Volume XXIV 2021
ISSUE no.1
MBNA Publishing House Constanta 2021

Scientific Bulletin of Naval Academy

SBNA PAPER • OPEN ACCESS

# Self-repairing mechanical components using artificial Intelligence

To cite this article: Mihail Iulian PLEȘA and Maria-Carmen PLEȘA, Scientific Bulletin of Naval Academy, Vol. XXIV 2021, pg.17-28.

Submitted: 24.02.2021
Revised: 15.06.2021
Accepted: 22.07.2021

Available online at www.anmb.ro

**ISSN: 2392-8956; ISSN-L: 1454-864X**

doi: 10.21279/1454-864X-21-I1-002

# Self-repairing mechanical components using artificial intelligence

**Mihail Iulian Pleșa and Maria-Carmen Pleșa**

Military Technical Academy "Ferdinand I"
Technical College "Costin D. Nenițescu" Pitești

E-Mail: mihai.plesa@mta.ro

**Abstract**. In this paper, we study the applicability of artificial intelligence for designing mechanical components that can repair themselves. We use the Cellular Automata (CA) model implemented as a Convolution Neural Network (CNN) to simulate the automatic growth and repair of a mechanical component from a small seed. Concretely, we start with an empty 2D grid of cells. Using the CNN, the cells will learn to self-organize into the image of a mechanical component. We simulate the damage to the component by deleting some parts of the imagine and show how they are automatically regenerated.

## 1. Introduction

For decades, humans have tried to create robots that can repair themselves. They are useful in situations where the presence of a human person is not possible. Consider for example the recent mission of NASA: Mars 2020. The mission consists of sending a rover to the planet Mars. No people have been sent to Mars to monitor the mission. What happens if a part of the robot is damaged? Currently, repairing a damaged part is based on others that are still working and remote assistance. Nevertheless, there are situations in which the robot is far too damaged to be repaired without direct human intervention. This was the case of the Opportunity rover.

This kind of behavior is already presented in living organisms. Living cells can self-organize into tissues with a well-defined role. Also, they possess the property of self-repairing: in the case that the tissue is damaged the cells can automatically regenerate the affected part.

One solution to our problem of designing self-repairing robots is to try to mimic nature. This path was pursued by research done in the field of cellular automata (CA). CA models are based on the idea that a grid of artificial cells can learn to update their states so that they can organize themselves into a well-defined shape with a well-defined purpose. The artificial cells can be programmable nanobots if we make a physical experiment or pixel in an image if we just simulate the behavior of the CA model.

At the moment, CA models learn to self-organize through a deep neural network (DNN). If we work in a simulated environment, the DNN is most of the time a convolutional neural network (CNN). The rapid increase of computational power has allowed the training of larger CNNs. For this reason, the research done in the field of CA has attracted much attention lately.

In this paper, we simulate the self-growing and the self-repairing behaviors of a CA model which aims to simulate mechanical components that can repair themselves. The paper is structured as follows: in section 2 we present the CA model, section 3 describes related work and our contributions, in section

4 we described a step-by-step tutorial on how to implement a CA model using a CNN in Python, section 5 studies several experiments on model performance and section 6 is left for the conclusions.

## 2. The model

In the first part of this section, we give a short introduction to convolutional neural networks (CNN). We present the general architecture of a CNN and some basic examples of convolutional and max-pooling operations.

In the second part of the section, we present the cellular automata model and what role CNNs play in building it. We present the model from a mathematical perspective without giving up the intuition behind it.

### *2.1. Convolutional neural networks*

A convolutional neural network is a type of artificial neural network specialized for computer vision tasks. Figure 1 shows the architecture of a CNN [1].

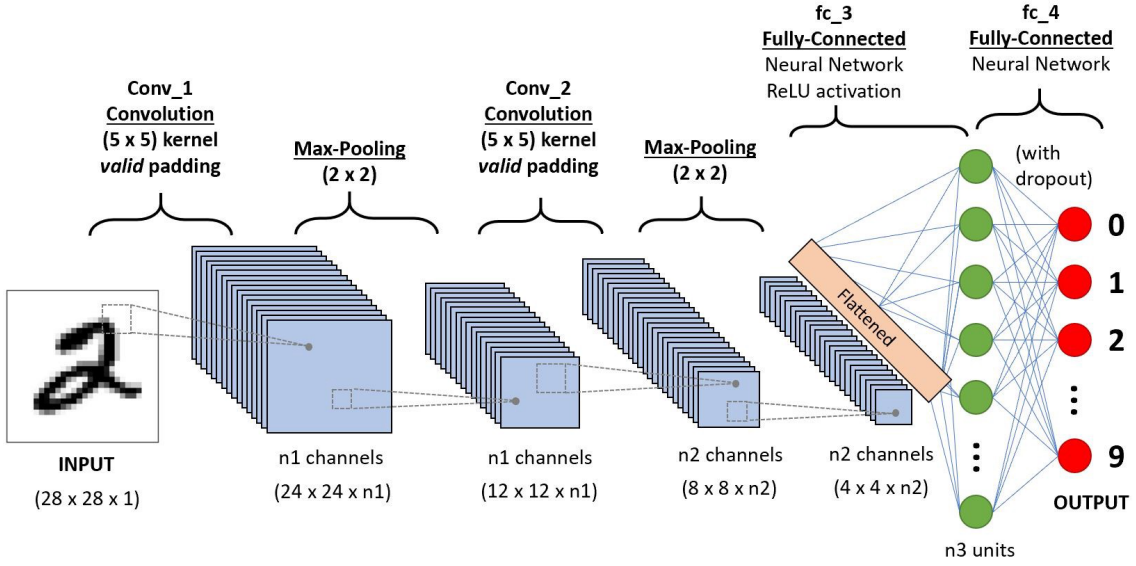


*Figure 1: CNN architecture*

Unlike a classical neural network, a CNN doesn't use the input imagine directly. The input is first filtered by several kernels. Each kernel produces a new filtered imagine. For example, some filters may extract features about the shape, others may extract features about the colors, etc. The core idea of a CNN is the convolution kernel. The input image is convoluted with some filters of fixed dimension. Figure 2 shows an example of convolution between a 3x3 input and a 2x2 filter [2].

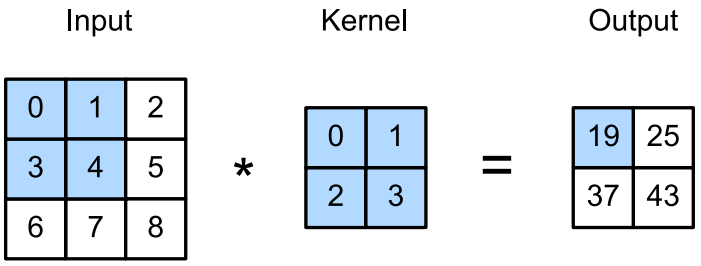


*Figure 2: Example of convolution*

The parameters of each kernel are learned by the network in multiple iterations using the backpropagation algorithm. As can be seen in Figure 1, there can be multiple convolutional layers. Between two such layers, a max-pooling layer is inserted. The role of this kind of layer is to reduce dimensionality by removing low-impact terms. The intuition behind this strategy relies on the fact that any neural network is based on dot products so there is no point in keeping terms with low values. Figure 3 shows an example of $2x2$ max-pooling between an input of $4x4$ [3].

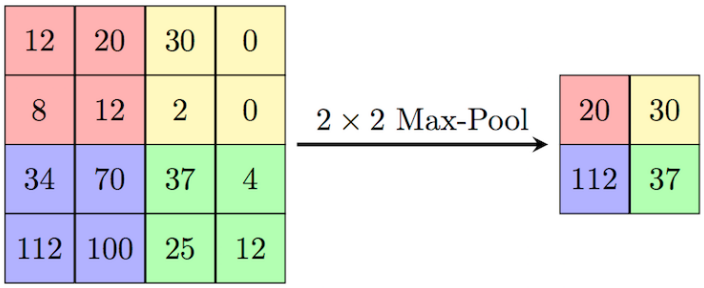


*Figure 3: Max-pooling example*

### *2.2. The cellular automata model*

The model used in this paper is called cellular automata (CA). This model is inspired by the self-organization behavior of the living cells. Consider for example the process of morphogenesis in which a simple bank of cells can organize itself in a well-defined form. This is possible simply by the fact that each cell can exchange information with its immediate neighbors. Besides the fact that the cells can organize themselves into a well-defined form, this shape is also a persistent one. After the process of self-organization took place, if any damage disrupts the shape, they can self-repair.

In this section, we present the mathematical formulation of the cellular automata model. In the real world, a cell has many physical and chemical properties. We model these proprieties of a cell by an array of $k$ elements. We call this array the state of the cell. A bank of cells is modeled by a grid of dimensions $nxm$. Each element of the grid is a cell i.e., an array of $k$ elements. Thus, a bank of cells is modeled by a matrix of dimensions $nxmxk$.

The process of self-organization is modeled by several iterations. At each iteration, the states of all cells are updated simultaneously according to the same update rule. The update rule consists of the addition between the current state of the cells and $k$ variables produced by a function $f$ applied over a neighborhood of dimension $pxp$ of that cell. The function $f$ models the exchange of information between a cell and its neighbors. More formally, let$c_{i,j}$ be the state of the cell that is on row $i$ and column $j$ of the grid:

$$c_{i,j} = \left(c_{i,j}^{1}, c_{i,j}^{2}, \dots, c_{i,j}^{k}\right) \tag{1}$$

The process of self-organization can be expressed by the following algorithm:

1. For each cell, a neighborhood of dimension $pxp$ is considered.
2. A function $f$ is applied over the states of the cells from the neighborhood. This function produces $k$ variables: $u_{i,j} = \left(u_{i,j}^1, u_{i,j}^2, \dots, u_{i,j}^2\right)$
3. The new state of the cell $c_{i,j}$ is obtained by adding the current state with the variables obtained in step 2:

$$c_{i,j} = c_{i,j} + u_{i,j} \tag{2}$$

All we have to do now is to define the function $f$. More exactly, given a bank of empty cells, we want to learn a function $f$ so that after a number of iterations in the self-organization process, the cell will organize themselves into a well-defined form. Given the fact that we talk about visual shapes, the most natural choice for the function $f$ is a convolutional neural network (CNN).

## 3. Related work and our contributions

Due to the rapid development of deep learning, both from a theoretical and practical perspective, several papers trying to implement and tests CA model on real scenarios have appeared. In [4], authors tried to grow an empty grid of cells to the image of a lizard emoji. The paper shows that if the size of the cell is 16 and the CNN model contains only one convolutional composed of 128 neurons, the image of the lizard can be obtained in 8000 iterations. In [5], the authors applied the CA model to solve the problem of classifying handwritten digits. The accuracy obtained by the model is 95.3% after 200 iterations. In [6], the CA model is applied to synthesize textures. Although the generated texture and the target texture are not pixel-perfect, the model generates textures that are consistent and robust.

This work is based on [4]. The main disadvantage of the original paper is that the authors don't provide an analysis of how CA model parameters influence the results. We extend the research done in the original work by trying different variations on the CA model. In particular, we explore different sizes of the cells and different CNN architectures. Also, we apply the model to generate images of a real mechanical component not artificial created emojis to simulate the self-growing and the self-repairing of mechanical components. Additionally, we present a tutorial on how to implement the CA model using well-known deep learning libraries.

## 4. Building the model

In this section, we present a step-by-step tutorial on how to implement the CA model using Python and Keras.

To load the image, we write the function `load_component` which downloads the image from a GitHub repository and resize it to the specified dimensions:

```
GRID_SIZE = 64
def load_component(url):
   return imread(url, GRID_SIZE)
```

The test imagine has dimensions of $64x64$ and it is shown in Figure 4. Greater dimensions will enhance the image quality but will increase the training time of the CNN.

*Figure 4: The image of 3/8-16 x 1-1/2 screw*

We want to implement a CA model that starting from an empty grid of cells of dimensions $64x64$ will automatically develop the image of the screw. In this way, we simulate the self-growing capability of a CA.

Next, we define the dimensions of a cell and the initialize the grid:

```
CELL_SIZE = 16
seed = np.zeros([1, GRID_SIZE, GRID_SIZE, CELL_SIZE], np.float32)
seed[:, GRID_SIZE//2, GRID_SIZE//2, 3:] = 1.0
```

The first test will use a CNN with just two convolutional layers and no max-pooling layer. The model is initialized as follows:

```
model = tf.keras.Sequential([
                              Conv2D(128, 3, padding='same',
activation=tf.nn.relu),
                              Conv2D(CELL_SIZE, 1,
kernel_initializer=tf.zeros)])
```

We use 128 kernels of dimension $3x3$. The summary of the model generated with the command `model.summary()` is given in Figure 5.

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (1, 64, 64, 128)          18560
_________________________________________________________________
conv2d_9 (Conv2D)            (1, 64, 64, 16)           2064
=================================================================
Total params: 20,624
Trainable params: 20,624
Non-trainable params: 0
_________________________________________________________________
```

*Figure 5: The summary of the model*

As we have stated in section 2.2, to update the state of the cells during the self-organization process, at each iteration, we add to the current state of the cell the output of the CNN. In code, $x$ is the grid of cells, the update rule is implemented as:

```
NUM_OF_ITERATIONS = 16
x = seed
for i in range(NUM_OF_ITERATIONS):
    x = x + model(x)
```

To train the model, we write the function `training_step`. This function executes 50 iterations of the self-organization process, compute the loss (i.e. the difference between the grid and the target image of the screw) and updates the parameters of CNN using gradient descend algorithm.

```
trainer = tf.optimizers.Adam(1e-3)
loss_log = []
def training_step():
  with tf.GradientTape() as g:
    x = seed
    for i in range(50):
      x = x + model(x)
    loss = tf.reduce_mean(tf.square(x[...,:3][0] - target_component))
  params = model.trainable_variables
  grads = g.gradient(loss, params)
  trainer.apply_gradients(zip(grads, params))
  return loss, x
```

We train the model for 1024 epochs and show in Figure 6 the imagine generated by the CA at certain milestones:

```
x_show = []
for k in range(1024):
  loss, x = training_step()
  loss_log.append(loss.numpy())
  if k%128 == 0:
    x_show.append(x[...,:3][0])
  print(k)
  imshow(zoom(x[...,:3][0]))
```

As it can be seen from Figure 6 the target imagine gets better and better with each epoch. To compare the result produced by the CA model with the original target component we use PSNR metric:

```
def PSNR(original, compressed):
    mse = np.mean((original - compressed) ** 2)
    if(mse == 0):
        return 100
    max_pixel = 1.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return psnr
```

The PSNR between the image generated by the CA model and the target imagine after 1024 epochs is 37.71 which means that the two images are almost identical. The loss after each of the first 1024 epochs is plotted in Figure 7.

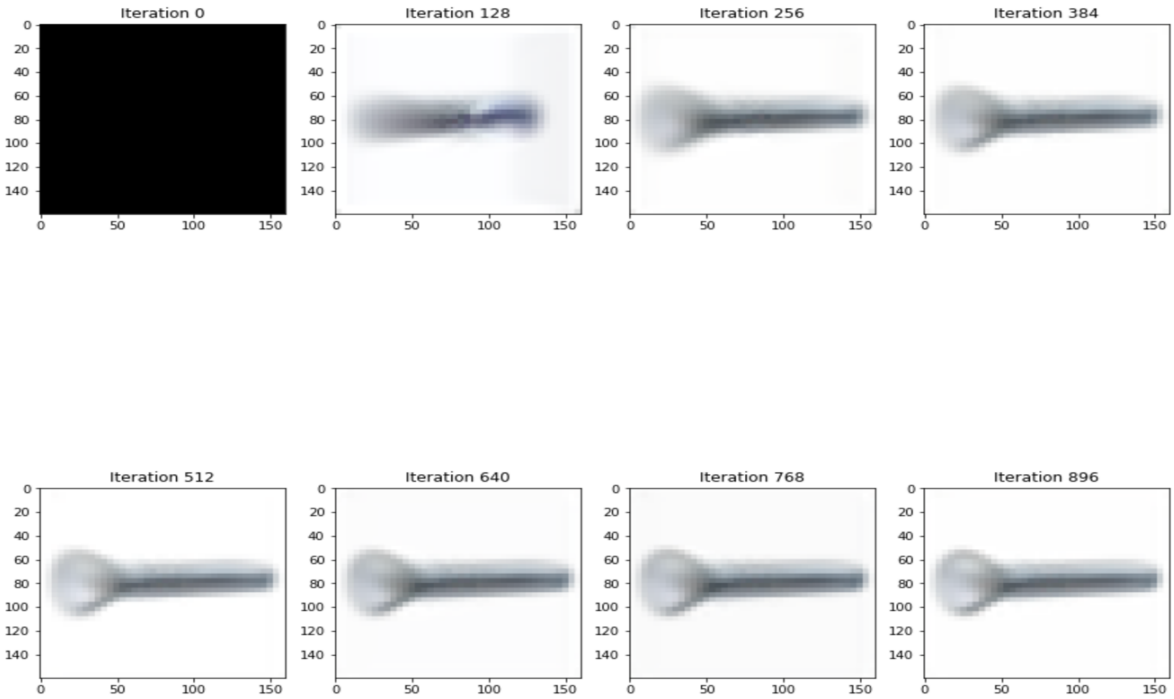


*Figure 6: The evolution of the self-organizing process*

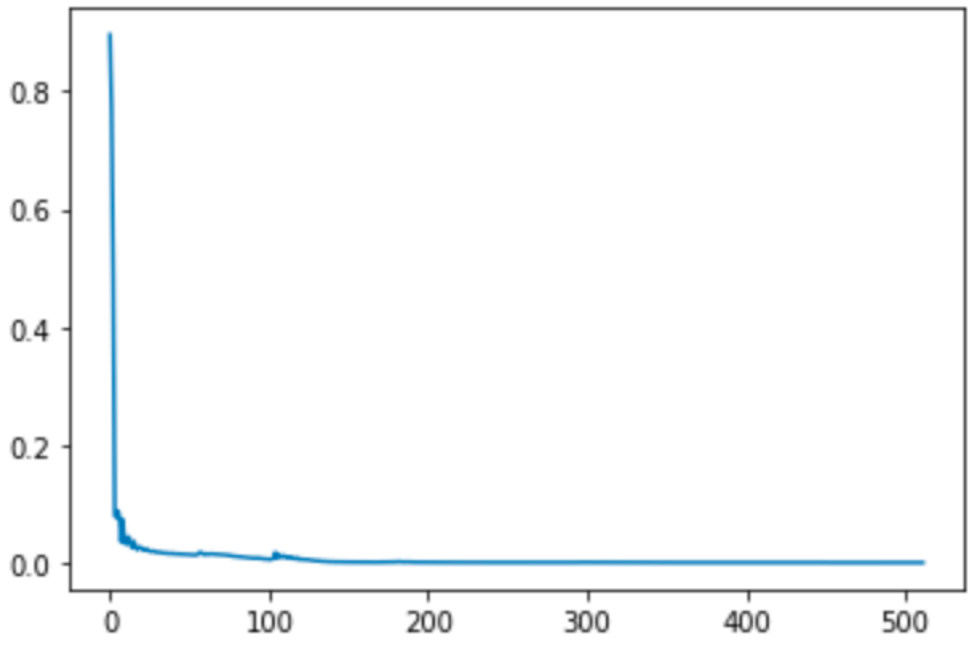


*Figure 7: The loss after the first 512 epochs*

## 5. The performance of the model

In this section, we will explore how different cell sizes and different CNN architectures affect the performance of the model. We also simulate the self-repairing behavior of the mechanical component by altering the seed of the cell grid.

Changing the cell size affect the PSNR between the image generated by the model and the target image according to Table 1:

| Cell size | PSNR |
|---|---|
| 4 | 32.54 |
| 8 | 31.66 |
| 12 | 19.19 |
| 16 | 37.71 |
| 24 | 30.13 |
| 32 | 18.83 |
| 64 | 17.83 |

*Table 1: PSNR for different cell sizes*

We cannot make a direct link between the cell size and the performance. A bigger cell does not necessarily imply a greater PSNR. The largest PSNR was obtained for a cell size of 16. Acceptable results were also obtained for cell sizes of 4, 8, and 24.

The architecture of the CNN also has an impact on the performance of the model. Adding an extra convolutional layer with 64 filters and keeping the cell size of 16 results in a decrease of the PSNR at 32:

```
model = tf.keras.Sequential([
                                Conv2D(128, 3, padding='same',
activation=tf.nn.relu),
                                Conv2D(64, 3, padding='same',
activation=tf.nn.relu),
                                Conv2D(CELL_SIZE, 1,
kernel_initializer=tf.zeros)])
```

Intuitively, the result must improve if we increase the size of the network. At the same time, a larger network needs more epochs to train. After another 1024 epochs, the PSNR becomes 34.79. From Figure 8 it is clear that increasing the number of epochs results in a greater PSNR.

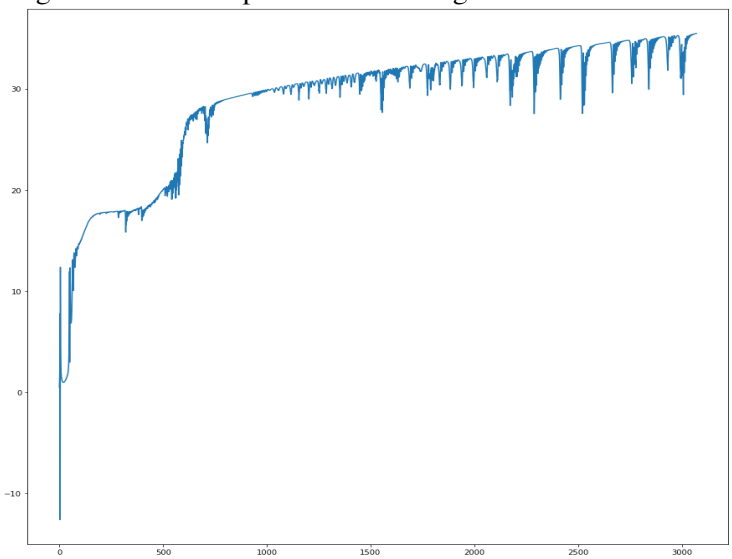


*Figure 8: The evolution of PSNR depending on the number of epochs*

Changing the seed to an altered image of the target component can emulate the self-repairing property of the model. After training the model is applied for a small number of iteration to a seed and as it is shown in Figure 9 the generated image simulates a self-repairing behaviour since the PSNR to the target image gets better and better:

```
x = seed
NUM_OF_ITERATIONS = 40
for i in range(NUM_OF_ITERATIONS):
    x = x + model(x)
NUM_OF_ITERATIONS = 8
for i in range(NUM_OF_ITERATIONS):
    x = x + model(x)
    plt.subplot(2, 4, i+1)
    plt.gca().set_title('PSNR: ' + str(PSNR(x[...,:3][0],
target_component)))
    plt.imshow(zoom(x[...,:3][0]))
```

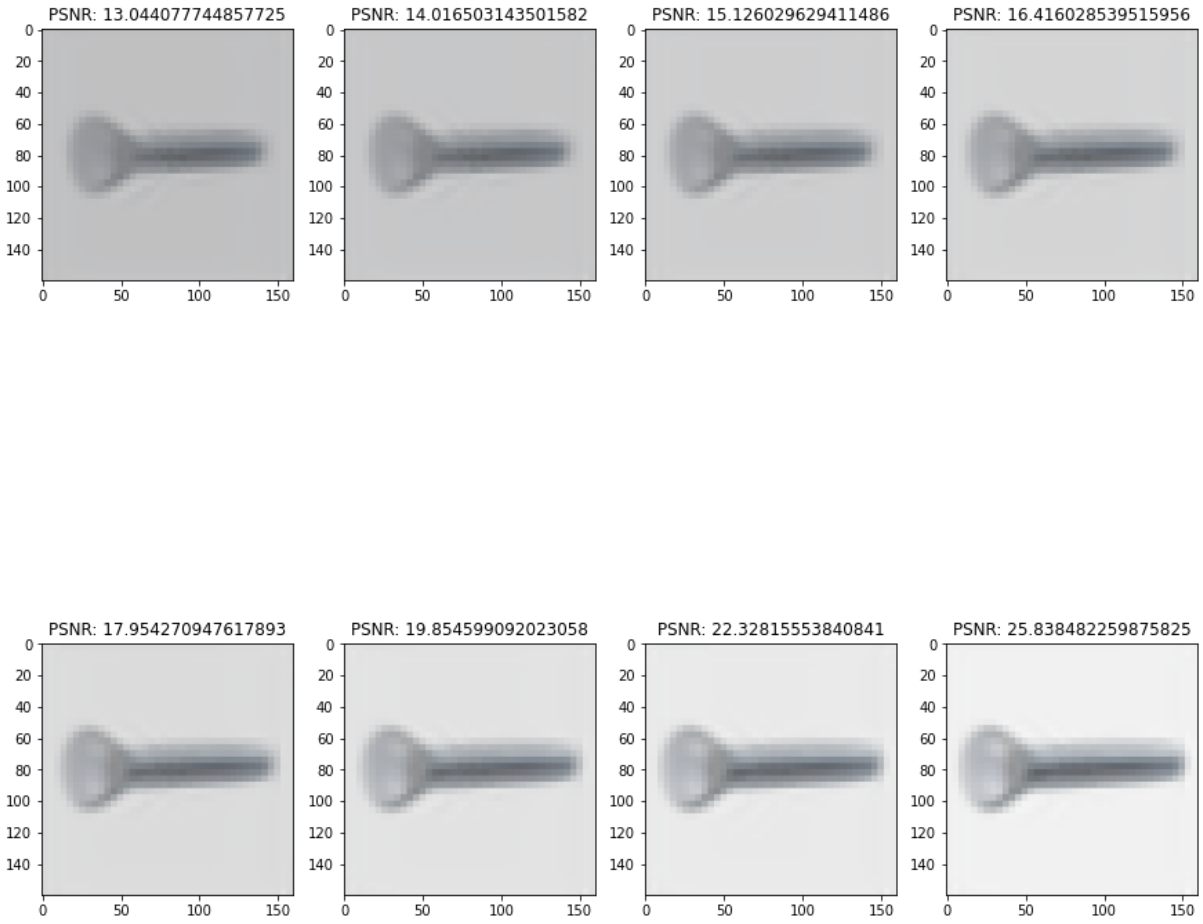


*Figure 9: Self-repairing behavior*

Our last test for the model involves generating the image of an electric motor. Unlike the image of the screw, which is preprocessed, the picture of the electric motor was taken with a mobile camera and was not processed before being loaded into the workspace. The target image is shown in Figure 10.

*Figure 10: The image of an electric motor*

The self-growing process generated by our CA model is presented in Figure 11. The model performed well in this task although the image was not preprocessed. The PSNR between the target image and the images generated by the model after 2688 iterations is 34.24. That means that the differences between the two images are visually imperceptible.

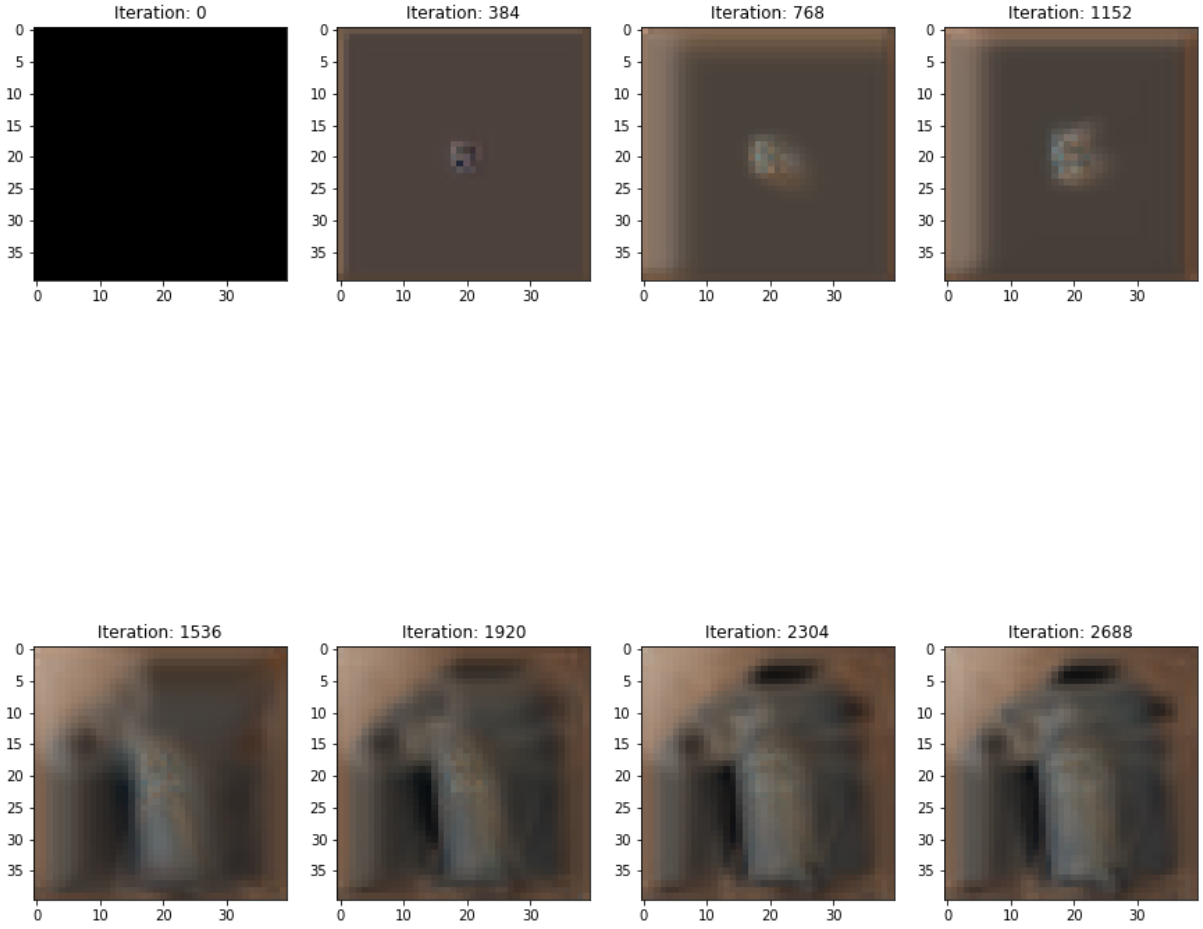


*Figure 11: The image of the electric motor at different iterations*

## 6. Conclusions and further direction of research

In this paper, we simulated the self-growing and the self-repairing behaviors of a mechanical component using the CA model. In the first part of the paper, we presented a step-by-step tutorial for implementing a CA with convolutional neural networks using the Keras library. All the code is available on a google colab notebook [7]. We saw that there is no direct link between the cell size and model performance. Our experiments showed that to maximize the performance it is necessary to run the model with several cell sizes and choose the best one. Adding an extra convolutional layer does not increase the PSNR between the images generated by the model and the target image but we showed that increasing the number of training epochs results in a better result. The self-growing behavior implies the self-repairing propriety of the CA model by simply apply the update rule to a pre-trained model as shown in the previous section.

A further direction of research consists of using CA models to generates 3D constructions like the one used in AutoCAD. This achievement could be much closer to the final target of building self-repairing robots without human assistance. The biggest challenge in doing this kind of simulation represents the computational power required to train and deploy a CNN large enough to produce CA models with high performance.

**References**

[1] "CNN model" [Online]. Available: https://editor.analyticsvidhya.com/uploads/90650dnn2.jpeg [Accessed: 20-Feb-2021].

[2] "Convolutions for Images" [Online]. Available: https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html. [Accessed: 20-Feb-2021].

[3] "Max-pooling / Pooling" [Online]. Available: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling. [Accessed: 20-Feb-2021].

[4] Mordvintsev, et al. , "Growing Neural Celular Automata", Distill, 2020.

[5] Randazzo, et al. , "Self-classifying MNIST Digits", Distill, 2020.

[6] Niklasson, et al. , "Self-Organising Textures", Distill, 2021.

[7] M. I. Plesa, "Google Colaboratory," *Google*. [Online]. Available: https://colab.research.google.com/drive/1YnSA1uR38zsMTgUzrV0zjNOCA4I91fkq?usp=sharing. [Accessed: 19-Mar-2021].