A Rotation-Invariant Embedded Platform for (Neural) Cellular Automata

Dominik Woiwode[†1], Jakob Marten[†2], and Bodo Rosenhahn[1]
[†] Contributed equally

[1]Institute of Information Processing, Leibniz University Hannover, Germany
{woiwode, rosenhahn}@tnt.uni-hannover.de

[2]Institute of Microelectronic Systems, Leibniz University Hannover, Germany
marten@ims.uni-hannover.de

Abstract

This paper presents a rotation-invariant embedded platform for simulating (neural) cellular automata (NCA) in modular robotic systems. Inspired by previous work on physical NCA, we introduce key innovations that overcome limitations in prior hardware designs. Our platform features a symmetric, modular structure, enabling seamless connections between cells regardless of orientation. Additionally, each cell is battery-powered, allowing it to operate independently and retain its state even when disconnected from the collective. To demonstrate the platform's applicability, we present a novel rotation-invariant NCA model for isotropic shape classification. The proposed system provides a robust foundation for exploring the physical realization of NCA, with potential applications in distributed robotic systems and self-organizing structures. Our implementation, including hardware, software code, a simulator, and a video, is openly shared at: `https://github.com/dwoiwode/embedded_nca`

Introduction

In nature, various phenomena, such as crystal formations and zebra stripes, can be described by cellular automata (CA) (Shah et al., 2022; Graván and Lahoz-Beltra, 2004). CA are used for modeling discrete and dynamic systems. They consist of simple cells that change their state simultaneously, following a uniform update rule applied to all cells. This rule is constrained in a way that allows only local interactions with neighboring cells. Despite this, life-like structures can emerge on 2D grids with relatively simple rules as in Conway's "Game of Life" (Gardner, 1970). Identifying update rules that produce emergent behavior is a difficult process, often carried out by hand in the past. Neural cellular automata (NCA) seek to address this by learning the update rule through a neural network (Mordvintsev et al., 2020). They keep the local property of CA (e.g. a 3×3 neighborhood on a 2D grid) and use stochastic gradient descent to optimize the update rule regarding a given target pattern.

Realizing software simulations in the physical domain often exposes gaps between modeled behavior and real-world dynamics. In modular robotic systems, this presents a unique challenge, as the absence of global communication restricts information exchange to strictly local interactions.

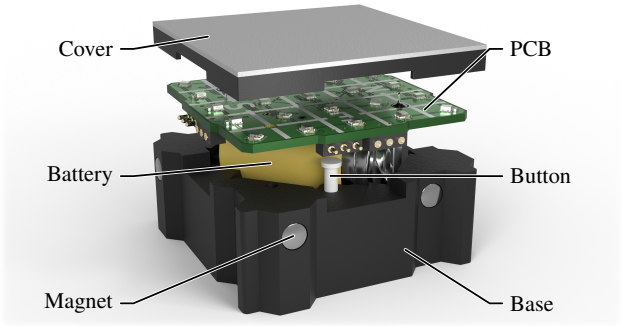


Figure 1: Structural breakdown of a single cell's hardware components.

Our work is closely related to the inspiring idea by Walker et al. (2022) to bring NCA into the physical world. However, several assumptions in their paper lead to limitations that are addressed by our proposed hardware. While prior work employs male and female header pins to ensure stable connections, this approach enforces a fixed orientation, which our design aims to overcome. The proposed hardware is symmetric, allowing any side to connect seamlessly with any side of another cell. The previous paper supplies power via cable, which introduces two key limitations: all active cells draw current through a single connection, leading to increased load issues, and modules cannot be rearranged while powered. In contrast, the proposed implementation utilizes battery-powered cells, enabling them to retain their state even when disconnected from the collective. Since each unit operates independently, they can be assembled into swarms of virtually any size.

To summarize, the **core contributions** of our work include:

- We developed a portable, square-shaped robotic module that operates independently of its horizontal orientation and communicates with its four adjacent neighbors (see Hardware Design).
- We present a novel rotation-invariant NCA capable of classifying shapes regardless of their orientation (see

Training a physical NCA).
- We analyze the effect of our rotation-invariant training methods and show their performance enhancements (see Experiments).

Related Work

A CA is a computational model with a set of "cells" on a d-dimensional grid that update their state over time based on their local neighbors using the same update rule (Neumann, 1966). A cell state s can range from a single boolean variable to a high-dimensional vector. A classic example of a CA is Conway's "Game of Life" Gardner (1970), illustrating the emergence of complexity from simple local rules on a two-dimensional grid. Throughout the remainder of this work, CA refers to a two-dimensional cellular automaton with a cell state $s \in \mathbb{R}^c$.

A neural cellular automaton (NCA) is a special type of CA which was introduced by Mordvintsev et al. (2020). It is characterized by the update rule which is represented by a neural network. Each **cell state** of a NCA has c different channels in which cell specific information can be encoded. The NCA can use certain channels arbitrarily as hidden channels, whereas others are reserved for interpretable purposes. Common examples include a RGB color representation in the first 3 channels (Mordvintsev et al., 2020; Niklasson et al., 2021b; Otte et al., 2021), a classification output (Randazzo et al., 2020), segmentation masks (Sandler et al., 2020) or even gene encodings (Stovold, 2023).

The cellular update rule of a NCA is split in two parts. First, a **cell perception** is computed by applying various convolutional kernels. The kernels operate channel-wise, and their outputs are concatenated to form the perception vector P. While fig. 7 shows the kernels applied in our setup, a typical NCA allows for arbitrary local kernels. Once the cell perception is computed, the **state update** Δs is determined solely based on the perceptual information. This process involves a relatively small neural network, typically containing fewer than 10,000 parameters, which makes them a perfect fit for small embedded hardware. More details on NCA, based on our architectural design, are provided in section Training a physical NCA.

Since the influential paper by Mordvintsev et al. in 2020, this subject has been the focus of many papers. In recent years, more than 50 publications have been produced, which are aggregated and continuously updated in a repository by Woiwode et al. (2025). Next, we present works which are relevant for this paper. Oliveira and de Oliveira (2008) published an approach to use CA for 2D pattern recognition. Randazzo et al. (2020) build on this idea and proposed an NCA which is capable of recognizing handwritten digits (Lecun et al., 1998). The task of each cell in the NCA is to locally predict the global digit encoded by the structure. Through repeated local exchanges of state, cells are able to indirectly communicate with distant parts and agree on a unified decision. Walker et al. applied this concept in 2022 in a real-world setting by constructing modular robotic devices that emulate NCA to classify their global shape. Hardware limitations required replacing the commonly used 3×3 Moore neighborhood with the more constrained Von Neumann neighborhood, reducing each cell's accessible information (Neumann, 1966). Similar to other NCA approaches, their system depends on robots being aligned in a specific direction.

Later studies introduce a concept where each cell is given an individual orientation (Mordvintsev et al., 2022; Randazzo et al., 2023). This concept is also used in our hardware implementation and further details will be presented in section "Hardware-aware training strategies".

Hardware Design

The Kilobots, presented by Rubenstein et al. (2012), set a new standard for developing scalable and affordable robot swarms. In this section, we introduce the proposed hardware architecture, which draws inspiration from Walker et al. (2022) but further enables distinctive novel features such as persistent state, diverse rotations, and enhanced output capabilities. Therefore, the hardware must meet three primary requirements:

1. Each cell must be capable of communicating its current state to its four neighboring cells in any of the four rotational positions.
2. Each cell should be able to display output via an array of colored Light Emitting Diodes (LEDs).
3. Each cell must be powered by a battery to enable seamless disconnection and reconnection with other cells without losing its state.

The first requirement has two major consequences for the design. Firstly, a cell needs to be fitted with a genderless connector. Walker et al. use pin headers for north/east and sockets for south/west placed in the center of each edge. This prevents the rotation of the cell. Our design uses two connectors per edge – one male and one female – which results in an interface independent of the cell's rotation. To further simplify the handling of the cells, spring actuated pins and corresponding counter parts were chosen. In total, each edge connects 6 pins, of which two pins each are used for power and ground. The remaining pins are used to communicate between the two cells using the Universal Asynchronous Receiver / Transmitter (UART) protocol.

Secondly this directly requires that the microcontroller is able to communicate over four UART channels at the same time. Many microcontroller in the low-budget range only support one or two interfaces, forcing to use a software driven implementation for the remaining channels. While technically possible, it reduced the amount of computation time available for the main calculations and additionally complicates the firmware design. In our case we selected the RP2350 microcontroller by Raspberry Pi Ltd (Raspberry

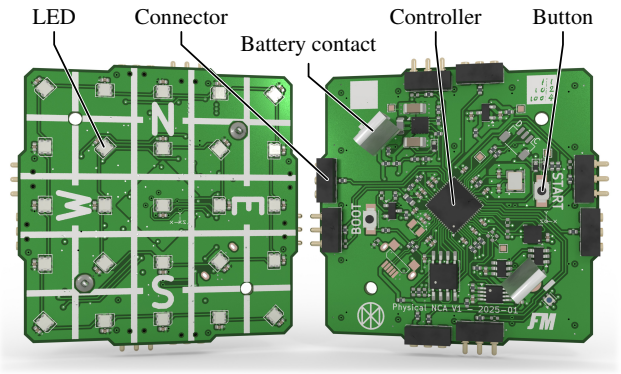


Figure 2: The top (left) and bottom side (right) of the custom PCB.

Pi, 2025) as it is both low-budget and can support more than four hardware-supported UART channels.

To satisfy our second requirement, we chose to implement an array of 25 LEDs in a grid of 5×5 which is shown on the left side of fig. 2. This enables many different output patterns and a straightforward output of the current cell state. As LED component a WS2812B-2020 was selected since it has RGB color output, a small spatial footprint and allows controlling a whole chain by only one data pin.

To be able to move a cell freely without any outgoing cables, an additional battery circuitry was added, which sits diagonally at the center of the Printed Circuit Board (PCB) (see right side of fig. 2). It stabilizes the battery voltage to usable supply voltages of 5 and 3.3 V for all components and enables its charging via universal serial bus (USB) or the edge connector. To prevent any damage to the battery, a battery protection circuit is included. To balance physical size, ease of maintenance and capacity a rechargeable battery in the CR123A form factor was selected ($34\,\mathrm{mm} \times 16\,\mathrm{mm}\ \varnothing, 700\,\mathrm{mAh}$).

The final design of the custom 4-layer PCB as presented in fig. 2 measures $49\,\mathrm{mm} \times 49\,\mathrm{mm}$. The top side (left) only contains the LED array; the bottom side houses any other components, i.e. the connectors, the microcontroller, buttons and an 3-axis accelerometer for user interaction, an USB and debug connector, the battery contacts, and the power and battery circuitry. Figure 3 presents the interaction of the different components.

Beyond the PCB, the hardware of a cell consists of four 3D-printed parts as well as eight magnets allowing a quick and stable connection of multiple cells. The whole stack is displayed in fig. 1. A 3D-printed base houses the battery and magnets. The PCB is placed on the top. To reach the buttons, two extension pins are inserted through the base. A translucent cover protects the electronics and improves the perceptibility by scattering of the emitted light.

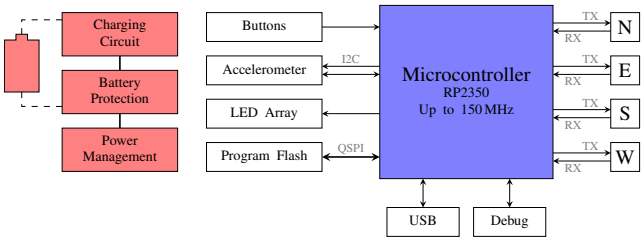


Figure 3: Different components used in the hardware design.

Firmware

The main purpose of the implemented microcontroller firmware lies in three tasks: The communication with neighboring cells and therefore receiving and sending state, the calculation of the next state based on the current and received state and the display of the output. The firmware is implemented in C using the "PICO C SDK" (Raspberry Pi, 2024).

For the first task, the four bidirectional ports are used. Each port is connected to one edge connector and uses the UART protocol. As the RP2350 only has two hardware UART interfaces, two Programmable Input Output (PIO) units of the microcontroller where used; one for transmitting data and one for receiving the incoming data. The PIO units provide four in- or outputs each and can be used to emulate different transfer protocols. This flexibility also give the possibility to adapt existing protocols beyond their typical configuration options. In our case, we chose to implement a UART protocol with a word with of $32\,\mathrm{bit}$, simplifying the transmission of single precision floating point numbers used for state representation. A baud rate of $115\,200\,\mathrm{kBd}$ was selected. The transfer is initiated by moving the data to be sent to the corresponding PIO data queues. On the receiving site, the data is deserialized and placed into the PIO's receiver queues. An interrupt is issued to allow further processing by the main core.

Within the second step, the calculation based on the current state of the cell and its neighbors is performed. Therefore, all states are combined to a single tensor and fed to a configurable calculation engine. Inspired by similar frameworks as TensorFlow Lite, a program is represented by a header, a set of tensors and an operations list. Its structure is presented in fig. 4.

The header contains mandatory information as the version of the model, the size of the cell state c, the number of tensors and operations and delays used to control the timing behavior. The set of tensors is represented by a list of pointer and can be split into two distinct types: On the on hand a tensor can be immutable (read only=R). Those tensors are used to provided values e.g. weights needed by the performed operations. Their content is stored directly in the program and cannot be changed. On the other hand there are modifiable tensors used to store intermediate re-

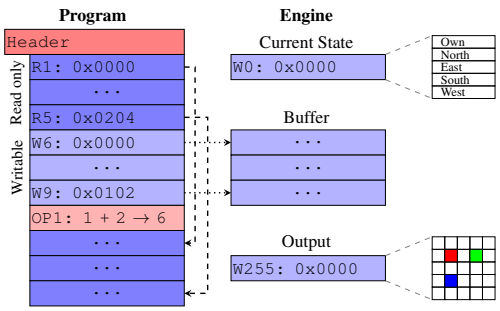


Figure 4: Program and data structure and its interaction with the engine.

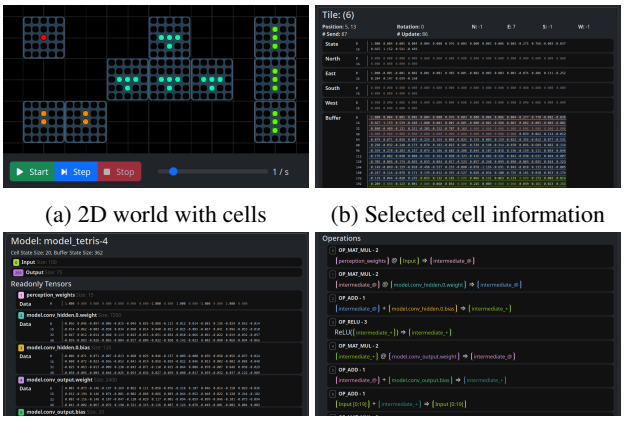
Code	Name	Description
0x00	NOP	No operation
0x01	ADD	Add two tensors
0x02	MAT_MUL	Matrix multiplication on two tensors
0x03	RELU	Apply relu on tensor
0x04	FILL	Fill tensor with value
0x05	MAX	Calculate maximum over tensor
0x06	SOFTMAX	Calculate softmax over tensor
0x07	STEP	Apply step function to tensor
0x08	MUL	Multiplication on two tensors
0x0B	FILL_RAND	Fill tensor with random value
0x0C	ARG_MAX	Calculate argmax over tensor

Table 1: List of implemented operations.

sults (writable=W). Their contents are stored within a global buffer area and referenced by the pointer value. Each tensor has a 8 bit, where 0 is reserved for the current state and 255 for the output tensor. The output tensor has the dimension 25×3 where each LED has three channels used to represent red, green, and blue. The operation list contains operation descriptors, which are processed linearly from start to end in each update cycle. Each descriptor starts with a operation identifier which is used to select the correct operation. Currently, the engine supports 11 different operations listed in table 1. It is also possible to add further operations by extending the engine to support other calculation steps. It is followed by a operation specific amount of source and destination tensor identifiers or constant values. For example the operation *ADD* takes two source and one destination tensor identifier as well as the length of the tensors. During program generation is must be assured that all tensors are at least as long as specified in the operations as those constraints are not checked during calculation.

In the last step, the written output is display via the LED array on the top. The third available PIO is used to generate the control signals for the *WS2812B* LEDs. This way the values can directly be shifted in an output queue and are automatically transferred to the LEDs.

In addition to the three main tasks, the firmware handles input data of the connected accelerometer. Currently, the values are only used to detect if the cell is flipped upside-



(a) 2D world with cells (b) Selected cell information

(c) Model weights (d) Operations list (model)

Figure 5: Different core parts of the simulation environment.

down to initiate a power off. In future work, it is planned to enable access to those value from the calculation engine to include them in a program. Also, a debug interface is implemented to transfer the current cell state and performance parameters via the USB interface.

The PCB layout, microcontroller firmware, simulator and 3D model files are open source and available in our code.

Simulator

To perform experiments quickly without the need to compile the firmware and apply it to all hardware cells, a simulator running in the browser was developed. It can be used to simulate and debug different programs.To ensure simulation accuracy, the firmware calculation engine is embedded as WebAssembly into the application.

Examples images of the simulator are displayed in fig. 5. The main part of the simulator are cells, displaying their current output, that can be moved and rotated (5a). A simulation can be controlled by different buttons as *start*, *step* and *stop*. Furthermore, it is possible to select other (precompiled) models, upload own models and chose different simulation techniques. The state of a selected cell can be inspected within an information overview (5b). These include the current state, the received neighbor states and the buffer containing the content of intermediate tensors as well as the output. The simulator also shows data on the used model e.g. the data of the immutable included tensors (5c). Additionally a list of operation is displayed (5d).

Training a physical NCA

To minimize the required capabilities of the embedded hardware, we separated the training loop from the firmware of the robots. This section will focus on the training procedure, which is implemented using a modular PyTorch framework. We present specific adaptations necessary for achieving rotation-invariant training. The framework and the train-

ing code is made open source and can be found in our code repository.

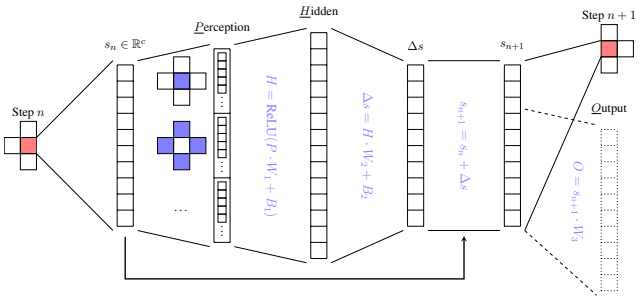


Figure 6: Overview of our NCA model. The use of a task head to generate task-specific outputs is optional and not implemented in all models.

An overview of our NCA model can be seen in fig. 6. Our CA is based on a 2D grid of cells, where each cell can see the state of their directly adjacent cells ("*4-neighborhood*") (Neumann, 1966)).

The perception is built upon a combination of different kernels K that are adapted to our 4-neighborhood, as shown in fig. 7. "Gradient X" and "Gradient Y" are simple gradient filters similar to Sobel filter (Sobel and Feldman, 1973). The "Von Neumann" is a isotropic kernel that sums all neighboring values. The "Identity" is simply the own state of the cell. These kernels are applied independently per channel and the results are subsequently concatenated to form a perception vector $P \in \mathbb{R}^{c \cdot n_k}$, where n_k is the number of kernels applied.

$$P = \text{concat}\left(K_1 \overset{\text{cw}}{\otimes} s,\ K_2 \overset{\text{cw}}{\otimes} s,\ \ldots,\ K_{n_k} \overset{\text{cw}}{\otimes} s\right), \quad (1)$$

where $\overset{\text{cw}}{\otimes}$ denotes a channel-wise convolution. In this context, s refers to the state configuration of the entire CA, where the kernels inherently account for the local 4-neighborhood interactions. Constant zero-padding is used at the boundary regions, as this accurately represents inactive cells.

We calculate the next state for a cell following a simple

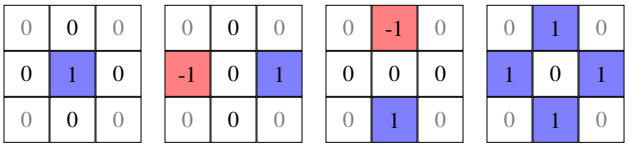


Figure 7: The perception kernels used for our NCA model.

update rule

$$H = \text{ReLU}(P \cdot W_1 + B_1), \quad (2)$$
$$\Delta s = H \cdot W_2 + B_2, \quad (3)$$
$$s_{n+1} = s_n + \Delta s, \quad (4)$$

where W_i and B_i are the learnable weights and bias of the i-th layer of the neural network respectively and ReLU is the Rectified Linear Unit activation function (Nair and Hinton, 2010).

A training iteration starts with a seed state s_0 and applies the NCA update rule T times, where T is randomly sampled between 10 and 40 for our experiments. Instead of the commonly used cross-entropy loss for classification, a mean squared error (MSE) against a one-hot encoded class label l is used, as it yielded more stable training results Walker et al. (2022); Randazzo et al. (2020).

In contrast to a classical NCA, we added an optional task specific head layer. Consequently, the number of channels needed is less constrained by the task. For example, classifying 29 categories typically requires at least 29 channels, yet one-hot encoding causes most channels to remain near zero, conveying minimal information. With a classification layer, the NCA is able to learn a latent space representation of the classification. To keep the spirit of the original NCA, only a part of the final state is forwarded to this layer. This preserves the benefit of hidden channels, which can serve purposes beyond the primary task, such as encoding spatial relationships with neighboring cells. The task head layer only has to be used when the NCA is evaluated and has no direct impact on the further state. We apply the loss defined in eq. (10) and train the model using gradient descent and the Adam optimizer (Kingma and Ba, 2014).

Hardware-aware training strategies

While the preceding section outlines the (mostly) standard training procedure for NCA, the characteristics and constraints of our robot hardware impose specific considerations that must be integrated into the training process.

Setting the first channel of each cell always to 1 ensures that the cells are able to sense each other. Otherwise, lacking neighbors can be indistinguishable from having a neighboring cell in a "dead" state, defined as a zero vector. We use this channel to apply a (modified[1]) "**Alive-Masking**" to the cells (Mordvintsev et al., 2020; Randazzo et al., 2020).

$$\Delta s' = \begin{cases} \Delta s & \text{if cell is active} \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

As our robots are designed to be **rotation invariant**, we also incorporate this aspect in our training. With our hardware, the cells can be physically rotated to achieve a different orientation. During training we add an extra channel

[1]The original Alive-Masking also takes neighboring cells into account. We limit this to the own cell state.

to our NCA that represents the orientation angle θ of this cell. This angle cannot be directly perceived by any cell, including itself, and it is inherently accommodated within the hardware design due to the physical capability to rotate the cells, rather than explicitly encoded. We then apply a 2D rotation matrix to rotate our gradient perceptions P_x and P_y at θ to get P'_x and P'_y using following formula

$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} P_x \\ P_y \end{bmatrix}. \tag{6}$$

The same concept already has been used by Mordvintsev et al. (2020) on a global scale and by Randazzo et al. (2023) for each cell. Their approaches are modified by discretizing θ into four predefined rotations: $0°, 90°, 180°, 270°$. θ is fixed during one training iteration and cannot be changed by the NCA itself.

While other implementations focus on getting a correct result at a specific simulation step (Walker et al., 2022), we want to achieve a stable configuration over a long time. We therefore make use of a **training pool** (Mordvintsev et al., 2020). A pool of samples is initialized with corresponding seed and target configurations. At the start of one training iteration, a batch of samples is drawn from this pool. Then, this batch is propagated through the NCA and evaluated as described above. At the end of each training iteration, the selected batch is added back to the sample pool, enabling longer episode durations over time. In this process, k_{new} random samples are substituted with new seeds and targets, preventing the unlearning of previous iterations. Furthermore, k_{replaced} random samples are altered to new targets, where only the channel indicating the living cells and θ are modified, but the overall cell state keeps intact. This modification was essential for enabling the models to learn to recognize new shapes. Otherwise, they remained stuck in a fixed state, despite changes in configuration.

We simulate an **asynchronicity** of cells by adding a random dropout mask to Δs. It has been shown that this makes the model more robust to pertubations in the state of the CA (Niklasson et al., 2021a). We also add a small normal distributed noise ($\sigma = 2 \cdot 10^{-2}$) to each update that is not affected by the dropout, which further helps the model to recover from small pertubations (Randazzo et al., 2020).

After training a model, the corresponding network structure and model weights have to be converted into a format that is recognizable by the firmware. During this step most modifications explained in this section are stripped, as they are only relevant during training. Because each cell determines its next state independently, the overall neural network is minimal and can be summarized by equations (1)–(4).

Experiments

We show the capabilities of our robotic models in two different experiments. In the first experiment, we evaluate our hardware using a hand-crafted model. For this we use a task called "firefly-synchronization", where different cells have to agree on a common clock frequency.

In the second experiment, we reproduce the classic shape classification for simple digits using a NCA. We also extend this experiment using rotation invariant polyominoes up to a degree of 5. Our results show that cells can be added, removed or shifted in the configuration and also keep their state while being alone.

Firefly Synchronization

To demonstrate the capabilities of our hardware and firmware, we choose the "firefly-synchronization" task. The synchronization of pulse-coupled oscillating cells is a widely researched topic (Mirollo and Strogatz, 1990; Brandner et al., 2016; Ramírez-Ávila et al., 2019). Each cell has an internal clock which can be adjusted by interacting with their local neighbors to achieve synchrony. When the internal clock reaches its period time of 1 it resets and the cell emits a pulse which can be perceived by other cells. Their goal is to reach a global synchronization with decentralized actions and is inspired by the natural behavior of fireflies flashing in unison. We adapt and implement this algorithm for our cell-like robots using the "Von Neumann" kernel shown in fig. 7. Algorithm 1 shows a pseudocode of our implementation.

Algorithm 1 Pseudocode of a single cell update step.

```
procedure CELLUPDATE( )
    state ← state + k
    if neighbor_flash_detected() then
        state ← state + (k · state) + random_noise
    end if
    if state ≥ 1 then
        FLASH( )
        state ← 0
    end if
end procedure
```

We compute a circular standard deviation $\sigma_\circ$ from directional statistics for all n cells with a phase angle φ:

$$R = \frac{1}{n}\sqrt{\left(\sum_{i=1}^{n}\sin\varphi_i\right)^2 + \left(\sum_{i=1}^{n}\cos\varphi_i\right)^2} \tag{7}$$

$$\sigma_\circ = \frac{\sqrt{-2\ln R}}{2\pi} \tag{8}$$

In contrast to a normal standard deviation, a circular standard deviation takes into account that the φ wraps around so that $0 \equiv 1 \mod 1$. As seen in fig. 9 the cells find a common phase after roughly 2 minutes in our simulation. This behavior can also be observed qualitatively as seen in fig. 10. This time could be adjusted by changing the phase shift parameters k.

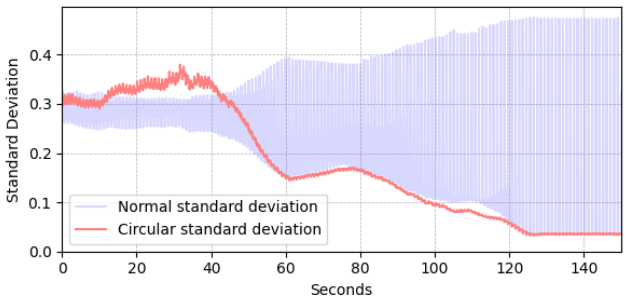


Figure 9: Standard deviation of the cell phases over time. Initially, the phases are randomly distributed. The simulation involves 29 cells arranged in a circle with a diameter of 5.

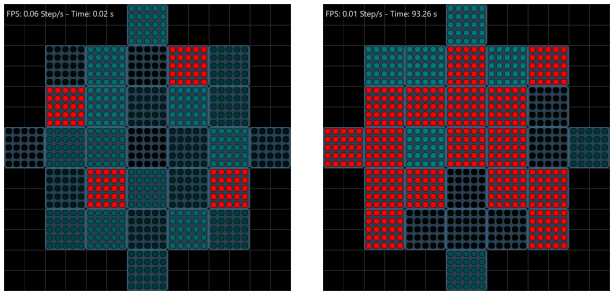
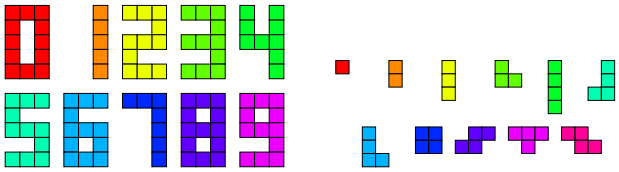


Figure 10: Simulator screenshots illustrating the firefly synchronization experiment setup. Cell brightness represents phase value; red cells indicate flashes. Left: initial random cell initialization. Right: setup after 90 s.



(a) *Digits* dataset. (b) *Polyomino-4* dataset.

Figure 11: Datasets used for shape classification.

Shape Classification

Inspired by Randazzo et al. (2020) and Walker et al. (2022), we test our robot model using a self-classifying shape detection task. We create four different datasets for this task:

- **digits**: Represents all digits from 0 to 9. This dataset is visualized in fig. 11a.
- **digits-symmetric**: Represents only digits from 0 to 8, as "6" and "9" look the same when rotated by 180°.
- **polyomino-4**: All different polyominoes up to a degree of 4 without accounting for the rotation of the shapes. This dataset is visualized in fig. 11b.
- **polyomino-5**: All different polyominoes up to a degree of 5 without accounting for the rotation of the shapes.

The architecture of the NCA is described in section Training a physical NCA. For the perception, we use the identity and two gradient kernels for X- and Y-direction which can be rotated during training as previously described. One might be tempted to use an already isotropic kernel, but this approach is limited by the inability to distinguish certain shapes (e.g., both polyominoes of degree 3). Therefore we use our rotation-invariant training described in section

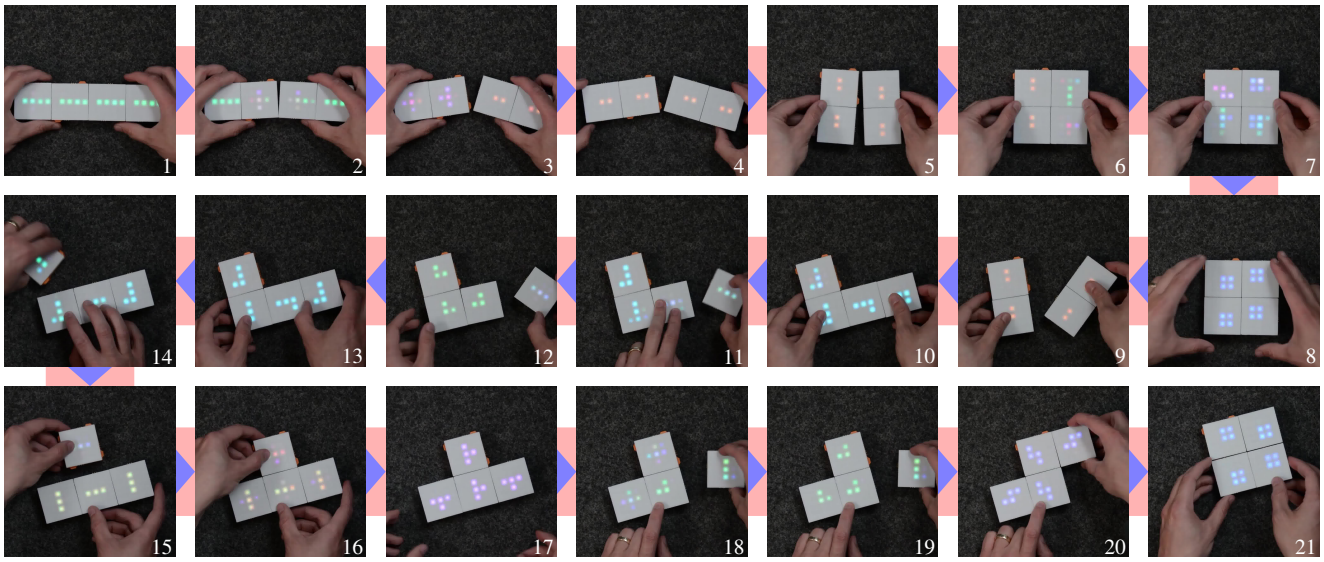


Figure 8: Example sequence of images from a video demonstrating the shape recognition. Some frames (e.g. 6 and 7) show intermediate states before classifying the correct shape again (frame 8). Frames 10–13 demonstrate the ability to rotate a tile. The full video can be seen on our project page.

Dataset	Inference Steps 50	100	150
`digits (n)`	0.90 ± 0.01	0.90 ± 0.02	0.90 ± 0.02
`digits`	0.81 ± 0.09	0.80 ± 0.08	0.80 ± 0.08
`digits-sym`	0.96 ± 0.03	0.95 ± 0.03	0.95 ± 0.03
`polyomino-4`	0.84 ± 0.04	0.83 ± 0.04	0.84 ± 0.04
`polyomino-5`	0.40 ± 0.02	0.40 ± 0.02	0.39 ± 0.03

Table 2: Classification accuracy after several steps for different datasets.

Hardware-aware training strategies.

The final classification output $o(i)$ for a cell i is either computed via the learned classification layer $W_C \in \mathbb{R}^{R \times C}$ applied to the state channels $s_n[:R]$, or directly taken from a subset of state channels.

$$o = \begin{cases} s_n[:R] \cdot W_C & \text{if classification layer is used} \\ s_n[1:C+1] & \text{otherwise} \end{cases} \tag{9}$$

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (o(i) - l)^2 , \tag{10}$$

where C is the number of classes, R is the number of channels used for the classification layer and N is the number of active cells.

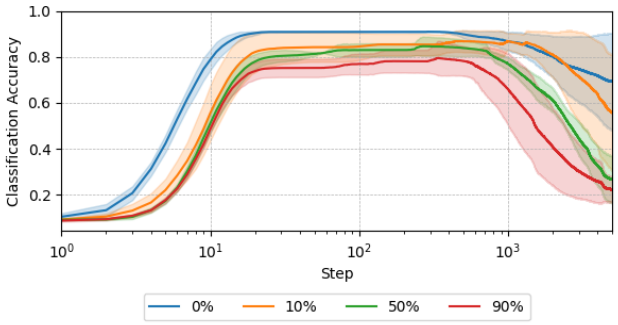
The classification output o is used to compute a weighted sum of a predefined 5×5 visual representation of each class, which can be rendered on the LED screen. When the model is confident, o approximates a one-hot vector, resulting in a clear class image; otherwise, a visual blend of multiple classes may appear (see e.g. frame 3 in fig. 8).

Per default, a batch size of 512, a sample pool size of 5120, 14 channels per cell state, 120 neurons for the hidden layer of the update function, and a classification layer which takes the first 10 channels as input is used. A series of images from the resulting default model can be seen in fig. 8.

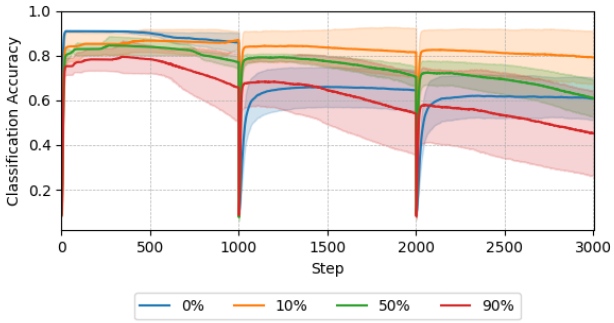
We evaluate the impact of the training methods by conducting several experiments. Each experiment is repeated 5 times, reporting the mean and standard deviation. Performance is measured using *classification accuracy*, which verifies whether the argmax of the classification output matches the target label.

Firstly, the default training parameters are used to train a NCA for each dataset. The classification remains stable As expected, the `polyomino-5` dataset is the hardest dataset, as it has the most different classes. The results can be seen in table 2.

We also investigate whether target mutation during training is essential. NCA for `polyomino-4` are trained with target replacement rates of 0, 0.1, 0.5, and 0.9, and their long-term continuity is evaluated across 5000 iterations. While fig. 12a indicates a negative impact of replacement, this is challenged by the next experiment with periodic target replacement every 1000 steps. Figure 12b suggests that changes in the cell configuration lead to even more stable classification results with a replacement rate of 0.1.



(a) Classification performance during inference with unchanged targets. Note the logarithmic scale on the x-axis.



(b) Classification performance during inference with target changes every 1000 steps.

Figure 12: Long-term stability analysis on the `polyomino-4` dataset with different values for target replacement during training. Shaded areas indicate standard deviation.

Conclusion

To summarize, we present an embedded hardware system that can emulate 2D CA. A central aspect of our design is its rotation-invariance, demonstrated through learning suitable NCA for shape classification. Our work can serve as an effective educational tool to demonstrate the principles and dynamics of (neural) CA to students. To this end, our work could be extended by incorporating other NCA variants. Additionally, our proposed output head could be leveraged to train more complex NCA models that generate structured outputs on the 5×5 LED matrix, enabling deeper insights into emergent behavior and learning dynamics. This work is open to extension, and we encourage the community to contribute using our openly available hardware and software designs.

Acknowledgements

This work was supported by the Federal Ministry of Education and Research (BMBF), Germany, under the AI service center KISSKI (grant no. 01IS22093C), the MWK of Lower Saxony within Hybrint (VWZN4219) and the European Union under grant agreement no. 101136006 – XTREME.

We used a large language model to refine and rephrase specific sentences.

References

Brandner, G., Schilcher, U., and Bettstetter, C. (2016). Firefly synchronization with phase rate equalization and its experimental analysis in wireless systems. *Computer Networks*, 97:74–87.

Gardner, M. (1970). Mathematical games - the fantastic combinations of john conway's new solitaire game "life" - m. gardner - 1970. *Scientific American*, page 6.

Graván, C. P. and Lahoz-Beltra, R. (2004). Evolving morphogenetic fields in the zebra skin pattern based on turing's morphogen hypothesis. *International Journal of Applied Mathematics and Computer Science*, 14(3):351–361.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. Conference Name: Proceedings of the IEEE.

Mirollo, R. E. and Strogatz, S. H. (1990). Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662.

Mordvintsev, A., Randazzo, E., and Fouts, C. (2022). Growing isotropic neural cellular automata. *Arxiv preprint*.

Mordvintsev, A., Randazzo, E., Niklasson, E., and Levin, M. (2020). Growing neural cellular automata. *Distill*, 5(2):e23.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA. Omnipress.

Neumann, J. V. (1966). Self-reproducing automata. *Urbana*, page 25.

Niklasson, E., Mordvintsev, A., and Randazzo, E. (2021a). Asynchronicity in neural cellular automata. In *ALIFE 2021: The 2021 Conference on Artificial Life*. MIT Press.

Niklasson, E., Mordvintsev, A., Randazzo, E., and Levin, M. (2021b). Self-organising textures. *Distill*, 6(2):e00027.003.

Oliveira, C. C. and de Oliveira, P. P. B. (2008). An approach to searching for two-dimensional cellular automata for recognition of handwritten digits. In Gelbukh, A. and Morales, E. F., editors, *MICAI 2008: Advances in Artificial Intelligence*, volume 5317, pages 462–471. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

Otte, M., Delfosse, Q., Czech, J., and Kersting, K. (2021). Generative adversarial neural cellular automata.

Ramírez-Ávila, G. M., Kurths, J., Depickère, S., and Deneubourg, J.-L. (2019). Modeling fireflies synchronization. In Macau, E. E. N., editor, *A Mathematical Modeling Approach from Nonlinear Dynamics to Complex Systems*, volume 22, pages 131–156. Springer International Publishing. Series Title: Nonlinear Systems and Complexity.

Randazzo, E., Mordvintsev, A., and Fouts, C. (2023). Growing steerable neural cellular automata.

Randazzo, E., Mordvintsev, A., Niklasson, E., Levin, M., and Greydanus, S. (2020). Self-classifying MNIST digits. *Distill*, 5(8):e00027.002.

Raspberry Pi (2024). Raspberrypi pico-sdk. `https://github.com/raspberrypi/pico-sdk`.

Raspberry Pi (2025). Rp2350 datasheet: A microcontroller by Raspberry Pi. `https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf`.

Rubenstein, M., Ahler, C., and Nagpal, R. (2012). Kilobot: A low cost scalable robot system for collective behaviors. In *2012 IEEE International Conference on Robotics and Automation*, pages 3293–3298. ISSN: 1050-4729.

Sandler, M., Zhmoginov, A., Luo, L., Mordvintsev, A., Randazzo, E., and Arcas, B. A. y. (2020). Image segmentation via cellular automata. *Arxiv preprint*.

Shah, V., Sedighiani, K., Van Dokkum, J. S., Bos, C., Roters, F., and Diehl, M. (2022). Coupling crystal plasticity and cellular automaton models to study meta-dynamic recrystallization during hot rolling at high strain rates. *Materials Science and Engineering: A*, 849:143471.

Sobel, I. and Feldman, G. (1973). A 3×3 isotropic gradient operator for image processing. *Pattern Classification and Scene Analysis*, pages 271–272.

Stovold, J. (2023). Neural cellular automata can respond to signals. *ALIFE 2023*, (arXiv:2305.12971).

Walker, K., Palm, R. B., Garcia, R. M., Faina, A., Stoy, K., and Risi, S. (2022). Physical neural cellular automata for 2d shape classification.

Woiwode, D., Schubert, F., and Rosenhahn, B. (2025). Awesome Neural Cellular Automata. `https://github.com/dwoiwode/awesome-neural-cellular-automata`.