that the dynamics of a soup might open up neural networks to a new kind of learning by not 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. Prokopenko (2013); Gabor et al. (2018), e.g.).

References


