NTNU

Kunnskap for en bedre verden

Department of Computer Science

IDATT2900 - Bachelor Thesis

# Evolutionary strategies as an alternative to backpropagation methods

*Authors:*
Bård Sørensen Hestmark and Tommy Duc Luu

*Supervisor:*
Ole Christian Eidheim

May, 2022

## Abstract

Gradient descent through backpropagation is the most common training method in the machine learning space. This method is however known for having some weaknesses with generalization and noisy environments. Alternatively, evolution strategies (ES) has been shown to be an alternative to the most recent reinforcement learning (RL) optimization methods, but its biggest challenge is training time. Adam, a well known backpropagation method based on gradient descent, is being compared to ES. The model is based on a neural cellular automaton (NCA), which is a self-organizing system of cells controlled by shared update rules. This system can be trained to achieve advanced behaviour, and we want to investigate how Adam and ES behave when training to grow a chosen image. The experiments are based on damaging the NCA in different ways and analyzing their reaction using mainly loss metrics. The results show that ES achieves better persistence than Adam. However, with Adam being specifically trained to persist, it is sometimes hard to observe any impressive regenerative capabilities between Adam and ES.

## Sammendrag

Den mest brukte treningsmetoden i maskinlæringsfeltet er backpropagation. Men denne metoden er kjent for å ha enkelte svakheter med generalisering og støymiljø. Evolutionary Strategies (ES) derimot, har vist seg å kunne være et alternativ til de mer populære treningsmetodene innenfor Reinforcement Learning, men den største utfordringen er den lange treningstiden. Det skal forskes på Adam, en velkjent backpropagation metode basert på gradient descent estimering, sammenlignet med ES. Modellen baserer seg på nevrale cellulære tilstandsmaskiner, som er et selvorganiserende system av celler kontrollert av felles oppdateringsregler. Denne tilstandsmaskinen kan oppnå avanserte egenskaper, og vi ønsker å undersøke hvordan Adam og ES oppfører seg når tilstandsmaskinen trenes til å danne et spesifikt bilde. Eksperimentene baserer seg på å skade disse tilstandsmaskinene på forskjellige måter og analysere resultatene. De viktigste resultatene viser at ES er mer robust enn Adam. Likevel, når modellen med Adam er optimalisert til å gro, er det vanskelig å oppdage signifikante forskjeller på de regenerative egenskapene mellom Adam og ES.

## Preface

We want to thank our supervisor, Ole Christian Eidheim, for providing quick feedback and inputs for our questions, and also for joining our discussions. We would also like to thank William Dalheim for contributing with thoughts and ideas in our discussions. Lastly, we would like to thank the Department of Computer Science at NTNU, specifically Aleksander Tandberg, for providing a powerful 128-core processing server.

Bård S. Hestmark

[illegible]

Bård Sørensen Hestmark

Tommy Duc Luu

# Contents

**List of Figures** vi

**List of Tables** vii

**1 Introduction** 1

**2 Related work** 2

2.1 Evolution strategies . . . . . . . . . . . . . . . . . . . . . . . . . 2

2.2 Cellular automata . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**3 Theory** 3

3.1 Neural Network . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

3.2 Black-Box Optimization . . . . . . . . . . . . . . . . . . . . . . . 3

3.3 Evolutionary Strategies . . . . . . . . . . . . . . . . . . . . . . . 4

3.3.1 Variants of ES . . . . . . . . . . . . . . . . . . . . . . . . 4

3.3.2 OpenAI NES . . . . . . . . . . . . . . . . . . . . . . . . . 4

3.3.3 Parallelizing ES . . . . . . . . . . . . . . . . . . . . . . . 5

3.4 Cellular automata . . . . . . . . . . . . . . . . . . . . . . . . . . 5

3.5 Gradient descent optimization . . . . . . . . . . . . . . . . . . . 5

3.5.1 Gradient descent with momentum . . . . . . . . . . . . . 6

3.5.2 Adam Optimizer . . . . . . . . . . . . . . . . . . . . . . . 6

3.5.3 Optimization through backpropagation . . . . . . . . . . 7

**4 Method** 9

4.1 Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

4.2 Parallelization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

4.3 Model Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

4.3.1 Training . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

4.3.2 Hyperparameters . . . . . . . . . . . . . . . . . . . . . . . 13

**5 Experiments** 15

5.1 Training a model to grow into an image . . . . . . . . . . . . . . 15

5.1.1 Parallelization . . . . . . . . . . . . . . . . . . . . . . . . 15

5.1.2 Different image sizes . . . . . . . . . . . . . . . . . . . . . 15

5.2 Damage to trained models . . . . . . . . . . . . . . . . . . . . . . 16

**6 Results** 17

6.1 Parallelization results . . . . . . . . . . . . . . . . . . . . . . . . 17

6.2 Models trained to grow - without sample pools . . . . . . . . . . 17

6.2.1 Quadratic erasing . . . . . . . . . . . . . . . . . . . . . . . 19

6.2.2 Adversarial attacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
6.3 Models trained to grow - with sample pools . . . . . . . . . . . . . . . . . . 21
6.3.1 Quadratic erasing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
6.3.2 Adversarial attacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**7 Discussion** **24**
7.1 NCA growth over many update steps . . . . . . . . . . . . . . . . . . . . . 24
7.2 Learning to regenerate . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
7.3 ES training time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
7.3.1 ES parallelization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**8 Conclusion** **30**

**9 Future work** **31**

**References** **33**

## List of Figures

1 Example of a neural network . . . 3
2 Model structure . . . 10
3 Seed and target image . . . 12
4 Target images . . . 15
5 Quadratic erasing example . . . 16
6 Adversarial attack example using FGSA with $\epsilon = 0.07$ . . . 16
7 Loss on growth without sample pools . . . 18
8 Damage 9x9 carrot without sample pools . . . 19
9 Damage 15x15 carrot without sample pools . . . 19
10 FGSA during growth without sample pools . . . 20
11 FGSA on step 51 without sample pools . . . 20
12 Loss on growth with sample pools . . . 21
13 Damage 9x9 carrot with sample pools . . . 22
14 Damage 15x15 carrot with sample pools . . . 22
15 FGSA on growth with sample pools . . . 23
16 FGSA on step 51 with sample pools . . . 23
17 ES loss on 40th and 200th update steps during training . . . 24
18 Results on larger canvases . . . 25
19 Results of quadratic erase . . . 26
20 Loss comparisons FGSA during growth . . . 27
21 Results of FGSA during growth . . . 28
22 FGSA on step 51 loss comparisons . . . 28
23 Results of FGSA attack on the 51th step . . . 29

## List of Tables

1 Shared hyperparameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3 Hyperparameters without sample pools . . . . . . . . . . . . . . . . . . . . . . 17
4 Table of multiple growing 15x15 NCA without sample pools . . . . . . . . . . 18
5 Hyperparameters with sample pools . . . . . . . . . . . . . . . . . . . . . . . 21
6 Table of multiple growing 15x15 NCA with sample pools . . . . . . . . . . . 22

# Abbreviations

ANN – Artificial Neural Network

CA – Cellular Automata

ES – Evolution Strategy

FGSA – Fast Gradient Sign Attack

MLP – Multilayer Perceptron

MSE – Mean Squared Error

NCA – Neural Cellular Automaton

NN – Neural Network

ReLU – Rectified Linear Unit

RGB – Red Blue Green

RGBA – Red Blue Green Alpha

RL – Reinforcement Learning

## 1 Introduction

Evolution Strategies (ES) has shown to be a scalable solution to Reinforcement Learning (RL) problems by Salimans et al. [1] In short they managed to show that ES could train neural networks to beat Atari games and MuJoCo-tasks etc. while heavily reducing the training time by increasing CPU-cores. In a report by a prevoius B.Sc. group at NTNU, Dalheim and Jacobsen [2], they discovered that ES showed some favourable qualities over multiple backpropagating algorithms like Adam and Stochastic Gradient Descent Method. Taking inspiration from Darwin's evolution theory and the principles about survival of the fittest, ES seeks to find the best possible solution to a problem by preserving the best mutations over many generations.

In this project we train a Neural Cellular Automaton (NCA) with ES. An NCA is a self-organizing system much like John Conway's "Game of Life" [3], but it uses an Artificial Neural Network (ANN) to set the update rules. As shown by Mordvintsev et al. [4], an NCA trained by backpropagating errors had promising results when specifically trained to stabilize or regenerate. We want to explore ES in an NCA to see if such qualities can be achieved without models being specifically trained for it. Perhaps there are other models that can benefit from these extra qualities gained from ES?

We hypothesize that ES will be able to learn in a generalized manner much like how natural selection manages to find favourable traits. Also from the discoveries in Dalheim and Jacobsen's [2] report, we think that ES will require a great amount of training time and that it will be able to regenerate from unknown forms of damage without necessarily being specifically trained to do so. Therefore, our research question is if ES is a scalable alternative to backpropagation methods based on loss-, speed- and robustness metrics.

# 2 Related work

The related work in this chapter are the main bases used for our research of evolution strategies as an alternative to backpropagation.

## 2.1 Evolution strategies

A variation of ES is Natural Evolution Strategies (NES), and is developed by Wierstra et al. [5] It involves calculating the gradients of a population and adding it to the parent parameters. The gradients help guide the distribution to a better solution using the fitness scores and mutation matrices for estimations. They also implemented shaping of fitnesses by ordering the fitness non-linearly across the population.

Advancements in modern technology has allowed the artificial intelligence space to take advantage of increasingly efficient computing capabilities. This also benefits ES as it can potentially be easier to parallelize in comparison to other optimization techniques. ES only requires gradients that do not need to be shared. This leads to a revisit of ES by Open AI as a "scalable alternative to reinforcement learning" [1]. By using a variation of NES, they showed lower training time and stability for certain MuJoCo-tasks. With this method, they managed to train a 3D human shaped object to walk in 10 minutes by parallelizing the workload across 1440 CPU cores. [1]

## 2.2 Cellular automata

John Conway's Game of Life [3] is an experiment based on cellular automata (CA). Although not the first works on CA, it is certainly one of the most famous ones. The game is based around an initial state set by the player, that from there on evolves into different patterns. These patterns can often be interesting or visually pleasant.

Mordvintsev et al. [4] developed an NCA to generate images of emojis by growing from a single cell. They also present the NCA's ability to recover from damage, however only when being trained for it. In a recent project by Dalheim and Jacobsen [2], they researched regenerative capabilities of an NCA using various training algorithms like ES. Their findings were similar to the ones by Open AI, such as consistency over a large amount of time steps, stability, and better regeneration in terms of loss data over time. Still, there were a few experiments where ES recovered worse than other optimization techniques.

# 3 Theory

This chapter will provide the reader with essential knowledge to understand important ideas in experiments and results discussed later in the paper.

## 3.1 Neural Network

A neural network (NN), in association to computer science, is a network or circuit made up of artificial neurons. Similarities to a biological NN are how wired neurons fire together thus forwarding a signal from the rear to the end as shown in figure 1. This means that in order for a neuron to send a signal it has to receive a signal first. A NN is used to solve complex problems such as classification in images, voice recognition etc. The NN adjusts the influence of each neuron using numerical weights in between the connections. An excitatory link has a positive weight, while inhibitory connections have a negative weight, which is applied to to all inputs before summation. This is also known as a linear combination. Finally, the output's amplitude is controlled by an activation function. For example, could an acceptable output range be between 0 and 1, or -1 and 1. The most standard ANN is a fully connected ANN, or a so called multilayer perceptron (MLP) like in figure 1. Each node has an input consisting of a linear combination of the nodes in the previous layer. [6]

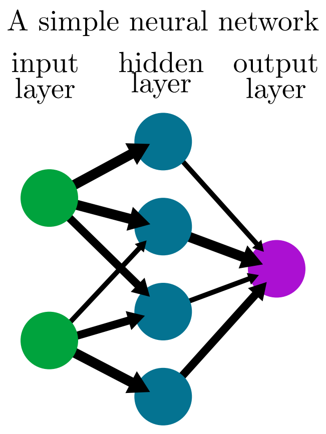


Figure 1: A multilayer perceptron, an example of a neural network [7]

## 3.2 Black-Box Optimization

Black boxes are typically made of numerous points which are connected in complex patterns. It would be difficult for a person to easily interpret such a system, therefore an ANN is an example of a black box. Black box optimization refers to algorithms that utilize millions of data points and correlates specific features in the data set to produce a desired output. Generally this type of optimization is mostly self-directed, and it is difficult to interpret the effect that the optimization has on the output.

## 3.3 Evolutionary Strategies

Evolution strategies is a class of algorithms utilizing black-box optimization to perform heuristic search procedures, inspired by natural evolution. The algorithms work by perturbing a population of parent parameters and evaluating their resulting objective function value known as the fitness score. Subsequently the parent parameters with the highest fitness score are recombined to form children parameters that will represent the parents of the next generation. This procedure is repeated until a best possible fitness score has been reached. Greater fitness score is better. [1]

### 3.3.1 Variants of ES

The differences between algorithms in this class are often in how mutations and recombination are performed and in how populations are represented. [1]

Perhaps the most extensively known form of ES is the covariance matrix adaptation evolution strategy (CMA-ES). Here the population is represented by a full-covariance multivariate Gaussian. This method has proven very successful at problem solving in lower dimension.

Natural Evolution Strategies (NES) are ES inspired black-box optimization algorithms derived from first principles while simultaneously providing state-of-the-art performance. With this the idea is to maintain and iteratively update a search distribution. The fitness is evaluated, and the distribution is adjusted accordingly in the direction of higher expected fitness. [1] [5]

### 3.3.2 OpenAI NES

Below is the pseudocode suggested by Salimans et al. [1] as presented in their article. The algorithm employed belongs to the NES class, and is the one that will mostly be used in onward.

**Algorithm 1** Evolution Strategies, OpenAI

```
1: Input: Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
2: for $t = 0, 1, 2, \ldots$ do
3:     Sample $\epsilon_1, \ldots \epsilon_n \sim \mathcal{N}(0, I)$
4:     Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
5:     Set $\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{i=1}^{n} F_i\epsilon_i$
6: end for
```

The algorithm works by adding normal distributed noise to all parameters. It then evaluates fitnesses, combines them, and calculates an estimate to update the networks parameters.

### 3.3.3 Parallelizing ES

The nature of ES makes it well suited for parallelization. As ES operates in complete episodes and the only thing necessary to communicate between episodes is a single scalar per worker. [1]

**Algorithm 2** Parallelized Evolution Strategies, OpenAI

```
    Input: Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
 2: Initialize: $n$ workers with known random seeds, and initial parameters $\theta_0$
    for $t = 0, 1, 2, \ldots$ do
 4:     for each worker $i = 1, \ldots, n$ do
            Sample $\epsilon_i \sim \mathcal{N}(0, I)$
 6:         Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
        end for
 8:     Send all scalar returns $F_i$ from each worker to every other worker
        for each worker $i = 1, \ldots, n$ do
10:         Reconstruct all perturbations $\epsilon_j$ for $j = 1, \ldots, n$ using known random seeds
            Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
12:     end for
    end for
```

Just like in Algorithm 1 the noise is added to all parameters and fitness is evaluated. These fitnesses are then shared between all workers to be combined and used to update network parameters.

## 3.4 Cellular automata

A cellular automaton defines a system of cells where each cell's state is determined by a set of rules for how a cell should behave based on the neighbouring cell states. A state can for example be a binary state, so that there are two indicators to determine if a cell is "*dead*" or "*alive*". The system is usually ordered in a 2-dimensional grid, and updates to the cell states happens in discrete time steps. All cells are updated simultaneously during an update step, and the resulting cell states after an update step is called a generation. There exist many different CA systems, but they all inhabit the same principles of uniformity, synchronicity, and locality. Meaning that the cells use the same set of rules, simultaneously for all cells, and that the cell's perception is limited to its surroundings. [8]

## 3.5 Gradient descent optimization

Gradient descent is a linear iterative optimization algorithm to find a local minimum of a function $F(\theta)$ where $\theta$ denotes a model's parameters. With repeated steps in the opposite direction of the gradient at the function's current point, the algorithm will reach the local minimum.

For a multi-variable function $F(x)$, if the function is differentiable in the neighborhood of point $\theta$, then $F(x)$ decreases the fastest when $\theta$ is moved in the direction of the negative gradient $-\nabla F(\theta)$.

With a small step size $\alpha \in \mathbb{R}_+$, for

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

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

### 3.5.1 Gradient descent with momentum

The issue with standard gradient descent is that the gradient could shift direction depending on the parameter set, which could diverge the parameters from the local or global minima. The variation with momentum allows it to converge more efficiently and avoid divergence. If the equation (1) is separated into two parts, one of which is the change applied at the timestep $t$ and the other one being the parameter update, the new equations are

$$c_{t+1} = \alpha \nabla F(\theta_t) \tag{2}$$

$$\theta_{t+1} = \theta_t - c_{t+1} \tag{3}$$

To apply momentum, the change $c$ in the previous time step is added to the next one. In addition to this, a hyperparameter $m$ is applied to the change. If $m \in [0, 1]$ and $\alpha \in \mathbb{R}_+$ the recursive formula is

$$c_{t+1} = \alpha \nabla F(\theta_t) + m \cdot c_t \tag{4}$$

$$\theta_{t+1} = \theta_t - c_{t+1} \tag{5}$$

### 3.5.2 Adam Optimizer

Adam, or adaptive moment estimation, is another optimization method only consisting of first order gradients. Making it both efficient and fast to calculate optimal parameters. The algorithm is a fusion of two other optimization methods, namely Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). AdaGrad is effective on sparse gradients, and RMSProp is better optimized for of handling non-stationary settings. Adam gets all these benefits and overcomes both these algorithms according to Kingma and Ba [9]. Given the gradient $g_t$ being defined as $\nabla F(\theta_{t-1})$, which gives the gradient of the function $F$ at a specific set of parameters, and without taking the bias into account, from the AdaGrad method the equations for the moment at $t$ is defined as

$$\beta_1 \in [0, 1\rangle \tag{6}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{7}$$

And with RMSProp, moment is defined as

$$\beta_2 \in [0, 1\rangle \tag{8}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{9}$$

Normally the betas are either 0.9 or 0.999 which will make the moments at $t$ move closer to the moments at $t - 1$. Since the moments are initialized as zeros, the moments will be moved towards zero within the first few iterations. These estimates are known as biased estimates, therefore the next step is to calculate the bias corrected estimates:

$$\hat{m_t} = \frac{m_t}{1 - \beta_1} \tag{10}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_2} \tag{11}$$

These moments are used to calculate the next iteration's parameters in a similar manner to equation (1):

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \tag{12}$$

where $\theta$ is the function's parameters, $\alpha \in \mathbb{R}_+$ and $\epsilon$ is a small floating point number for preventing division by zero.

### 3.5.3 Optimization through backpropagation

Backpropagation is an algorithm for supervised learning of NN. Given a NN structure and an error function which calculates the difference of the NN output and the desired output, the method calculates the gradient of the error function with respect to the NN's weights. By calculating the gradient of the output from the last layer of the NN, one can use the gradient descent algorithm to change the weights of the next to last layer and so on. Eventually the network will have the weights which minimizes the error function for given inputs. [10]

In mathematical terms, one is trying to minimize a given error function. If $\hat{\boldsymbol{y}}$ is the predicted outcome by the NN and $\boldsymbol{d}$ is the desired output, the error function is

$$E = \frac{1}{2}(\hat{\boldsymbol{y}} - \boldsymbol{d})^2 \tag{13}$$

With $\boldsymbol{y_{j-1}}$ as the output from the previous layer, the linear input for j-th layer is

$$\boldsymbol{x_j} = \boldsymbol{y_{j-1}}\boldsymbol{w_j} + \boldsymbol{b_j} \tag{14}$$

where the weights are $\boldsymbol{w_j}$ and $\boldsymbol{b_j}$ are the biases. And with $a(x)$ as the activation function of the input from the previous layer the NN output is

$$\boldsymbol{\hat{y}} = a(\boldsymbol{x_j}) \tag{15}$$

The gradient of the error function (13) with respect to $\boldsymbol{w_j}$ using the chain rule multiple times

$$\frac{\delta \boldsymbol{E}}{\delta \boldsymbol{\hat{y}}} = \boldsymbol{\hat{y}} - \boldsymbol{d} \tag{16}$$

$$\frac{\delta \boldsymbol{E}}{\delta \boldsymbol{x_j}} = \frac{\delta \boldsymbol{E}}{\delta \boldsymbol{\hat{y}}}\frac{\delta \boldsymbol{a}}{\delta \boldsymbol{x_j}} = (\boldsymbol{\hat{y}} - \boldsymbol{d})(a'(\boldsymbol{x_j})) \tag{17}$$

$$\frac{\delta \boldsymbol{E}}{\delta \boldsymbol{w_j}} = \frac{\delta \boldsymbol{E}}{\delta \boldsymbol{\hat{y}}}\frac{\delta \boldsymbol{a}}{\delta \boldsymbol{x_j}}\frac{\delta \boldsymbol{x_j}}{\delta \boldsymbol{w_j}} = (\boldsymbol{\hat{y}} - \boldsymbol{d})(a'(\boldsymbol{x_j}))\boldsymbol{y_{j-1}} \tag{18}$$

Now the last layer's weights can be updated as follows

$$\boldsymbol{w_{t+1}} = \boldsymbol{w_t} - \alpha(\boldsymbol{\hat{y}} - \boldsymbol{d})(a'(\boldsymbol{x_j}))\boldsymbol{y_{j-1}} \tag{19}$$

The gradient of the layer j-1 then equals to

$$\frac{\delta \boldsymbol{E}}{\delta \boldsymbol{y_{j-1}}} = \frac{\delta \boldsymbol{E}}{\delta \boldsymbol{\hat{y}}}\frac{\delta \boldsymbol{a}}{\delta \boldsymbol{x_j}}\frac{\delta \boldsymbol{x_j}}{\widehat{\boldsymbol{y_{j-1}}}} = (\boldsymbol{\hat{y}} - \boldsymbol{d})(a'(\boldsymbol{x_j}))\boldsymbol{w_j} \tag{20}$$

With this the weights of the j-1th layer and so on can be updated. If one were to use multiple nodes per layer, then one would have to take the sum of error gradients $\sum_i(\hat{y}_i - d_i)$ where $i \in \mathbb{R}+$ and use that to adjust the model weights. [10]

# 4 Method

For this chapter, we describe our NN model, training methods, and hyperparameters used in training. The code used in this report can be found at `https://github.com/tdl1304/IDATT2900-45B`.

## 4.1 Algorithms

Based on findings in a similar project by Dalheim and Jacobsen [2], we decided to primarily use the NES variant of the ES algorithm on our NCA. As our control group to compare against, we used the Adam optimizer.

**OpenAI NES**. This algorithm is described in 3.3.2, and references to ES will be made to the OpenAI NES.

**Adam**. This optimizer is described in 3.5.2, and it is one of many gradient descent methods with backpropagation. According to Mordvintsev et al. [4] the optimizer showed good results with a relatively short training time.

## 4.2 Parallelization

ES has the potential to benefit greatly from parallelization. To help us achieve this we were given access to a CPU-server powered by NTNU. This server ran on an AMD EPYC 7742, a 128 core vCPU with 128GB of RAM.

To achieve parallelization in python we want to distribute the workload over all available CPU-cores. Python itself does not support this out of the box. It is hard to parallelize python using threading due to restrictions in CPython. We decided to use the torch.multiprocessing library. The multiprocessing library allows for spawning of sub processes that are allowed to run in parallel. The torch.multiprocessing library acts as a wrapper around multiprocessing, extending it and allowing for tensors sent through a multiprocessing queue to make use of shared memory.

The division of labor was done through assigning each mutant in a population its own process through forking, allowing them to run in parallel, reporting back to the main process on completion.

## 4.3 Model Structure

The NCA we use in this project consists of a feature extraction layer followed by a MLP as shown in figure 2. In the MLP, there is an activation layer using the Rectified Linear Unit

(ReLU) activation function, and is defined as

$$z \in \mathbb{R} \tag{21}$$

$$Relu(z) = max(0, z) \tag{22}$$

The final layer outputs a tensor with the same dimensions as the input tensor. A tensor is simply a multidimensional vector that stores numerical values. The number of channels is set to 16 where the three first channels are the RGB channels and the fourth channel is the $\alpha$ - channel. The 12 other channels are not predetermined and it is up to the NN to define a rule for these. We refer to these 12 channels as hidden channels.

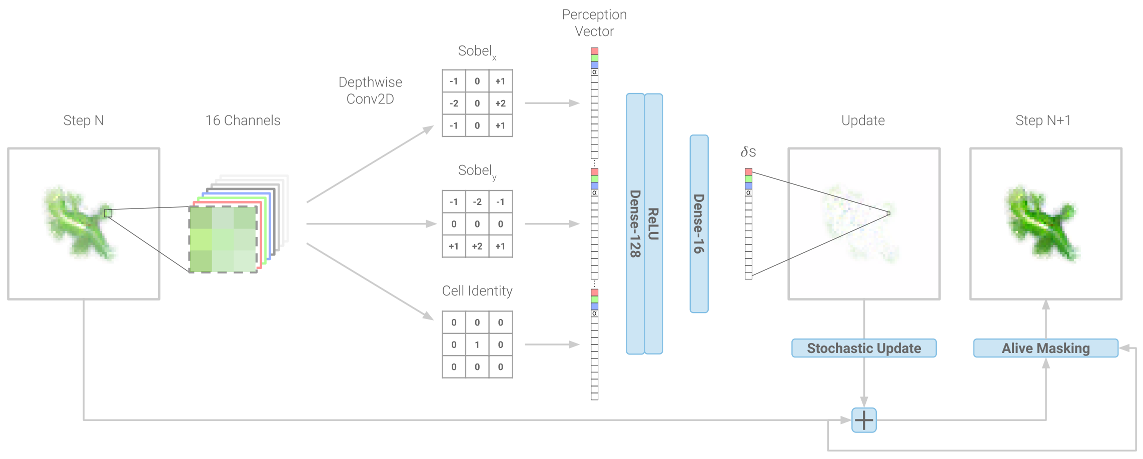


Figure 2: Model structure based on model by Mordvintsev et al. [4]. The input tensor is an RGBA image with additional 12 hidden channels. There is a total of 16 channels for the input. That tensor is fed through three filters for feature extraction, and then into a dense or MLP layer with 128 nodes and a ReLU activation layer. The output consists of 16 nodes because of the 16 initial input channels, and this creates an output with the same dimensions as the input tensor. A stochastic update and alive masking is performed with the input tensor and the output tensor to create the next update step.

The feature extraction layer uses three filters and convolves the filters with the input before feeding it through the network. This means that we utilize a matrix, or a so called "kernel", and for each point in the input image the pointwise product of the corresponding cluster of pixels are summed. The result is written to the corresponding position of each point. Two Sobel filters, one for the x- and y-direction, and an identity filter are used. The Sobel filters are commonly found in computer vision for edge detection, and the identity matrix essentially makes a copy of the input to store the existing cell state information. $I$ is the identity filter and $S$ are the Sobel filters in each direction, and these are shown below.

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

The input and output to the NN is a tensor of shape (batch size, cell grid height, cell grid width, channels). The output is of the same shape because it is added to the input tensor using an update mask. The process is referred to as "alive masking", and it performs updates on the cell state. A living cell is a cell with its $\alpha > 0.1$ or any of its neighbour's $\alpha > 0.1$. Dead cells however, do not update or share information. During an update step there is also an update probability, or fire rate, that determines if a cell should update or not. The purpose of the fire rate, according to Mordvintsev et al. [4], is to achieve asynchronous behaviour. The fire rate is by default set to 0.5. After alive masking, the next generation should behave like it is expanding or growing outwards.

Training with smaller images greatly reduces the difficulty and a need for a big number of neurons. Therefore, unlike the model in figure 2, we have reduced the "Dense-128" layer to a "Dense-32" layer. We also decided to omit the bias since testing done by Dalheim and Jacobsen [2] showed that it was unnecessary for an NCA. We keep these changes for training with both algorithms mentioned in 4.1 to standardize our models. Everything else remains the same as the model shown in figure 2.

By initiating the weights of the "Dense-32" layer to zeros, we achieve a "no action" behaviour at the beginning of training. This means that there will be no updates to the cell states, thus no growth. For ES training, we are also disabling gradients because it is not necessary for the algorithm. If gradients are not disabled then the training will result in a memory overflow error. With Adam optimization however, gradients are needed [9].

### 4.3.1 Training

The self structuring nature of the NCA can be viewed as a Reinforcement Learning (RL) problem. An agent is tasked with grouping up a grid of cells in a specific pattern and colour. By "agent", we are referring to the optimization algorithm. The NCA problem is more supervised than other RL problems though. For this instance, the agent receives a greater reward when the output is more similar to the target image. Therefore, we can use a more supervised approach to calculate the rewards by utilizing the Mean Squared Error (MSE) between the images. Since our goal is to grow an image after a certain amount of time steps, the agent is assessed after an episode.

As a starting point, we use a cell grid that is set to **zero** on all channels except for the middle which is set to **one** on all channels. This is referred to as the seed. The NCA grows outwards from the seed.. An example of a target image and the seed are illustrated in figure 3.

(a) Seed

(b) Target image

Figure 3: Shows a 15x15 cell grid referred to as the seed **(a)** and an example of a 15x15 target image **(b)** which is used for the loss calculation

Early in our ES training, we found that training from a sample pool and fitness shaping improved our model. Fitness shaping is a normalization method that is rank based, which means that the relative positions of the fitnesses affect the overall score of the population fitness [5]. This helps the algorithm by guiding the agent more consistently to an optimal solution. A sample pool involves sampling from a batch and only storing the sample with the best score. These samples were used as a starting point instead of the seed. This technique was also implemented by Mordvintsev et al. [4].

We make use of some of the code published by Dalheim and Jacobsen [2] for our ES training, likewise for the code published by Mordvintsev et al. [4] as the base for the model structure and training with Adam. The model structure for both code bases are identical, they differ only in training method. The loss functions and hyperparameters are also mostly the same.

### 4.3.2 Hyperparameters

Python version 3.10.0 and PyTorch version 1.11.0 [11] were used during training and experimenting with ES and Adam. Tensorboard was useful for logging and visualizing data. The models have the shared hyperparameters as listed in table 1.

| **Hyperparameter** | **Value** |
|---|---|
| In-channels | 16 |
| Grid size | 9x9 and 15x15 |
| Update steps | 30-40 |
| Fire rate | 0.5 |
| Pool size | 1024 |

Table 1: Shared hyperparameters. In-channels refers to the amount of channels for the input tensor, grid size decide how large our target image is, update step refers to how many iterations the NCA have in an episode, fire rate is the probability for a cell to update and pool size is the amount of samples to keep at the same time during training.

Update steps determine how many iterations the cell grid is updated, and it is set to a random number ranging from 30 to 40. We use a sample pool of size 1024. This technique acts as a tool to prevent destroying fully grown images during training. Before the update steps, we retrieve cell grids from the sample pool, at random, depending on our batch size. After the update steps we only commit the best result to the indices it was sampled from. This is due to the randomness of the ES algorithm that are more likely to produce outliers. Additionally, to prevent a sample pool of only dead cells in the cell grids, we only commit the grids that have living cells. A variation of this, used by Mordvintsev et al. [4], is to first swap out the worst result with a seed and then commit the whole batch to the sample pool. But using this variation for ES is more likely to produce a sample pool of outliers. Therefore, we use the variation by Mordvintsev et al. [4] for Adam, and the other one for ES.

Our models trained on a 9x9 grid size to compare against the results of Dalheim and Jacobsen [2] since they also used a 9x9 grid size. In addition to this, we also trained models on a 15x15 grid size, as incremental testing showed that this size gave a good compromise between training time and image quality. Mordvintsev et al. [4] trained on a 40x40 grid size. We could not train on such a large cell grid using ES as we found that the training time increased tremendously at this size.

As mentioned before, the loss is calculated as the MSE between the generated image and the target image. If $\mathbf{x}$ is the generated image, $\mathbf{y}$ is the target image and i iterates through a batch with size n, MSE is defined as

$$\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x_i} - \boldsymbol{y_i})^2 \tag{23}$$

ES use the negative MSE loss to sort a population's score in an ascending order. Given the fitnesses, the population is evenly mapped from -0.5 to 0.5. The shaped fitness F of an instance in the sorted population with size n is given by

$$F = \frac{\text{position}}{n-1} - 0.5, \text{ position} \in \{0, 1, 2 \ldots, n-1\} \tag{24}$$

We also use a normalized shaped fitness to decrease the importance of a few single fitnesses, or outliers. If **A** denotes a one-dimensional array of numerical values, mean($\mathbf{x}$) returns the average of a list and std($\mathbf{x}$) returns the standard deviation, then the normalization is defined as

$$Normalized(\boldsymbol{A}) = \frac{\boldsymbol{A} - mean(\boldsymbol{A})}{std(\boldsymbol{A})} \tag{25}$$

ES use a $\sigma$ of 0.01 to scale the noise added to the model weights. The population size varies in the experiments, but parallelization allows for scalability in terms of increasing population size. We will later discuss the effect that the population size has on the training time in chapter 7.

For ES, we are decaying the learning rate, or step size, by a factor of 0.3 if the mean loss $\leq$ 0.03. This occurs again when the mean loss $\leq$ 0.01 by a factor of 0.5. By contrast, Adam uses the default learning rate of 0.001 provided by PyTorch, which does not decay [11].

# 5 Experiments

This chapter covers all experiments done on the models using ES and Adam with different configurations. These experiments aim to compare the model's growing- and regenerative capabilities, while also aiming for shorter training time.

## 5.1 Training a model to grow into an image

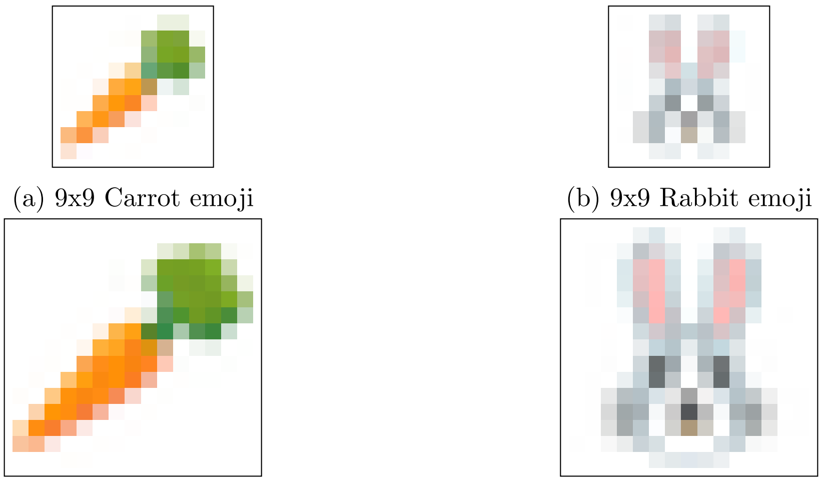
The models used for these experiments are trained with and without sample pools. We tested these model's growing and regenerative capabilities without being trained for self healing properties and stability over time.

### 5.1.1 Parallelization

We tested our implementation of parallelization to see if we could find improvements to the training time on ES.

### 5.1.2 Different image sizes

We trained with 9x9 and 15x15 cell grids for our models. We used a rabbit emoji and a carrot emoji as our target images. The main differences between these images were the amount of leftover white pixels in the canvas and the texture details as shown in figure 4.



(a) 9x9 Carrot emoji

(b) 9x9 Rabbit emoji

(c) 15x15 Carrot emoji

(d) 15x15 Rabbit emoji

Figure 4: Shows a 9x9 and 15x15 cell grid of the target images used in our experiments. Figure **(a)(c)** has more similar texture in terms of colours in one area, while **(b)(d)** has a wider spread of colours in one area.

## 5.2 Damage to trained models

In this section we apply different kinds of damage to trained models to see how they respond. We are testing with:

**Quadratic erasing on 51st update step.** By setting all channels, including the RGBA channels, to zero. We test with three different sizes of damage, 2 x 2-, 3 x 3-, and removing the bottom half of an image. An visualization of the latter is shown in figure 5. The damage is done at the 51st update step. By giving it approximately 10 update steps above the number of update steps during training we can better see how many steps are required to fully regenerate.

(a) Fully grown rabbit (b) Rabbit 15x15 erased lower half

Figure 5: Quadratic erasing example

**Adversarial attacks under and after growth**. A very minor amount of noise is added to the whole cell grid at each update step, and also with a fully grown image at the 51st step. The noise use the error gradient from the backward propagation and adds it to the input image to maximize the loss. This algorithm is called the Fast Gradient Sign Attack (FGSA) and is developped by Goodfellow et al. [12] We adjust the scale of noise by the parameter $\epsilon$.

(a) Fully grown rabbit (b) After adversarial attack

Figure 6: Adversarial attack example using FGSA with $\epsilon = 0.07$

# 6 Results

In this chapter the results from the experiments mentioned in chapter 5 are shown and visualized. The experiments are ran several times. Not all models are included in all experiments as we found some results to be largely similar.

## 6.1 Parallelization results

The training time is measured for training a model with parameters as specified in chapter 4.3.2. The models trained in these runs are 9x9 carrots, all to a fitness goal of -0.003. All the models are trained with ES once, and time is taken for each run.

| **Parallelization results** | | | | |
|---|---|---|---|---|
| **Pop size** | **Gen/s** | **Mut/s** | **Time** | **Tot gen** |
| 6 | 25 | 150 | 36m 7s | 53.9k |
| 10 | 20 | 200 | 43m 36 s | 51.2k |
| 40 | 7 | 280 | 2h 24m 58s | 63.53k |
| 100 | 3 | 300 | 5h 38m 38s | 57.85k |
| 1000 | 0.25 | 250 | - | - |

Table 2: Mut/s is a metric for the number of mutants per second, and is the product of pop size (population size) and gen/s (generations per second). We did not finish training with pop size 1000.

## 6.2 Models trained to grow - without sample pools

Hyperparameters are shown in table 3. In total there are four models trained for 9x9 and 15x15 cell grids for the images shown in figure 4.

| **Hyperparameters** | | | | |
|---|---|---|---|---|
| **Model** | **lr** | **Pop size** | **Batch size** | **Using sample pool** |
| ES | 0.005 | 6 | 1 | No |
| Adam | 0.001 | - | 8 | |

Table 3: Hyperparameters used for training models without activating sample pools

Figure 7 shows the loss over update steps. The models used for this test was trained with sample pools disabled as stated in table 3. During training the models are only evaluated after an episode with 30-40 update steps each episode. Table 4 is showing the growth from the fully trained models for size 15x15. Models trained with 9x9 images are excluded from the table, but several graphs showing loss can be found in figure 7.

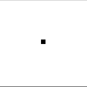
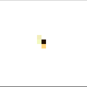
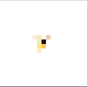
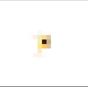
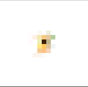
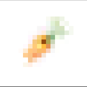
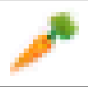
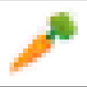
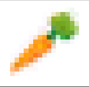
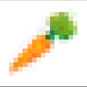
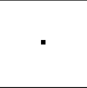
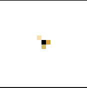
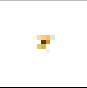
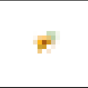
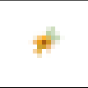
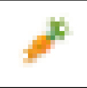
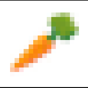
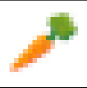
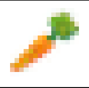
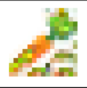
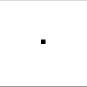
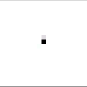
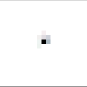
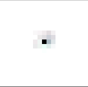
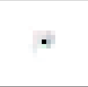
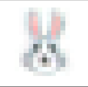
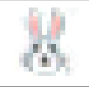
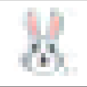
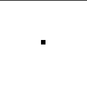
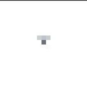
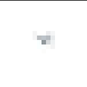
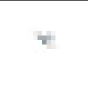
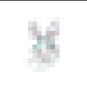
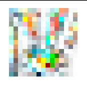
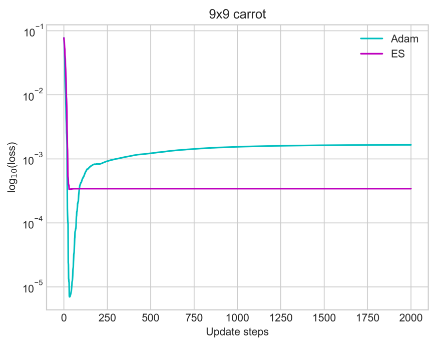
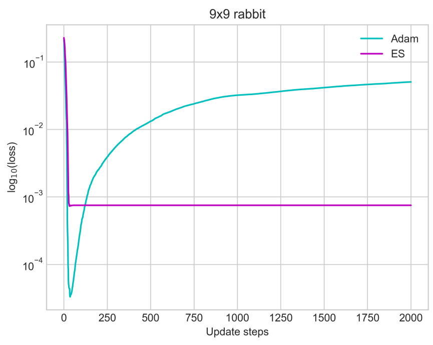
| Iteration | 0 | 1 | 2 | 3 | 4 | 10 | 30 | 40 | 100 | 1000 |
|---|---|---|---|---|---|---|---|---|---|---|
| Carrot ES |  |  |  |  |  |  |  |  |  |  |
| Carrot Adam |  |  |  |  |  |  |  |  |  |  |
| Rabbit ES |  |  |  |  |  | |  | |  |  |
| Rabbit Adam |  |  |  |  | |  | | | |  |

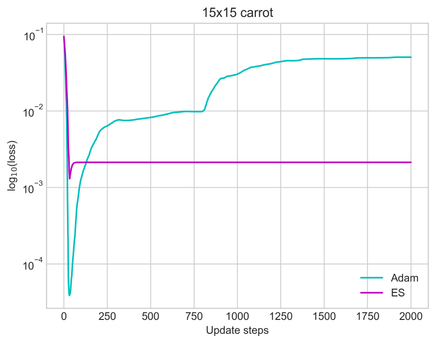
Table 4: Table of multiple growing 15x15 NCA. The horisontal axis are showing the growth stages with a number specifying the update step. The vertical axis orders the NCA into target images and training methods.



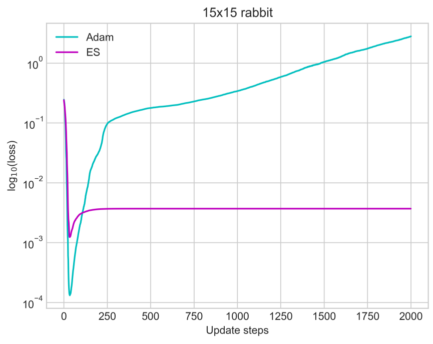
(a) Growing a 9x9 carrot emoji



(b) Growing a 9x9 rabbit emoji



(c) Growing a 15x15 carrot emoji

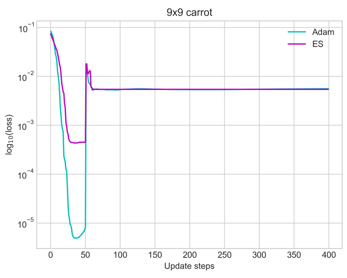


(d) Growing a 15x15 rabbit emoji

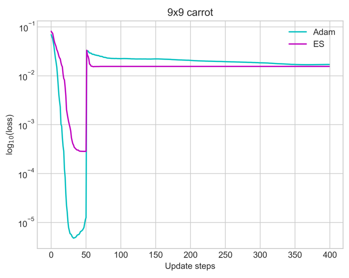
Figure 7: Shows loss graph over 2000 update steps. For each image size and type. The loss is measured as log of MSE from the growing NCA to the actual target image.

### 6.2.1 Quadratic erasing

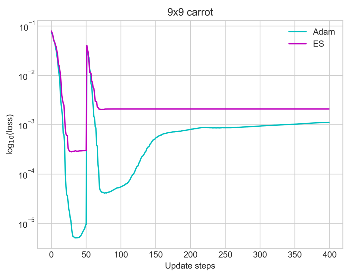
Loss before and after quadratic erasing done at the 51st update step. The graphs in the figures show an increasing amount of damage done to the models, starting with removing a 2x2 box in the middle of the image, then 3x3 in the middle and then removing the entire bottom half in the right graph.



(a) 2x2

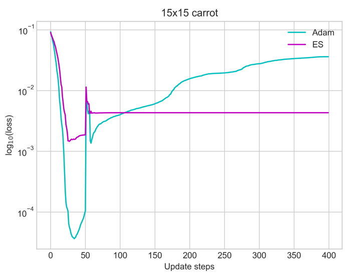


(b) 3x3

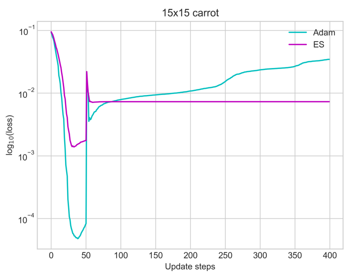


(c) Bottom half

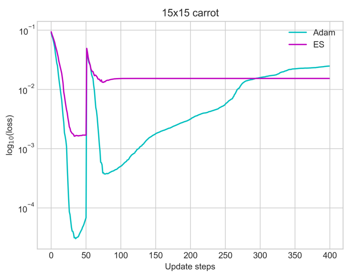
Figure 8: Loss graph after damage over 400 update steps for 9x9 carrot



(a) 2x2



(b) 3x3

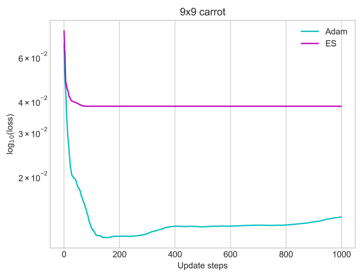


(c) Bottom half

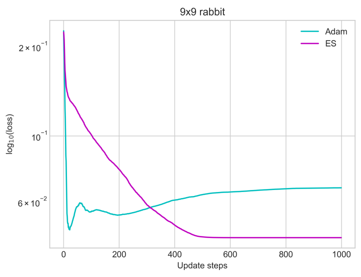
Figure 9: Loss graph after damage over 400 update steps for 15x15 carrot

### 6.2.2 Adversarial attacks

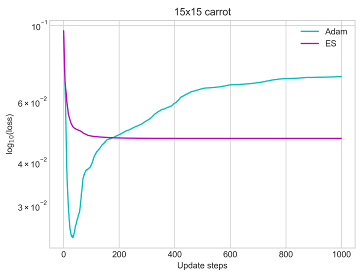
Different values for $\epsilon$ was tested. With an $\epsilon$ larger than 0.007, which is a default value, it cannot grow at all. Therefore, we induce the NCA with a very small noise dosage so that the model is able to stabilize. We used a value of $\epsilon = 7 \cdot 10^{-7}$ for every update step till update step 40. This is shown in figure 10. The other test is with the FGSA on the 51st update step, and these results are visualized in figure 11.



(a) 9x9 carrot emoji

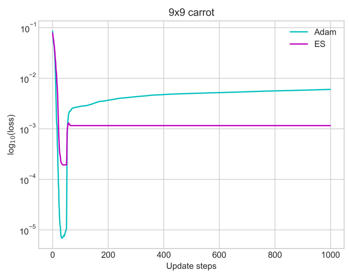


(b) 9x9 rabbit emoji

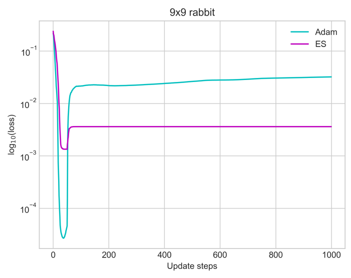


(c) 15x15 carrot emoji

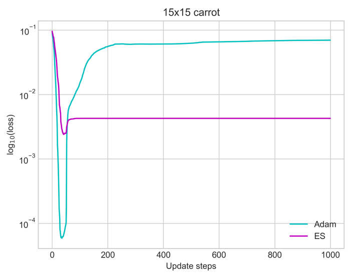
Figure 10: Shows loss graph over 1000 update steps. FGSA with $\epsilon = 7 \cdot 10^{-7}$ is applied on each update step until update step 40.



(a) 9x9 carrot emoji



(b) 9x9 rabbit emoji



(c) 15x15 carrot emoji

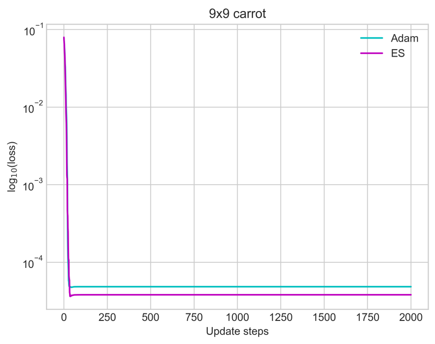
Figure 11: Shows loss graph over 1000 update steps. FGSA with $\epsilon = 0.007$ is applied on update step 51.

## 6.3 Models trained to grow - with sample pools

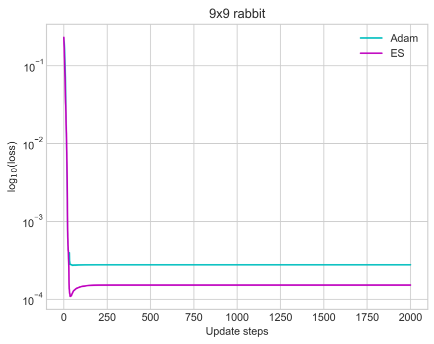
| Hyperparameters | | | | |
|---|---|---|---|---|
| **Model** | **lr** | **Pop size** | **Batch size** | **Using sample pool** |
| ES | 0.005 | 40 | 1 | Yes |
| Adam | 0.001 | - | 8 | |

Table 5: Hyperparameters

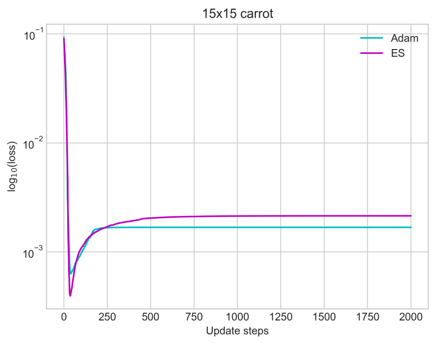
The following graphs and tables shows the growth of each of the four models. The only difference between the models trained in chapter 6.2 is the inclusion of sample pools. The results are shown below.



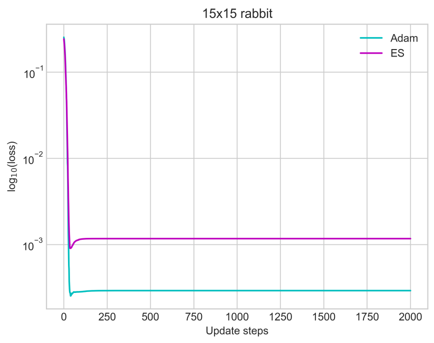
(a) Growing a 9x9 carrot emoji



(b) Growing a 9x9 rabbit emoji



(c) Growing a 15x15 carrot emoji



(d) Growing a 15x15 rabbit emoji

Figure 12: Shows loss graph over 2000 update steps, but compared to figure 7, the models are instead trained with sample pools. In graph **(c)**, ES has increasing loss, but stabilizes after roughly 500 update steps.

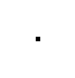
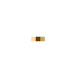
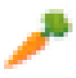
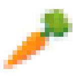
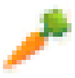
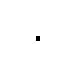
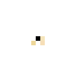
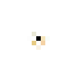
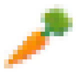
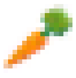
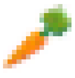
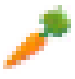
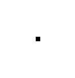
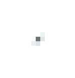
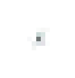
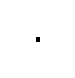
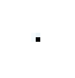
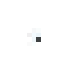
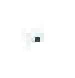
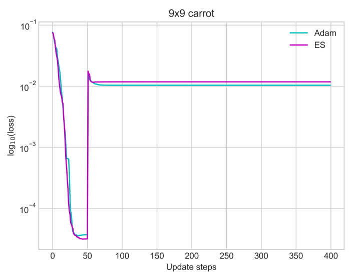
| Iteration | 0 | 1 | 2 | 3 | 4 | 10 | 30 | 40 | 100 | 1000 |
|---|---|---|---|---|---|---|---|---|---|---|
| Carrot ES |  |  | | | | |  |  |  | |
| Carrot Adam |  |  |  | | | |  |  |  |  |
| Rabbit ES |  |  |  | | | | | | | |
| Rabbit Adam |  |  |  |  | | | | | | |

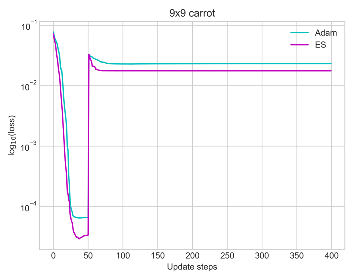
Table 6: Table of multiple growing 15x15 NCA

### 6.3.1 Quadratic erasing

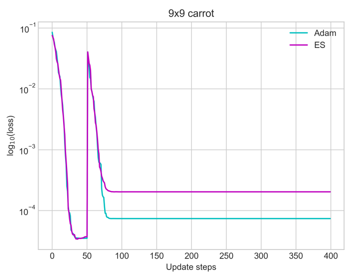
Loss before and after quadratic erasing done at the 51st update step. The graphs in the figures show an increasing amount of damage done to the models, starting with removing a 2x2 box in the middle of the image, then 3x3 in the middle and then removing the entire bottom half in the right graph.



(a) 2x2

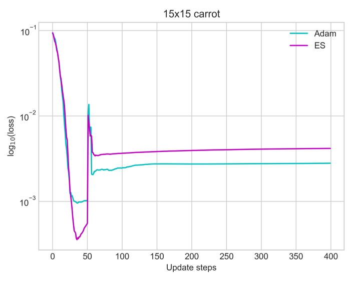


(b) 3x3

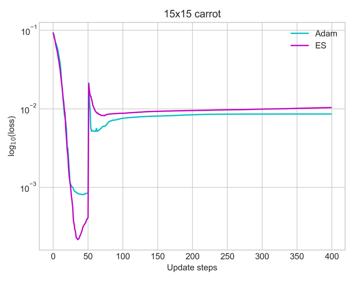


(c) Bottom half

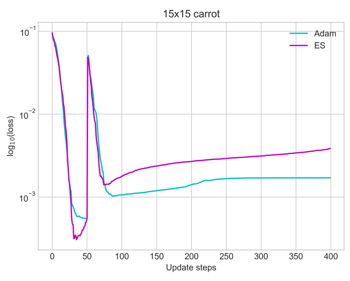
Figure 13: Loss graph after damage over 400 update steps for 9x9 carrot



(a) 2x2



(b) 3x3

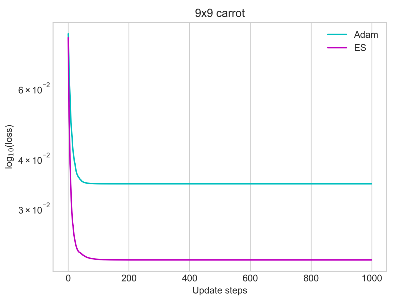


(c) Bottom half

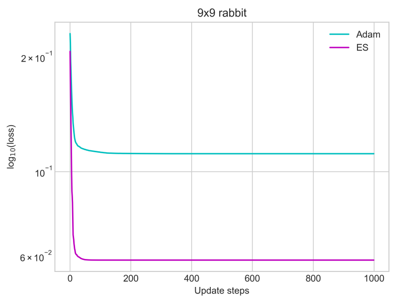
Figure 14: Loss graph after damage over 400 update steps for 15x15 carrot

### 6.3.2 Adversarial attacks

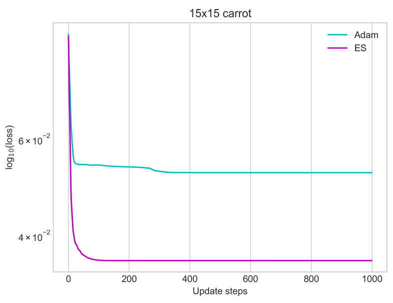
We used an $\epsilon = 7 \cdot 10^{-7}$ to be able to better compare results to the models trained without sample pools. The loss is recorded after noise induction in each update step. For the damaging after growth, it is done with the same method as in chapter 6.2.2.



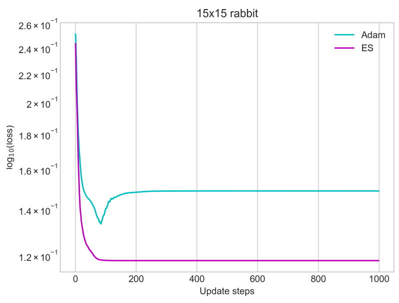
(a) 9x9 carrot emoji



(b) 9x9 rabbit emoji

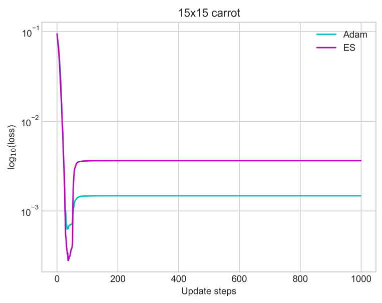


(c) 15x15 carrot emoji

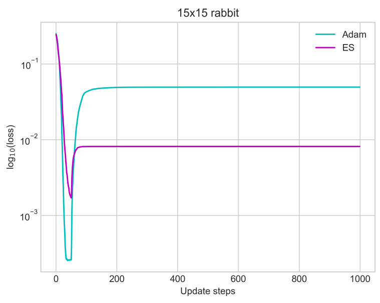


(d) 15x15 rabbit emoji

Figure 15: Shows loss graph over 1000 update steps. FGSA $\epsilon = 7 \cdot 10^{-7}$ is applied on each update step until update step 40.



(a) 15x15 carrot emoji



(b) 15x15 rabbit emoji

Figure 16: Shows loss graph over 1000 update steps. FGSA $\epsilon = 0.07$ is applied on update step 51.

# 7 Discussion

In this chapter we discuss the results and findings from chapter 6 related to the thesis question presented in chapter 1.

## 7.1 NCA growth over many update steps

As mentioned in chapter 4, all models are trained with update steps between 30 and 40. This means that the NCA is only trained to grow to the target image at max 40 update steps. In figure 17 below, the loss for the 200th update step is also sinking after many training iterations although it only being evaluated at 30-40 steps. The difference in loss for the 40th and 200th updates step is converging to 0. Referring to table 4, the models trained without sample pools retained the integrity of the target images for the first 100 steps. However, after many iterations, we found that the loss was rapidly increasing for the Adam trained models. The fast increase in loss may be due to the complexity of large scale images, and also due to the rabbit emoji's perhaps more complex colour patterns. Albeit these challenges with NCA, the ES trained models manage to retain the image characteristics even at 1000 update steps. To support this further, we can also observe that the loss graphs for ES on all the models in figure 7 show a constant loss after the first initial update steps. The fact that ES models can learn to stop growing without being trained to do so is also confirmed by Dalheim and Jacobsen [2] in their research. They also theorize that this behaviour could be the result of passing the fully grown image to the first layer which then will pass negative values to the ReLu functions so that the fully grown image is updated with a zeroed grid. Therefore, this leads to no change after the growing phase.

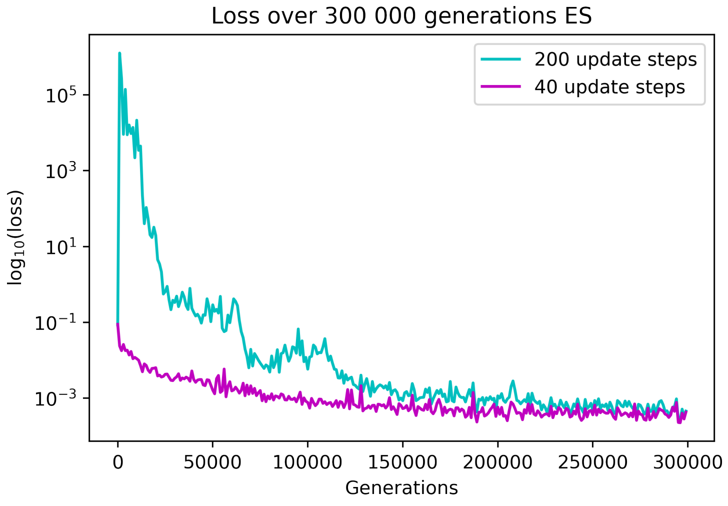
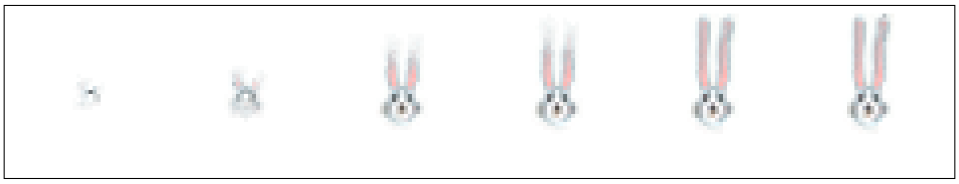


Figure 17: Shows loss after 40 and 200 update steps for an ES model trained for 300 000 iterations/generations. During training the model was evaluated at 30-40 update steps, without sample pool.

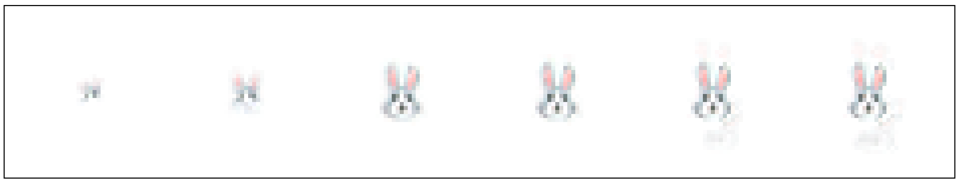
Training the NCA using sample pools greatly improved the stability of Adam trained models beyond 40 update steps. This technique seems to lead the models to stop growing out of control, something that ES managed to achieve without sample pools. However, using sample pools on ES seemed to grant consistency during training. Due to the nature of ES, it can randomly mutate its parameters to something that will leave the grid with an undesired state. During our testing, we found that learning rates greater than 0.005 often lead to such problems. Incrementally, we also optimized ES using for example fitness shaping and normalization to prevent this behaviour. It is also possible that it was a pure coincidence with an unknown bug in our program since we had many iterations of ES during development.

In the report by Dalheim and Jacobsen [2] all their models were trained to reach a loss lower than or equal to 0.0033. With this target loss, the Adam model did not show any signs of stability over many update steps. When training with the Adam method with an even lower target loss, it was able to achieve the same results as ES in terms of stability over many update steps. In the model by Mordvintsev et al. [4], they used Adam and also reached stability using sample pools. Hence Adam, while using sample pools, seems to be able to compete with the ES method's stability when growing.

During out testing we we decided to check how the models would behave in environments not seen during training. Our models were trained on canvases as large as their own size, that is, a 9x9 model would be trained on a 9x9 canvas. We tried to examine the models on grids larger than what they were initially trained for, and discovered some interesting results. We saw that the Adam model was not able to retain its shape on larger canvases, while ES generally maintained it although there were some small artifacts. We also tested on smaller canvases, but here both models gave the same results.



(a)



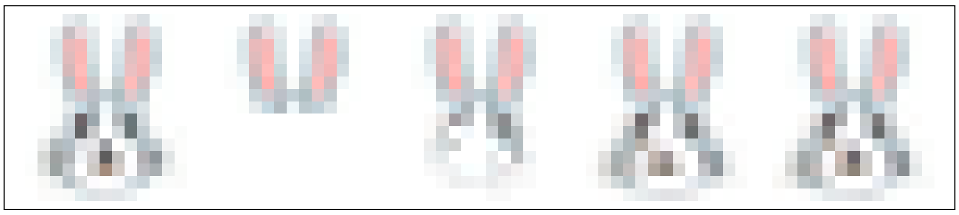
(b)

Figure 18: The figures shows the update steps: {4, 10, 30, 40, 100, 1000}. Figure **(a)** shows a model trained with Adam using sample pools for a 15x15 rabbit emoji, and figure **(b)** is using a ES trained model without sample pools. The models were used with a 39x39 grid which was larger than the initial training size.

## 7.2 Learning to regenerate

Mordvintsev et al.'s [4] NCA models showed no sign of regenerative capabilities when only trained to grow, and this is equivalent to training the NCA with the Adam optimizer without sample pools. They also trained an NCA for persistence using sample pools, which they found to have some regenerative capabilities even at bigger grid sizes like 40x40. Referring to experiment 3 in their paper, the CA was damaged during training to become more robust. The outcome was a highly durable model, but after many update steps there were signs of random artifacts occurring in the CA. Even though, ES is only trained on smaller images, one can observe a constant loss in figure 7 and figure 12. This signals stability when undisturbed during growth. As our Adam models trained without sample pools are unstable, it is purposeless to look for regenerative capabilities in those models. Therefore, we only compare ES with Adam models trained with sample pools.

In our experiments we found that ES regardless of it being trained with sample pools or not, showed to have the same regenerative qualities. In figure 9 and 14, the loss on ES is quickly increasing after being damaged, then slightly recovers and stabilizes. This seem to be true for all the damage experiments therefore, sample pools do not seem to affect the results for ES. In this case Adam has an overall lower loss than ES when being subjected to quadratic erasing as shown in the results in chapter 6.3.1. Figure 19 illustrates the differences of regeneration for the models. The model trained with Adam is regenerating better in the central area than ES, but despite this, both models seem to retain the shape and key traits of the rabbit emoji.


(a)

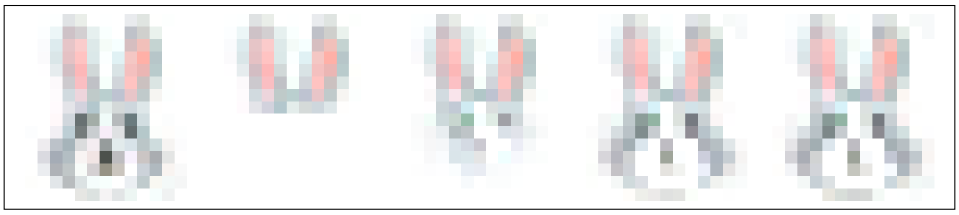
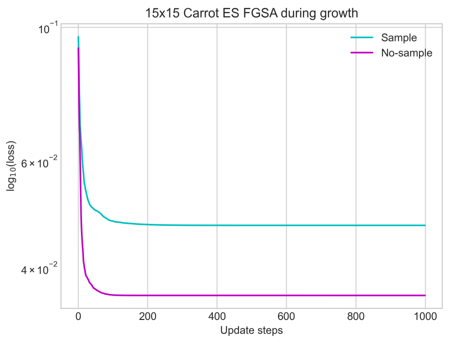
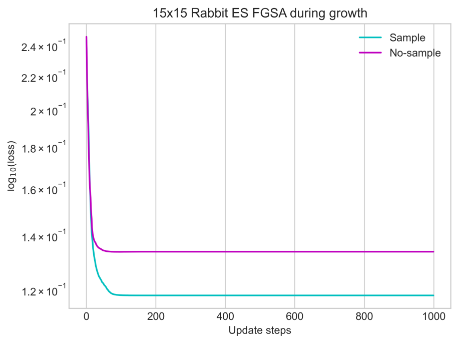

(b)

Figure 19: The figures shows the update steps: {50, 51, 60, 80, 100}. At update step 51, the bottom half of the image is removed. Figure **(a)** shows quadratic erase on a model trained with Adam using sample pools for a 15x15 rabbit emoji, and figure **(b)** is using a ES trained model without sample pools.

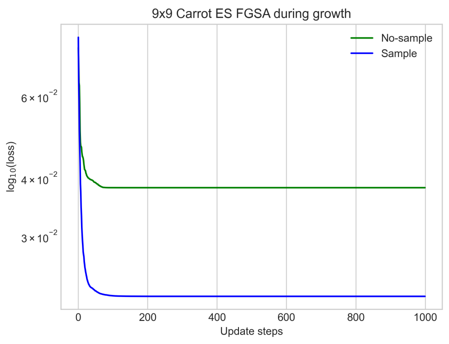
When applying the FGSA during growth there were unexpected results. As mentioned earlier, whether ES is trained with or without sample pools, it should not affect the overall results when excluding natural randomness within the model structure. The different results shown in figure 20 could be caused by how differently the ES models were trained and at which loss they ended their training at. However, we could observe that ES had a lower loss during growth with adversarial attacks than for the Adam trained models as shown in figure 15.



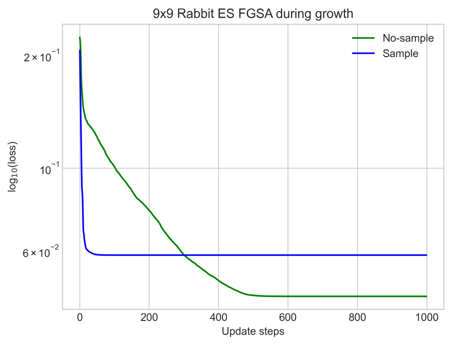
(a)



(b)



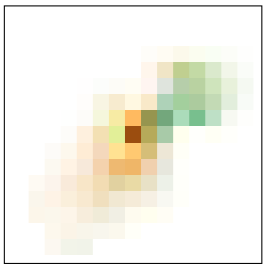
(c)



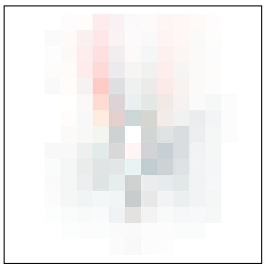
(d)

Figure 20: Figure **(a)** is showing the loss during adversarial attacks in the growth stages for an ES model with and without sample pool for a 15x15 carrot emoji, and **(b)** is showing the same for a rabbit emoji. **(c)** and **(d)** are showing the same, but for the size 9x9.

Despite ES having better results than Adam in this experiment, adversarial attacks during growth, the images that were generated were not that impressive. The 100th update step of both NCA are shown in figure 21, and we decided to showcase these models as they had the best results from the graphs shown above in figure 20. Still, it is hard to decipher what the generated image is supposed to resemble.



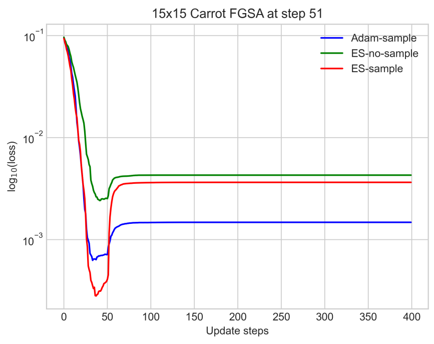
(a) Carrot 15x15 FGSA during growth



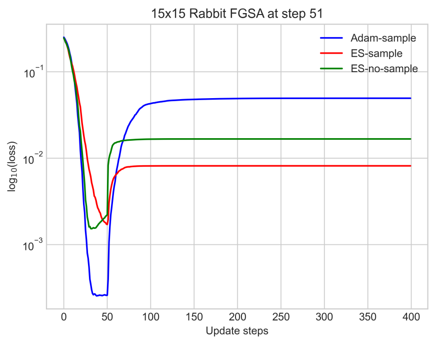
(b) Rabbit 15x15 FGSA during growth

Figure 21: Figure **(a)** shows the 100th update step of an ES trained model without sample pools for a 15x15 carrot, and figure **(b)** shows the 100th step for a 15x15 rabbit trained ES model with sample pools. Both NCA are subjected to adversarial attacks during growth. FGSA with $\epsilon = 7 \cdot 10^{-7}$ is used to damage the NCA.

The results from the adversarial attack after growth is put together into one graph for each emoji type in figure 22. By only observing the results in the figures, it seems like Adam is performing best on the carrot emoji, while ES is doing better on the rabbit emoji. For both graphs, ES is again performing similarly even with the difference in sample pools. Both Adam and ES losses are constant after being subjected to FGSA at update step 51. Unlike how the models behaved with quadratic erasing, there does not appear to be any regeneration happening. This is observed by the loss instantly decreasing after receiving damage at step 51 as in figure 14, which does not occur here.



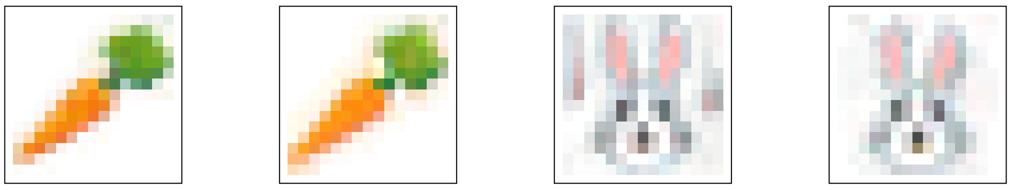
(a)



(b)

Figure 22: Shows loss comparisons for models trained with Adam with no sample pools, and ES with- and without sample pools over 400 update steps. FGSA $\epsilon = 0.07$ is applied on update step 51 for all models. Figure **(a)** is displaying the loss comparisons for the 15x15 carrot emoji models, and figure **(b)** is showing it for the 15x15 rabbit emoji.

At the 100th update step as shown in figure 23, even when subjected to a strong adversarial attack of $\epsilon = 0.07$, the models shows robustness. Although the ES trained model for the 15x15 carrot emoji had a greater loss than the Adam method, when viewing the images side by side, the only difference are the artifacts around the carrot itself. However, both generated images can be classified as carrot emojis. For the rabbit emojis, it seems like the Adam model is cloning itself, while the ES model manages to keep its shape. The difference between the loss on the rabbit- is greater than the carrot emojis, which shows that the ES method appear to be more robust than Adam.



(a) 15x15 Adam (b) 15x15 ES (c) 15x15 Adam (d) 15x15 ES

Figure 23: The figures show the 100th update step after an adversarial attack at step 51. The models are trained as 15x15 NCA with either Adam or ES, and all models were trained with sample pools.

## 7.3 ES training time

It quickly becomes clear that training time for ES is a major challenge, and that in this case ES are not able to compete with the likes of Adam. Although, the implementation of ES used in this report is largely written in non-optimized Python, and the libraries used for Adam are much more optimized. It is possible that given as much optimization, ES could be more competitive in this regard. Given enough time ES is able to evolve models that reach results that are comparable to what is produced by Adam, and in some cases seems to develop favourable traits. Given traits that are favourable enough, one could argue that the extra training time is worth it. Also, given enough computing power on models that are able to benefit from such, ES might also be able to compete on the time aspect as well.

### 7.3.1 ES parallelization

In our testing with parallelization we did not see any clear advantages or significantly reduced training times. We think that the models trained during this project may simply be to small to gain any practical benefit from parallelization. Also taking into account that our implementation of parallelization, might not be optimal. And that with the choice of Python as our programming language, while it is very viable for proof of concept implementations, it might not be the best choice for implementations where efficiency is key.

## 8 Conclusion

ES appear to have a natural ability to adapt to different environments and has shown to compete well against backpropagation methods like Adam. However, based on Mordvintsev et al's [4] model, we have found both methods to have issues with regeneration when testing for variations of adversarial attacks and quadratic erasing. Depending on the target image and the image size, we have found both methods to perform better than the other. In the core element of each optimizer, without using sample pools, ES has proven to be more general and robust than Adam. An example of this behaviour is how ES learns how to stop growing after the initial growing stage.

The main drawback of ES is without a doubt its training time. During training, we experienced the most complex model, a 15x15 rabbit emoji NCA to take up to two days to finish training with a comparable loss to an Adam trained model for 10 minutes on a GPU. Although, training with only Adam resulted in a less robust model, when applying specific techniques it could be able to have the robustness of ES while also being able to scale to larger images in a shorter training time. The techniques could involve sample pools, damage during training as Mordvintsev et al. [4] did, and possibly add negative rewards when the model does unnecessary changes.

With only 2 main types of experiments, we do not deem this research sufficient to explore the differences of generalization and regeneration between ES and Adam. However, by experimenting with relatively large scale images, and with better optimization and parallelization, then ES could become a worthy contender to backpropagation methods. It has already been shown that ES could compete in RL problems, so with the inclusion of the aforementioned optimization techniques the ES method could become a beneficial asset to the artificial intelligence space.

## 9 Future work

To improve upon this research, we would have tested the differences with ES and Adam, or other training methods, in more ways. A random thing we happened to find was the test with growing on a larger grid as explored in figure 18. Perhaps there are more experiments like this one, that could provide several interesting differences.

Another approach to CA, could be with the GAN or the generative adversarial network, which explores classifier models and their ability to distinguish between altered images and the actual image. Instead of using altered images, one could use CA generated images and in that way improve both the classification model and the NCA. That way the classification becomes more precise and the NCA improves on mimicry.

It may also be an interesting idea to use smaller models to grow bigger ones. With the smaller models having the same basic characteristics of the bigger model, similar to how RL problems within games have a checkpoint when the player has progressed to a certain point. ES and NCA could also take advantage of this. By starting with a small cell grid size, the model could incrementally increase the grid size after each training session has reached a certain loss threshold. Since the model should be able to grow into the fundamental shape with the same texture and colours, we think it should be able to learn faster than starting from a seed.

## Broader impact

Real world use cases for the technologies discussed such as cellular automata and an improved form of evolution strategies, though far in the future, could be interesting and many. Of course one needs to account for research like this potentially being used in a negative sense, like for military purposes, and review the ethics surrounding this. But otherwise we imagine that implementations could involve generating false images to train classification models for more efficient training. Perhaps medicinal research, aiding fields like folding in chemistry. Lastly, maybe teach programmable cells, either biological or in form of a swarm to organize them into specific shapes or clusters, be it lab grown meat, body parts, or structures.

## References

[1] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017. https://arxiv.org/abs/1703.03864.

[2] Dalheim William & Jacobsen Jonas B. Regeneration and generalization of cellular automata through evolution strategies. *NTNU Open*, pages 1–53, May 2021. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2778039.

[3] Carter Bays. *Introduction to Cellular Automata and Conway's Game of Life*, pages 1–7. 01 2010.

[4] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 2020. https://distill.pub/2020/growing-ca.

[5] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, and Jürgen Schmidhuber. Natural evolution strategies, 2011. https://arxiv.org/abs/1106.4487.

[6] Hopfield J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, page 2554–2558, 1982. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC346238/.

[7] Wikipedia. Neural network, March 2022. https://en.wikipedia.org/wiki/Neural_network.

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

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. https://arxiv.org/abs/1412.6980.

[10] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. URL: https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox,

and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[12] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2014. `https://arxiv.org/abs/1412.6572`.