Goals for Self-Replicating Neural Networks

Thomas Gabor, Steffen Illium, Maximilian Zorn, Claudia Linnhoff-Popien

LMU Munich
thomas.gabor@ifi.lmu.de

Abstract
Self-replicating neural networks can be trained to output a representation of themselves, making them navigate towards non-trivial fixpoints in their weight space. We explore the problem of adding a secondary functionality to the primary task of replication. We find a successful solution in training the networks with separate input/output vectors for one network trained in both tasks so that the additional task does not hinder (and even stabilizes) the self-replication task. Furthermore, we observe the interaction of our goal-networks in an artificial chemistry environment. We examine the influence of different action parameters on the population and their effects on the group’s learning capability. Lastly we show the possibility of safely guiding the whole group to goal-fulfilling weight configurations via the inclusion of one specially-developed guiding particle that is able to propagate a secondary task to its peers.

Introduction
Self-replication has been identified as a central tool of life in general (e.g., Kauffman et al. (1993); Dawkins (2016)). Maybe for that reason, the emergence of self-replication has also been sought for and been studied intensively in the context of computer programs in their various representations (Bäck et al. (1997); Sipper (1998); Dittrich et al. (2001)). While self-replication has often been considered an interesting goal in its own right, evolved self-replicating structures often tend to be of little use to anyone but themselves. While this is their point, of course, it seems promising to construct entities that can both self-replicate and fulfill an external goal given by the programmer, similarly to how apple trees can self-replicate and do something useful for the gardener (i.e., produce edible fruit).

Neural networks have been the prime method for modeling computational structures (i.e., functions) in recent years. They have been shown to be able to learn basic self-replication by Chang and Lipson (2018) and to form a self-replicating group by Gabor et al. (2019). We augment this previous research by constructing neural networks that can self-replicate their own weight configuration and efficiently respond to a given (albeit primitive) auxiliary task. The goal here is to show that self-replicating neural network soups, which we consider interesting in their own right, can maintain additional goals and thus might some day (in more complex setups) be used for machine learning just like their non-replicating counterparts. We show the robustness of self-replication even when we change any additional goals for the networks. Finally, we observe that knowledge about the auxiliary task can even be propagated within a group of networks and we use that propagation to control the group via guided self-organization (cf. Prokopenko (2013)), i.e., we influence the group by adjusting a single designated particle, the so-called beacon.

Related Work
Due to conceptual proximity to the counterpart research on cellular automata, many familiar approaches can be found in that direction, of which Von Neumann and Burks (1966) and Langton (1984, 1986) represent the foundation. As for neural network approaches, Mordvintsev et al. (2020) for example present a way to grow neural cellular automata with self-regenerating capabilities. Gabor et al. (2019) forms our central foundation, which we augment with the use of additional goal functions rather than just achieving self-replication. Chang and Lipson (2018) already combine self-replicating neural networks with additional goals, but use a different design for the single network case where they predict both primary and secondary task results simultaneously, leading them to observe a constant trade-off between the accuracy of self-replication versus the auxiliary task. Furthermore, they do not consider the case of network soups.

Single Networks
Foundations
For the foundations, we follow Gabor et al. (2019) throughout Definitions 1–6. We also briefly define our approach and notation for the underlying neural network basics.

Definition 1 (neural network) A neural network is function \( N : \mathbb{R}^p \rightarrow \mathbb{R}^q \) with \( p \) inputs and \( q \) outputs. This function is defined via (a) the architecture of the network and (b) a vector of \( r \) weights \( \mathbf{w} \in \mathbb{R}^r \). The length of the weight vector is \( |\mathbf{w}| = \text{def} \ r \).
Note that during all experiments, the network architecture remains fixed so that we can deduce the network function using only the weight vector. To better understand the internal structure of a neural network and how the function is derived from architecture and weights, please refer to literature (e.g., Kruse et al. (2011)) or the treatment by Gabor et al. (2019), which matches our formalism.

To even allow for self-replication on a formal level, we need to define neural networks that are able to process other neural networks of at least equal size to their own. As a network’s weight vector always contains at least as many weights as its input and output dimensions combined, this cannot be done trivially by simply giving the whole weight vector as input. Instead, we need to define reductions that allow for a shorter representation of the neural network to be processed by another neural network. As Gabor et al. (2019) found the weightwise reduction (which was also suggested by Chang and Lipson (2018)) most effective, we adopt it for this paper. We would also like to refer to Gabor et al. (2019) for a more elaborate treatment of these goal-less networks’ buildup and the reasons it was chosen for analysis.

Definition 2 (weightwise application) Let $\mathcal{M}, \mathcal{N}, \mathcal{O} : \mathbb{R}^d \rightarrow \mathbb{R}$ be neural networks. Let the vector of all weights $v_i$ of $\mathcal{M}$ be $\vec{M} = \langle v_i \rangle_{0 \leq i < |\vec{M}|}$. Neural network $\mathcal{O}$ is the result of weightwise application of $\mathcal{N}$ onto $\mathcal{M}$, written $\mathcal{O} = \mathcal{N} \triangleleft_{ww} \mathcal{M}$, iff

$$\vec{O} = \langle N(v_i, L(i), C(i), P(i)) \rangle_{0 \leq i < |\vec{M}|}$$

where $L(i)$ 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 going into the same cell.

Using this notion, we can further re-iterate what we formally mean when talking about self-replication in neural networks:

Definition 3 (fixpoint, self-replication) Given a neural network $\mathcal{N}$. We call $\mathcal{N}$ a fixpoint with respect to $\triangleleft_{ww}$ iff $\mathcal{N} = \mathcal{N} \triangleleft_{ww} \mathcal{N}$, i.e., iff $\mathcal{N}$ is its own self-application. We also say that $\mathcal{N}$ is able to self-replicate.

For practical purposes, we relax the fixpoint property:

Definition 4 ($\varepsilon$-fixpoint) Given a neural network $\mathcal{N}$ with weight vector $\vec{N} = \langle v_i \rangle_{0 \leq i < |\vec{N}|}$. Let $\varepsilon \in \mathbb{R}$ be the error margin of the fixpoint property. Let $\mathcal{N}' = \mathcal{N} \triangleleft_{ww} \mathcal{N}$ be the self-application of $\mathcal{N}$ with weights $\vec{N}' = \langle w_i \rangle_{0 \leq i < |\vec{N}'|}$. We call $\mathcal{N}$ an $\varepsilon$-fixpoint or a fixpoint up to $\varepsilon$ iff for all $i$ it holds that $|w_i - v_i| < \varepsilon$.

Note that for the purpose of this paper, we use $\varepsilon = 10^{-5}$ unless stated otherwise.

In most applications of artificial intelligence, the weights for neural networks are generated via training:

Definition 5 (training) Given a training data set $\mathbb{D} = \{ (X_i, Y_i) \mid 0 \leq i < |\mathbb{D}| \}$ with data points consisting of an input $X_i$ and a desired output $Y_i$. We train a network $\mathcal{N}$ on $\mathbb{D}$ when we create a new network $\mathcal{N}'$ with adjusted weights $\vec{N}'$ so to minimize the loss $\sum_{i=0}^{|\mathbb{D}|-1} |\mathcal{N}'(X_i) - Y_i|$ compared to $\mathcal{N}$. This update of weights is written as $\mathcal{N}' = \mathcal{N} \triangleright \langle X_0, Y_0 \rangle \triangleright \ldots \triangleright \langle X_{|\mathbb{D}|-1}, Y_{|\mathbb{D}|-1} \rangle$ or shorter $\mathcal{N}' = \mathcal{N} \triangleright \mathbb{D}$.

Training is usually performed via backpropagation, which is explained by Kruse et al. (2011), for example. When we are interested in training self-replicating neural networks, we can define a weightwise training operator $\triangleright_{ww}$:

Definition 6 (weightwise training) Given neural networks $\mathcal{M}, \mathcal{N}, \mathcal{O}$ with weight vector $\vec{M} = \langle v_i \rangle_{0 \leq i < |\vec{M}|}$. Neural network $\mathcal{O}$ is the result of weightwise training of $\mathcal{N}$ for $\mathcal{M}$, written $\mathcal{O} = \mathcal{N} \triangleright_{ww} \mathcal{M}$, iff

$$\mathcal{O} = \mathcal{N} \triangleright \{ (v_i, L(i), C(i), P(i), v_i) \mid 0 \leq i < |\vec{M}| \}$$

where $L, C, P$ are defined as in Definition 2.

As shown by Chang and Lipson (2018), subjecting a randomly initialized neural network $\mathcal{N}$ to iterated self-application, i.e., computing $\mathcal{N} \triangleleft_{ww} \mathcal{N} \triangleleft_{ww} \ldots$, causes the network to converge/diverge to trivial fixpoints $\mathcal{N} \in \{0, +\infty, -\infty \}$. Gabor et al. (2019) have shown that subjecting randomly initialized neural networks to self-training, i.e., computing $\mathcal{N} \triangleright_{ww} \mathcal{N} \triangleright_{ww} \ldots$, results in these networks becoming (usually non-trivial) $\varepsilon$-fixpoints up to very small $\varepsilon$. Evidently, this process is then not interrupted by interjecting some self-application steps in between self-training steps as long as they are relatively few. Also note that Gabor et al. (2019) have shown that the presented approach does not produce higher-order fixpoints or periodic cycles, so we exclude those from our analysis. Achieving them for setups like these surely is important future work.

Introducing Goals

Self-training introduces an inherent goal to the network, i.e., minimizing the loss when reproducing its own weights. We now extend the model of weightwise self-replicating neural networks by introducing an auxiliary goal, i.e., an externally given goal function $f : \mathbb{X} \rightarrow \mathbb{Y}$ that the neural network is supposed to train for in addition to its self-replication task.

If $f$ is of the type $f : \mathbb{R}^d \rightarrow \mathbb{R}$, we could re-use our network structure given in Definition 2; but this would only generate ambiguity as self-replication and the auxiliary goal $f$ might not agree on some data points. Having evaluated various encodings, we decided on the following solution: For the remainder of this paper, we focus on exemplary auxiliary goals of the form $f : \mathbb{R}^2 \rightarrow \mathbb{R}$. We then use neural networks of the structure $\mathcal{N} : \mathbb{R}^0 \rightarrow \mathbb{R}$, which allows the neural network to have separate incoming neurons for the replicative and the auxiliary task. Sample data for either of the task is expanded by having the input for the other task filled with zeroes. Note that both tasks share the same outgoing neuron.
Definition 7 (replicative and auxiliary application) Given neural networks $\mathcal{M}, \mathcal{N}, \mathcal{O} : \mathbb{R}^6 \to \mathbb{R}$ with weight vector $\mathcal{M} = (v_i)_{0 \leq i < |\mathcal{M}|}$. Also given an input value $X = (x_1, x_2) \in \mathbb{R}^2$ and an output value $Y = y \in \mathbb{R}$. Neural network $\mathcal{O}$ is the result of replicative (weightwise) application of $\mathcal{N}$ for $\mathcal{M}$, written $\mathcal{O} = \mathcal{N} \preceq_{\text{rep}} \mathcal{M}$, iff

$$\mathcal{O} = \langle N(v_i, l(i), c(i), p(i), 0, 0) \rangle_{0 \leq i < |\mathcal{M}|}$$

where $L, C, P$ are defined as in Definition 2. Result value $Y$ is the result of auxiliary application to input $X$, written $Y = \mathcal{N} \bullet_{\text{aux}} X$, iff

$$Y = \mathcal{N}(0, 0, 0, 0, x_1, x_2).$$

Definition 8 (replicative and auxiliary training) Given neural networks $\mathcal{M}, \mathcal{N}, \mathcal{O} : \mathbb{R}^6 \to \mathbb{R}$ with weight vector $\mathcal{M} = (v_i)_{0 \leq i < |\mathcal{M}|}$. Also a finite training set $T \subset \mathbb{R}^2 \times \mathbb{R}$. Neural network $\mathcal{O}$ is the result of replicative training of $\mathcal{N}$ for $\mathcal{M}$, written $\mathcal{O} = \mathcal{N} \triangleright_{\text{rep}} \mathcal{M}$, iff

$$\mathcal{O} = \mathcal{N} \triangleright \{(v_i, l(i), c(i), p(i), 0, 0), v_i \} \; 0 \leq i < |\mathcal{M}|$$

where $L, C, P$ are defined as in Definition 2. Neural network $\mathcal{O}$ is the result of auxiliary training of $\mathcal{N}$ for $\mathcal{T}$, written $\mathcal{O} = \mathcal{N} \triangleright_{\text{aux}} \mathcal{T}$, iff

$$\mathcal{O} = \mathcal{N} \triangleright \{((0, 0, 0, 0, x_1, x_2), y) \} \; ((x_1, x_2), y) \in \mathcal{T} \}.$$

Definitions 7 and 8 augment the respective operations from previous work by including the option to work (i.e., apply or train) on a given goal function using a single neural network. Note that said network can only work on one of the operations at a time. However, as we later show it is possible for a single network to both fulfill the fixpoint property as well as fulfill a given goal function.

Definition 9 ($\varepsilon$-goal-fulfillment) A neural network $\mathcal{N}$ has fulfilled a goal function $f : \mathbb{R}^2 \to \mathbb{R}$ up to an error margin $\varepsilon$ iff $|\langle \mathcal{N} \bullet_{\text{aux}} (x_1, x_2) - f(x_1, x_2) \rangle| < \varepsilon$ for all (practically tested) $(x_1, x_2) \in \mathbb{R}^2$.

Definition 10 (goal-fixpoint) A neural network that is an $\varepsilon$-fixpoint and fulfills its goal function up to $\varepsilon$ is also called a goal-fixpoint.\(^1\)

Experiments

Mainly, goal-fixpoints are what we are looking for in the course of this paper. Our definitions made above can only be called sound when we can find a reasonable amount of goal-fixpoints using the operators in Definitions 7 and 8. To set up experiments, we further need to introduce a few things. First, we define two (very simple) auxiliary tasks that we will be using in the experiments.

\(^1\)We intentionally share the parameter $\varepsilon$ for both the fixpoint property and the goal fulfillment property here, interpreting it as a more general error margin we are willing to accept in all real-valued computations.

---

Figure 1: A singular network (2 hidden layers with width 3) trained over 1000 steps by alternating between replicative and auxiliary training. The 30 weights in total per network were visualized in a two-dimensional space based on the transformed bases X and Y derived via PCA. The network performed full self-applications at time-steps 500 and 1000 to test its convergence to a goal-fixpoint. The position in the black box is the transformed final weight configuration (after the second self-application), at which point the network is a goal-fixpoint up to $\varepsilon < 10^{-14}$.

Task 1 (addition) Let $f_{\text{add}}(x_1, x_2) = x_1 + x_2$.

Task 2 (mean) Let $f_{\text{mean}}(x_1, x_2) = \frac{x_1 + x_2}{2}$.

Fortunately, a relatively simple setup is able to produce goal-fixpoints: We just alternate between replicative training and auxiliary training on a single sample input. An exemplary run is shown in Figure 1. For the replicative training the next time step’s neural network is given via $\mathcal{N}' = \mathcal{N} \triangleright_{\text{rep}} N$; for the auxiliary training the next time-step’s neural network is given via $\mathcal{N}' = \mathcal{N} \triangleright_{\text{aux}} \{(x_1, x_2), f(x_1, x_2)\}$, where $(x_1, x_2)$ is sampled randomly (uniform) from $\mathbb{R}^2$.

Note that in order to display the evolution of a neural network throughout these steps, we compact its weight space into two dimensions using principal component analysis (PCA) so that we can then plot the evolution of the weights in 3D space as it has been done by Gabor et al. (2019). This allows us to reason about the training process (and later: interactions) displayed by our neural networks in a more intuitive manner.

Compared to training just for self-replication, it takes more training steps to reach a weight configuration where both the fixpoint and the goal fulfillment properties apply. Figure 2 shows the comparison between normal and goal-networks. We can clearly observe an additional training requirement for networks to reach a goal-fixpoint state.
Figure 2: Rates of fixpoints occurring for various training setups. We train 20 independent networks for each amount of training steps (100, ..., 600). We compare self-replicating neural networks without a goal function (green) versus goal-networks (blue) and goal-networks which before evaluation are also subjected to one round of self-application (orange).

Figure 2 also showcases the destabilization effect of self-application, especially in the early learning stages. Simply speaking, networks with less training predict weights still inaccurately, therefore only replicating themselves crudely. With more refined abilities, the replication becomes more and more accurate, reducing the relatively straining impact of one self-application.

As we noticed that we already needed a sensible amount of additional weights to accommodate for the additional inputs to the auxiliary task, we also analyzed the impact additional weights have on finding goal-fixpoints. We did so by varying the width of the network, i.e., the amount of cells per (hidden) layer.

We observe that, in general, more neurons per hidden layer help the networks learn the combination of tasks faster and more reliably with increased training time, as shown in Figure 3. This is surprising as larger width increases the search space for weight configurations and also increases the amount of weight replications that may be flawed. However, the weightwise training scales well as the network’s input and output neurons remain the same as we increase its width. This result may also indicate that goal-fixpoint configurations at least do not become less dense when scaling up the weight space.

For our next experiment we tested if networks can adapt to a new auxiliary task midway. After training stable fixpoints with the goal property, we found that it is possible to change the goal function from \( f_{\text{add}} \) to \( f_{\text{mean}} \), e.g., effectively forcing the network to reinterpret the values of the auxiliary training. Figure 4a shows multiple individual networks undergoing this procedure in the context of the same weight space. The total weight change is seemingly small considering the broadness of the weight space but the complexity of this transformation can be seen in the detailed focus on one of the networks in Figure 4b.

Figure 3: The effect of network width. We train 20 fresh, independent networks for each amount of training steps (100, 200, ..., 600) with a uniformly random choice of replicative or auxiliary training at each time step. The y-axis shows the rate of found goal-fixpoints after one round of self-application for various settings of network width.

This change can also be performed multiple times in succession. The weight change induced by this is shown in Figure 5. We observed that eventually all networks relocate from one fixpoint position to another with enough training time. Midway through this transition the networks are not fixpoints, however. Only when the network can compute the new task reliably enough, i.e., after reaching \( \varepsilon \)-accuracy, the back-propagated error becomes small enough that the self-training regains the upper hand and guides the trajectory back to a fixpoint position. We train self-replication or the auxiliary task in turn, resulting in a stable schedule that prevents divergence. Upon re-training a former task the fixpoint configuration is different in all cases we observed.

The average graph in Figure 5 also shows that the total change of weights during the goal-switch transformation does not seem to follow a reducing trend, i.e., the goal changes do not simply induce a solution (approximately) fit for both goals but actually re-focus on the new goal. This is important as it suggests that the network can adapt to different auxiliary tasks multiple times, without loosing any learning capacity of its weights, but also without remembering much from its past.

**Network Soups**

**Foundations**

We again follow Gabor et al. (2019) by defining interactions between two neural networks and thus having them form an artificial chemistry system or shorter: a soup. Each network within a soup is called a particle. One or more particles can be involved in certain actions, which may change the respective particles’ weight configuration. Actions that involve two or more particles are also called interactions. Actions are performed in sequence with one particle being involved
in at most one action per time step. A sequence of actions for a soup is then called an evolution. Gabor et al. (2019) only use the following actions for self-replicating neural networks:

**Action 1 (attack)** A particle \( N \) attacks another particle \( M \), substituting the other’s weights with its own prediction, given the other network as input, i.e., setting \( M := N \triangleleft_{\text{rep}} M \).

**Action 2 (train-self)** A particle \( N \) trains self-replication with full batch size, i.e., sets \( N := N \leftrightarrow_{\text{rep}} N \).

As frequent interactions tend to destabilize the particles, we use train-self as a background activity, i.e., most of the time a network \( N \) evolves by computing an updated network \( N := N \leftrightarrow_{\text{rep}} N \). However, at certain time steps called interaction points, we select pairs of random particles from the soup and let them perform a random interaction, with attack being the only interaction we defined so far.

In this setup, Gabor et al. (2019) showed that the attack interaction actually has a meaningful effect and (under certain conditions) draws the particles together in the weight space. Thus, some soups show emergent behavior so that the particles, while striving towards fixpoints, end up at fixpoints closer together than they would be without any interactions.

**Introducing Goals**

For a goal-network of format \( N : \mathbb{R}^6 \rightarrow \mathbb{R} \) we can now define a new training action that allows training with respect to the auxiliary function and based thereon we can define a mixed training action.

**Action 3 (train-aux)** A particle \( N \) performs auxiliary training (cf. Definition 8) with batch size \( m \), i.e., sets \( N := N \leftrightarrow_{\text{aux}} \{(x_1, x_2, f(x_1, x_2))\} \) times consecutively with \((x_1, x_2)\) sampled randomly (uniform) from \( \mathbb{R}^2 \).

**Action 4 (train-both)** A particle \( N \) performs either one train-self action (cf. Action 2) with full batch size or one train-aux action (cf. Action 3) with equal chance.

Note that when we employ train-both iteratively, we achieve an effect similar to employing both train-self and train-aux within the soup (hence the name). For a clear comparison, we set the batch size \( m \) equal to the amount of weights present in each network of the soup, i.e., \( m = |N| \).

To allow for more complex interactions between particles, we introduce the learn interactions that have been hinted at in the outlook of Gabor et al. (2019). They work similarly to the train actions but use two distinct particles: one provides the input, i.e., the data to be trained on, and another provides the weights to be trained.

**Action 5 (learn-rep)** A particle \( N \) learns self-replication from another particle \( M \) by training on every weight of particle \( M \), i.e., setting \( N := N \leftrightarrow_{\text{rep}} M \).
Action 6 (learn-aux) A particle $N$ learns the auxiliary task from another particle $M$ by letting $M$ perform an auxiliary application on randomly sampled input $(x_1, x_2)$ and using that same input vector and $M$’s output for one round of training, i.e., setting $N := N \xrightarrow{\text{aux}} \{(x_1, x_2), M \cdot \text{aux} ((x_1, x_2))\}$. We include Actions 5 and 6 to allow particle interactions in ways beyond re-writing their respective weights (i.e., attacking). A soup of particles bound together via repeated learning interactions can also learn as a group as we show in the experiments shortly. Since the impact of learning interactions on the respective particles’ weights is weak compared to the attack interaction (cf. Action 1), we configure them with a severity parameter (1 by default) that indicates how often the network will repeat the interaction with this same partner at this time step. A higher severity will increase the learning effect and will thus provoke a more pronounced change. The rates at which each of the available actions is executed and the number of action occurrences are hyper-parameters that should be tuned depending on the desired objective of the soup’s evolution. We call a soup where all remaining particles fulfill the requirement of a goal-fixpoint (cf. Definition 10) at the end of the evolution a goal-soup.

Experiments

Naturally, we can achieve a goal-soup simply by applying train-both (i.e., Action 4) at every time step. Since we have already shown in the first part of this paper that single networks can converge towards goal-fixpoints, this soup of independent networks can do so as well. However, what we are looking for in the following experiments is meaningful group interaction within a soup. In these cases, we use some form of training action as a background action, which is performed every time step per default. As shown by Gabor et al. (2019) using train-self (or train-both) in this way has a stabilizing effect on the particles’ weight trajectories. Every 50 time steps, we choose another (inter-)action for each soup particle. Time steps where interactions may occur are called interaction points and when considering soups our plots will use these to account for time. For soups we are mostly interested in interactions as they introduce the potential for emergent group dynamics.

The intent of the newly introduced Actions 3–6 can be seen in the following experimental setup: We use only train-self (Action 2) as background activity, which makes drawing train-aux (Action 3) a rarer event. In between the other actions (train-aux, learn-rep, and learn-aux) the background training is not enough for the particles to converge to any fixpoint positions immediately but it helps stabilizing the weights. Figure 6 shows an exemplary evolution over 100 interaction points towards a goal-soup. Keeping the attacking rate low and the learn rates equal and moderately low, we can ensure that the interactions give meaningful, balanced support for the learning process with the occasional opportunity to adjust their abilities with inputs from other networks. We observe that both learning and attacking cause the weights to change their trajectory visibly along the weight space, especially in the early stages. The bigger steps are caused by attacks from still unskilled networks. The smaller steps in contrast are the result of learning actions.

It is clear that adjusting the rates of the performed actions can impact the course of evolution. As shown by Gabor et al. (2019) (for a goal-less case) already, high attacking rates make for a more chaotic evolution in the beginning, causing many particles’ weights to diverge. To an already well-trained soup like the one shown in Figure 6, attack actions have little effect even when executed in large numbers. Such soups will also remain constant even when training actions are removed entirely.\footnote{For these experiments we do not show the plots (a) for space reasons and (b) because they hardly show anything. In this case, the particles’ weight lines just continue onwards.} This shows that at least for a stable soup, the learn interactions can take over the stabilizing function of the train actions. Simply put, learning from a good teacher is similar to learning from the objective truth (both for self-replication and the auxiliary task).

However, leaving out the train actions from the beginning naturally never produces goal-fixpoints, most clearly because train-aux is the only way to convey the actual goal function $f$ to the soup. Thus, for the next experiment we opt for a middle ground: We use train-aux for the beginning of a soup and the turn it off before the auxiliary task.
is learned fully by the soup’s particles. The result is shown in Figure 7a: After a brief initialization period, training is disabled and the learning rates are set to high values with an increased severity. We found that this uninformed information gain was neither damaging enough to collapse/diverge the soup nor helpful enough to contribute to converging in most cases, resulting in the continually unstable trajectories seen in Figure 7a. This makes sense, since we provide no opportunity to increase the accuracy after a certain point in time (as the particles have no way to access the task goal \( f \) any more) and learning from peer particles with the same insufficient auxiliary performance will only destabilize the collective weight quality over time. This instability is also reflected in the fluctuating auxiliary task error seen in Figure 7c (blue line).

However, we discovered that the stability of the whole soup can be helped with the introduction of a single guiding particle or beacon.

**Definition 11 (Beacon)** A particle \( M \) is called a beacon if its weights cannot be changed by any action from another particle, i.e., it cannot learn, or be attacked, but can train and be learned from.

Thus, from the beacon’s perspective it operates alone and thus shows an evolution much like the single network in Figure 1. However, other particles can learn from the beacon and thus the soup can be influenced by its pure existence. We show that we can thus control the soup using only the beacon in accordance to the principles of guided self-organization (cf. Prokopenko (2013)). Since the beacon is able to retain its originally trained weights (being immune to attacks), it can also act like a memory of the previously available auxiliary goal function in the setting discussed above. Figure 7b shows such an evolution: Simply by providing the chance of drawing the guiding particle as a possible learn partner, the soup manages to converge to a goal-soup. Figure 7c (red line) also shows how the auxiliary error converges to 0 after a certain amount of time.

To further test interaction with the guiding particle, we confirm that this guiding particle is strong enough to propagate an auxiliary skill throughout a whole soup on its own. Setting up all other particles without the option to apply Action 3 at all, the only way to reach goal-fixpoint status is by gathering information via Action 6. An exemplary run is shown in Figure 8. As the soup only gets one source of information about the auxiliary goal, the spread of information is slow at the beginning and the trend towards goal-soup convergence only increases as the particles that interacted often with the guiding particle are in turn learned from by other particles. Figure 9 shows that this approach takes quite a bit longer to converge to a goal-soup but can be sped up by favoring the beacon particle for learning interactions. This, of course, again introduces external guidance (as we tell the soup that the guiding particle is probably the best learning

![Figure 7](https://example.com/figure7.png)
Figure 8: A 15-particle soup guided towards a goal-soup configuration by a beacon (circled). Normal particles perform Action 2 for a background action, the beacon performs Action 4. The action rates are: Action 1 at rate 0.05, Action 3 at rate 0, Action 5,6 at rate 0.25.

Figure 9: Comparison of goal-soup convergence speed with different approaches to acquiring knowledge about the auxiliary task until total auxiliary error $\leq \varepsilon$. We compare particles training the auxiliary task normally via train-aux on their own (blue), only learning from the soup but with 30% chance of focusing on the beacon particle (orange) and learning from the group with equal chance for any partner including the beacon (green).

target) and thus quite defeats the purpose of using a beacon in the first place. However, we show that this approach is able to interpolate between using train-aux on all particles and using a beacon only. It could be used to speed up learning for simulation before running a purely beacon-guided soup or in a setup where the auxiliary goal is the soup’s main purpose and self-replication and learning interactions are but an afterthought.

Lastly, we examine the influence of a beacon particle when the soup is already in a bad state: Figure 10 shows an experimental setup of a barely stable soup that has been almost collapsed by a phase of high attack rate. However, when the beacon arrives it can quickly recover to a stable state without any change to the particles besides the beacon’s influence.

Figure 10: A 15-particle soup repaired by a beacon (circled). This setup shows stable training until time step 100, when the soup is heavily strained by high attack rates for 20 turns and then repaired to a functional goal-soup by learning both Action 5 and 6 exclusively from the guiding particle with 30% chance (severity 5). Points left at the top represent a goal networks that did not collapse in the intermediate phase and thus did not need to be recovered. (Some networks are filtered to better show some of the recovery trajectories.)

Conclusion

We provided an architecture for integrating auxiliary goals efficiently into the training process sketched for (non-trivial) self-replicating networks by Gabor et al. (2019). We analyzed the effects of changing the goals on self-replicating networks and noticed their ability to re-learn tasks while also re-learning a fixpoint weight configuration. We lifted the goal networks to a soup and introduced learning interactions for both self-replication and the auxiliary task. Soups bound together by learning interactions can then be aided by a single beacon particle, providing means of setting goals for the soup particles without actually altering their programming but just via controlling the one beacon particle.

To keep both formal descriptions and execution times short, we choose fully trivial goal functions for this first study. While we discovered no reason why more complex goals should be problematic, follow-up research should explore the possibility of applying our paradigm to state-of-the-art tasks for neural networks. This idea could also be tackled the other way around: Instead of going for full self-replication, maybe there are other ‘background’ tasks that can be implemented more easily into larger state-of-the-art networks and still show some improved properties regarding stability or re-training. Even in a soup, all particles learn the same goal in a redundant manner as of now. We thus suggest that further research should tackle the idea that a soup could achieve certain goals as a group, maybe via the specialization of its particles for certain sub-tasks?
References


