Growing 3D Artefacts and Functional Machines with Neural Cellular Automata

Shyam Sudhakaran[1], Djordje Grbic[1], Siyan Li[1], Adam Katona[2]
Elias Najarro[1], Claire Glanois[3], Sebastian Risi[1]

[1] IT University of Copenhagen, [2] University of York, [3] Shanghai University
shyamsnair@protonmail.com, djgr@itu.dk, lisiyansylvia@gmail.com, ak1774@york.ac.uk, sebr@itu.dk

Abstract

Neural Cellular Automata (NCAs) have been proven effective in simulating morphogenetic processes, the continuous construction of complex structures from very few starting cells. Recent developments in NCAs lie in the 2D domain, namely reconstructing target images from a single pixel or infinitely growing 2D textures. In this work, we propose an extension of NCAs to 3D, utilizing 3D convolutions in the proposed neural network architecture. Minecraft is selected as the environment for our automaton since it allows the generation of both static structures and moving machines. We show that despite their simplicity, NCAs are capable of growing complex entities such as castles, apartment blocks, and trees, some of which are composed of over 3,000 blocks. Additionally, when trained for regeneration, the system is able to regrow parts of simple functional machines, significantly expanding the capabilities of simulated morphogenetic systems. The code for the experiment in this paper can be found at: `https://github.com/real-itu/3d-artefacts-nca`.

Introduction

Life begins as an embryo. From the single cell, the interacting forces of morphogenesis and cell growth then create tissues and organs, stacking them together into a functional organism. Morphogenesis, the shape organization of cell populations, has been a focus in regenerative medicine for tissue engineering and organ assembly (Joshi and Davidson, 2012; Kinney et al., 2014; Nakashima and Reddi, 2003; Hubbell, 2003). These cell populations operate on a local level, communicating information to immediate neighbors. Despite relying solely on local dependency, cells are able to form complex biological entities, consisting of various organs and the connections between them. This local dependency also allows for very useful properties, such as growth and regeneration from a small set of cells.

One approach for understanding morphogenes is via emulating local interactions of cells in *Cellular Automata* (CA). CAs contain grids of cells iteratively updated based on cell-level rules. The specific update of a cell depends upon the states of the cell itself and the neighboring cells. Therefore, these rules can be expressed as a function of a cell's and its neighbors' states. In Conway's Game of Life (Games, 1970), a less complex case, this function only counts the neighboring alive and dead cells. When equations alone are unable to encapsulate the desired update rules, Neural Networks are often introduced to replace the function. Models resulting from such substitutions are referred to as *Neural Cellular Automata* (NCA).

The sophisticated nature of morphogenesis makes NCAs a promising modeling approach. Works on simulating morphogenesis with NCAs have primarily focused on easily-replicable artefacts, such as pixelated emojis (Mordvintsev et al., 2020) and 2D images (Ruiz et al., 2021). Recent progress has also demonstrated its capability in generating high-fidelity 3D shapes (Zhang et al., 2021) and regenerating damaged morphology (Horibe et al., 2021). However, significant gaps still exist between the current state-of-the-art and real-life applications for morphogenesis simulation.

To improve the generalizability of NCAs to real-world applications, we propose an extension of Mordvintsev et al. (2020) to facilitate complex structure generation in three dimensions. Specifically, we develop a 3D NCA for generating 3D Minecraft structures and functional robots (Figure 1).

We train out NCA to grow complex structures with up to 3,584 blocks and 50 unique block types through a series of reconstruction tasks on Minecraft designs. Minecraft is a good test domain for our approach because it allows the creation of both static (e.g. an oak tree) and dynamic (e.g. a simple flying machine) structures. The results show that, in most cases, NCAs scale to 3D effectively, able to generate complex entities from a single cell. The NCAs also hold the same "regenerative" properties as their counterparts in 2D.

The main contributions of our work include: (1) An extension of NCA in 3D on multi-class voxels; (2) A cellular automation for generating Minecraft structures of varying complexity, as well as moving functional robots. In addition, we develop a particular loss function that should be generally helpful in domains that require an NCA to deal with varying discrete block types.
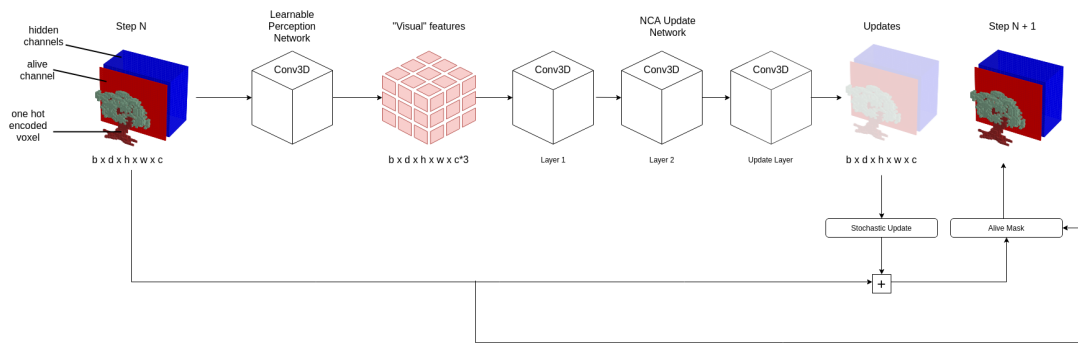
Figure 1: 3D Neural Cellular Automata update rule.

Related work

Cellular Automata

Originally proposed in 1940s (Neumann et al., 1966), Cellular Automata mimic developmental processes in multi-cell organisms including morphogenesis and irregular growth. The architecture contains a grid of similarly structured cells, which are updated periodically in discrete time steps. At every time step, the status of each cell can be represented as a state, which is then transitioned into the next state per the update rule. The specific transition depends on the current state of the cell and the neighboring cells. Despite individual cells being simplistic, the entire CA is capable of complex behaviors. Cellular Automata have been speculated to aid understanding of biological pattern formations (Wolfram, 1984), modeling of lattice-based physical systems in percolation and nucleation (Vichniac, 1984), and synthesis of artificial life (Langton, 1986).

Neural Cellular Automata

CAs that replace the update rules with a neural network are referred to as *Neural Cellular Automata* (NCA). Li and Yeh (2002) models development of multiple land use utilizing a densely-connected neural network to simulate conversion probabilities between different land types. Nichele et al. (2017) replaces the update process of CAs with a continuously evolved *Compositional Pattern Producing Network* (CPPN) to replicate simple 2D patterns; the network maps neighboring cell states to the next state for the central cell.

The primary inspiration for our work is Mordvintsev et al. (2020), where a NCA learns to generate specific 2D images starting from a single pixel. Every cell within the CA encodes cell state information using a continuous size-16 state vector, representing the RGB values, whether a cell is alive, and hidden channels for simulating local morphogens concentrations. Sobel filters estimate the partial derivatives of cell state channels, which are then concatenated with individual cell state vectors to form the "perception vector" of the cell. This perception is fed into a series of neural layers to evoke a potential update of the cell. Sharing the neural network layers throughout the CA enforces a universal, differentiable update rule. Trained via supervised learning, the model can be driven towards a goal using a reconstruction loss, more specifically the error between the target image pixels and those that were generated. The work also includes a "sample pool", which helps relieve issues with "catastrophic forgetting". Additional operations include stochastic update of cells to imitate independent development of individual cells, as well as alive-masking to prevent participation of empty ("dead") cells in the update process.

Ruiz et al. (2021) extends upon Mordvintsev et al. (2020) by learning NCA's manifold, where each point encode a different update rule corresponding to a different image. The weights of the neural NCA's layers are functions of the target image, after passage through an encoder. Although improving generalizability, the resulting model is still constrained to 2D images.

Zhang et al. (2021) proposes a Generative Neural Cellular Automata for generating diverse 3D structures from an arbitrary starting shape. Instead of pre-defined, the update rules are neural network outputs from a Markov Chain. As Ruiz et al. (2021), this enables the model to generate multiple structures, contrary to the typical NCA, which is only specific to one target. One limitation is that there is only one type of building block. They also only predict the "outer" layer of the 3d structures, improving efficiency with the added sparseness, but the tradeoff is the lack of detail within the interior of a structure.

Horibe et al. (2021) equips simulated soft robots with the ability to partially regenerate and regain locomotive capacity using two NCAs, one for generating the initial morphology and the other for restoration once the robot is damaged. The transition for a voxel (cell) is determined by a neural network using cellular states from the neighboring four voxels. A "voxel cost" is also introduced to prevent box-like morphologies. We employ a similar cost, using an intersect over union cost to steer the model to only generate cells that it needs to reconstruct the target.

Compared to previous literature, our work has higher dimensionality (3D instead of 2D) than Mordvintsev et al.

(2020) and Ruiz et al. (2021), more construction unit types than Zhang et al. (2021), and we capture all surrounding cells with a 3D convolution instead of only the four immediate neighbors as in Horibe et al. (2021).

Approach: Growing Minecraft Entities

Similar to Mordvintsev et al. (2020), we teach a Neural Cellular Automata to generate 3D entities through reconstruction tasks. We represent a Minecraft entity as a 3D grid of cells, each with a cell state vector containing channels for (1) its block type; (2) its living status; (3) its hidden states. The block type is embedded into a one-hot vector, whose entries correspond to unique block types in the structure, including the empty "air" block as its first entry. We reproduce the mechanism for cells' living status from Mordvintsev et al. (2020): Each cell state has an "alive channel" with an alpha value; a cell is "alive" when it or one of its neighbors has an alpha value greater than 0.1 and "dead" otherwise. "Dead" cells are nullified by setting the "air" block channel to 1 and everything else to 0. A cell's hidden states are represented as a continuous vector. This vector is used to carry information through steps and differentiate individual cells.

Model Architecture

As previously stated, we replace the 2D convolutions in Mordvintsev et al. (2020) with 3D ones to allow generation of 3D structures. Furthermore, we also employ a learnable perception network instead of the static one with Sobel filters. The dynamic perception net is implemented as a 3D convolutional layer with `kernel size = 3`, `stride = 1`, and `output channels = cell_state_channels * 3`.

The cell updates are retrieved by passing the visual features from the perception net into 2 configurable linear layers, which are implemented as 3D convolutions with `kernel size = 1` and `stride = 1` (See configurations: Table 2, which are passed into a non configurable update layer. Like the previous work in 2D, we use a stochastic update, multiplying half of the updates by zero, as a form of "dropout" (Srivastava et al. (2014)), which helps with overfitting. We then apply an "alive mask" to the updates, which uses the interaction with the living channel stated before. This is implemented by multiplying the updates with the outputs of a MaxPool layer and a boolean mask where 1 if > 0.1 and 0 otherwise. We initialize all models with standard normal initialization, with mean = 0 and standard deviation ranging from 0.0001 to 0.1

Training Procedure

We use a similar training process as the original work where the NCA tries to generate a target entity from a single living cell. optimizing over a reconstruction loss using supervised learning. However, because each cell can only be of a single block type, we treat the structure reconstruction task

as a multi-class classification problem, predicting the type of a given cell. Therefore, our objective becomes the minimization of the cross entropy loss between cells from the target structure and from the predictions. We utilize a combination of *LogSoftmax* and *NLLLoss*, provided by Pytorch's `cross_entropy` method. With just this loss, we noticed that the training performance was unstable and the model demonstrated a preference for "air" blocks. We hypothesize this to be due to an imbalance in our training data. "Air" blocks often take up the majorities of our selected structures, and these blocks can be either alive or dead. Such an imbalance might result in the model over-predicting "air" blocks. This may also be alleviated by removing padding, but doing so causes training instability for most cases.

To address this, we divide up the loss calculations between cells that are classified as non-"air" blocks and the cells that are classified as "air" blocks. We also add an *Intersect Over Union (IOU)* cost, which measures the absolute difference between the non-"air" blocks in the target and the generated entities. We anticipate this additional cost component to impose structural constraints on generated structures and improve regeneration accuracy.

$$CE(\hat{y}_{i,c}, y_i) = -I(y_i = c)\hat{y}_{i,c} + \log \sum_{j=1}^{M} (exp(\hat{y}_{i,j})) \quad (1)$$

$$Inter(\hat{y}, y) = \sum (I(\operatorname*{argmax}_c \hat{y}_c > 1) \cap I(y > 1)) \quad (2)$$

$$Uni(\hat{y}, y) = \sum (I(\operatorname*{argmax}_c \hat{y}_c > 1) \cup I(y > 1)) \quad (3)$$

$$IOU(\hat{y}, y) = (Uni(\hat{y}, y) - Inter(\hat{y}, y))/(Uni(\hat{y}, y) + 1e^{-8}) \quad (4)$$

$$Loss = \frac{1}{N} \sum_{i=1}^{N} (\sum_{c=1}^{M} (CE(\hat{y}_{i,c}, c)) + IOU(\hat{y}_i, y_i)) \quad (5)$$

When a generated entity's non-"air" blocks are identical to its respective target, the IOU cost is 0. The gradients for this loss function are accumulated over time, with multiple forward passes ranging from anywhere between 48 to 64 steps, see Table 2.

Consistent with Mordvintsev et al. (2020), the system becomes unstable when generating samples for more steps than what it was trained for. To alleviate this, we include a "sample pool" of size 32, which is updated by the outputs of each batch. The pool is initialized with a set of "seed states", which are composed of a single living cell. Every training iteration a batch of samples is taken from the pool. The best-performing sample from each batch (the sample that has the 'least loss compared to the target) is replaced with a single living cell to prevent catastrophic forgetting. The sample

Minecraft Entity Details				
Entity	No. Unique Blocks	No. Non Air Blocks	Padding	Padded Size (W×D×H)
Village House	10	84	Yes	10×10×10
Blacksmith	17	280	Yes	10× 10×10
MiniCastle	35	1253	Yes	20×20×20
Jungle Temple	15	1283	Yes	20×20×20
Tree	3	1622	Yes	30x30x30
Apartment Block	50	3136	No	18×18×23
Cathedral	23	3584	No	33×27×31
Flying Machine	6	8	Yes	10×10×10
Caterpillar	7	137	No	8×27×6

Table 1: Minecraft Entity Details

Experiment Hyperparameter Details					
Entity	No. hidden	Min & Max steps	NCA Update Net layer 1 & 2 no. channels	layer init. stdev	lr
Village House	10	(48, 64)	(32, 32)	0.1	0.0002
Blacksmith	10	(48, 64)	(32, 32)	0.1	0.002
MiniCastle	10	(48, 64)	(32, 32)	0.1	0.002
Jungle Temple	12	(48,64)	(64, 64)	0.1	0.002
Tree	12	(64,64)	(64, 64)	0.1	0.002
Apartment Block	12	(64, 65)	(64, 64)	0.1	0.002
Cathedral	12	(50,51)	(64, 64)	0.2	0.002
Flying Machine	10	(48, 64)	(32, 32)	0.001	0.002
Caterpillar	12	(48, 64)	(64, 64)	0.02	0.002

Table 2: Experiment Hyperparameter Details

pool imitates experience replay. The outputs of the batch are obtained by passing the batch into the NCA for a random number of forward steps, parameterized by "min_steps" and "max_steps". At the end of the training iteration, the sample pool is updated with the outputs.

EvoCraft environment

To interact with Minecraft, we use the EvoCraft API (Grbic et al., 2020). EvoCraft includes a Python programmable interface that allows reading and writing blocks. The API provides a connection over the local network to a running Minecraft server. The interface contains an enumeration of supported block types and block orientations.

The three available functions are *spawnBlocks*, *readCube*, and *fillCube*. *spawnBlocks* takes a list of *Block* objects and sends them server side to be spawned in the Minecraft world. A *Block* object contains the block type, block orientation, and a coordinate where the block should be spawned. Similarly, the *readCube* function takes a bounding box corner coordinates and returns a list of Block objects representing the blocks contained within the box. This function reads the current state of the world with regards to block positions and to follow if the blocks are moving through the world (redstone moving contraptions). Finally, the *fillCube* function takes a bounding box corner coordinates and a block type. The result is the box filled with blocks of that type. These three methods allow us to implement a cellular automaton by rearranging the blocks depending on the current state.

The API uses gRPC protocol to send/receive messages and the interface definition (e.g. Listing 1). Most of the block types have no functional purpose except aesthetics, while some have unique functional purposes and interactions with other blocks. Blocks like "clay", "terracotta", "stone", "glass", "dirt" are inert but have different textures and colors making them useful to build aesthetically pleasing structures. Blocks like "lava" and "concrete powder" can interact with "water" blocks to produce "obsidian" blocks and "concrete" blocks. Similarly, "lava" and "TNT" blocks can destroy if they come into contact with or explode near to nearby blocks. Most interestingly for our purposes, there are "piston" and "sticky piston" blocks that push/pull other blocks if they are adjacent to a "redstone" block. "Slime" blocks glue to the adjacent blocks and pull them if the "slime" block gets pushed/pulled by a "piston" block. The existence of these blocks allows players to build large and dynamic moving structures. The unique interactions between the blocks provides ways for the Minecraft world to be dynamic independent of the rules of the cellular automa-
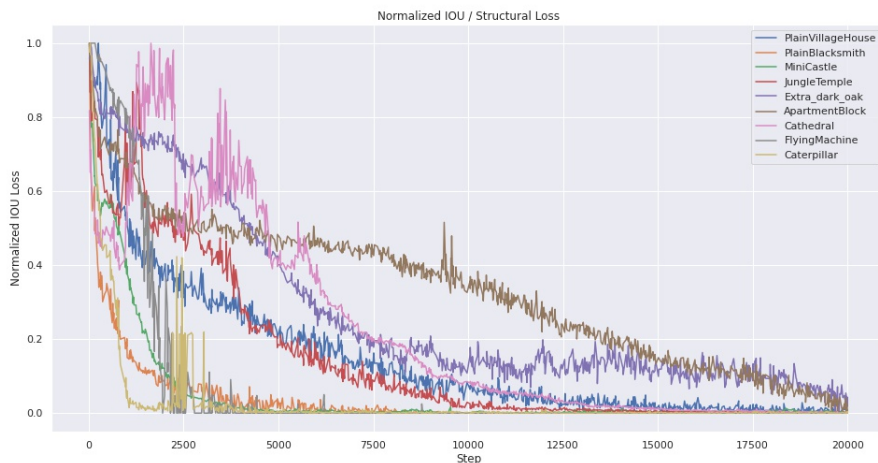
(a) Normalized IOU / Structural Loss
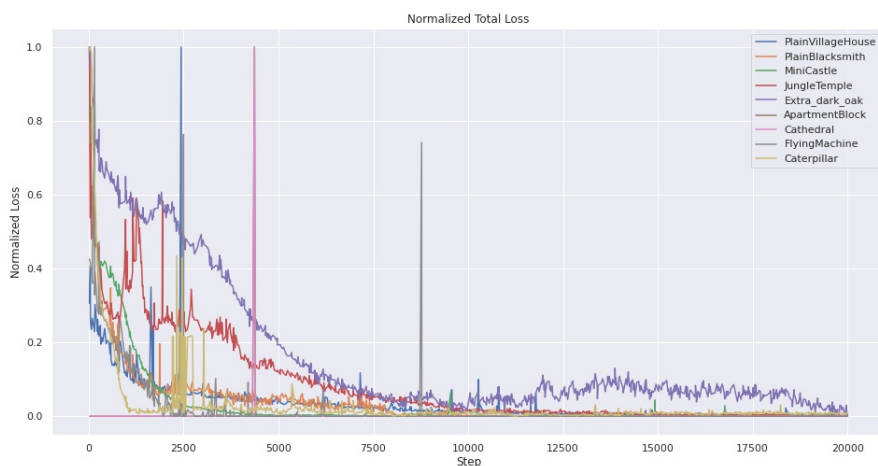
(b) Normalized Total Loss

Figure 2: Normalized training curves for total loss and IOU / Structural loss

ton operating beyond. For the full list of supported blocks consult (Grbic et al., 2020) appendix.

```
1  service MinecraftService {
2      /** Spawn multiple blocks. */
3      rpc spawnBlocks (Blocks) returns (↩
       Empty)
4
5      /** Return all blocks in a cube */
6      rpc readCube (Cube) returns (Blocks)
7
8      /** Fill a cube with a block type */
9      rpc fillCube (FillCubeRequest) ↩
       returns (Empty)
10 }
```

Listing 1: The EvoCraft gRPC API definition. For brevity the headers and message definitions are omitted.

Dataset

As an inspiration for the structures we trained the CA to reproduce, we used several publicly available artefacts built by the Minecraft community. Namely we used structures found on `www.planetminecraft.com/project/111-structure_block-saves-to-share-nbt` website. This library contains files in the NBT format, which is a JSON-like object that contains block types, positions, etc; everything needed to serialize a MineCraft build for later use. We used EvoCraft API to spawn the NBT files into the running Minecraft server. We added a structure called "apartment complex" from YouTuber Pixlriffs website. Finally, for the moving "redstone" builds we took a caterpillar found in this video

(a) Target MiniCastle   **(b)** MiniCastle step – 10   **(c)** MiniCastle step – 25   **(d)** MiniCastle step – 50   **(e)** MiniCastle step – 100

(f) Target JungleTemple   **(g)** JungleTemple step – 10   **(h)** JungleTemple step – 25   **(i)** JungleTemple step – 50   **(j)** JungleTemple step – 100

(k) Target Tree   **(l)** Tree step – 10   **(m)** Tree step – 25   **(n)** Tree step – 50   **(o)** Tree step – 100

(p) Target ApartmentBlock   **(q)** ApartmentBlock step – 10   **(r)** ApartmentBlock step – 25   **(s)** ApartmentBlock step – 50   **(t)** ApartmentBlock step – 100

Figure 3: Structural generation over time

(a) Target Cathedral   **(b)** Cathedral step – 10   **(c)** Cathedral step – 25   **(d)** Cathedral step – 50   **(e)** Cathedral step – 100
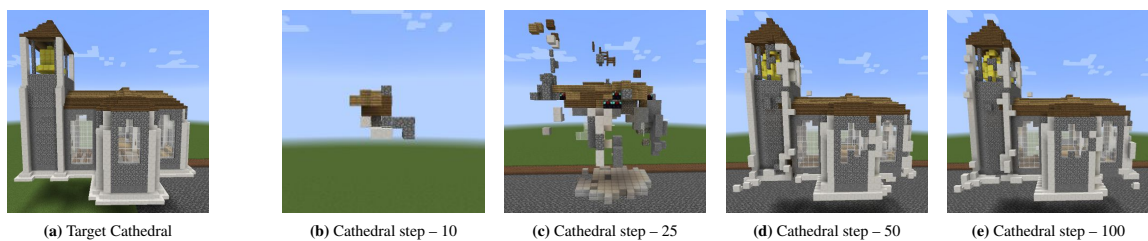
Figure 4: Cathedral generation failure case

www.youtube.com/watch?v=wNqzwAPdFbs as an inspiration for our caterpillar build (Figure 5).

Experimental Details

Our model is evaluated on a series of regeneration tasks for both static structures and dynamic functional machines, their relevant details are recorded in Table 1. We initialize the NCA's convolutional layers using standard normal initial-ization and zero bias. We run each experiment with a batch size of 5 for 20k steps, with early stoppage when losses are < 0.005. Network size, number of steps, and other relevant hyperparameters are recorded in Table 2 Some entities, such as functional robots, have redstone and pistons which are orientation sensitive components. Because the NCA model only predicts block type and not orientation, we set these orientations by default to NORTH.
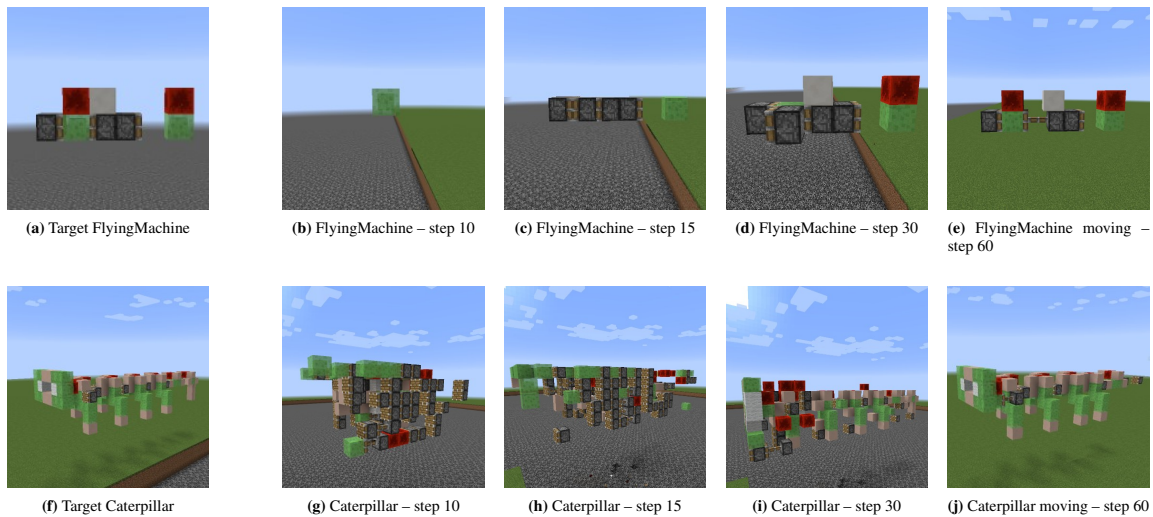
| (a) Target FlyingMachine | (b) FlyingMachine – step 10 | (c) FlyingMachine – step 15 | (d) FlyingMachine – step 30 | (e) FlyingMachine moving – step 60 |

| (f) Target Caterpillar | (g) Caterpillar – step 10 | (h) Caterpillar – step 15 | (i) Caterpillar – step 30 | (j) Caterpillar moving – step 60 |

Figure 5: Functional machine generation over time

Experiments were run using a 2080ti GPU and a Titan RTX GPU. On average, smaller structures took 1-3 seconds each epoch, and bigger structures took around 15-24 seconds each epoch. The full library used for training the CA will be made available soon.

Results

Growing static structures

Despite having simple and local individual updates, our NCA is capable of generating complex structures, some almost identical to the targets. It also scales well to increasing number of individual block types. See Figure 3 and videos at `https://youtu.be/-EzztzKoPeo` for results.

The NCA, contrary to the work presented in Zhang et al. (2021), is able to generate diverse and complex interiors like those of ApartmentBlock and JungleTemple. Surprising to us, in the case of JungleTemple, the NCA even generates a functional arrow trap, which uses a working redstone circuit.
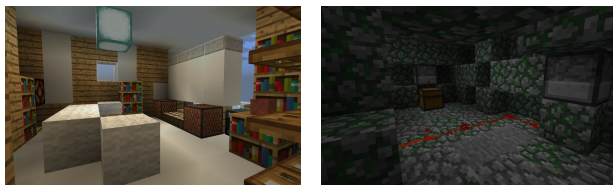
| (a) ApartmentBlock: tv living room | (b) JungleTemple: working redstone circuit |

The NCA's ability to reconstruct structures is proficient with respect to a wide range of size and number of unique blocks, as seen in the decreasing training curves in Figure 2. However, larger entities, such as the cathedral, are more challenging to reconstruct than their smaller counterparts, as the model is often stuck in local minima and takes much
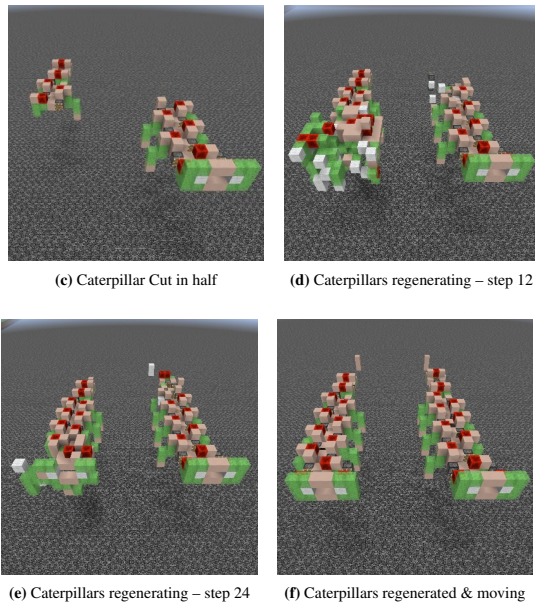
longer to train. Even though the cathedral model achieved a lower loss than some of the other entities, the generated structure has many random artifacts and is less natural than the others (Figure 4). Entities that are more random in nature are also harder to generate. This is especially apparent in the oak tree, where the loss over time is very sporadic. However, unlike the cathedral, the trained model generates very natural trees that resemble the target very closely, but this may also be due to the structure's inherent randomness.

Growing functional machines

The NCA was also successful in replicating functional machines (Figure 5). As a result, after 60 steps each trained NCA generated a working machine, even a complex one like the caterpillar. Even though all the different NCAs shared the same initial seed, we observed different growth patterns: the flying machine started generating small cells then gradually expanding out into the final functional state. The caterpillar, however, started growing wildly in the earlier steps, then culling cells to form the final functional state.

Regenerative Properties

One of the attractive properties of NCAs are their similarity to the natural process of developing complex organisms from a single cell only using few parameters encoding local update rules. Beside growing complex structures from a single cell, these local update rules also allow for regenerating or repairing damage. NCAs were shown to be able to regrow parts of their structure when damaged, even when they are not specifically trained for regeneration (Mordvintsev et al., 2020). This ability can be enhanced further by specifically training the model to be able to recover from damaged states, by providing damaged variants of the samples from the sample pool during training.

(c) Caterpillar Cut in half   **(d)** Caterpillars regenerating – step 12

(e) Caterpillars regenerating – step 24   **(f)** Caterpillars regenerated & moving

Figure 6: Caterpillar's regenerating after being sliced in half.

(a) Tree cut in half   **(b)** Tree - step 96
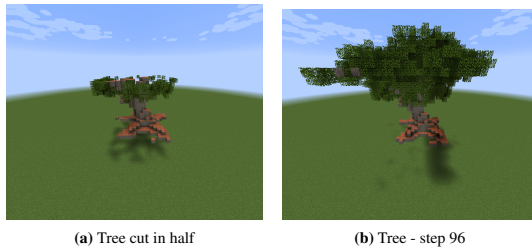
Figure 7: Tree regenerating after cut in half, without training to be regenerated.

The metric we used for evaluating regeneration capability is the ratio of regenerated blocks after the structure was cut in half. We compare a model trained without regeneration and a model trained with regeneration on the task of regenerating a Caterpillar (functional machine). When the model was trained for regeneration, it was able to regrow 99% of the blocks in case of the Caterpillar when cut in half (Figure 6). When the model was not trained for regeneration, it only regenerated 30% of the Caterpillar.

Even when the NCA is not trained for regeneration, it still has some ability to regenerate from damage for certain structures such as the tree (Figure 7), while less so on functional machines. However it is clear that a model trained for regeneration performs much better, while also being able to grow entities at the same level as its counterpart.

Discussion and Future Work

In this paper, we propose an improved Neural Cellular Automata from Mordvintsev et al. (2020) that reproduces 3D Minecraft structures and functional robots with high accu-

racy. We modify the original automata to include 3D convolutional layers with learnable parameters, cell state vector representations compatibile with the Minecraft voxel environment, and a combined loss function with an emphasis on structural resemblance to the target. To assess our model's efficacy, we teach our NCA to generate a set of Minecraft structures, including both static architectures and simple functional robots. The NCA is able to generate structures that are, in some cases, identical to their respective targets. In addition, these generations are robust to perturbations and damage, able to recover blocks that are destroyed.

Although the NCA demonstrates promising results in complex 3D structure regeneration, there is still room for improvement. Our approach is not robust enough, since the model has to be trained separately for every new entity, making the model parameters unsharable between tasks. This may be relieved by adding a global one-hot vector for all possible block types. Such an improved embedding can be extended multiple entities. Another approach would be to adopt the autoencoder from Ruiz et al. (2021) to encode entities' structural information.

Another potential improvement would be to shift from supervised to reinforcement learning. Reinforcement learning have demonstrated potency in complex environments (Lillicrap et al., 2015; Vinyals et al., 2019; Mnih et al., 2013), even with sparse rewards Vecerik et al. (2017). We can take advantage of SOTA reinforcement learning algorithms to augment the iterative training process of the NCA.

Training only on reconstruction tasks would not teach the NCA the inner workings for a machine to be functional. A potential approach to facilitate understanding of mechanisms is via additional contextual costs (e.g. evaluating a reconstruction of a flying machine by the distance travelled by the generation). This would allow for a more diverse set of generated machines.

We also look forward to attempting to generate increasingly complex and larger machines, such as a functional redstone computer. This would require the model to become much more efficient and utilize multi-gpu training procedures such as Rajbhandari et al. (2020)

Conclusion

Our paper broadens the scope of modalities for tasks suitable for Neural Cellular Automata. Although the work presented is in a simplified 3D environment, the model's capability of generating increasingly complex 3D entities brings us one step closer to real-life, self-organizing, and regenerative physical artefacts. Self-organizing capabilities in 3D can lead to more scalable, even growable, physical systems. Furthermore, regenerative abilities in NCA may inspire self-repairing structures, including buildings and artificial organs. Through investigating and expanding the capabilities of Neural Cellular Automata, we hope to facilitate further developments in various fields of engineering.

References

Games, M. (1970). The fantastic combinations of john conway's new solitaire game "life" by martin gardner. *Scientific American*, 223:120–123.

Grbic, D., Palm, R. B., Najarro, E., Glanois, C., and Risi, S. (2020). Evocraft: A new challenge for open-endedness. *arXiv preprint arXiv:2012.04751*.

Horibe, K., Walker, K., and Risi, S. (2021). Regenerating soft robots through neural cellular automata. *arXiv preprint arXiv:2102.02579*.

Hubbell, J. A. (2003). Materials as morphogenetic guides in tissue engineering. *Current opinion in biotechnology*, 14(5):551–558.

Joshi, S. D. and Davidson, L. A. (2012). Epithelial machines of morphogenesis and their potential application in organ assembly and tissue engineering. *Biomechanics and modeling in mechanobiology*, 11(8):1109–1121.

Kinney, M. A., Hookway, T. A., Wang, Y., and McDevitt, T. C. (2014). Engineering three-dimensional stem cell morphogenesis for the development of tissue models and scalable regenerative therapeutics. *Annals of biomedical engineering*, 42(2):352–367.

Langton, C. G. (1986). Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena*, 22(1):120–149. Proceedings of the Fifth Annual International Conference.

Li, X. and Yeh, A. G.-O. (2002). Neural-network-based cellular automata for simulating multiple land use changes using gis. *International Journal of Geographical Information Science*, 16(4):323–343.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mordvintsev, A., Randazzo, E., Niklasson, E., and Levin, M. (2020). Growing neural cellular automata. *Distill*. https://distill.pub/2020/growing-ca.

Nakashima, M. and Reddi, A. H. (2003). The application of bone morphogenetic proteins to dental tissue engineering. *Nature biotechnology*, 21(9):1025–1032.

Neumann, J., Burks, A. W., et al. (1966). *Theory of self-reproducing automata*, volume 1102024. University of Illinois press Urbana.

Nichele, S., Ose, M. B., Risi, S., and Tufte, G. (2017). Ca-neat: evolved compositional pattern producing networks for cellular automata morphogenesis and replication. *IEEE Transactions on Cognitive and Developmental Systems*, 10(3):687–700.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. (2020). Zero: Memory optimizations toward training trillion parameter models.

Ruiz, A. H., Vilalta, A., and Moreno-Noguer, F. (2021). Neural cellular automata manifold.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., Heess, N., Rothörl, T., Lampe, T., and Riedmiller, M. (2017). Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*.

Vichniac, G. Y. (1984). Simulating physics with cellular automata. *Physica D: Nonlinear Phenomena*, 10(1):96–116.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

Wolfram, S. (1984). Cellular automata as models of complexity. *Nature*, 311(5985):419–424.

Zhang, D., Choi, C., Kim, J., and Kim, Y. (2021). Learning to generate 3d shapes with generative cellular automata. In *ICLR 2021*.