Supplementary material for "Learning Graph Cellular Automata"

Daniele Grattarola
Università della Svizzera italiana
grattd@usi.ch

Lorenzo Livi
University of Manitoba

Cesare Alippi
Università della Svizzera italiana
Politecnico di Milano

A Experimental details

A.1 Hardware and software

We ran all experiments on an NVIDIA Titan V GPU with 12GB of video memory. We implemented all models in Python 3.7 and TensorFlow 2.4 [1]. The code to reproduce our experiments is available (with documentation) at the following URL: `https://github.com/danielegrattarola/GNCA`.

A.2 Architecture of the GNCA

The architecture of the GNCA can be summarised as follows:

$$\begin{aligned}
&\mathbf{h}_i \leftarrow \mathrm{MLP}_{pre}(\mathbf{h}_i);\\
&\mathbf{h}_i \leftarrow \mathbf{h}_i \Big\| \sum_{j \in \mathcal{N}(i)} \mathrm{ReLU}(\mathbf{W}\mathbf{h}_j + \mathbf{b});\\
&\mathbf{h}_i \leftarrow \mathrm{MLP}_{post}(\mathbf{h}_i).
\end{aligned}$$

$\mathrm{MLP}_{pre|post}$ are two-layer multi-layer perceptrons with $d_{hidden} = 256$ hidden units and ReLU activation. The pre-processing MLP has 256 output units while the post-processing MLP has a number of units equal to the size of the state. The final activation of the post-processing MLP depends on the state space: we use a sigmoid for binary state spaces, hyperbolic tangent if the states are bounded between [-1, 1], and no activation otherwise. The message-passing layer maps the node features to a vector of size $2 \cdot d_{hidden}$, *i.e.*, $\mathbf{W} \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$ and $\mathbf{b} \in \mathbb{R}^{d_{hidden}}$.

The architecture and hyperparameters are motivated by the results of You et al. [2].

A.3 GNCA on Voronoi tessellation

For each experiment, we generate a random Voronoi tessellation starting from 1000 random points sampled uniformly in $[0, 1] \times [0, 1]$, and making sure that we don't sample collinear points. We compute graph $\mathcal{G}$ as the Delaunay triangulation of the points (*i.e.*, the graph describing the adjacency of Voronoi cells).

Training To train the model, we sample batches of random binary states $[\mathbf{S}^{(1)}, \dots, \mathbf{S}^{(K)}]$, with $\mathbf{S}_i \in \{0, 1\}^n$ and $K = 32$. We train the model on 1000 such batches using the Adam optimiser [3] with learning rate of 0.01. At every step, we also generate a random validation batch to monitor the performance of the model. We train the model by minimising the negative log-likelihood between the true successor states $\tau(\mathbf{S}^{(k)})$ and the predicted next states $\tau_\theta(\mathbf{S}^{(k)})$.

Entropies To compute the entropy measures, we generate a batch of states that we keep fixed throughout the training. At each step, we evolve the GNCA for 1000 steps autonomously, *i.e.*, feeding

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

its predictions (rounded to 0 or 1) back as input. We then compute the two measures from the state trajectory of each individual cell and report the average measures over all cells. We report the values only for the first 100 training steps for computational reasons.

Minimal exact implementation To identify a possible set of weights that implement the transition rule of the Voronoi GCA, we generate a dataset of 198 transitions of the form $([\mathbf{s}_i, \rho_i], \mathbf{s}'_i)$, for all possible combinations of $\mathbf{s}_i = 0, 1$ and $\rho_i = 0.01, \ldots, 0.99$, where $\mathbf{s}'_i$ is computed with the true transition rule. We then train a two-layer MLP with two neurons in the first layer and one neuron in the second layer. The MLP has hidden ReLU activation and sigmoid output. We apply L_2 weight decay with a coefficient of 0.001 to all weights. We initialise the weights uniformly in the $[-1, 1]$ interval to improve the chances of convergence. We train the model by minimising the mean squared error between the prediction and the target $\mathbf{s}'_i$, using Adam with learning rate 0.001 and batch size 198 (*i.e.*, the entire dataset). We train the model for 100000 epochs and we reduce the learning rate by a factor of 0.1 if the training loss does not improve by at least 10^{-8} in 10000 steps. We trained multiple models until we obtained 100% training accuracy. Finally, we inspected the weights of the model and rounded the values to the lowest possible precision that still preserved 100% accuracy. We report these final values in the main text.

A.4 GNCA for agent-based modelling

Our implementation of the Boids algorithm is inspired by the original implementation [4], with additional measures to facilitate the creation of flocks.

Each boid is a vector $[p_x, p_y, v_x, v_y]$ where $\mathbf{p}_i = [p_x, p_y]$ represent the position and $\mathbf{v}_i = [v_x, v_y]$ the velocity relative to the boid. The simulation happens in the square $[-1, 1] \times [-1, 1]$. We compute graph $\mathcal{G}$ at each step of the algorithm by connecting those boids that are within a radius of 0.15 from each other.

At each step of the algorithm, we apply the following transformations synchronously to all boids to compute the change in each boid's velocity:

1. If a boid is within a radius of 0.2 from the bounding box, a force is applied to steer the boid towards the centre of the box;
2. If the distance between a boid and its neighbours is lower than a user-defined threshold, a separation force is applied to steer the boid away from these excessively close neighbours. We set this threshold to 0.015;
3. An alignment force is applied to match the velocity of each boid to the average velocity of its neighbours. The force is scaled by a factor of $1/8$;
4. A cohesion force is applied to bring the position of each boid closer to the average position of its neighbours. The force is scaled by a factor of $1/100$;
5. For each boid, the resulting force is summed to the current velocity;
6. A speed limit of 0.01 per step is enforced, as well as a maximum turn angle of 5 degrees relative to the previous direction;
7. The position of each boid is updated according to the new velocity.

For all details, we refer the reader to the code: `https://github.com/danielegrattarola/GNCA`.

Data To generate the training, validation, and test sets, we initialise the positions and velocities of the boids uniformly in $[-1, 1]$. We instantiate 100 boids and evolve the system for 500 steps to obtain the trajectories. We generate 300 trajectories for training, 30 for validation, and 30 for testing.

Architectural changes As described above, the transition function of the boids algorithm maps the current state and position $[p_x, p_y, v_x, v_y]$ to the updated velocity $[v'_x, v'_y]$, which we then use to update the position to $[p_x + v'_x, p_y + v'_y]$. For this reason, we configure the GNCA to have a similar behaviour: instead of training the GNCA to predict the full state $[p'_x, p'_y, v'_x, v'_y]$, we train the model to only predict the new velocity, and then we compute the updated state by summing the predicted velocity to the old position. Finally, we obtain the new state by concatenating the updated positions and velocities of each boid, and the we compute the loss between the full input and output states. By

doing this, we greatly improve the performance of the GNCA and obtain a behaviour similar to the true GCA. A second architectural modification that we include for this experiment is related to the separation force in the algorithm. Since this force is computed from the distance between the boids and their excessively close neighbours, we include a message-passing layer based on the EdgeConv model of Wang et al. [5]:

$$\mathbf{h}'_i = \sum_{j \in \mathcal{N}(i)} \mathrm{MLP}([\mathbf{h}_i - \mathbf{h}_j, \|\mathbf{h}_i - \mathbf{h}_j\|])$$

where MLP is a two-layer perceptron with hidden ReLU activation and linear output.

The output of this additional message-passing layer is added to the output of the base GNCA architecture to compute the updated velocity of the boids.

Training We train the model to minimise the mean squared error between the prediction and the true next state, using Adam with initial learning rate of 0.001 and batch size 30. We decease the learning rate by a factor of 0.1 if the validation loss does not improve for 10 consecutive steps. We train the model to convergence, by monitoring the validation loss with a patience of 20 epochs.

Complexity measures Sample entropy (SampEn) is a measure of complexity for time series. Let $x(t) = \{x_1, \ldots, x_t, \ldots, x_N\}$ be a time series and $X_m(i) = \{x(i), \ldots, x(i+m-1)\}$ a subsequence of length m. Consider also a distance function between subsequences $d(X_m(i), X_m(j)) \geq 0$. Define $P(m)$ as the number of subsequence pairs $X_m(i), X_m(j)$ for which $d(X_m(i), X_m(j)) < r$ for a given tolerance r and for all possible subsequences of $x(t)$. The SampEn is given by the ratio

$$\text{SampEn} = -\log \frac{P(m+1)}{P(m)}$$

for a given *embedding dimension* m.

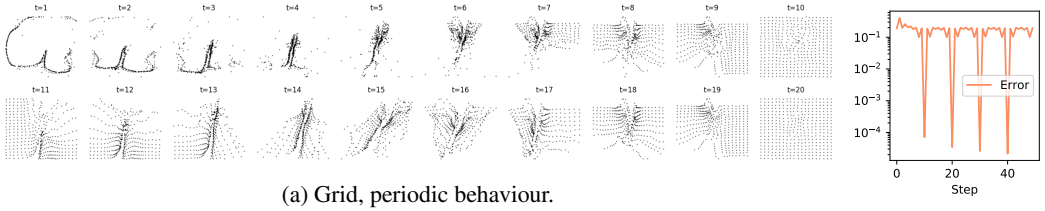
Correlation dimension (CD), instead, measures complexity using the correlation sum of the time series, *i.e.*, the fraction of overall subsequences for which $d(X_m(i), X_m(j)) < r$. Given a sufficiently large amount of data, the correlation sum $C(r)$ will tend to r^D where D is the CD.

To compute the complexity measures, we let the GNCA evolve autonomously for 1000 steps. We then compute the sample entropy and correlation dimension from the trajectories of each cell, and then average them over all cells. We use the Nolds library [6] to compute both measures. For SampEn, we consider the default $m = 2$ and $r = 0.2\sigma$ where σ is the standard deviation of the time series. To estimate the correlation dimension, we set $m = 10$ as it is suggested by the library documentation that higher values help avoid scaling error, although we could not find significant differences between different values. We also use the default strategy of the library for choosing r.

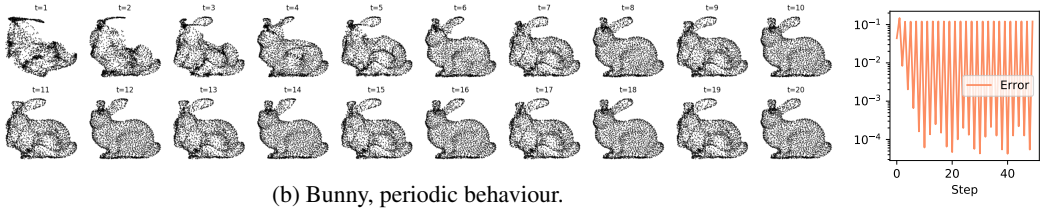
A.5 GNCA that converge to a fixed target

Training We train the model to minimise the mean squared error between the prediction and the target, using Adam with a an initial learning rate of 0.001 and batch size of 8. We divide the training in epochs of 10 batches, to have a more accurate estimate of the loss and accuracy of the model. We decrease the learning rate by a factor of 0.1 if the validation loss does not improve for 750 consecutive epochs (7500 batches). We train the model to convergence, by monitoring the validation loss with a patience of 1000 epochs (10000 batches). We also apply gradient clipping by global norm to stabilise training.

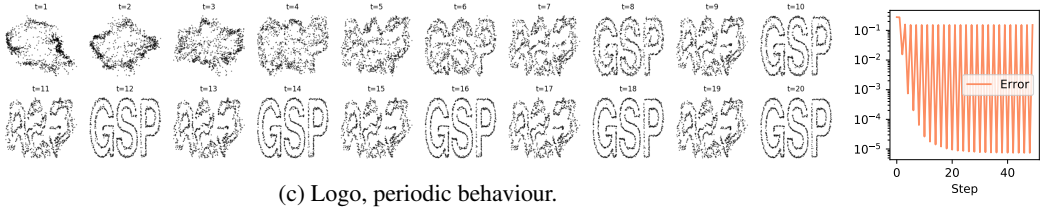
Additional results Figures 1, 2, and 3 show the trajectories and error w.r.t. the target for all combinations of graphs and t that we have considered in our experiments.



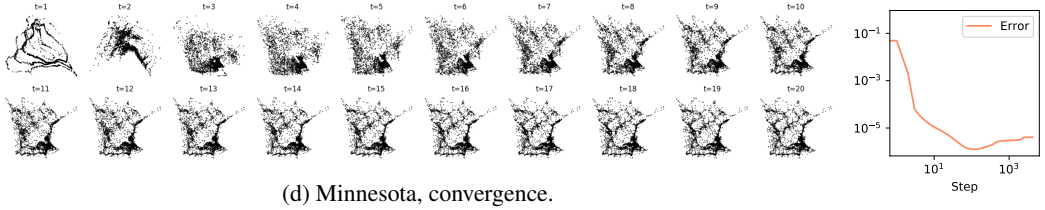
(a) Grid, periodic behaviour.



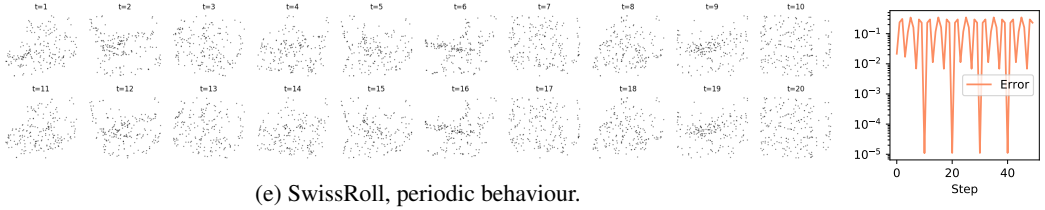
(b) Bunny, periodic behaviour.



(c) Logo, periodic behaviour.

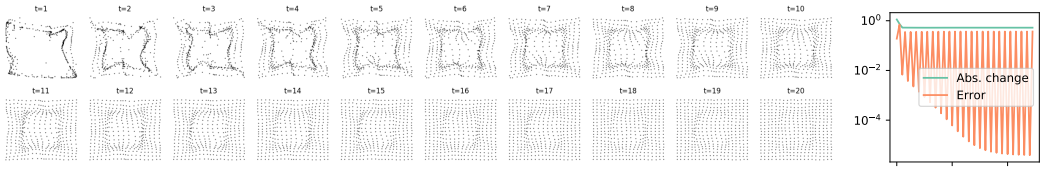


(d) Minnesota, convergence.

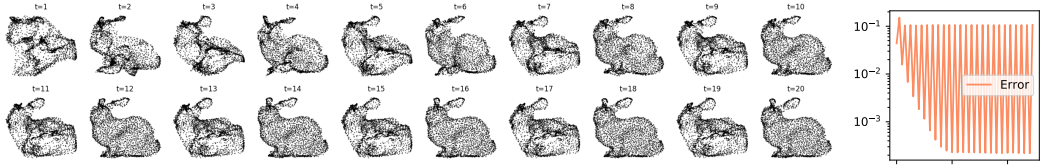


(e) SwissRoll, periodic behaviour.

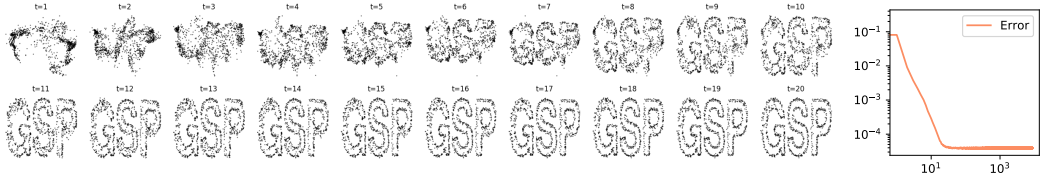
Figure 1: State trajectories of the GNCA trained with $t = 10$. We show 20 steps starting from $\bar{\mathbf{S}}$. The plots to the right show the mean squared error between the current state and the target (50 steps if the GNCA has periodic behaviour and 10^4 steps if the GNCA converges).



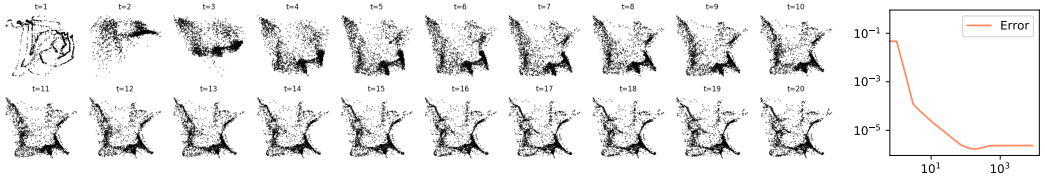
(a) Grid, periodic behaviour.



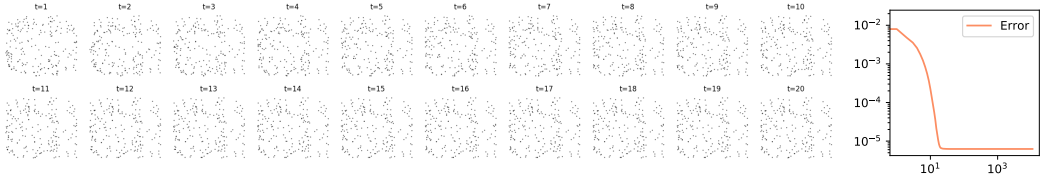
(b) Bunny, periodic behaviour.



(c) Logo, convergence.

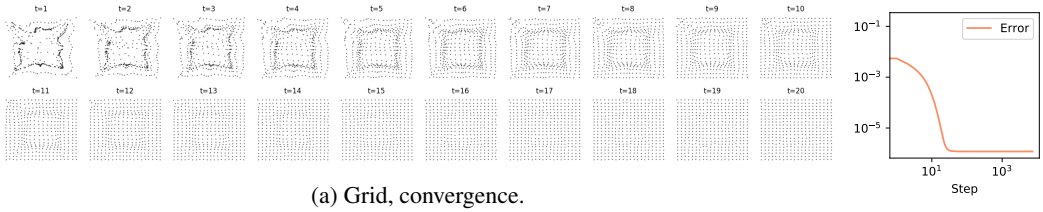


(d) Minnesota, convergence.

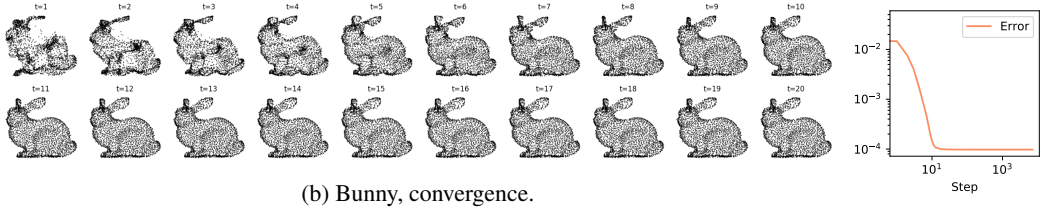


(e) SwissRoll, convergence.

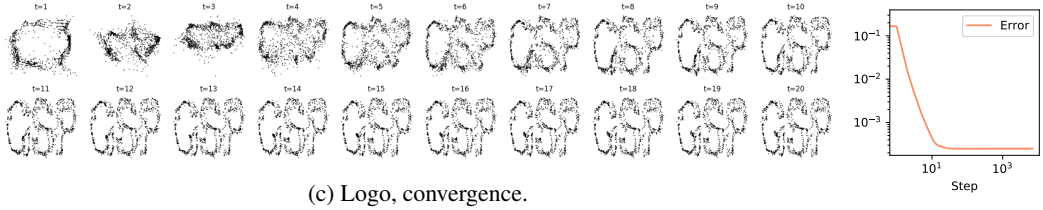
Figure 2: State trajectories of the GNCA trained with $t = 20$. We show 20 steps starting from $\bar{\mathbf{S}}$. The plots to the right show the mean squared error between the current state and the target.



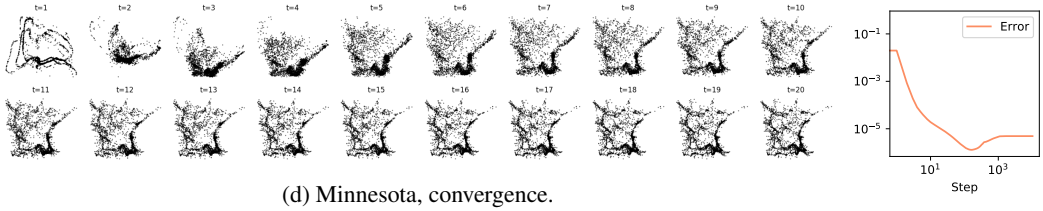
(a) Grid, convergence.



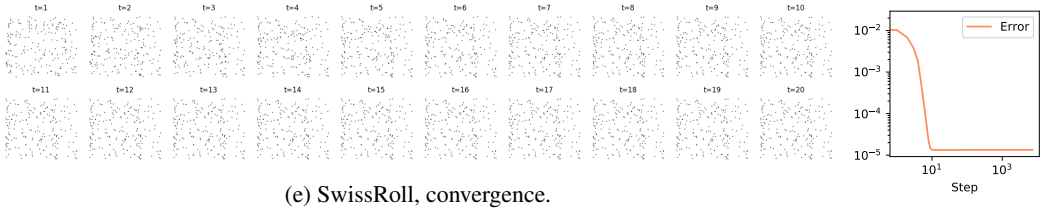
(b) Bunny, convergence.



(c) Logo, convergence.



(d) Minnesota, convergence.



(e) SwissRoll, convergence.

Figure 3: State trajectories of the GNCA trained with $t \in [10, 20]$. We show 20 steps starting from $\bar{\mathbf{S}}$. The plots to the right show the mean squared error between the current state and the target.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[2] J. You, R. Ying, and J. Leskovec, "Design space for graph neural networks," *arXiv preprint arXiv:2011.08843*, 2020.

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[4] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 25–34.

[5] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds.(2018)," *arXiv preprint arXiv:1801.07829*, 2018.

[6] C. Schölzel, "Nonlinear measures for dynamical systems," Jun. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3814723