Machine learning models to predict the evolution of Conway's Game of Life under different levels of coarse-grainings of the system

Ían Patón Vázquez[1,†] Dr. Tamás Kriváchy[1]

[1] ICFO - Institut de Ciències Fotòniques, The Barcelona Institute of Science and Technology, Castelldefels, Spain

[†] ian.patonvazquez@gmail.com

Abstract

This work investigates the capability of machine learning models to predict the evolution of Conway's Game of Life, a cellular automaton governed by simple local rules, under different levels of coarse-graining in the system. By developing simulation and dataset generation techniques, we trained multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs) to predict the state of the central cell in future generations from temporally-evolved coarse-grained grids. Despite the known challenges of training neural networks on such a system, models showed high accuracy when no coarse-graining was applied. However, predictive performance significantly degraded for low coarse-graining levels, often approaching random guessing. Nonetheless, this is theorized to be due to the learning parameters of the models. Interestingly, providing the central cell's neighbourhood sum as an input feature drastically improved accuracy, serving as a useful baseline for sanity checks. The study highlights the difficulty of extracting emergent patterns from complex systems and the importance of dataset balancing.

I. INTRODUCTION

Conway's Game of Life [1] is one of the most popular cellular automaton in the world. Starting from a grid in which each cell can be either alive or dead, each generation (one time-step) evolves according to some basic rules:

- Any live cell with fewer than two live neighbours dies of underpopulation.
- Any live cell with more than three live neighbours dies of overpopulation.
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours will come to life.

Therefore, the only requirement for the Game of Life (GoL) is the initial pattern, as we can iterate it over infinite generations. This, a priori, simple game has been shown to have many interesting patterns and figures emerging from these rules.

With the rise of popularity in machine learning, several attempts to create neural network models that successfully predict the next generation of the Game of Life given an initial distribution of alive and dead cells have been made. Nonetheless, as found by Springer and Kenyon [2], neural networks rarely converge when training them on predicting the next step of the game. The aim of this internship is to study different machine learning models and apply them to the Game of Life, paying special attention to the influence of coarse-graining the system.

Every piece of code used for the project can be found in the following *GitHub Repository*

II. GAME OF LIFE SIMULATOR

The first step of the project has been to develop a Game of Life simulator. A Python function that has a grid, i.e. a two-dimensional array (matrix), as an argument is an easy way to start. In this function, we can generate a grid that contains the neighbours' sum of each cell, hence being able to later apply the rules that govern the system.

Note that it is important to determine the boundary conditions of the grid. We have worked by imposing periodic boundary conditions, thus turning our two-dimensional grid into a torus by wrapping it around its edges. Even though a torus is a three-dimensional figure, we will be discussing everything in terms of grids as it is easier to visualize.

Apart from generating this simulator, we made it visually pleasant by creating GIFs that reproduced the grid over different generations. This can be done with the *animation* library in Python. We also tried visualizing the game with a torus, but it is not as aesthetic.

III. COARSE-GRAINING METHODS

Apart from a function that predicts the next generation of a grid, we need a function that coarse-grains our grids. The main approach to this step has been the block average method. Aside from this, a pseudo-convolutional coarse-graining method has been tried, too.

A. Block-average

Given an $M \times M$ grid and a coarse-graining parameter β, we generate a coarse-grained grid by dividing the original one into $\beta \times \beta$ blocks. Each block is then replaced by a single cell in the coarse-grained grid, set to 1 or 0 following the majority rule. It is important to also implement periodic boundary conditions for coarse-graining parameters that are not divisors of M. By using this method, we end up having a $m \times m$ coarse-grained grid, where $m \equiv M/\beta$.

To code this, we just basically have to reproduce what was discussed in the previous paragraph. We therefore define a function called *block-average* that receives a grid and a coarse-graining parameter and returns a coarse-grained grid.

B. Pseudo-convolutional coarse-graining

Introduced to have a smoother coarse-graining method, the pseudo-convolutional method basically applies a convolution-like coarse-graining operation without reducing the grid's dimension. Each cell is replaced by the majority value in its local $\beta \times \beta$ neighbourhood.

In order to code it, we can apply a uniform kernel that convolves around the grid. As stated before, this method was introduced because we wanted a smoother coarse-graining method. Let us explain this more thoroughly.

IV. EXPECTED RESULTS

Before conducting any simulation, we thought that the accuracy with which the neural network predicted the next step of the Game of Life would increase as the coarse-graining was set at a higher value.

To a greater or lesser extent, a phase transition-like graph ($fig.$ 1) was to be expected, where for low coarse-graining the neural network had difficulties learning meaningful patterns, whereas for high coarse-graining, the neural network would properly identify patterns; similar to how no clear pattern emerges when a bird flies alone, yet distinct formations appear when it flies as part of a flock, patterns in the Game of Life become apparent in the context of collective behaviour.

Lastly, when the coarse-graining parameter is very high, comparable to the grid's size, we expect to see a decrease in the accuracy in which the neural network predicts patterns, as the random guess will probably predict.

Bear in mind that these are the expected results prior to any simulation. Nonetheless, the neural network had a high accuracy at low coarse-graining when us-

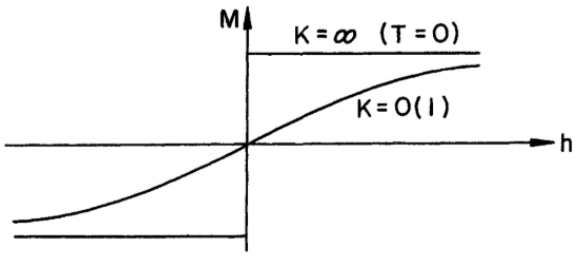

Figure 1: Example of a phase transition graph: Magnetisation as a function of external field at zero temperature (K = ∞ and at non-zero temperature (K = O(1)). Extracted from Figure 3.1, p. 94 of *Lectures on Phase Transition and the Renormalization Group*[3, p. 94, fig. 3.1]

ing the block average method. That is why the pseudo-convolutional method was added, although the results did not differ that much.

V. DATASET GENERATION

It is crucial to have a way of generating data to feed the neural network with. We are going to take advantage of the fact that infinite Game of Life settings can be generated. The discussion now relies on how the data should be created, as we do not have a problem with generating a huge amount of it.
According to Bibin and Dereventsov [4], the best accuracy is obtained when the training dataset is crafted manually, with grids in which symmetry and typical patterns predominate. However, our study has been done with randomly generated grids.

A. Parameters

Once defined how our data is refined, there are different parameters that need to be discussed for the sake of generating our dataset.

A.1 Number of samples, K

A dataset is made up of K samples. In other words, K random grids will be created.

A.2 Density, ρ

The density in which alive cells will appear in our initial grid.

A.3 Grid size N

The K randomly generated grids will be of size $N \times N$.

A.4 Subgrid size M

From the $N \times N$ grid, we will extract a subgrid of size $M \times M$. This is done to see whether the model is able to extrapolate the behaviour of the Game of Life from a part of the whole data, as well as to ease the computational power for large grids.

A.5 Coarse-graining parameter, β

A parameter that enables us to modify the coarse-graining of our data, hence turning our subgrid's size to $m \times m$, where $m = M/\beta$, as defined before.

A.6 Number of generations, τ

τ Generations of data will be generated = Iterate the initial grid τ times according to the Game of Life rules.

A.7 Initialization time, τ_0

Instead of starting with the randomly generated grid and iterate it τ times, we may want to see what happens if we start from a generation in which the patterns have been stabilised.

B. Neural Network Input

B.1 Data

The data that will be fed to the neural network will be a collection of K samples of $m \times m$ subgrids iterated over τ generations. This results in a tensor of shape $K \times \tau \times m^2$.

B.2 Label

As we are dealing with supervised learning, apart from feeding the data to the neural network, we need to associate a label with it. Thus, the label that will be

associated with each sample will be the value of the central cell of the $\tau + 1$ iteration. If the cell is alive, its value is 1; if it is dead, 0.

C. Neural Network Output

The value the neural network model has to predict is then the central cell of the $\tau+1$ generation. Note that we do not need neither the label nor the output to be the whole $\tau + 1$ grid as there is always the possibility of rolling it over when predicting.

D. Normalization

The process of normalization bears some importance as the neural network works better when the dataset is properly normalized.

The normalization is done by features and it is only done on the data, not the labels. Doing the normalization by features means that the statistical values (mean and standard deviation) are obtained for each component of the tensor fed to the neural network. Normalizing all the components equally does not make sense as they do not represent the same.

It is advised to compute the statistical values on a large dataset for them to be very precise and then reuse them every time a dataset with the same parameters is computed.

E. Balancing

Another key aspect of the dataset generation is balancing the data. It has been found that if we only generate random data and feed it to the neural network, the model tends to learn only the ratio of zeros and ones in the labels, not guessing higher than that. That is why it is essential to balance the dataset. Basically, instead of generating K random samples, we generate them establishing a threshold for both labels, which will set to be half the amount of samples. Then, we generate data up until both thresholds are met. This way we can ensure the dataset has the same amount of zeros than ones.

The brute force method, which is just to generate many samples up until this target amount of labels is hit, has a huge disadvantage: a lot of samples are discarded, thrown away. The higher the coarse-graining is, the more difficult it is to get one as a label, hence having to waste a lot of computational power to generate balanced datasets. That is why, if we set $N = M$ at the beginning of our dataset generation (which will not affect drastically the results obtained), we can take advantage of the periodic boundary conditions as, instead of waiting for the central cell to be alive, we can roll over the whole grid to wherever a one appears. Simulations show that, by implementing this *rolling the grid over* method, the computational time decreases significantly, resulting in easier and faster code executions.

F. Grid multipliers

At the beginning of the project, the block average method was only implemented for β that were divisors of M. That is why there were restrictions in the values of M and β we could have. Although this was solved and explained in section $III.A$, it is worth mentioning why grid multipliers is not a good option.

Grid multipliers made us generate data the other way around. We set β to be some value and we then create N, M and τ according to some multipliers λ_N , λ_M, λ_τ.

$$N = \lambda_N \cdot \beta$$

$$M = \lambda_M \cdot \beta$$

$$\tau = \lambda_\tau \cdot \beta$$

This way, we made sure there was no divisibility issue between any parameter. Nevertheless, a deeper issue arose, as the coarse-graining had no effect because we were just creating bigger grids, not applying any real coarse-graining.

VI. NEURAL NETWORK MODELS

A. Recommended literature

For information regarding neural networks and basic machine learning, there are plenty of resources online to learn the conceptual basics. In my view, the video

series *Neural Networks* by Sanderson [5] provides a comprehensive and insightful introduction to the topic. The library used for the machine learning part in this project has been *PyTorch*[6] and it is advised to follow a tutorial to familiarise yourself with it, such as the one provided by Mulla [7].

B. Multilayer Perceptron

The Multilayer Perceptron model, often called MLP, is a feedforward neural network that consists of fully connected neurons with nonlinear activation functions. The neurons are organized in layers and there are three different types.

B.1 Input layer

The input layer depended on the size of the sample we worked with. For example, if the samples generated in the dataset were made up of grids of size $m \times m$ and they were iterated τ times, the input dimension was of size $m^2 \times \tau$.
When feeding our data to the neural network, it is important to turn the tensor into a vector.

B.2 Hidden layers

Whereas the input and output layers have fixed values, there is more freedom to define the hidden layers. For not overly complicated models, people tend to use few hidden layers made up of fewer neurons than the input.
In our case, we studied how much variation there was for different hidden layers' architectures. We usually worked with two hidden layers. If the input layer had more than 200 neurons, then the hidden layers would be of size half and a quarter of the input layer's size, respectively. If the input size was less than 200 neurons, the hidden layers would be of size 128 and 64, independently of the input layer's size.
The first step is always to connect the layers linearly. This is done in every layer. Then, if the data is fed in batches, we can apply a batch normalization method, too. More importantly, we need an activation function when connecting each layer. The activation function

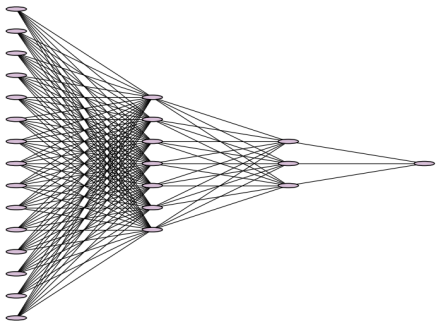

Figure 2: Schematic diagram of the multilayer perceptron (MLP) architecture used in this work. Diagram created by the author.

used has been the Rectified Linear Unit (ReLU)(1)

$$\mathrm{ReLU}(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leq 0 \end{cases} \tag{1}$$

Apart from these steps, a dropout layer has also been added.

B.3 Output layer

In our case, it consists of only one neuron, as the model needs to only have one output: the central cell of the $\tau + 1$ generation. Furthermore, this output has to be binary. That is why in this last layer a Sigmoid (2) activation function is used.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

C. Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a feedforward neural network that consists of using convolutional layers that apply filters to the input data. It is great to capture spatial patterns such as in the Game of Life. Nonetheless, it was not the primary aim of this project.
In this case, it is important to not turn the data tensor into a vector, but to preserve the data as it is. The

architecture of the CNN used has been the same as the recursive CNN used by Bibin and Dereventsov [4] with the $\tanh x$ activation function (3), as it was the one that obtained the best results. If we use the same weights and biases as the ones proposed in the paper, the neural network accurately predicts the next step.

$$f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

VII. TRAINING AND VALIDATION

Having created the neural network model and its architecture, we have to train it properly so as to succeed in then predicting the $\tau + 1$ central cell of our data.

A. Loss function & Optimizer

A neural network has its weights and its biases, and it has to optimize them in a way in which the loss between the predicted value and the real value is minimized.

A.1 Loss function

In order to do that, we use a so-called *loss function*. It is a pre-established function that computes this loss. There are different types of loss functions, and it is important to know what type of data you are working with to ensure that you choose the best option.
In our case, the best loss function is the Binary Cross Entropy function. This is because this function quantifies the difference between class labels and the predicted probabilities output by the model. Mathematically, it can be defined as (4):

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log p_i + (1 - y_i) \log (1 - p_i)] \tag{4}$$

where N is the number of observations, y_i is the actual label of the i^{th} observation, and p_i is the probability of the i^{th} observation being in class 1.
There is also the option to use the binary cross entropy with logits loss function (BCEWithLogitsLoss), which is a function that combines the Sigmoid activation function (2) and the binary cross entropy function (4). As this one already includes the Sigmoid, if we use this second option we must erase it from the output layer in our model so as not to apply the Sigmoid twice.

A.2 Optimizer

The optimizer is then the tool that adjusts the model's parameters (weights and biases) minimizing the loss function during training. Step by step, it first computes the gradients of the loss with respect to each parameter using backpropagation. Then, it updates the parameters in the direction that reduces the loss. This process is iterated for many batches or epochs to improve the model's performance up until the user desires. Last but not least, there is the learning rate that controls the step size of each update.
Adam optimizer is the one that has been used to do the work. It is a very popular algorithm in this context of machine learning as it converges faster than others and uses adaptive learning techniques. In Bibin and Dereventsov [4], there is a comparison of the convergence of different optimizers and the one that performed the best was Adam.

A.3 Regularization techniques

Apart from our loss function, we can employ some regularization techniques to effectively perform feature selection. We have used L1 as a regularization technique. This one adds a penalty to the loss function based on the absolute values of the model's weights. This encourages the network to make many weights exactly zero. It helps to reduce some possible overfitting and promotes sparsity in the model, as many weights become zero.

B. Training process

The training process consists of giving many samples to the neural network model, thus reducing the loss between the predicted and actual value.
Depending on the type of data data, the feeding process can be done in different ways. Suppose we are working with medical data. In that case, there is a limitation in data generation as we will not be able to

produce data as many times as we please. Therefore, there are different methods to efficiently train and validate the model on limited resources.
However, that is not our case, as we can infinitely produce as many data as we want. That is why what we really need is a huge dataset on which to train the model: feed a lot of samples to the neural network and hope the results converge.
The training has been done in batches, as it is more efficient and practical for the neural network and the hardware, i.e. the computational power required. This means that the data is not given to the neural network sample by sample but in small chunks of data. Dividing into batches allows processing smaller chunks at a time, and this fits better into the GPU/CPU memory; noise is also reduced as, by feeding the data in batches, we can apply techniques such as batch normalization that allow us to reduce the noise.
The process is then to go through a batch of data, in which we will define the input vector $\vec{x}$ and the label $\vec{y}$. Flattening the data to feed the MLP model, we will compute the predicted class label $\vec{p}$ and compare it, with the loss function, to $\vec{y}$. If we want to add the regularization technique, we should do it afterwards. Once the loss is computed, we compute the gradient for the weights with the backpropagation method and update them accordingly to our optimizer. Then we iterate through all the batches up until we reach convergence or we have been through all the training samples. It is also advised to include objects that prevent the model from exploding gradients and vanishing gradients.

C. Validation process

In order to check how well the model performs while training, the same process as the training one has been implemented. This way, we ensure that there was neither overfitting nor underfitting. To prevent not updating the model's parameters while doing the validation, we have to be sure that the code is in evaluation mode, not in training mode.
Note that, as we can generate infinitely many data, we can generate a dataset only to do this validation process, completely different from the training one. This way, we ensure that the neural network is not just memorizing the data but actually learning from it.

VIII. TESTING

When the model is properly trained, we now have to test how good the performance is when shown new data.
Setting the model to evaluation mode, we have to see how accurate the model is at predicting the correct label. To do that, we generate another dataset and go through it, comparing the predicted output to the actual output and counting how many guesses are correct. From that, we can generate the accuracy (the ratio of correct guesses in all the testing samples).
In addition to this accuracy parameter, we can also count if the predictions and actual values are zeros or ones. This way, aside from the general accuracy, we can see if there is bias toward guessing one label or the other.
It is also very advisable to add this accuracy step to the validation and training data, to see if the performance becomes stalled at some point and/or the learning is not going well.

IX. RESULTS

A. Values used for the study

The vast majority of results were done on datasets where $N = M$, as no significant difference in results was found between $N > M$ and the case studied. The typical values for N and M used were between 15-45. The density of alive cells ρ was set to be 50% almost all the time. However, Bibin and Dereventsov [4] found that, for their study, the optimal density to train on was 38%, but the accuracy results obtained between both of these densities were not significant.
The amount of times the data was iterated (τ) was, at least, the coarse-graining parameter β, as the information of each cell needs to at least move β times to go to another coarse-grid cell.

$$\tau \geq \beta$$

The initialization time τ_0 is an interesting parameter, as, depending on the grid size, there is an average time in which the grids start to form stable patterns. In spite of that, increasing τ_0 too much will result in using huge computational power.
To train the models, we used training datasets made up of 16,000-48,000 grids. To validate and test them, the datasets were smaller, around 1,000-2,000 grids. This has been found to be enough to do the process.
Even though these are the values proposed, playing around with different values is highly encouraged.

B. No-balancing

If we did not balance the datasets, the model just memorized the ratio in which alive and dead labels appear, hence not learning anything, just guessing either zero or one always. That is why we decided to train on balanced datasets.

C. Coarse-graining

The most influential parameter is the coarse-graining parameter, the block-size β.

C.1 No coarse-graining

Setting our grids to have $\beta = 0$, the models are surprisingly good at predicting the data, better than we expected, moving around between 80% and 90% accuracy.

C.2 Non-zero coarse-graining

For low coarse-graining, $\beta = 3$, $\beta = 5$; the results have been underwhelming. The learning has been found to be stuck between 55% and 60%, even decreasing up until random guessing 50% sometimes. It seems to be a problem with how the model is learning. Maybe changing some parameters of the optimizer will improve this.
For higher coarse-graining, $\beta = 7$, $\beta = 9$, $\beta = 11$, tests have not been properly made, but the results are expected to be similar to the low coarse-graining ones unless architectural changes are made.

D. Neighbours'sum

If instead of feeding the data proposed previously to the neural network, we also feed the neighbours' sum of the central cell, the neural network always predicts the next step correctly.
As a result, this can be used as a sanity check to look for possible issues with the model before jumping to conclusions.

X. CONCLUSIONS

This study explored the capacity of machine learning models, particularly MLPs and a bit of CNNs, to predict the evolution of Conway's Game of Life under various coarse-graining levels.
A simulator and dataset generation methods were developed, enabling high flexibility in parameters such as grid size, density, number of generations, and coarse-graining level. Results showed that while models performed surprisingly well with no coarse-graining ($\beta = 0$), their performance degraded significantly at non-zero coarse-graining levels, often stagnating near random guessing accuracy. This is theorized to be due to the model's learning parameters.
It is important to remark on the necessity of balanced datasets, as unbalanced ones lead to models memorizing label frequencies instead of learning underlying dynamics.
To summarise, the project highlights the challenges of neural networks to learn patterns in coarse-grained complex systems.

XI. FURTHER RESEARCH

Obviously, the project has not been finished yet, but there are different leads to follow.
Apart from running simulations and improving the model's architecture for MLPs and CNNs to have some conclusive results for different dataset parameters, it would be very interesting to study the patterns obtained with the predictions from our models and compare them with the ones found in the literature[1].

XII. ACKNOWLEDGEMENTS

I would especially like to thank Dr. Tamás Kriváchy for his invaluable guidance throughout this internship and for giving me the opportunity to begin my journey as a young researcher. I am also grateful to Prof. Antonio Acín and his Quantum Information Theory group at ICFO for making these months such a meaningful experience, during which I had the chance to meet many inspiring and passionate individuals.

REFERENCES

[1] *Conway's Game of Life*. https://conwaylife.com/. Accessed: 2025-06-12.

[2] Jacob M. Springer and Garrett T. Kenyon. "It's Hard for Neural Networks to Learn the Game of Life". In: *arXiv preprint arXiv:2009.01398* (2020). URL: https://arxiv.org/abs/2009.01398.

[3] Nigel Goldenfeld. *Lectures on Phase Transitions and the Renormalization Group*. Reading, MA: Addison-Wesley, 1992.

[4] Anton Bibin and Anton Dereventsov. "Data-Centric Approach to Constrained Machine Learning: A Case Study on Conway's Game of Life". In: *arXiv preprint arXiv:2408.12778* (2024). URL: https://arxiv.org/abs/2408.12778.

[5] Grant Sanderson. *Neural Networks*. 3Blue1Brown, YouTube, https://www.youtube.com/watch?v=aircAruvnKk. Accessed: 2025-06-12. 2017.

[6] *PyTorch*. https://pytorch.org/. Accessed: 2025-06-12.

[7] Rob Mulla. *PyTorch – Full Course for Beginners*. YouTube, https://www.youtube.com/watch?v=tHL5STNJKag. Accessed: 2025-06-12. 2021.