Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jonas Brager Jacobsen and William Dalheim

Regeneration and Generalization of Cellular Automata through Evolution Strategies

Bachelor's project in Computer Engineering
Supervisor: Ole Christian Eidheim
May 2021

NTNU
Kunnskap for en bedre verden

Jonas Brager Jacobsen and William Dalheim

Regeneration and Generalization of Cellular Automata through Evolution Strategies

Bachelor's project in Computer Engineering
Supervisor: Ole Christian Eidheim
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Cellular automata are systems of cells that are able to exhibit advanced behavior, only relying on shared update rules and local communication. These update rules can be trained using machine learning to evolve systems specialized on predefined tasks. Evolution strategies (ES) have been rediscovered as a scalable alternative to state of the art reinforcement learning optimization methods and have also been shown to adapt well to noisy environments. We study ES along with a variant of ES using meta-learning through Hebbian plasticity, investigating their abilities to regenerate and generalize on noisy neural cellular automata (NCA), when trained to grow a specific image. The experiments are performed by damaging the models during the growth and evaluate their response. In some experiments, we train the model on damage similar to the damage they are exposed to in the tests to measure their ability to generalize. Our results prove that the ES methods are capable of evolving NCA, but do not indicate to better generalize or regenerate than Adam and SGDM optimizers. We find that ES demonstrates exceptional persistence on NCA where it applies very few changes after growing the target image, keeping the image consistent far longer than it had the chance to during training.

Preface

We would like to thank our supervisor, Ole Christian Eidheim, for providing feedback and participating in discussions. His enthusiasm for our project amplified our motivation. We would also like to thank the Department of Computer Science at NTNU, and specifically Aleksander Tandberg, for providing us with a server cluster to run our experiments.

We thank the reader for showing interest in our work.

Table of Contents

List of Figures

List of Tables

Acronyms

CA Cellular Automaton.

CMA Covariance Matrix Adaptation.

EA Evolutionary Algorithm.

ES Evolution Strategy.

GAN Generative Adversarial Network.

MLP Multilayer Perceptron.

NCA Neural Cellular Automaton.

NES Natural Evolution Strategy.

ReLU Rectified Linear Unit.

RL Reinforcement Learning.

SGD Stochastic Gradient Descent.

SGDM Stochastic Gradient Descent with Momentum.

1 Introduction

Cellular automata (CA) has been researched as a method of developing intelligent self-organizing systems, similar to cells in nature. Such systems show the ability to exhibit advanced behavior, even though a single cell is restricted to the information it gets from its immediate neighbors, without any sense or overview of the system as a whole. Cellular automata in nature show great robustness to outside perturbations, like the way the skin on our fingers heals after a cut. Imagine being able to harness this technology to create self-repairing robots or regenerate damaged organs in a lab. A common pitfall in machine learning is training models in constrained environments, using training data that does not properly reflect real-world scenarios. There is a challenge of developing robust and adaptable models if cellular automata are to be deployed in the physical world where noise is a certainty.

Evolution strategies (ES) have recently been rediscovered by Salimans et al. as a scalable alternative to reinforcement learning [28] and have also been shown to adapt well to noisy optimization tasks [2, 10]. Mordvintsev et al. have previously demonstrated that it is possible to develop CA using neural networks that show incredible regenerative capabilities [21]. In this report we investigate the possibility of developing CA, based on the work by Mordvintsev et al., that better adapt to perturbations using ES instead of state of the art gradient descent optimization techniques. Another motivation for investigating the use of ES is that they are more biologically plausible, which may be a long-term requirement to imitate biological cells. To take this further, we also investigate a variant of ES created by Najarro and Risi [22] that uses a type of meta-learning modeled after the synaptic activity of brain cells, that can adapt the network after training. We refer to this method as Hebbian. Najarro and Risi have demonstrated this method to show great adaptability when introduced to data not seen during training on a quadrupedal robot locomotion task [22].

We hypothesize that Hebbian will learn to regenerate partly erased images in a manner similar to how it shows resilience to weight perturbations as seen in the research by Najarro and Risi. We observe parallels in how weights self-organize in Hebbian with how cellular automata operates as self-organizing systems, speculating that these properties will act complementary to each other. We extend the idea of regeneration to study how Hebbian handles damage not seen during training.

The goal of this report is to investigate how Hebbian and ES's robustness and regenerative capabilities affect their performance in a noisy CA. We hypothesize that ES and Hebbian display better robustness to perturbations than gradient descent optimization methods, and that Hebbian outperforms the other methods at generalizing.

1.1 Outline

This report is divided into nine chapters. The first three, *introduction*, *related work*, and *theory* will give the reader insight into our motivation for the research we present, and provide

them with the appropriate knowledge to comprehend the succeeding chapters. In *method* and *experiments*, we present our approach for selecting and setting up the experiments. In *results*, we visualize the outcomes of our experiments and review these in the next chapter, *discussion*. We summarize our findings and encourage some paths for future work in the conclusion before ending our report by discussing some ethical aspects in *broader impact*.

2 Related Work

The related work presented in this chapter is what we have used as a basis in our research on the generalization and regeneration of cellular automata with ES. Some history is presented to give the reader a clear overview of earlier research.

2.1 Evolution strategies

The first documented use of Evolution strategies (ES) was in 1964, by students at the Technical University of Berlin when trying to find an optimal state of a 3D shape in a wind tunnel to minimize drag. Their experiments involved an iterative process of updating a set of parameters and evaluating the outcome. Every iteration followed two simple rules. The first one was to change each parameter in a random direction, which resembles mutations occurring on offspring in nature. The second rule was to set the perturbed parameters as the new basis if they produced better results than the parent. This rule can be associated with the concept that only the fittest individuals will survive. This form of ES is today known as (1 + 1)-ES, or two-membered ES. In the following years, Schwefel and Rechenberg studied the use of different mutation distributions. They found the Gaussian distribution to be applicable, which is commonly used today. [7]

The first thesis in the field of ES was completed in 1971 by Rechenberg [17]. This thesis described multimembered ES, namely $(\mu + 1)$-ES. This form of ES uses μ parents and selects two at random every iteration to recombine into a mutant. The parents are evaluated alongside the mutant, and the worst of these are discarded. A downside of this method is that the mutation intensity, meaning the distance between parents and offspring, will naturally reduce during training with no way for the developer to control it. As external adjustment of hyperparameters is important to optimize the training process, Schwefel presented two new forms of multimembered ES in 1974. $(\mu + \lambda)$-ES creates λ mutants every iteration and discards the λ worst of the $\mu + \lambda$ population. In (μ, λ)-ES the next iteration's parents will always be μ of the mutants from the current iteration. [7]

Additional variants of ES have been proposed, one of them being Natural ES (NES) by Wierstra et al. [38]. Instead of directly replacing parents with their offspring and disposing of information that the low-fitness mutants hold, NES estimates the gradient using the fitness and the mutation matrices. This gradient is used to update the parent parameters in the direction of the highest expected fitness. The gradient estimate is similar to how weights are adjusted using REINFORCE-algorithms [40]. Wierstra et al. also introduced a technique of shaping the fitnesses non-linearly across the population [38].

In recent years, computational resources have become more available and distributed across cores, which benefits ES as it is easier to parallelize compared to other optimization techniques. ES does not require the communication of gradients across threads, reducing bandwidth and time between iterations. This led to ES being rediscovered by OpenAI in 2017

as a scalable alternative to the current state of the art reinforcement learning algorithms. Their experiments with using a form of Natural ES showed significant improvements in speed and stability when solving certain MuJoCo-tasks [28]. Using this method, they trained a 3D humanoid to walk in only 10 minutes by parallelizing the workload across 1440 CPU cores.

2.2 Meta-Learning through Hebbian Plasticity

Meta-learning is the process of learning to learn. The goal is to create a network that can learn adaptively as the environment changes [35, 30]. The networks are typically introduced to some variation of tasks during training and then evaluated on their ability to adapt to unseen tasks.

Najarro and Risi explored the idea of initializing a network with random weights and then evolving Hebbian rules for each connection in the network [22]. The network is updated using the trained rules which enables the agent to adapt during an episode. This differs from common ES and reinforcement learning algorithms where the policy is modified when the episode is finished. The Hebbian rules modify the network based on the firing of neurons, with neurons that fire together being more likely to develop stronger connections with each other. Najarro and Risi present results indicating that meta-learning through the evolution of a plastic network can generalize well on data not seen during training.

2.3 Cellular Automata

Von Neumann worked on cellular automata (CA) as a model for self-replication in the 1940s [23]. His goal was to design a machine that could grow similar to biological organisms under natural selection. The machine he used consisted of a two-dimensional grid representing the cells, each with its own state. The cells update every time step depending on the state of their neighboring cells. Even though one individual cell is simple, a group of cells is able to express complex behaviors, depending on the starting state and the transition rules of the system. John Conway's Game of Life is a famous experiment where the player sets an initial condition and observes how the cellular automaton evolves, often in interesting and visually pleasing patterns. The game has been proven to be Turing complete [1]. Miller used evolutionary algorithms (EA), a superclass of evolution strategies, to create and regenerate a French flag starting from one cell, proving that EA has the ability to evolve growing cellular automata and that it exhibits some regenerative capabilities [20].

CA that use a neural network in place of the update rules are referred to as neural cellular automata (NCA). Mordvintsev et al., who has been the main inspiration for our research, developed a growing NCA to generate images of emojis, starting from a single cell [21]. Mordvintsev et al. also present the growing neural cellular automata's ability to recover from damage, when being trained for it. Similar research on regeneration has been done by Horibe, Walker, and Risi who have written about regenerating soft robots through growing NCA [16], and Sudhakaran et al. who observed regenerative properties on an NCA network trained to

grow 3D structures [32].

2.4 Self-organization

The NCA described in chapter 2.3 show the ability to express complex systems through local communication, showcasing examples of advanced self-organization. This type of order arising in a seemingly disordered system that only reacts to local interactions is observed in many fields. Examples include crystallization, animal swarming, and languages [5]. Related to machine learning, Wu et al. wrote a popular paper on graph neural network models that send messages through the system by passing information through the graph edges [41]. Research has also been done on autonomous robots, exhibiting self-organizing swarm behavior [36, 18].

2.5 Robustness to Noisy Environments

Beyer identifies Darwinian theory as an optimization technique heavily affected by many forms of noise and argues that Evolutionary Algorithms (EA) should handle noise well as it is built on ideas inspired by this theory. Traditional optimization methods are often reliant on sequential and deterministic information to find optimal solutions, in contrast to evolution in nature which evolves solutions even when being exposed to high degrees of noise [6]. Evolution strategies (ES) have proved to be robust in noisy environments [13, 3]. Back and Hammel show that convergence does not take more time when introduced to noise, as long as the noise does not exceed a certain threshold. They argue that the robustness to noise is due to the non-deterministic selection process of the next generations [3].

Genetic algorithms (GA), a sibling of ES, creates the next generation's population by combining parents with elements of mutation. GA share some similarities with natural evolution strategies (NES) in the way a new population is created by recombining the individuals of the last generation, based on their fitness. There exist a lot of research on GA in noisy environments. Fitzpatrick and Grefenstette have in different studies explored the effect of varying the population size in genetic algorithms, and shown that reducing the time spent on evaluating a population's fitness and instead increasing the number of generations evolved, might improve the performance of the algorithms. In both studies, genetic algorithms have shown to perform well in noisy environments, and interestingly, better than without noise [11, 12]. In the experiments by Then and Chong they recognize these findings and further demonstrates that GA is also effective at finding global optima in noisy environments [34].

3 Theory

The theory presented in this chapter is meant to provide the reader with the necessary knowledge to understand the concepts and experiments discussed later in the report.

3.1 Evolution Strategies

Evolution strategies (ES) are a class of algorithms that uses heuristic searches to perform a type of black-box optimization [28]. The algorithms construct a population at every generation that has been mutated from a parent and evaluate their performance, called fitness. There exist many variants of ES, where most differ in how the next generation's parents are derived from the mutants. $(\mu + \lambda)$-ES and (μ, λ)-ES use a technique of directly replacing the parents with some of the offspring for the next generation [7]. In natural evolution strategies (NES), the mutants weighted by their fitness are used to adjust the parameters of the current parent to create the next generation's parent [38].

The variant of ES we employ is identical to the one used by Salimans et al. [28], which is a type of NES. This method, referred to as OpenAI-ES, is characterized by its mutation distribution being a constant Gaussian distribution which differs from conventional NES where this is adapted throughout training. Algorithm 1 presents the OpenAI-ES pseudocode by Salimans et al. [28, Algorithm 1] with some minor modifications.

Algorithm 1 Evolution Strategies (OpenAI-ES)

1: **Input:** learning rate α, noise standard deviation σ, population size n, initial policy parameters θ_0
2: **for** $t = 0, 1, 2, ...$ **do**
3: Sample $\epsilon_1, ..., \epsilon_n \sim \mathcal{N}(0, \sigma^2)$
4: Compute fitness $F_i = F(\theta_t + \epsilon_i)$ for $i = 1, ..., n$
5: $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
6: **end for**

Algorithm 1 iterates two steps. The first step, line 3-4, is to add normal-distributed noise to the network's policy parameters to create a perturbed population and evaluate their fitness. Line 5 combines the fitnesses, calculates a stochastic gradient estimate, and uses this to update the network's policy parameters. θ_t denotes the parent parameters, and $\theta_t + \epsilon_i$ is the ith mutant.

The fitnesses of a population can be tweaked to better guide the network towards optimal solutions, generally called fitness shaping. The fitness shaping used in this report is rank-based, meaning that the fitnesses assigned to a population depend on their positions relative to each other, and not the actual objective value. This method of fitness assignment is more robust than proportional fitness, as it reduces the influence of outliers and makes the algorithm less likely to fall into local optima early in the training [39, 4, 37].

3.2 Meta-Learning through Hebbian Plasticity

Meta-learning through Hebbian plasticity is a type of ES where a set of "Hebbian rules" are trained instead of the network parameters. The network is initialized to random values at the start of each generation and is updated during the episode according to the trained rules. There are several variations of the algorithm. The one used in this report is the algorithm that yielded the best results in the paper written by Najarro and Risi [22].

The algorithm is based on concepts of correlation-based learning from Hebb's postulate, written by Donald O. Hebb, in 1949, about neuroscience [15]. There are two sections of Hebb's postulate that are needed to formulate the algorithm. The first is about locality in a synapse (the connection between two neurons), which implies that change of synaptic efficacy is only dependent on local variables, such as presynaptic and postsynaptic firing rate and the value of the synaptic efficacy. The other section is about joint activity. In this section, it is argued that pre- and postsynaptic neurons have to fire together to change their synaptic weight. This is the origin of the saying "what fires together, wires together". We use a generalized Hebbian ABCD update rule [31] inspired by the aspects of locality and joint activity, as given by Najarro and Risi [22, Equation 1]:

$$\Delta w_{ij} = \eta_w \left(A_w o_i o_j + B_w o_i + C_w o_j + D_w\right) \tag{1}$$

Here, w_{ij} is the synaptic efficacy/weight between two neurons i and j, o_i and o_j the pre- and postsynaptic activations, η_w the evolved learning rate. A, B and C is the correlation, presynaptic and postsynaptic term, and D can be viewed as a bias. The coefficients are unique for every parameter in the network, allowing each connection to evolve its own learning rates, rules, and bias.

Inspired by the method used by Najarro and Risi [22] we formulate algorithm 2 as pseudocode for meta-learning through Hebbian plasticity in ES:

Algorithm 2 Meta-Learning through Hebbian Plasticity

1: **Input:** learning rate α, noise standard deviation σ, population size n, weight update rule H
2: Initialize uniform distributed hebbian coefficients h_0,
3: **for** $t = 0, 1, 2, ...$ **do**
4: Sample $\epsilon_1, ..., \epsilon_n \sim \mathcal{N}(0, \sigma^2)$
5: **for** $m = 1, 2, 3, ..., n$ **do**
6: Sample $\theta_0 \sim U$
7: **for** $i = 0, 1, 2, ...$ **do**
8: Perform step in environment using θ_i
9: $\Delta w_i \leftarrow H(\theta_i,\ h_t + \epsilon_m)$
10: $\theta_{i+1} \leftarrow \theta_i + \Delta w_i$
11: Normalize θ_{i+1}
12: **end for**
13: $F_m \leftarrow F(h_t + \epsilon_m)$
14: **end for**
15: $h_{t+1} \leftarrow h_t + \alpha \frac{1}{n\sigma} \sum_{m=1}^{n} F_m \epsilon_m$
16: **end for**

In contrast to algorithm 1, which starts with initial policy parameters θ_0, algorithm 2 starts with Hebbian coefficients h_0 and a weight update rule H. The inner loop, line 7-12, represents an episode for a mutant m. Every step i consists of calculating the weight change from the post- and presynaptic activity and the Hebbian coefficients using the weight update rule H. The fitness of mutant m is then used to update the parent Hebbian coefficients on line 15.

3.3 Cellular Automata

A cellular automaton (CA) is a system of cells, normally living on a 2D lattice, that update their state based on the state of their neighbors. There exist several different cellular automata, distinguished by their update rules and grid formation. CA generally evolves by having all cells transition to their next state synchronously at discrete time steps. The systems are characterized by three fundamental concepts, uniformity, synchronicity, and locality. Cells use the same rules to update their state, simultaneously across all cells. Similar to nature, the cell's perception is limited to its close surroundings. [29]

3.4 Gradient Descent Optimization

Gradient descent is a method to find a local minimum of a function $f(\theta)$, $\theta \in \mathbb{R}^d$ being a model's parameters, by updating the parameters in the direction opposite to the gradients at the current point of the function. This path is the direction of the steepest descent.

If $f(\theta)$ is a multivariable function that is defined and differentiable, we move θ in the direction of the negative gradient $-\nabla f(\theta)$, to decrease f the fastest. Using a small step size

$\alpha \in \mathbb{R}_+$, if

$$\theta_{t+1} = \theta_t - \alpha \nabla f(\theta_t) \tag{2}$$

then $f(\theta_t) \geq f(\theta_{t+1})$.

3.4.1 Stochastic Gradient Descent with Momentum

Stochastic gradient descent (SGD) is a variation of gradient descent where instead of iterating through all training data before updating the parameters, the parameters are updated after each data sample by estimating the gradient. If the dataset is large, gradient descent may become costly, and SGD can run a lot more iterations than standard gradient descent and thus converge faster, even though the stochastic approach is noisier. [8]

In an article by Rumelhart, Hinton, and Williams a simple addition to SGD appeared, called momentum [27]. SGD has been shown to have trouble navigating ravines [33] and momentum helps accelerate convergence in the desired direction. The momentum is incorporated by adding the last step update vector v_{t-1}, multiplied by the momentum parameter γ, to the current update vector v_t. γ is a value between 0 and 1, often set to 0.9. [26]

$$v_t = \gamma v_{t-1} + \alpha \nabla f(\theta_t) \tag{3}$$

$$\theta_{t+1} = \theta_t - v_t \tag{4}$$

3.4.2 Adam

Adam is an efficient stochastic optimization method requiring only first-order gradients. It was introduced by Kingma and Ba [19]. Adam was made to combine the benefits of two other optimization methods, Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). AdaGrad works well on sparse gradients, and RMSProp has the advantage of handling non-stationary settings well. Similar to AdaGrad, Adam maintains individual adaptive learning rates for all network weights using estimated first and second moments of the gradients. These moments are the mean and the uncentered variance respectively of the gradient, and are calculated as follows:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{5}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{6}$$

Where $\beta_1, \beta_2 \in [0, 1\rangle$ are exponential decaying rates and g_t being the gradient of the function to be minimized at time step t. Since the moments are initialized as empty vectors, the next step is to bias-correct them:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} \tag{7}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{8}$$

With the moments calculated, they can be used to create the next iteration's parameters:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{9}$$

where α denotes the learning rate and ϵ is a small floating-point number whose purpose is to avoid division by zero.

4 Method

In this chapter, we describe the process, methods, and hyperparameters used to train the algorithms that will be compared in later chapters. The code used in this report can be found at `github.com/jobrajac/ca-es`.

4.1 Algorithms

Four different evolution strategies (ES) algorithms were tried on neural cellular automata (NCA):

OpenAI-ES. The algorithm is described in chapter 3.1. Moving forward, all references to the term ES indicate the use of OpenAI's algorithm.

CMA-ES. Perhaps the most popular variant of ES, covariance matrix adaptation evolution strategy is a method to update the pairwise dependencies between the variables in the distribution (the covariance matrix) before mutation. In contrast to ES, CMA-ES requires fewer hyperparameters, only initial weights, initial step size, and size of population. The algorithm has been proven to outperform 31 other black box algorithms in a survey made by Hansen et al. [14].

Sep-CMA-ES. Separable-CMA-ES is a modification to CMA-ES that reduces the degrees of freedom in the covariance matrix, and in turn, lowers the time and space complexity from quadratic to linear. Sep-CMA-ES has indicated to better adapt to high-dimensional problems [25].

Meta-learning through Hebbian Plasticity. The algorithm is described in chapter 3.2. We refer to the algorithm as Hebbian for the remainder of the report.

We could not get CMA-ES to converge on NCA, even on simple 5 x 5 grids. With very few hyperparameters to tune other than initial conditions and population size, we suspect the algorithm is not suited for NCA with many parameters. Our implementation contains 2048 parameters, and to the best of our knowledge, there are no CMA-ES implementations on a problem with over two thousand parameters where the output of the network is kept and evaluated as floating-point numbers. The failure of CMA-ES motivated the use of sep-CMA-ES, which as mentioned above has shown promising results on larger dimension spaces. Unfortunately, like CMA-ES, sep-CMA-ES failed to converge. Only the standard ES and the Hebbian algorithm were able to improve their fitness over time, and are the only two methods we will be comparing against gradient descent methods in this report.

There are a variety of optimizers available using gradient descent optimization with back-propagation. In this report, we use Adam, a popular and advanced method that has been shown to achieve great results fast. This was also the method used by Mordvintsev et al. [21].

The other method we employed was stochastic gradient descent with momentum (SGDM). We chose this as the original stochastic gradient descent (SGD) has shown to better generalize than Adam [42], but SGD alone was not able to evolve the NCA in our tests. Adding the momentum parameter to the equation solved this issue. Both methods are described in chapter 3.4.

4.2 Parallelization

Part of the task was to parallelize ES on a cluster of CPU machines. The Department of Computer Science at NTNU provided access to five servers running on their own Dell Inc. PowerEdge R630[1]. Each server is equipped with two Intel Xeon E5-2620 v4[2], has 32 GB of allocated RAM, and 32 virtual cores using hyperthreading.

The first step in implementing parallelization was to distribute work across the cores of each machine. For this, we used the built-in multiprocessing library in Python. We ran subprocesses instead of threads, as the Global Interpreter Lock does not allow multiple threads to run Python bytecode at the same time in CPython[3]. When exploring the use of multiprocessing for NCA, we experienced that PyTorch by default was set to use all cores to run its computations. We had to block this behavior to fully utilize the cores for our purpose. For every generation, we assign one subprocess to each mutant in the population.

In the first implementation of multiprocessing for NCA, we overlooked a critical flaw that made the training appear as if it converged more steadily and at a steeper rate than a single-processed run. We later discovered that new subprocesses fork the runtime, including the random number generator and its state. Since no calls to create random numbers were made in the main process, the subprocesses would always produce the exact same noise across generations and the population. The solution was to seed the random number generator in every subprocess as soon as they had been forked.

With the ability to parallelize across each machine's cores, the next step was to run all machines together as a cluster. We used Redis[4], an in-memory data structure store, to handle communication between the machines in the cluster.

We assigned one machine to be the master, which in this case means being in charge of coordinating all workers and updating the parameters. An iteration starts when the workers get notified that the master has pushed updated parent parameters to the Redis server. A worker will then create a random seed to be used in creating the perturbations on the parent parameters, and calculating the fitnesses from these. It then sends the seed and the fitnesses back to the master. The master will use the seeds to reconstruct the perturbations and

[1] Dell PowerEdge 360
[2] Intel Xeon E5-2620 v4
[3] docs.python.org/3/library/threading
[4] redis.io

update the parent parameters with the fitnesses. The described relationship can be seen in figure 1.

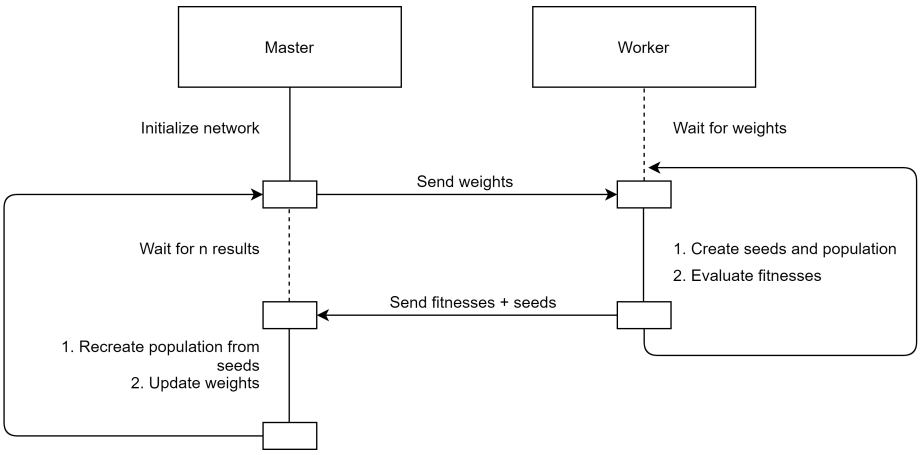


Figure 1: The relationship between a worker and the master. The master sends the initialized weights to the worker. The worker creates the seed that will be used to construct its population. The fitnesses and seed are sent to the master who then recreates the population and updates the weights. The exit condition triggers after a set amount of iterations of the loop described.

4.3 Cell Perception and States

The NCA used in this report applies three filters to a cell's perception of its neighbors before feeding it through the network. An identity filter and two Sobel filters, one in the x- and one in the y-direction. The identity filter is a single-entry matrix with the middle element set to 1, thus only selecting the existing information about the cell. The Sobel filters are a common technique used in computer vision for better edge detection. The filters:

$$Id = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, S_y = \begin{bmatrix} -1 & 2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Every cell's state is represented by RGB values, a channel α to determine if the cell is alive, and 12 additional channels to convey arbitrary information to neighboring cells. There is no predefined rule for what information should be stored in the last 12 channels, it is the network's task to decide how they will be utilized. We refer to these additional channels as hidden channels. A cell is determined alive if its α or any of its neighbors $\alpha > 0.1$. Any cell that is not alive will not update nor convey information to its neighbors. Earlier work on CA often uses sequential updates of cells [23], but biological cells are not synchronous. To model an NCA that is asynchronous in its updates, with no global clock or external synchronization, Mordvintsev et al. introduced an update-probability [21]. This probability determines if the cell should update or not during an update step. The probability is set to 0.5.

4.4 Training a Neural Cellular Automaton

The process of training an NCA can be viewed as a reinforcement learning (RL) problem where an agent has to coordinate a group of cells in forming a specific structure. Each cell acts in a unique environment consisting of its state and its neighboring cells' states. This makes it a single-agent-multi-environment problem where the environments affect each other. This problem is more supervised than conventional RL, like the common benchmark environments seen in OpenAI Gym[5]. The agent is not evaluated at each step, but rather at the end of an episode. There are many possible paths in the growing process, making it difficult to determine the best one in an unsupervised manner and reward the agent accordingly. The agent is rewarded as the mean squared error of the RGBα values from the target image it is learning to grow. As this reward is calculated directly and the agent is having a fixed target, we consider this task to be more supervised. On the other hand, the agent is not evaluated on a cell-by-cell basis, thus not rewarding the agent on how its policy affected a specific cell. ES has not been shown to train faster than gradient descent optimization methods on more supervised tasks. OpenAI found ES to require 1000 times longer training time on the MNIST-dataset[6]. Still, our focus is to study regenerative properties and generalization, and we do not consider longer training time an obstacle.

We use the code published by Mordvintsev et al.[7] as a basis for the NCA, using a similar network architecture, the same cell perception, and the same loss function for evaluation. To best compare the methods, we use as many shared functions and hyperparameters as possible. Even though the network architecture was designed for using the Adam optimizer, we found it to work great with ES. Some other architectures were tried to see if ES could be evolved quicker using different layers and activation functions, but we did not find any where that was the case.

4.4.1 Network Structure

In the code published by Mordvintsev et al. [21], the network structure consists of two-dimensional convolutions with a kernel size of 1, also known as pointwise convolutions or 1 x 1 convolutions. Using this setup, the network iterates over all cells and multiplies its weights with the channels of an individual cell. A pointwise convolution maintains the height and width of the input tensor but modifies the number of channels. This operation can be seen as performing multiple passes through one layer in a multilayer perceptron (MLP). A pointwise convolution takes in input with shape (h, w, ch_0) and produces an output of shape (h, w, ch_1) as seen in figure 2. h and w denotes the height and width of the tensor respectively, and ch is the number of channels.

[5] gym.openai.com/envs

[6] openai.com/blog/evolution-strategies

[7] github.com/google-research/self-organising-systems/blob/master/notebooks/growing_ca.ipynb

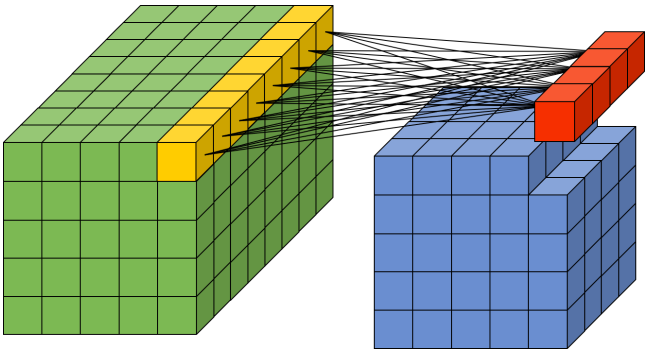


Figure 2: 1 x 1 convolution. The green box is the cell grid and the blue box is the output tensor. The yellow vector can be viewed as the channels of one cell and the perception of its neighbors. The red vector is the output. Connecting these are the weights of a 1 x 1 kernel or a single layer in an MLP, the latter being apparent when removing the green and blue shapes. With this convolution, only the channel dimension is altered.

As emphasized by Najarro and Risi, attempting to define any presynaptic and postsynaptic activity for a convolution layer is challenging [22]. We changed from a 1 x 1 convolution network to a fully connected structure, as it does not increase the number of parameters, and allowed the implementation of Hebbian Learning on the NCA problem. As the net is trained on small images and thus reducing the difficulty of the problem, we decided to use 32 neurons in the hidden layers instead of 128 as used by Mordvintsev et al. Najarro and Risi's network for Hebbian Learning did not accommodate bias, while Mordvintsev et al.'s did. Since early testing showed that bias was not necessary for NCA, no bias was used for all models to strive for a standardized network structure across all tests. Other than these design changes, the network structure was kept similar to Mordvintsev et al. [21]. This structure consists of two fully connected layers with a Rectified Linear Unit (ReLU) activation function after the first layer. Their design also includes a "do-nothing" behavior at the start of training, which is accomplished by initializing the weights of the second layer to zero.

The input to the network is a tensor of shape (cell grid height, cell grid width, channels $\cdot$ filters), and the output of shape (cell grid height, cell grid width, channels). The output is of the same shape as the cell grid and is directly added to the cells. Using a shared network for all cells ensures that they share the same update rule.

4.4.2 Hyperparameters

All training and experiments were run using Python version 3.8.5. ES and Hebbian were trained with PyTorch[8] 1.8.0. The code used to train ES and Hebbian can be found at `github.com/jobrajac/ca-es`. Adam and SGDM were trained using TensorFlow[9] 2.4.1 with the code published by Mordvintsev et al.[10].

[8] pytorch.org

[9] tensorflow.org

[10] github.com/google-research/self-organising-systems/blob/master/notebooks/growing_ca.ipynb

The models share some static hyperparameters, given in table 1.

Cell channels	16
Cell grid size	9 x 9 pixels
Update steps	30 - 40
Cell update probability	0.5

Table 1: Shared static hyperparameters.

The 16 cell channels represent the cell state and consist of R, G, B, α, and 12 hidden channels. We use a 9 x 9 cell grid size as a compromise between the time required to train the models and image clarity. ES and Hebbian take time to train, and we would not have been able to train all the models on a larger cell grid. The update steps determine how many iterations the cell grid will update each generation, and is set to a number between 30 and 40 at random. The cell update probability is the probability for a cell to change its state with the output of the network, each update step.

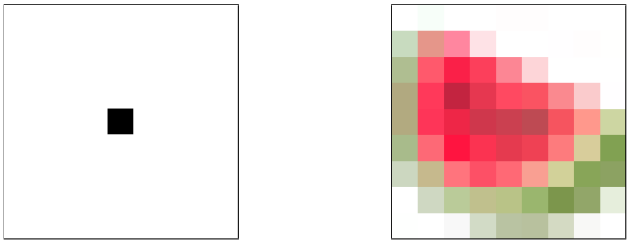
The loss is calculated as the mean squared error between the produced image and the target image. ES and Hebbian use the loss as a basis to sort the population's performance, which is then given fitnesses evenly mapped from -0.5 to 0.5. The shaped fitness F of an individual in the sorted population with size n is given by

$$F = \frac{\text{position}}{n-1} - 0.5, \text{ position} \in \{0, 1, 2, \ldots, n-1\} \tag{10}$$

SGDM uses a momentum of 0.9 in all but one experiment. ES and Hebbian use a σ of 0.01 to scale the mutations added to the weights and coefficients. Most experiments are run with a population size of six. While we later demonstrate in table 2 that a population size of six is not the optimal size for fast convergence, keeping the population size low allowed us to train several models in parallel on a single machine.

For ES and Hebbian the learning rate varies but is always decreased by a factor of 0.3 when the mean loss ≤ 0.03 and again by a factor of 0.5 when the mean loss ≤ 0.01. Adam and SGDM use a learning rate of 0.002 which is decreased by a factor of 0.1 after 2000 iterations. Hebbian initializes its coefficients from a uniform distribution between -0.001 and 0.001. The network Hebbian creates for each generation is initialized to values from a uniform distribution between -0.1 and 0.1.

The starting cell grid is set to zero for all cells on all channels, except for the one cell in the middle which is set to one on all hidden channels and α. This grid is referred to as the seed. The seed and the target image are visualized in figure 3.



(a) (b)

Figure 3: **(a)** The starting cell grid, referred to as the seed, where the middle cell has its hidden channels and α set to one. **(b)** The target image used to calculate the loss. Both images are 9 x 9 pixels.

Some of the models were trained retrieving previous cell grids from a storage to use as a starting point instead of the seed. We refer to this storage as a sample pool. It acts as a tool to train the models to not destroy a fully grown image and as an arena to apply damage to images, which the models can learn to repair. When a model asks for a cell grid from the pool it is chosen at random from one of the 1024 saved cell grids. At the end of each generation, the models commit their cell grids to the pool, replacing the ones they sampled. ES and Hebbian use a batch size of four, keeping a slot reserved for the seed to prevent catastrophic forgetting. Adam and SGDM always use a batch size of eight, regardless of training with a pool or not. ES and Hebbian are more reliant on randomness to evolve, which in earlier generations produce multiple cell grids containing only dead cells. To prevent the pool to be filled with cell grids that cannot be improved upon, we restrict ES and Hebbian from committing the cell grid if all cells are dead. ES and Hebbian also produce more outliers than Adam and SGDM. Therefore, to save time during training we only commit the best solution from each generation to the sample pool.

Any hyperparameters or variations during training not mentioned above are listed for the specific models in chapter 6.

5 Experiments

In this chapter, we describe the four categories of experiments used to compare the model's regenerative properties and ability to generalize.

5.1 Models trained to grow

In the first experiments, we compare models trained to grow from the seed to the target image, without being trained with a sample pool or any type of damage. We perform three tests:

Quadratic erasing at the 35th update step. To evaluate how the models regenerate when they lose parts of their cell grid, we damage the models by setting all channels, including RGB and α, to zero. We test with three different sizes of damage, 2 x 2-, 3 x 3-, and 4 x 4 cells. The damage is done at the 35th update step to best evaluate regeneration when the image is close to fully grown.

Random erasing. As a more thorough approach to evaluating regeneration, we run the test with random widths of the damage box, between 1 and 4 on both axes, on random locations in the cell grid, and at random update steps between 20 and 40.

Random noise at every update step. In this test, we evaluate the models' regenerative capabilities when exposed to noise. Instead of setting the channels to 0 we add noise sampled from $\mathcal{N}(0, 0.01^2)$, $\mathcal{N}(0, 0.03^2)$ and $\mathcal{N}(0, 0.05^2)$. The noise is added at every update step, to every cell where $\alpha > 0.1$. This is done to disturb the cells where there is change, not adding random noise to the edges of the cell grid where there are no cells alive.

5.2 Models trained with random damage to all channels

In these experiments, we compare models trained using a sample pool with applied damage to the sampled cell grids. The damage is applied to the bottom half of the cell grid by setting all 16 channels to zero inside a rectangle with side lengths between 1 and 4 at random. Figure 4 shows an example of a batch sampled from the pool.

Figure 4: Example of a batch sampled from the pool and exposed to random damage in the bottom half of the image. The damage is applied to all channels and has a width and height between 1 and 4 pixels.

We evaluate the models by applying damage at the 35th update step when the cell grid is close to grown since the damage done during training after sampling from the pool is also applied to grown cell grids.

Two tests are performed, one with damage in the bottom half of the image, and one with damage in the top half of the image. The main reason for dividing the damage into the two halves of the grid is to test Hebbian's alleged ability to better generalize on damage not seen during training, but having experienced damage similar to it.

5.3 Models trained with damage to the green channel

In this chapter and chapter 5.4, we further test the models' abilities to regenerate and generalize damage not seen during training but having experienced damage that affects the cells in similar ways.

The models in these experiments are trained with setting the channel representing the color green to zero in the cell grids sampled from the pool. Figure 5 is an example of how a batch sampled from the pool might look like.

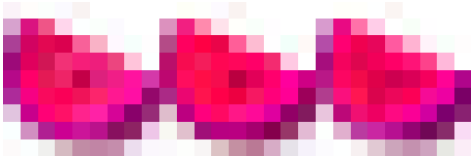


Figure 5: Example of a batch sampled from the pool and having the channel representing the color green set to zero for all cells.

Two tests are performed where a channel representing a color is zeroed out, first green which they were trained with, then blue which they were not trained with. The damage is applied at the 35th update step.

5.4 Models trained with damage to hidden channels

In preparation for these experiments, we train models with a 0.5 probability at every generation of damaging the last four channels. The damage consists of making these channels unresponsive to changes and is achieved by zeroing them at every update step for that generation. This results in 50% of all generations having the input to the network as 0 for these channels, rendering them effectively useless in those generations.

Six tests are performed. The first three involves damaging the first four-, the middle four- and the last four hidden channels. In the next three tests, we damage one, two, and three random channels of the first eight hidden channels, which were always present during training. For all experiments, the RGBα channels are not affected. The damage is applied at every update step.

These experiments are inspired by the tests Najarro and Risi performed in the quadruped environment with Hebbian. For every episode, their agent may start with its right front leg

damaged, making the corresponding joints non-responsive to its actions. Their results show that the agent achieves great performance when exposed to damage on another leg.

6 Results

Adam, SGDM, ES, and Hebbian all learn to grow an image of a watermelon, but in quite different time frames. Table 2 shows how much time and how many generations are required to reach a $\log_{10}$(loss) of -2.5 with the different methods. Table 3 is an overview of how much time it takes to run ES with large population sizes for 1000 generations, when using one machine and when using five machines in parallel. The data in both tables are gathered using the hardware specifications listed in chapter 4.2. Figure 6 shows how an evolution in ES with population size 6 might look like.

Model	**Batch size**	**Population size**	**Training time (minutes)**	**Generations**	**gens/second**
Adam	8	-	0.44 ± 0.05	671 ± 66	25.49 ± 0.94
SGDM	8	-	1.21 ± 0.08	1880 ± 162	25.87 ± 1.26
ES	1	6	286 ± 73	193067 ± 38991	11.38 ± 0.73
ES	1	12	184 ± 20	94607 ± 11130	8.57 ± 0.61
ES	1	18	140 ± 33	82583 ± 9616	5.84 ± 1.33
ES	1	24	229 ± 47	74113 ± 12883	5.42 ± 0.24
ES	1	30	276 ± 21	77523 ± 6919	4.67 ± 0.06
Hebbian	1	6	1223 ± 95	276290 ± 21599	3.77 ± 0.00
Hebbian	1	12	945 ± 203	112293 ± 37003	2.10 ± 0.92
Hebbian	1	18	773 ± 283	105540 ± 38431	2.28 ± 0.01
Hebbian	1	24	762 ± 171	88097 ± 21512	1.92 ± 0.04
Hebbian	1	30	932 ± 129	87350 ± 8849	1.57 ± 0.07

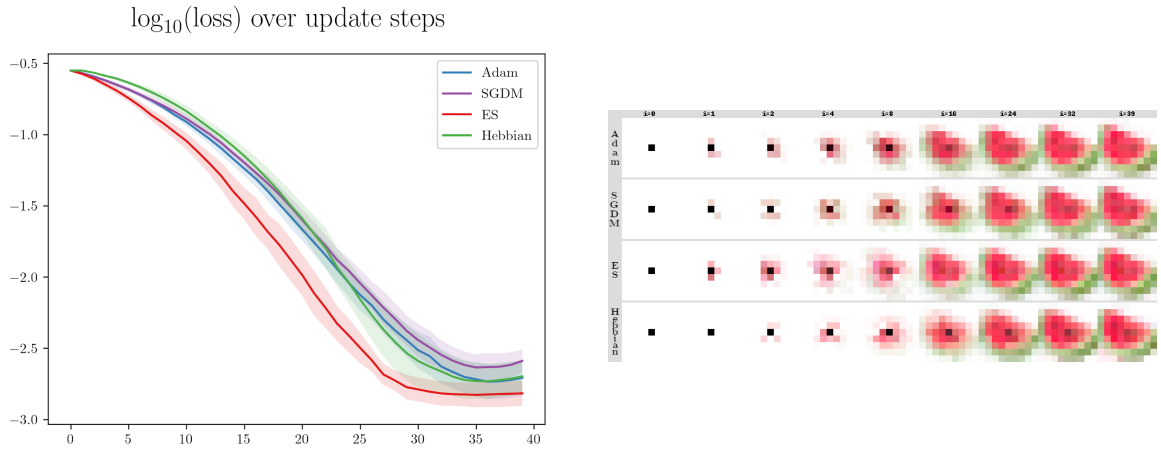
Table 2: Comparing the time and generations required to train the models to $\log_{10}(\text{loss}) = -2.5$. The data is calculated as the average across three runs for every row in the table. Hardware specifications are listed in chapter 4.2. Adam and SGDM: lr 0.002. ES and Hebbian: lr 0.005 (Hebbian with population 6: lr 0.003).

Nodes	Population size	$\frac{\text{minutes}}{\text{1000 gens}}$	$\frac{\text{gens}}{\text{sec}}$	$\frac{\text{mutants}}{\text{sec}}$
1	30	3.52 ± 0.07	4.73 ± 0.09	142
1	60	6.67 ± 0.05	2.50 ± 0.02	150
1	120	12.40 ± 0.20	1.34 ± 0.02	161
1	300	32.30 ± 0.46	0.52 ± 0.01	155
1	600	66.08 ± 0.49	0.25 ± 0.00	151
1	1200	142.89 ± 2.46	0.12 ± 0.00	140
5	30	4.70 ± 0.01	3.55 ± 0.01	106
5	60	5.33 ± 0.02	3.13 ± 0.01	188
5	120	6.71 ± 0.03	2.49 ± 0.01	298
5	300	11.03 ± 0.07	1.51 ± 0.01	453
5	600	18.47 ± 0.02	0.90 ± 0.00	542
5	1200	34.15 ± 0.63	0.49 ± 0.01	586

Table 3: Comparing the time required to run ES for 1000 generations with varying population sizes on one node and five nodes. The data is calculated as the average across three runs for every row in the table. The last column, $\frac{\text{mutants}}{\text{sec}}$, shows how many mutants are evaluated every second. Hardware specifications are listed in chapter 4.2.

Figure 6: Evolution in ES with population size 6, visualized on selected generations.

Figure 7a shows the models compared to each other when only trained to grow a static image, and with no interference other than the randomness they were trained with. The lines in this graph and the ones presented in the following chapters represent the models' average across several runs, usually 10 per model unless otherwise specified, and their shadows the standard deviation from the average at each update step. Figure 7b is an example of how the cells grow over 40 update steps with the trained models.



(a) (b)

Figure 7: **(a)** Loss over 40 update steps. **(b)** Example of how cells grow over 40 update steps with the featured models.

Figure 8 visualizes all channels across several update steps, as enumerated in the grey column. The first image column shows the composite of the following four columns, channels R, G, B, and α. These channels are clipped to fit in the range [0, 1]. The remaining columns are the 12 hidden channels where negative values are visualized in blue and positive values in red. These values are scaled in relation to each other with the strongest color representing the highest absolute value. Empty cell channels (with 0 as value) are visualized as white pixels.

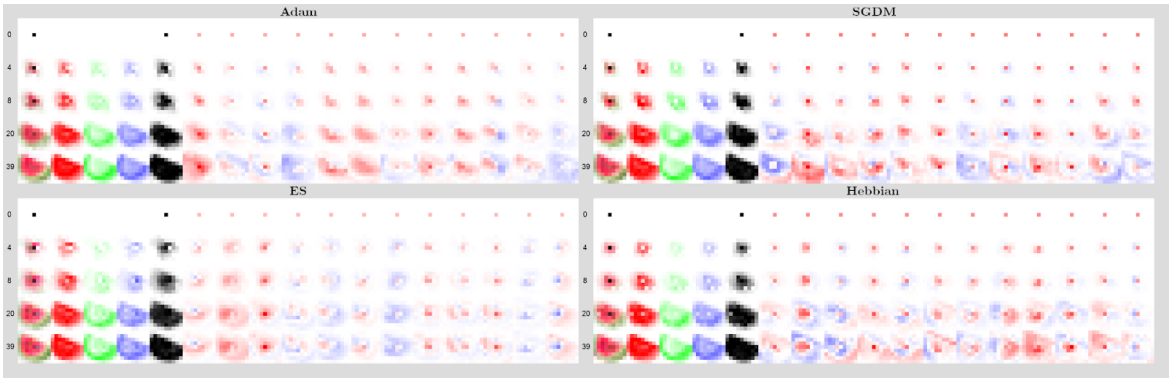


Figure 8: Visualization of all channels of the featured models. The column furthest to the left is the composite of the following four columns, channels RGBα. For the hidden channels, negative values are shown in blue while positive values are red. The grey columns indicate which update steps the rows are sampled from.

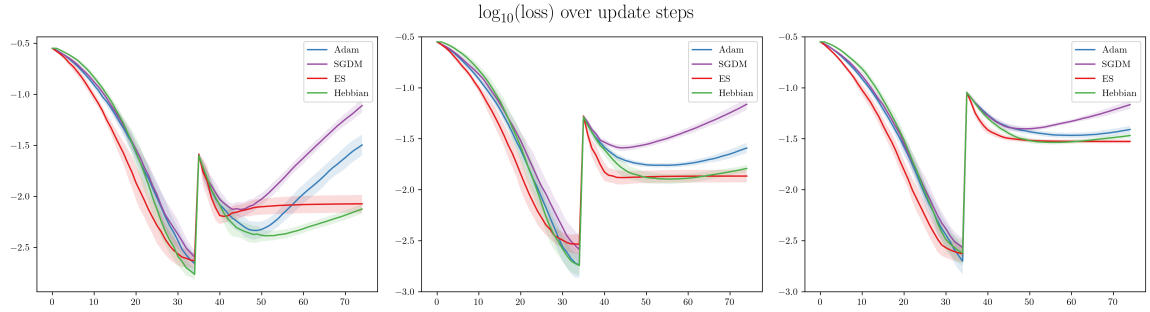
6.1 Models trained to grow

Hyperparameters				
Model	**lr**	**Pop size**	**Using pool**	**Damage**
Adam	0.002	-	No	None
SGDM				
ES	0.005	6		
Hebbian				

Table 4: Hyperparameters for models only trained to grow.

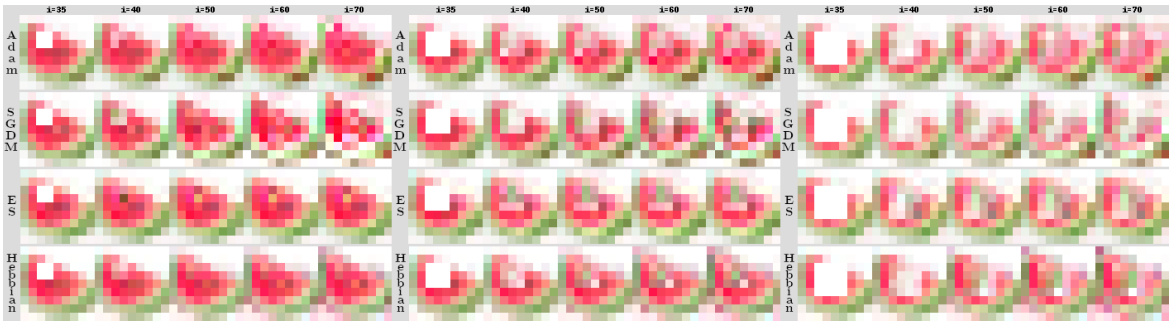
6.1.1 Quadratic erasing

Figure 9 shows how the models regenerate after damage. In the left graph, the damage is done in a 2 x 2 box. In the middle graph, the damage is done in a 3 x 3 box, and in the right a 4 x 4 box. The damage is done at the 35th update step to best evaluate regeneration when the image is nearly fully grown. Figure 10 visualizes the image in the update steps after the damage has been applied.



(a) 2 x 2 (b) 3 x 3 (c) 4 x 4

Figure 9: Loss over 70 update steps with quadratic damage of various sizes applied at update step 35.



(a) 2 x 2 (b) 3 x 3 (c) 4 x 4

Figure 10: Visualization of how the featured models regenerate when exposed to three magnitudes of quadratic damage.

6.1.2 Random erasing

Figure 11 shows how the models regenerate after being damaged with random widths of the damage box, between 1 and 4 pixels on both axes, on random locations in the cell grid, and at random update steps between 10 and 35. The models are run 100 times each.

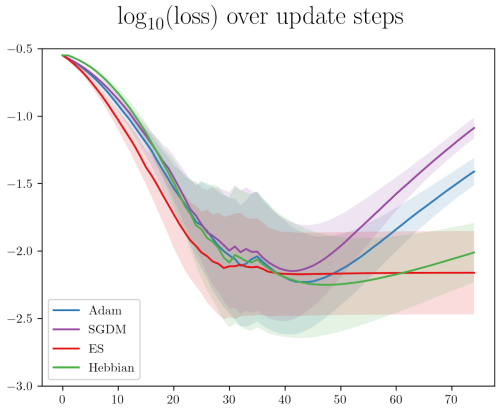
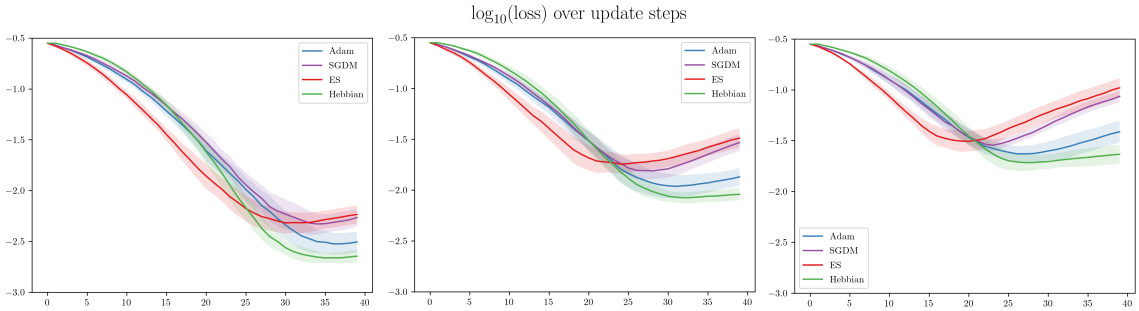


Figure 11: Loss over 100 runs for each model. For every process, the image is damaged at a random update step between the 10th and the 35th. The damage consists of zeroing out a portion of the image with random rectangular size.

6.1.3 Random noise

In the next tests, the cells were exposed to random sampled noise, applied to every living cell at every update step. Figure 12 shows how the models react to noise from $\mathcal{N}(0, 0.01^2)$, $\mathcal{N}(0, 0.03^2)$ and $\mathcal{N}(0, 0.05^2)$.



(a) $\mathcal{N}(0, 0.01^2)$ (b) $\mathcal{N}(0, 0.03^2)$ (c) $\mathcal{N}(0, 0.05^2)$

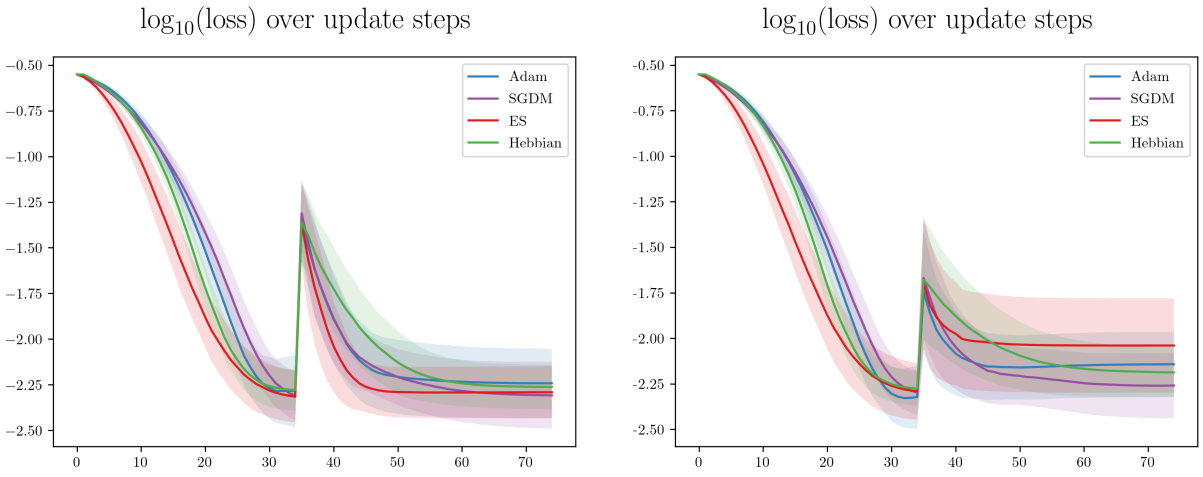
Figure 12: All cells with $\alpha > 0.1$ are disturbed with randomly sampled values from a normal distribution at every update step. The three plots differ in what variance is used in the normal distribution.

6.2 Models trained with random damage to all channels

Hyperparameters					
Model	**lr**	σ	**Pop size**	**Using pool**	**Damage**
Adam	0.002	-		Yes	From pool. Random location and size in the bottom half, on all channels
SGDM					
ES	0.03 (0.01)	0.04 (0.01)	128		
Hebbian	0.2	0.1			

Table 5: Hyperparameters for models trained with random damage to all channels on the lower half. σ and learning rate were both manually changed for ES during training after 13000 generations. The new values are listed in parentheses. SGDM was trained with momentum 0.95.

Figure 13a shows how the models perform when the images are exposed to damage at their lower half. This damage is applied in the same way as during training. Figure 13b shows how the models perform when exposed to damage on their upper half. Each model is run 100 times.



(a) Damaged lower half

(b) Damaged upper half

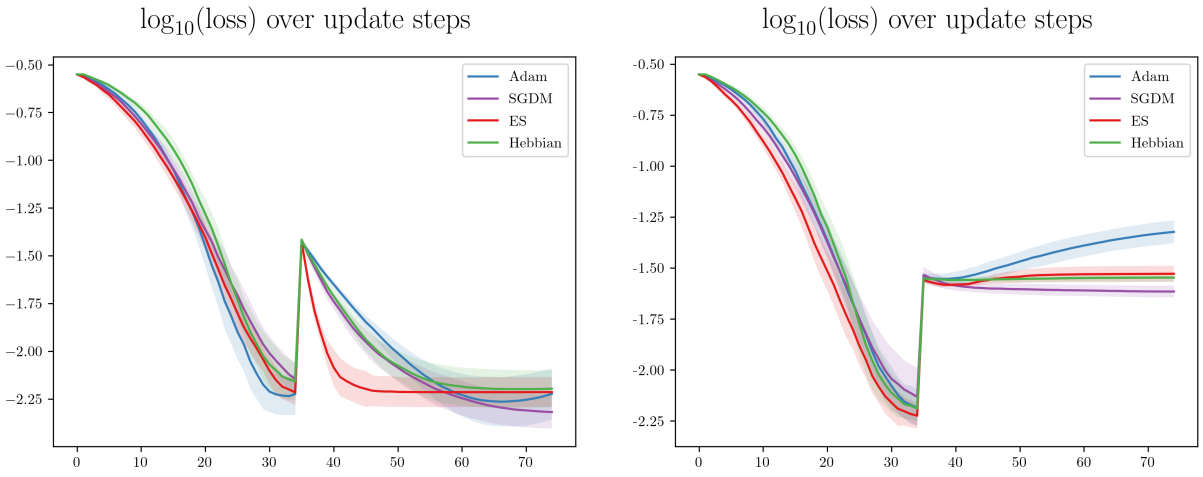
Figure 13: **(a)** All models undergo damage to the lower half of the image at the 35th update step. The damage is applied in the same area as during training. **(b)** All models undergo damage to the upper half of the image at the 35th update step. The damage is in a different area than during training. Each model is run 100 times.

6.3 Models trained with damage to the green channel

Hyperparameters				
Model	**lr**	**Pop size**	**Using pool**	**Damage**
Adam	0.002	-	Yes	From pool. Channel green set to 0
SGDM				
ES	0.003	6		
Hebbian	0.002			

Table 6: Hyperparameters for models trained with damage to the green channel.

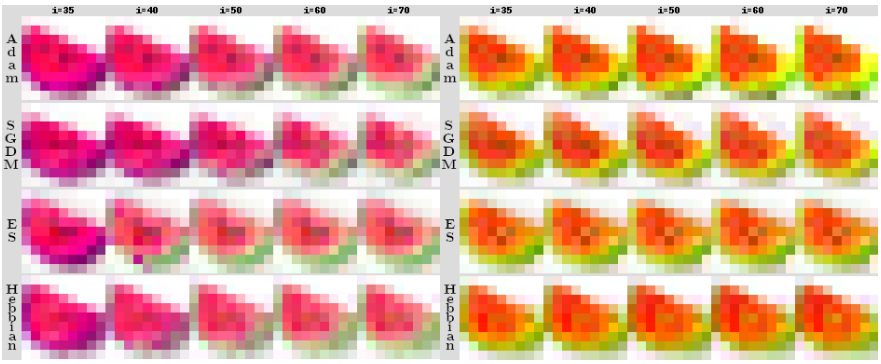
Figure 14a shows how the models regenerate after being damaged at the 35th update step by setting the green channel to zero for all cells. This is the same damage as the models were trained with. The models are all able to recover from the damage applied. They recover back to, or close to, the same loss as before the damage. Notice how ES is quicker at regenerating. Figure 14b shows how the models regenerate after being damaged at the 35th update step by erasing the blue channel for all cells. The models were not trained with damage to the blue channel. Figure 15 visualizes the image in the update steps after the damage has been applied.



(a) Green channels erased

(b) Blue channels erased

Figure 14: **(a)** All cells have their green channel erased at update step 35. The models were exposed to this type of damage during training. **(b)** All cells have their blue channel erased at update step 35. The models were not exposed to this type of damage during training.



(a) Green removed (b) Blue removed

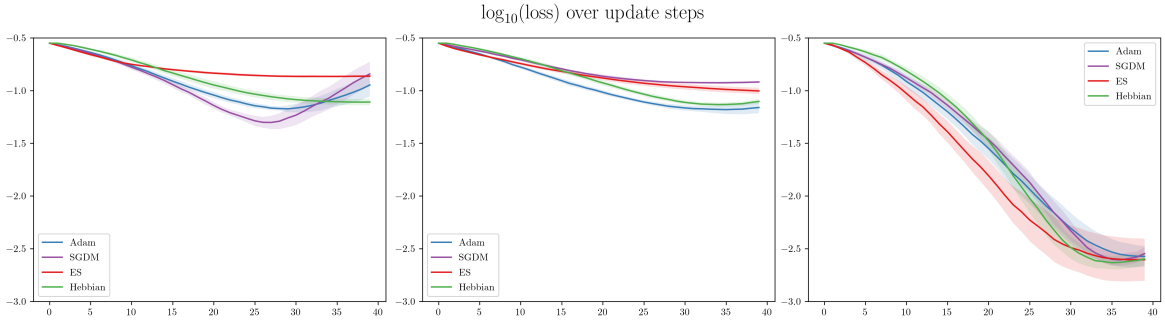
Figure 15: Visualization of how the featured models regenerate after the **(a)** green and **(b)** blue channel is removed.

6.4 Models trained with damage to hidden channels

Hyperparameters				
Model	**lr**	**Pop size**	**Using pool**	**Damage**
Adam	0.002	-	Yes	0.5 probability every generation of disabling the last four channels
SGDM				
ES	0.003	6		
Hebbian				

Table 7: Hyperparameters for models trained with damage to hidden channels

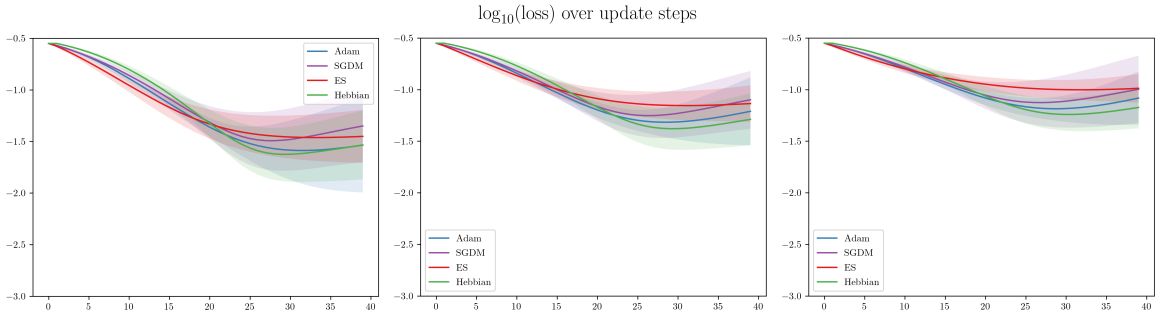
Figure 16 demonstrates how the models regenerate after having their first four, middle four, and last four hidden channels disabled. The models were trained with having their last four channels disabled in 50% of the generations.



(a) (b) (c)

Figure 16: Plot of the losses when four hidden channels are disabled. **(a)** The first four hidden channels are disabled. **(b)** The inner four hidden channels are disabled. **(c)** The last four hidden channels are disabled, same as during training.

Figure 17 shows how the models regenerate after disabling one, two, and three random channels from the first eight hidden channels, who were always present during training. The tests were performed 3000 times on each model to ensure that every possible combination of hidden channels disabled is tested a considerable number of times.

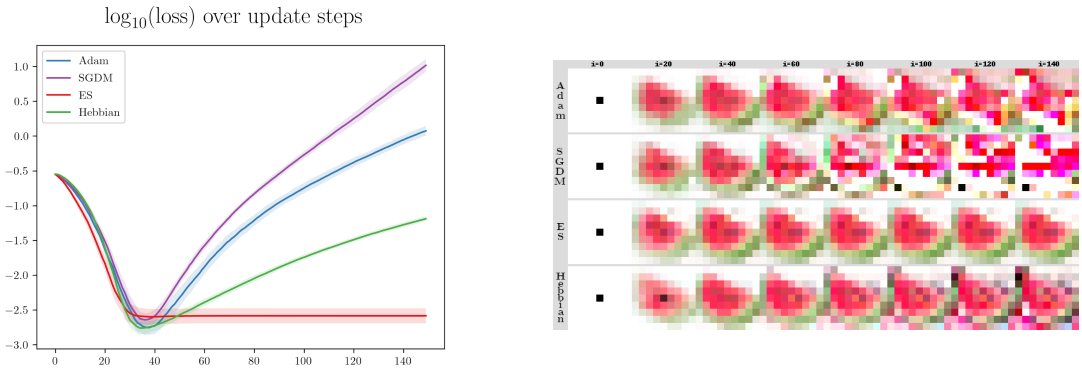


(a) (b) (c)

Figure 17: Randomly chosen hidden channels are disabled, excluding the ones disabled during training. Each model is evaluated 3000 times. **(a)** One random hidden channel is disabled. **(b)** Two random hidden channels are disabled. **(c)** Three random hidden channels are disabled. Hebbian performs slightly better on average in the three tests.

6.5 Consistency of ES over additional update steps

In figure 9, ES keeps a constant loss after regenerating, while the other models' loss increases steadily. Figure 18a shows how ES maintains its loss over 150 update steps. The cells at update step 40 are very similar to the cells at update step 140 using ES, which is not the case for the other methods, visualized in figure 18b.



(a) Loss (b) Consistency

Figure 18: **(a)** The models' loss over 150 update steps with no exposure to any damage. These are the same models as used in chapter 5.1. ES keeps its loss constant through all update steps after reaching its minima. **(b)** Visualization of the image in every 20th update step.

Looking at the change, dx, applied to the cell grid at every update step we see that ES keeps

changes to a minimum after a certain point. dx is calculated as the sum of the absolute values of changes made to the cells. With N cells each having C channels, $dx_{i,c}$ is the change applied to channel c of cell i:

$$dx = \sum_{i=1}^{N} \sum_{c=1}^{C} |dx_{i,c}| \tag{11}$$

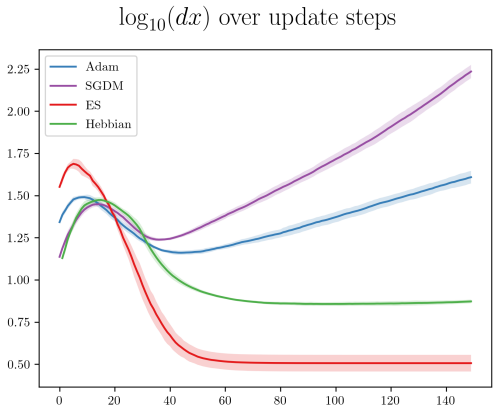
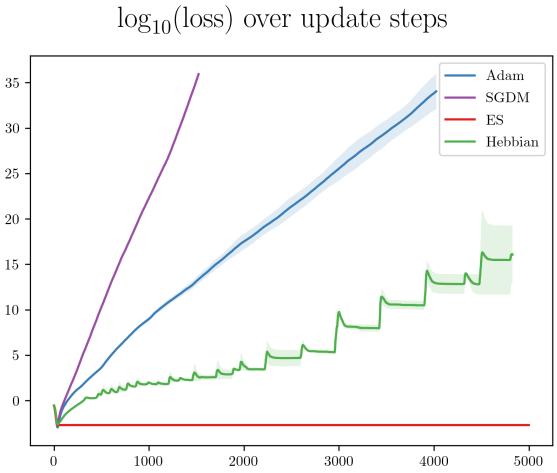
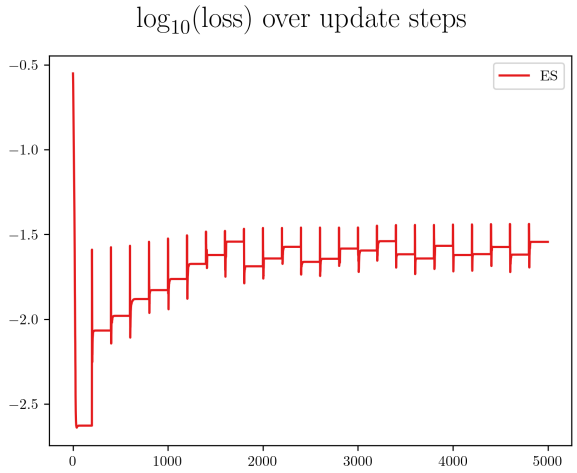


Figure 19: The sum of changes the network applies to the cell grid at every update step.

This property does not disappear over time, shown in figure 20a across 5000 update steps, and in figure 20b with damage every 200th update step.



(a) Loss



(b) ES damaged every 200th step

Figure 20: The same models as in chapter 5.1, run over 5000 update steps. **(a)** ES maintains a consistent loss while the other models' loss increases evenly with time. **(b)** Damage to ES at every 200th update step in a 2 x 2 square as in chapter 5.1. ES maintains stable loss between damages, as seen by the horizontal lines between the spikes.

7 Discussion

ES and Hebbian take a significantly longer time to train than Adam and SGDM. Table 2 shows that ES takes around 40 times longer to reach $\log_{10}$(loss) of -2.5 than Adam. This was expected as backpropagation is usually more efficient on supervised learning tasks, where an exact error can be computed. The gradient descent methods can complete significantly more iterations per second than ES and Hebbian. Some optimization could possibly be done to reduce this difference as we have used heavily optimized machine learning libraries for Adam and SGDM, while the weight and coefficient updates in ES and Hebbian are implemented with Python. Given time the ES methods are able to evolve CA that grows the target image with comparable loss to the gradient descent optimization methods. If hardware resources are not a limit, ES might have the potential to be faster than gradient descent optimization on larger CA, where a large population size is advantageous, due to its ease of parallelization as discussed in chapter 4.2 and demonstrated in table 3. This table shows that when the population size surpasses the number of subprocesses that can be run in parallel on one machine, which in our case is 32, multiple machines can be used to share the workload and reduce training times.

While we have shown that ES and Hebbian can compete with gradient descent optimization given enough time, they have not proven to better regenerate after damage or generalize on data not seen during training. ES and Hebbian do not perform significantly better than Adam or SGDM in any of the proposed experiments.

The experiments in chapter 6.1 show how the models regenerate to roughly the same loss, with few differences between them. Adam and Hebbian perform marginally better than ES and SGDM, as seen in figure 12 with a greater tolerance to noise, but no model shows a significantly higher degree of robustness to perturbations than the others. In chapter 6.2 the models have no problem regenerating from damage in the lower half of the image, which is the damage they were trained with. When damaged in the upper half, the loss after regeneration varies. Damage in the upper right of the image, where there are fewer cells alive, will have less impact on the image than in the upper left, which may be a cause for the greater deviations in the loss in this test. Generally, the models are able to repair damage to some degree, but not as successfully as when damaged in the bottom half. In chapter 6.3 the models are unable to recover the removed blue channel, even when trained to recover the green channel. In figure 14b we observe that the models do not seem to react to the damage, with minor variation between update steps 35 and 70. Even Hebbian with its ability to adjust the network in reaction to the removed blue channel is unable to recognize and repair the damage.

In the experiments in chapter 6, we observe that ES grows and recovers faster than the other models. This is especially noticeable in figure 14a, where ES regenerates in about 10 update steps quicker than the other models. This might be a difference between the methods used, but could also be a result of the long training times required for ES, compared to Adam

and SGDM, giving the model many generations to learn how to quickly regenerate.

In the experiments on damage to sets of hidden channels, in chapter 6.4, Hebbian performs slightly better than the other models, seen in figure 17. However, the difference is minimal, and no model regenerates to an impressive loss. It might have been Hebbian's ability to change its weights after training that gave it a better result, but it is important to note that how the models use their hidden channels will vary, as they are not predefined, and it might be that this evolution of Hebbian was less reliant on specific channels than the other three models.

The effect of the models diversifying their dependence on the hidden channels can also be seen in figure 16 where four predetermined hidden channels are disabled. Taking SGDM as an example, we observe in figure 16a that it is reliant on one or more hidden channels from the first four hidden channels during the late stages of growth. It is possible that another trained instance of SGDM with the same hyperparameters would make use of other hidden channels to fulfill the same purpose. Taking the deviations seen in figure 17a into consideration, it is definitive that all models have some channels they rely less on than others. Such dependencies vary across all models, making it challenging to evaluate how trustworthy these results are.

Najarro and Risi showed promising results using Hebbian meta-learning to develop a model that generalized well when exposed to damage it had not been introduced to during training. In our experiments, however, we did not observe Hebbian to be able to greatly generalize better than its peers when introduced to damage not experienced before. It might be that NCA is not a suitable environment for Hebbian. If the model does not have a sense of target goal or direction, being able to adapt to changes will not help it perform any better than static weights. The locality and noise in CA coupled with sharing the weights across several local environments might not be enough for Hebbian to sufficiently understand the changes that are being made to its environment. An important aspect of CA is that a cell should act only based on the information given by its neighbors. In Hebbian, the shared weights are changed using the post- and pre-synaptic activity of all cells, which contradicts the principle of locality as non-neighbor cells will affect each other's behavior in the next step. We used this approach to ensure that the shared update rules would be in the form of a neural network, as in ES, Adam, and SGDM. An alternative implementation would be to assign each cell its own network and have the Hebbian coefficients indirectly represent the update rules. This solution would have preserved the principle of locality, but the cells would not share the exact same update rules, as their weights would be different. In the experiments by Najarro and Risi they trained the agent on a walking task, where movements are repeated periodically to move the agent forward across several hundred update steps. In our NCA the process is linear, growing from the seed to the target image in 40 update steps or less. This difference might be a reason Hebbian has not shown the same adaptability in our experiments on CA, with fewer recurring patterns to follow and fewer update steps to adjust the network weights.

Why we did not see greater differences between the models in the experiments could be explained as asynchronous CA being a noisy environment that develops robustness, regardless of the methods used. By evolving generations in an irregular number of update steps and unpredictability in when cells update, the network might be forced to learn to be adaptive. The methods are not able to follow any general procedure as they do not know how many times a cell has been updated or at what update steps.

Perhaps the most interesting observation was that of ES's consistency over time, shown in figure 18. While the other methods continue making changes to the cells at every update step, ES keeps changes at a minimum after reaching the loss it was trained to achieve, seen in figure 19. This property can be learned on gradient descent optimization methods by using a sample pool during training, as shown by Mordvintsev et al. [21], but ES exhibits this behavior by default without being trained to be persistent. Our immediate assumption was that the randomness in cell firing and number of update steps was enough for ES to be able to learn to grow fast and then stop changing. However, when training a model on a static amount of updates steps and full cell fire rate, making the growth deterministic, the model behaved the same way. As this setup removed any noise affecting the environment and ES still showed persistence, we do not see this robustness originating from the fundamentals described in chapter 2.5 that make ES robust in noisy environments.

This property may be a consequence of ES learning to grow the image faster than the other models, as seen in figure 7a, which we speculated was caused by longer training times. The more update steps ES has left when the image is already grown, the more it will learn to not change it. In figure 21, we see how ES performs over the first 65000 generations of its training. The red line shows its loss after 40 update steps and the black line after 200 update steps. The persistence is attained approximately after 20000 generations. This is less than 1/20 of the training time used by the model in figure 7a.

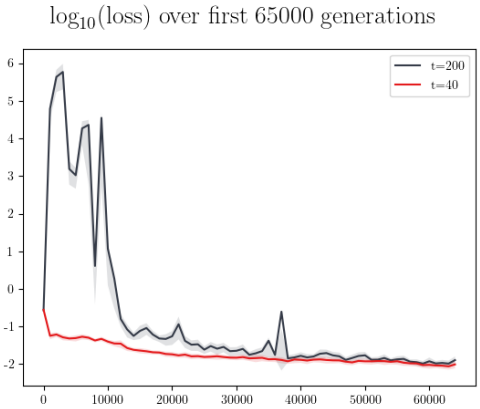
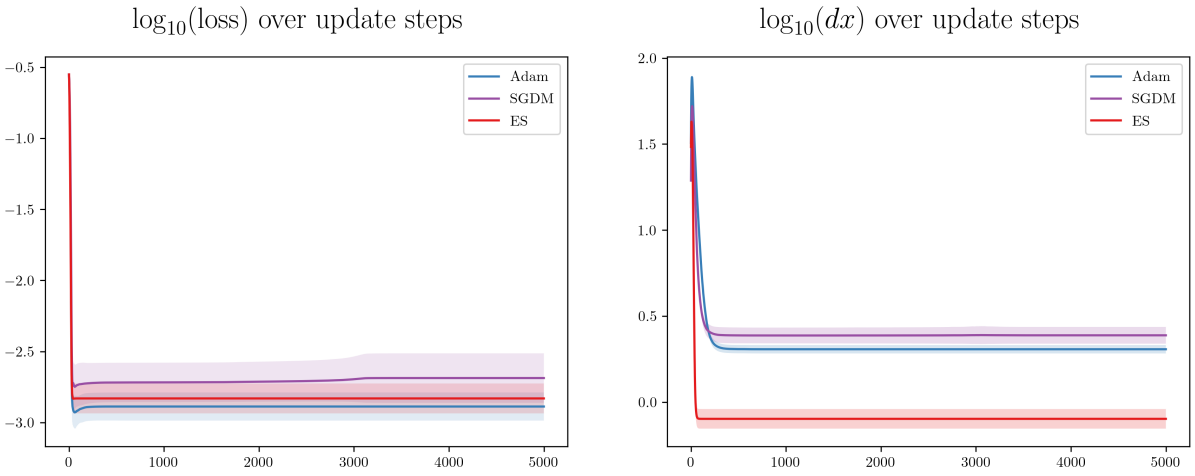


Figure 21: The performance of ES at the first 65000 generations of training. The model is trained to the usual 30 - 40 update steps. The red line shows the loss after 40 update steps and the black line after 200. The checkpoints used are sampled every 1000th generation during training.

The loss seen in figure 21 is still converging. The persistence property is developed at the same pace as the cells learn to form the target image, not after a long time as a way to reinforce the current image as we initially thought. Still, one could argue that 20000 generations are adequate for any of the featured models to gain such abilities. We studied a model trained with Adam for one million iterations and found it not to behave persistently on additional update steps. Another thing to note is that while training Adam and SGDM with a sample pool can be done to keep the loss persistent, they still apply significantly more changes per update step than ES, as seen in figure 22.



(a) (b)

Figure 22: Comparison of **(a)** loss and **(b)** dx over 5000 update steps. Adam and SGDM are trained with a sample pool, while ES is not.

An intuitive explanation of which techniques ES employs to be naturally persistent is that

it exploits the ReLU activation function by producing mainly negative values from the first layer, making most elements in the input to the second layer zero. This leads to the second layer giving an output of mostly zero values, thus not applying any change. We could not find any reasonable explanation as to why ES develops this behavior and not any of the other methods. We do not exclude the possibility that Adam and SGDM could have gained the same ability with the right hyperparameters.

If this type of CA is to be deployed in an environment where communication is cheap, but changes are expensive, e.g. a set of small robots on a limited battery, ES could be considered the preferred method to keep changes at a minimum while still being able to respond to new information. On the other hand, it might be possible to train gradient descent optimization techniques with a penalty in the loss for unnecessary changes, to achieve similar behavior.

8 Conclusion and Future Work

Evolution strategies (ES) and Hebbian meta-learning have both shown to be able to compete with Adam and stochastic gradient descent with momentum (SGDM), evolving neural cellular automata to form a predefined image. However, we have not found the methods to better regenerate or generalize than the gradient descent optimization techniques when trained and tested with various types of damage. There are slight variations in the models' performance in the experiments, but no method stands out conspicuously. Hebbian meta-learning has previously shown to generalize well on data not seen during training, but we did not observe the same properties in our experiments. Cellular automaton (CA) as an environment is different from the environments where Hebbian has been deployed in earlier research, and we believe CA consists of some factors that make it hard for Hebbian to generalize.

The reason we did not obtain greater differences between the models might be a consequence of the small cell grid size used and that the limited scope of tests is not able to fully measure the differences between the methods. We also speculate that asynchronous cellular automaton is an environment that develops robustness by default and that using algorithms that are more robust in other settings does not necessarily develop more robust models in this environment. The most interesting find has been the observation of ES's consistency over time. In our experiments, we have noticed that ES learns to stop making changes to the cells after some update steps and that the cells can keep their final form static. While this consistency can be achieved in Adam and SGDM when trained to keep the loss persistent with time, they still apply far more changes to the cells than ES per update step. ES achieves this behavior by producing mostly negative values in its first layer when the cells are fully grown, which are then zeroed by a ReLU activation function. Why ES evolves this behavior and the other methods do not is unclear. This property should be further researched to understand why this happens and to investigate possible use cases for this persistence.

The experiments conducted in this report are not comprehensive enough to fully explore the differences in generalization and regeneration between evolution strategies and gradient descent optimization. An interesting future work direction is to test the models on larger images. This could also be an opportunity to research the possibility of ES being faster to train compared to Adam, when parallelizing becomes advantageous to be able to train the model with larger population sizes. This will also be the case in future work on CA as reinforcement learning where the error can not be directly calculated, where ES is already competitive with other optimization techniques.

While Hebbian only showed slightly better performance on the experiments tailored to showcase its adaptabilities, meta-learning and non-static weights may still be a compelling direction for future research on CA. It will be interesting to explore more Hebbian update rules, as the ones proposed by Najarro and Risi. At the time of writing this, Pedersen and Risi published a paper showing that reducing Hebbian connections can improve the model's

robustness and adaptabilities [24], which may further separate Hebbian's performance from the other models.

Neural cellular automata exhibit great abilities in regenerating images in general. In this report, we only explored NCA on static images, but NCA are not limited to this and can be trained to generate different images where the subjects are all in the same category, using a generative adversarial network (GAN). We attempted to train a model with ES to generate digits using the MNIST dataset[11]. The result can be seen below in figure 23. This has been studied before by Chen and Wang where they combined variational auto-encoders with NCA [9]. In their article, they used gradient optimization techniques.

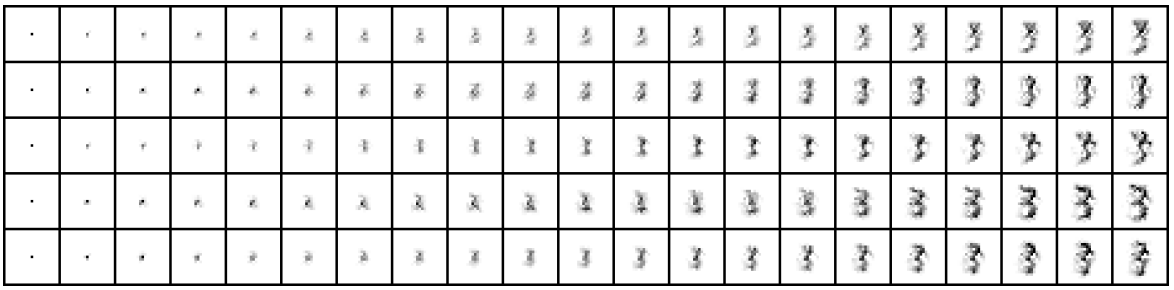


Figure 23: NCA growing five instances of the digit 3 in a GAN environment. The model is trained with ES. The differences between the growing processes are caused by the cells being asynchronously updated.

The results presented in figure 23 are not fairly impressive yet, but we see that NCA with ES shows potential when it comes to generalizing and learning with feedback from a discriminator and we consider this an interesting direction for future work.

[11] yann.lecun.com/exdb/mnist/

9 Broader Impact

The type of cellular automata investigated in this report is far from being deployed to any real-world scenario, but it is relevant to discuss potential long-term ethical consequences early on. As with a lot of similar research on robotics and machine learning, both helpful and destructive applications can be imagined for these types of systems. Advanced self-organizing robots capable of regenerating could have a positive impact in fields like medicine, entertainment, or automation. Cellular automata could also be applied for military purposes like self-repairing field robots or clusters of drones that can self-organize to execute attacks.

References

[1] Andrew Adamatzky. *Collision-Based Computing*. Springer-Verlag London, 2002. ISBN: 978-1-4471-0129-1. DOI: 10.1007/978-1-4471-0129-1.

[2] D. V. Arnold. "Evolution Strategies in Noisy Environments — A Survey of Existing Work". In: *Theoretical Aspects of Evolutionary Computing*. Ed. by Leila Kallel, Bart Naudts, and Alex Rogers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 239–249. ISBN: 978-3-662-04448-3. DOI: 10.1007/978-3-662-04448-3_11. URL: https://doi.org/10.1007/978-3-662-04448-3_11.

[3] T. Back and U. Hammel. "Evolution strategies applied to perturbed objective functions". In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. 1994, 40–45 vol.1. DOI: 10.1109/ICEC.1994.350045.

[4] Thomas Bäck and Frank Hoffmeister. "Extended Selection Mechanisms in Genetic Algorithms". In: Morgan Kaufmann, 1991, pp. 92–99.

[5] N. Barry. *The Tradition of Spontaneous Order*. @Literature of liberty. Inst. for Humane Studies, 1982. URL: https://books.google.no/books?id=rKRrYgEACAAJ.

[6] Hans-Georg Beyer. "Evolutionary algorithms in noisy environments: theoretical issues and guidelines for practice". In: *Computer Methods in Applied Mechanics and Engineering* 186.2 (2000), pp. 239–267. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(99)00386-2. URL: https://www.sciencedirect.com/science/article/pii/S0045782599003862.

[7] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategies - A comprehensive introduction". In: *Natural Computing* 1 (Mar. 2002), pp. 3–52. DOI: 10.1023/A:1015059928466.

[8] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[9] Mingxiang Chen and Zhecheng Wang. *Image Generation With Neural Cellular Automatas*. 2020. arXiv: 2010.04949 [cs.AI].

[10] Dirk V. Arnold and Hans-Georg Beyer. "A Comparison of Evolution Strategies with Other Direct Search Methods in the Presence of Noise". In: *Computational Optimization and Applications* 24 (Jan. 2003), pp. 135–159. DOI: 10.1023/A:1021810301763.

[11] J. Michael Fitzpatrick and John J. Grefenstette. "Genetic algorithms in noisy environments". In: *Machine Learning* 3.2 (Oct. 1988), pp. 101–120. ISSN: 1573-0565. DOI: 10.1007/BF00113893. URL: https://doi.org/10.1007/BF00113893.

[12] John J. Grefenstette and J. Michael Fitzpatrick. "Genetic Search with Approximate Function Evaluation". In: *Proceedings of the 1st International Conference on Genetic Algorithms*. USA: L. Erlbaum Associates Inc., 1985, pp. 112–120. ISBN: 0805804269.

[13] Ulrich Hammel and Thomas Bäck. "Evolution strategies on noisy functions how to improve convergence properties". In: *Parallel Problem Solving from Nature — PPSN III*. Ed. by Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 159–168. ISBN: 978-3-540-49001-2.

[14] Nikolaus Hansen et al. "Comparing Results of 31 Algorithms from the Black-Box Optimization Benchmarking BBOB-2009". In: *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '10. Portland, Oregon, USA: Association for Computing Machinery, 2010, pp. 1689–1696. ISBN: 9781450300735. DOI: 10.1145/1830761.1830790. URL: https://doi.org/10.1145/1830761.1830790.

[15] D. O. Hebb. "The organization of behavior: A Neuropsychological Theory". In: New York: Wiley and Sons, 1949. ISBN: ISBN 9780471367277.

[16] Kazuya Horibe, Kathryn Walker, and Sebastian Risi. *Regenerating Soft Robots through Neural Cellular Automata*. 2021. arXiv: 2102.02579 [cs.NE].

[17] Alois Huning. In: *ARSP: Archiv für Rechts- und Sozialphilosophie / Archives for Philosophy of Law and Social Philosophy* 62.2 (1976), pp. 298–300. ISSN: 00012343. URL: http://www.jstor.org/stable/23679080.

[18] Mauro S. Innocente and Paolo Grasso. "Self-organising swarms of firefighting drones: Harnessing the power of collective intelligence in decentralised multi-robot systems". In: *Journal of Computational Science* 34 (2019), pp. 80–101. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2019.04.009. URL: https://www.sciencedirect.com/science/article/pii/S1877750318310238.

[19] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2017). arXiv: 1412.6980 [cs.LG].

[20] Julian Miller. "Evolving a Self-Repairing, Self-Regulating, French Flag Organism". In: vol. 3102. June 2004, pp. 129–139. ISBN: 978-3-540-22344-3. DOI: 10.1007/978-3-540-24854-5_12.

[21] Alexander Mordvintsev et al. "Growing Neural Cellular Automata". In: *Distill* (2020). https://distill.pub/2020/growing-ca. DOI: 10.23915/distill.00023.

[22] Elias Najarro and Sebastian Risi. *Meta-Learning through Hebbian Plasticity in Random Networks*. 2021. arXiv: 2007.02686 [cs.NE].

[23] John Von Neumann. *Theory of Self-Reproducing Automata*. Urbana, University of Illinois Press, 1966.

[24] Joachim Winther Pedersen and Sebastian Risi. *Evolving and Merging Hebbian Learning Rules: Increasing Generalization by Decreasing the Number of Rules*. 2021. arXiv: 2104.07959 [cs.NE].

[25] Raymond Ros and Nikolaus Hansen. "A Simple Modification in CMA-ES Achieving Linear Time and Space Complexity". In: *Parallel Problem Solving from Nature – PPSN X*. Ed. by Günter Rudolph et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 296–305. ISBN: 978-3-540-87700-4.

[26] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG].

[27] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[28] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: (2017). arXiv: 1703.03864 [stat.ML].

[29] J.L. Schiff. *Cellular Automata: A Discrete View of the World*. Wiley Series in Discrete Mathematics & Optimization. Wiley, 2011. ISBN: 9781118030639. URL: https://books.google.no/books?id=uXJC2C2sRbIC.

[30] Jürgen Schmidhuber. "Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks". In: *Neural Computation* 4.1 (1992), pp. 131–139. DOI: 10.1162/neco.1992.4.1.131.

[31] Andrea Soltoggio et al. "Evolving neuromodulatory topologies for reinforcement learning-like problems". In: *2007 IEEE Congress on Evolutionary Computation*. 2007, pp. 2471–2478. DOI: 10.1109/CEC.2007.4424781.

[32] Shyam Sudhakaran et al. *Growing 3D Artefacts and Functional Machines with Neural Cellular Automata*. 2021. arXiv: 2103.08737 [cs.LG].

[33] Richard S. Sutton. "Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks". In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986.

[34] T.W. Then and Edwin Chong. "Genetic algorithms in noisy environment". In: Sept. 1994, pp. 225–230. ISBN: 0-7803-1990-7. DOI: 10.1109/ISIC.1994.367813.

[35] Sebastian Thrun and Lorien Pratt. "Learning to Learn: Introduction and Overview". In: *Learning to Learn*. Ed. by Sebastian Thrun and Lorien Pratt. Boston, MA: Springer US, 1998, pp. 3–17. ISBN: 978-1-4615-5529-2. DOI: 10.1007/978-1-4615-5529-2_1. URL: https://doi.org/10.1007/978-1-4615-5529-2_1.

[36] Vito Trianni, Stefano Nolfi, and Marco Dorigo. "Evolution, Self-organization and Swarm Robotics". In: *Swarm Intelligence: Introduction and Applications*. Ed. by Christian Blum and Daniel Merkle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 163–191. ISBN: 978-3-540-74089-6. DOI: 10.1007/978-3-540-74089-6_5. URL: https://doi.org/10.1007/978-3-540-74089-6_5.

[37] Darrell Whitley. "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best". In: 89 (June 2000).

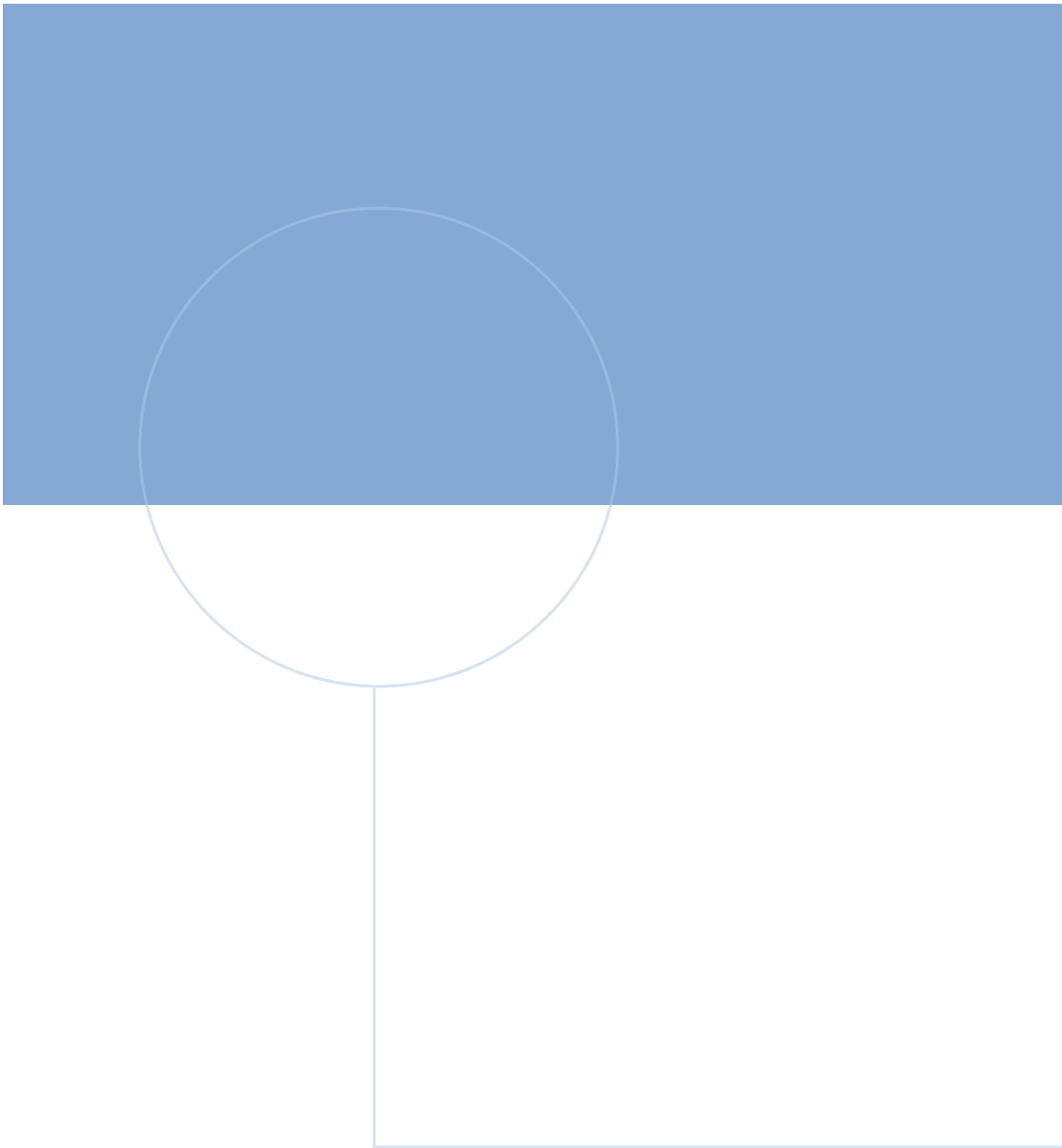
[38] Daan Wierstra et al. "Natural Evolution Strategies". In: (2002). URL: https://people.idsia.ch//~tom/publications/nes.pdf.

[39] Daan Wierstra et al. "Natural evolution strategies". In: *Journal of Machine Learning Research* 15(1) (2014), pp. 949–980.

[40] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: 8 (1992), pp. 229–256. DOI: 10.1007/BF00992696. URL: https://link.springer.com/content/pdf/10.1007/BF00992696.pdf.

[41] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021), pp. 4–24. ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.2978386. URL: http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[42] Pan Zhou et al. *Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning*. 2020. arXiv: 2010.05627 [cs.LG].



NTNU

Kunnskap for en bedre verden