Emergent Models – Research Proposal

Giacomo Bocchese
Wolfram Institute
brightstarlabs.ai

Nicola Giacobbo
brightstarlabs.ai

Nicolo Monti
brightstarlabs.ai

April 14, 2025

Abstract

This proposal aims to introduce the concept of Emergent Models (EMs): dynamical systems inspired by Cellular Automata and Turing Machines that can be used as Machine Learning (ML) models instead of neural networks. By encoding complexity in a large state space and recursively applying a fixed update rule until a halting condition is met, these models aim to emulate modeling and learning behaviors in a fundamentally different way from neural networks and other traditional machine learning models. Current ML models, especially neural networks, usually tend to capture surface patterns rather than achieve true generalization. This shortfall may arise from their reliance on a one-shot, complex, and highly parametric transition function. In contrast, biological systems and physical processes evolve by applying many times a simple transition function on a large state space, much like a Turing machine. EMs, which are inspired by Turing Completeness and use simple local rules applied iteratively over a large state space, offer a highly expressive yet efficient alternative to neural networks. This approach likely reduces overfitting and enhances generalization, making EMs a promising pathway toward artificial general intelligence (AGI). Furthermore, due to the inherent flexibility of their architecture, they could present advanced meta-learning capabilities. To leverage all this flexibility, EMs are trained with black box algorithms, like with genetic and Bayesian optimization.

We propose to implement and compare EMs on a toy reinforcement learning (RL) task, driving a dummy car in a 2D environment, to evaluate expressivity, stability, learning speed, spontaneous emergence of meta-learning abilities, and searching for the best configurations to allow the model to work and learn efficiently. All findings will then be formalized into a research paper, and we also plan to release a GitHub repository to reproduce our work.

1 Introduction

Traditional ML models, such as Neural Networks, encode learned complexity within their trainable parameters, forming complex transition functions. Furthermore, multilayer perceptron [2] (MLP) neural networks generally operate in a *one-shot* manner: they predict an output $\hat{y}$ by applying a very complex closed-form parametric function in a single time for each input.

Recurrent neural networks [11] and Transformers [14] for autoregressive tasks operate in a slightly different way since they allow *loops* where information can be processed taking into account past states. However, even if they are somewhat *stateful*, most complexity is not derived from the state space (which is low dimensional, e.g. hidden state in neural networks or context in transformers) but from the transition function [1]. Moreover, even autoregressive ML systems operate by applying the transition function only once per input, which increases the complexity of correctly modeling the relation. Additionally, since the model complexity for each prediction step is fixed, it poses limitations in reasoning tasks, where *computational depth* should be adaptable.

In contrast, EMs employ a fundamentally different strategy for processing information and making predictions. Rather than applying a fixed and highly complex transition function in a single step, EMs iteratively build their estimates by updating internal representations over multiple computational cycles. This iterative process allows EMs to dynamically allocate computation time depending on the complexity of the input and the task at hand, to reduce the parameter count needed, and to improve generalization.

[1] The transition function transforms the previous state into the next state, which in case of neural network is the network itself

1.1 Motivation

Unlike traditional neural networks, which primarily encode learned complexity within their trainable parameters, EMs emphasize a more structured approach in which reasoning and inference emerge through repeated transformations of an evolving state. Indeed, EMs rely on:

- a **simple fixed transition function** analogous to the local rules of Cellular Automata [17] or Turing Machines [5];
- a **large, flexible state space** where the *program* (i.e., the learned behavior) is embedded and evolves over time;
- **iterative computation** until a halting condition is met. The state space continues to evolve with the same input until a halting condition becomes true, reminiscent of Turing machines;
- an **encoding function** that injects information from an external input into the state space;
- a **decoding function** that retrieves information from the state space to form the output;
- a **black-box optimizer** that improves state space values to select and propagate emergent behaviors that increase reward or reduce loss. Here, the initial values of the state space are optimized rather than the transition function.
- **reward feedback fed into the state space as input**. In addition to inputs, reward information can be encoded and fed into the state space. Since the model's behavior is entirely encoded within the state space, an initial state exists that enables reward information to propagate into the state space and induce self-improvement. In other words, the model can **emergently** build an optimizer within itself to drive arbitrary improvements.[2]

Furthermore, applying a simple transition function repeatedly (with adaptive halting) allows building complex representations without requiring a complex transition function. This may improve reasoning abilities and generalization on complex tasks with fewer parameters.

This paradigm shift - from optimizing the parameters of a transition function (like in neural networks) to tuning the initial (or ongoing) state of a dynamical system - may unlock new capabilities. Inspired by natural processes, EMs could offer more adaptable, general, and interpretable machine learning mechanisms.

1.2 Why Emergent Models

Initially, dynamical systems like Cellular Automata were studied descriptively and have not been used for practical applications. EMs aim to be the first general-purpose computational framework for using Cellular Automata as a machine learning model. As dynamical systems, they work as computational models that process inputs and generate outputs, much like ML models. Inputs are encoded by perturbing the initial state of a complex system, and outputs are retrieved from its terminal value.

The learning abilities of these systems can be thought of as emergent, since they weren't initially built for this goal. However, they can be used with this scope via black-box optimization techniques, like genetic algorithms, which tunes the automata to behave in the desired way in response to inputs.

These systems can be viewed as *blank canvases* applicable to almost any scope. They are highly flexible, even more than current ML algorithms (including neural networks). Depending on their degree of flexibility, high-exploration optimization algorithms might be required to let the optimal state *emerge*. Moreover, unlike neural networks and other differentiable models, EMs rely on search rather than deterministic optimization, which makes them inherently emergent. In those systems, optimization continues until a desired property in the space of possibilities appears. Furthermore, they (theoretically) exhibit higher-order emergent behaviors that add further value: meta-learning, memory capabilities, and self-healing to maintain stability (see Section 5 for more details).

The base idea behind those models is *to take a complex, near-chaotic dynamical system, connect it to input and output proxies, and search for initial states that achieve high performance on a given task*.

Encoding and decoding should also be done properly to allow correct processing of information, but computation theory tells us that encoding, decoding, and transition functions can be kept simple and fixed (see Section 2.4.1).

[2] If a model is Turing complete, it can theoretically represent every optimizer into itself and use them to drive arbitrary improvements of the state space.

2 Mathematical abstraction

In this section, the mathematical formalism is used to describe EMs in their discrete and continuous representations. In the following, it is assumed that EMs is a model that maps an input $x \in \mathscr{X}$ to an output $y \in \mathscr{Y}$.

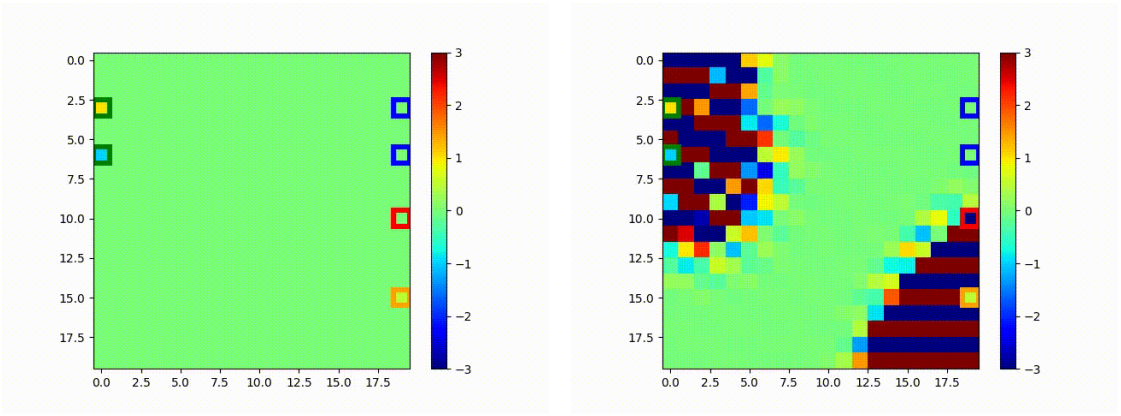
We define the state space S as one of the possible configurations represented by the set $\mathscr{S}$, where each *cell* of the system has a value $v \in \mathbb{V}$, and thus the state space can be expressed as:

$$\mathscr{S} = \mathbb{V}^n$$

where $n \in \mathbb{N}$ represents the dimension of the system or the number of cells. We might have $\mathbb{V} = \{0,1\}$ for a binary domain, $\mathbb{V} = \mathbb{N}$ for the natural numbers domain, or $\mathbb{V} = \mathbb{R}$ for the real numbers domain. In general, $\mathscr{S}$ may have an arbitrary topology.

To achieve Turing completeness [16], the state space must have a countable infinite number of states. If $\mathbb{V}$ is finite (as in binary or floating-point representations), the state space must have the cardinality of the natural numbers: $\mathscr{S} = \mathbb{V}^{\mathbb{N}}$. In practice, infinite memory is infeasible, so we limit the state space to a finite n, that is, $\mathscr{S} = \mathbb{V}^n$. Although this limitation may reduce expressiveness, depending on the task complexity, it is negligible for sufficiently large n. Thus, we are exploring ways to allow a dynamic dimension n so that the model can self-adapt its memory requirements.

2.1 Intuition



(a) (b)

Figure 1: Snapshots at different instants of a system composed of a 20×20 cells. Where different border colors indicate: inputs x in green, outputs y in blue, in orange rewards, and halting in red. In particular, image (a) represents the initial state of the system, while image (b) shows the system at a generic instant t.

Think of a system as a grid of n cells, each holding a value. In particular, S_0 represents the initial state while S_t is a snapshot of the same system at a generic time t (see Figure 1). We define the **encoding function** (E) as the function that injects the input x into the state space domain:

$$E(x) \to \mathscr{S} \quad \forall x \in \mathscr{X},$$

and symmetrically the **decoding function** (D) extracts the output y from the state space:

$$D(S) \to \mathscr{Y} \quad \forall S \in \mathscr{S}.$$

Then, we define H as the **halting function** that determines when the iterations stop:

$$H(S) \to \{0,1\} \quad \forall S \in \mathscr{S}, \quad \text{where} \quad H(s) = \begin{cases} 1 \Rightarrow \text{it stops} \\ 0 \Rightarrow \text{it continues} \end{cases} \tag{1}$$

While the **transition function** (f) maps the current state, S_t to the next instant S_{t+1}

$$f(S) \to \mathscr{S} \quad \forall S \in \mathscr{S}.$$

Generally, f is a simple and local function (though it can sometimes be non-local). For example, f may be a convolution:

$$f(S) = K * S \quad \text{(with } K \text{ as the kernel)},$$

or a convolution + non-linear pointwise application:

$$f(S) = \sigma(K * S) \quad \text{(similar to Lenia CA [3])},$$

or it may represent the aggregate effect of Game of Life rules [3] or even a stochastic function for probabilistic transitions.

2.2 Formulation of Emergent Models

The inference process of the emergent model, from input to output, is described as follows:

1. **Encoding/Initialization**: we initialize the emergent model with a non-perturbed initial state S_0 and with the input term encoded $E(x)$:
$$\tilde{S}_0 = S_0 \oplus E(x), \tag{2}$$
where the two terms are combined to form the actual *perturbed* initial state $\tilde{S}_0$, comprehensive of the input. The combination operator $\oplus$, in the simplest form can be a sum $(+)$, and equation 2 becomes $\tilde{S}_0 = S_0 + E(x)$, or more complex combinations.

2. **Evolution**: the system evolves in time by applying recursively the transition function until halting:
$$\tilde{S}_{t+1} = f\left(\tilde{S}_t\right), \quad t = 0,1,2,\ldots,T \quad T = \min\{t \in \mathbb{N} \mid H(\tilde{S}_t) = 1\}, \tag{3}$$
where the instant T represents the moment at which the system reaches the configuration of the state space that makes the system halt. By definition, it depends on both the initial state S_0 and the input x.

3. **Decoding**: the decoding function is used to extract the output from the state space at T:
$$y = D(\tilde{S}_T) \tag{4}$$

The behaviour of the model is highly influenced by the initial state S_0; in fact, the model built this way is entirely programmable by the initial state. The non-perturbed state S_0 can be considered as the program in a Turing machine or as the weights in a neural network; it encodes the *knowledge* and the *behaviour* that the model follows. Obviously, S_0 is not programmed in a deterministic way; in contrast, it is searched using optimization algorithms to minimize a loss function or to maximize a reward metric.

The model can be described by the following function:

$$\phi : \mathscr{S} \times \mathscr{X} \to \mathscr{Y}$$

$$y = \phi(S_0, x) = D\Big(\tilde{S}_T\Big) = D\Big(f^T\Big(S_0 \oplus E(x)\Big)\Big), \tag{5}$$

where f^T denotes the recursive composition of f for T times. It is also important to notice that even the halting time T depends on both the initial state S_0 and the input x.

Additionally, it is useful to define an auxiliary function that operates like ϕ but it also returns the state space value at halting time $\tilde{S}_T$:

$$\phi' : \mathscr{S} \times \mathscr{X} \to \mathscr{Y} \times \mathscr{S}$$

$$(y, \tilde{S}_T) = \phi'(S_0, x) = \big(\phi(S_0, x), \tilde{S}_T\big)$$

This is useful since in many cases $\tilde{S}_T$ is used as the non-perturbed initial state S_0 in the next prediction.
An important note: ϕ is a partial recursive function, meaning that it may not be valued for some inputs, since there is a set of inputs/programs for which the recursion never halts.

[3] At this link the reader can find more details and some examples.

2.3 Sequential operation and state retention

EMs can operate sequentially, it is enough not to reset the state space. It is common to have sequential input data, where we have x_i as the i-th input x. In this case, the goal of the model is to produce an output y_i for each i. In this contest, ϕ' returns the output y_i and the perturbed state at halting time of the system for the i-th iteration $\tilde{S}_T^i$, used as the non-perturbed initial state for the next iteration S_0^{i+1}.

Description of sequential operation:

- Initialize the model with a state space S_0^0
- Feed the first input x_0 in the model, wait halting and predict y_0, $S_0^1 = \phi'(S_0^0, x_0)$
- Feed the second input x_1, wait halting and predict y_1, $S_0^2 = \phi'(S_0^1, x_1)$
- So on, for the i-step y_i, $S_0^{i+1} = \phi'(S_0^i, x_i)$

In practice, this means continuing to infer without resetting the initial state at every input. In this way, we allow the model to retain memory over time.

State retention has to be used in every task, even if non natively sequential (even in the absence of explicit temporal sequences or time causality). This is useful for training and optimization, and fundamental for meta learning and higher-order behaviours.

2.4 Turing Completeness of Emergent Models

We show that, under the proper hypotheses, certain EMs are Turing complete and can perform any computation. Hypotheses require countably infinite cardinality of the states, which implies infinite memory requirements. Furthermore, computations may never halt for some inputs. This requires a practical adaptation that cuts to finite memory and finite computation time, lowering the maximum expressivity but still being able to handle almost any computation of practical use, or designing variable-dimension memories to allow the model to extend its memory. For example, an external read-write memory can be added to the model.

2.4.1 Turing completeness definition

If the state space $\mathscr{S}$ is sufficiently large, having a at least the same cardinality of the natural numbers, then the existence of the key functions - transition f, encoding E, decoding D, and halting H - ensures that for every computable function $g : \mathscr{X} \to \mathscr{Y}$, we can find an initial state $S_0 \in \mathscr{S}$ such that $\phi(S_0, x) = g(x)$ for all $x \in \mathscr{X}$. In formal terms:

$$\exists f, D, H, E \text{ s.t. } \forall g \in \text{COMP}, \ \exists S_0 \in \mathscr{S} \text{ s.t. } \forall x \in \mathscr{X}, \ \phi(S_0, x) = g(x)$$

In practice, it means that by varying S_0 - keeping f, E, D, and H fixed - the model can be programmed to represent any algorithm.

A cellular automata [15, 10] with a Turing complete rule can be expressed as an EM. This implies that there exist EMs that are Turing-complete.

2.5 Halting problem and workarounds

As for Turing machines, we do not guarantee that the system halts on every input and on every program. We propose various methods to address this practically:

- **Option 1**: by using stochastic transition functions (or stochastic halting function), there is always a little likelihood that the value of halting becomes true, halting the computation. This makes it very unlikely that the computation will run for a long time and prevents it from running forever.
- **Option 2**: add a mechanism such that if the number of iterations exceeds a threshold, the system halts.
- **Option 3**: reward penalty for computation time. This disincentivizes models that require excessive computation time.

The best way to handle the halting issue could be a mix of these three strategies: setting a high threshold for maximum number of iterations, to avoid excessively long processes in edge cases; adopting a probabilistic transition function to keep the whole system flexible; and by using a small penalty for computation time, we let the model learn to self-weight the importance of long computations and develop *least computation time* strategy.

A similar thing could be done to make EMs memory efficient: a penalty over the number of non-zero cells, which should induce a minimization of Kolmogorov complexity.

2.6 Generalization to the Continuous

EMs are initially formulated through a discrete representation but can be extended to continuous. Here, we present two generalizations: first, the temporal evolution of a finite number of cells as a continuous process, which is modeled using an ordinary differential equation (ODE); second, a spatially continuous representation, which leads to a partial differential equation (PDE) model.

2.6.1 ODE case (continuous time, spatially discrete)

We model this concept as an ODE, which dictates a relationship between the time derivative of the state depending on the transformed state $f(s)$, where f is the transition function. In this case, any state is represented as an n-dimensional real vector:

$$S \in \mathbb{R}^n \quad \text{where} \quad n \in \mathbb{N}.$$

Thus, for a given input x and a chosen initial state S_0, the state can at $t = 0$ can be written as $\tilde{S}(0) = S_0 \oplus E(x)$ and its evolution is represented by an ODE:

$$\frac{d\tilde{S}(t)}{dt} = f\big(\tilde{S}(t)\big) \quad t \geq 0.$$

Then the halting condition becomes:

$$T(x, S_0) = \min\{t \geq 0 \mid H(\tilde{S(t)}) = 1\},$$

while the final output y is obtained by:

$$y = \phi(S_0, x) = D\Big(\tilde{S}\big(T(x, S_0)\big)\Big).$$

2.6.2 PDE case (continuous time, continuous space)

In this case, both space and time are continuous, similarly to a fluid simulation or a reaction-diffusion simulation, so a PDE can be used to model it. Here, the state space is represented by a function $s : \Omega \to \mathbb{R}$ and its evolution can be described by an arbitrary PDE:

$$\frac{\partial s(\cdot, t)}{\partial t} = f\Big(s(\cdot, t), \nabla s(\cdot, t), \nabla^2 s(\cdot, t), \dots\Big), \quad t > 0,$$

and the halting function can be written as:

$$T(x, s_0) = \min\{t \geq 0 \mid H(s(t)) = 1\}.$$

Thus, the final output y is computed as:

$$y = \phi(S_0, x) = D\Big(s\big(T(x, S_0)\big)\Big).$$

2.7 Further Generalizations

The concept of EMs can be generalized further. For simplicity, we considered a one-dimensional state space (like a tape in a Turing machine), where the state cells are distributed along a single axis ($S = \mathbb{R}^n$ or $S = \{0, 1\}^n$). An example of this can be a 1-dimensional cellular automata; but the concept of EM can be generalized even to more dimensions (e.g., forming a 2-dimensional grid like the *Game Of Life* [4]); or even further, allowing for a more complex set

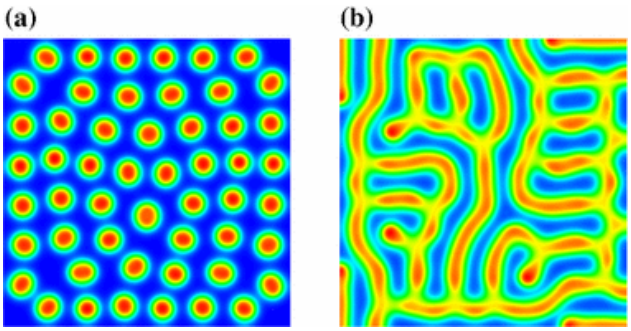


Figure 2: Gray-scott simulation with two reagents (for more details, take a look at this link).

of values for each cell. The set $\mathbb{V}$ can be represented by real numbers ($\mathbb{R}$), but can also hold two real values for each cell ($\mathbb{R}^2$). An example of this is reaction-diffusion systems, where the state space is modeled as the continuous concentration of two chemical reagents in space, having $\mathbb{V} = \mathbb{R}^2$, on a bidimensional domain $\mathbb{R}^2$.

A final generalization involves allowing an arbitrary topology for the state space, eventually non-Euclidean, for instance, enabling custom (e.g., graph-based) spatial interactions. This might permit the expression of spiking neural networks [7] (SNNs) as EMs. This is consistent with the current formulation of EMs, as, by definition, the state is defined as a set of cells, while the interaction between cells is entirely encoded in the transition function f.

3 Practical Implementation

After the theoretical introduction, we analyze two practical implementations of EMs that actually allow us to perform an inference from an input x to an output y: Convolutional Turing Machine and Game Of Life Language Model.

3.1 Convolutional Turing Machine

The Convolutional Turing Machine (CTM) is an innovative implementation of an emergent model that is inspired by Lenia Cellular Automata [3] (which uses convolution + non-linear activation). In essence, the CTM employs a continuous-valued, grid-based state space where a simple convolution is repeatedly applied to update the system's state. Input values are directly injected into specific cells, and outputs are extracted from designated output cells, ensuring a clear mapping from input to output. Additionally, a halting mechanism is implemented, based on a threshold on the halting cell. An example implementation of a CTM is available, as a public repository, at link.

3.1.1 Details of the Convolutional Turing Machine

- **State Space:** Continuous-valued, discrete-dimension state space: $\mathbb{V} = \mathbb{R}$, $\mathscr{S} = \mathbb{V}^n$.
- **Evolution**: Discrete time evolution with $s_{t+1} = f(s_t)$, where $f(s) = K * s$ or $f(s) = \sigma(K * s)$.
- **Geometry:** Square geometry of the state space: $n = m^2$, where m is the number of cells per side, indicating a square geometry $\rightarrow$ 2D geometry $\rightarrow$ interaction between nearest cells in an Euclidean sense.
- **Kernel:** Generally, $K \in \mathbb{R}^9$ or $\mathbb{R}^{25}$ (corresponding to a 3x3 kernel or 5x5 kernel, respectively).
- **Input Space:** $\mathscr{X} = \mathbb{R}^{d_{\text{input}}}$.
- **Output Space:** $\mathscr{Y} = \mathbb{R}^{d_{\text{output}}}$.
- **Encoding:** Direct injection: the value of certain cells (green border cells), called *input cells*, is directly set as the input value. That is, $E(x) = x_i$ for each i in the set of input cells, and 0 elsewhere.
- **Decoding:** Direct extraction: the output values are obtained by copying the value of certain cells (blue border cells), called *output cells*, i.e., $D(s) = s_i$ for each i in the set of output cells.

- **Halting:** Direct extraction: a cell (red border cell) in the state space, named *halting cell*, provides the halting value. If the value of the halting cell s_{halt} exceeds a threshold, the iterations of f halt, i.e., $H(s) = (s_{\text{halt}} > \text{threshold})$.
- **Reward Injection:** Direct injection of reward information into a designated *reward cell* (yellow border cell), which holds the reward value.

3.2 Game Of Life Language Model

It's an emergent model that uses Game Of Life principles, designed to generate sequences of symbols belonging to a discrete set. It can be used for language modeling since it generates symbols autoregressively until an end-of-text token. Under current settings, it is designed for modeling arithmetic computations, so it allows numbers 0–9 as inputs and outputs, and an end-of-text symbol $|$. Since we feed the model one token at a time, the model should learn to build a memory mechanism that allows it to memorize information from previous tokens (context) in order to predict future ones coherently. This memory should emerge inside the state space, with the ability to retain previous information as cell states. This is possible since the state space is not reset when passing to the next token, but is kept as it was on the last iteration. A Python implementation can be found at this link.

3.2.1 Details of the Game Of Life Language Model

The model generates one token at a time until the end-of-text token, and here we describe the inference process for a single token:

- **State Space:** Discrete-valued (binary), discrete-dimension state space: $\mathbb{V} = \{0,1\}$, $\mathscr{S} = \mathbb{V}^n$.
- **Geometry:** Square geometry of the state space: $n = m^2$, where m is the number of cells per side, indicating a square geometry $\rightarrow$ 2D geometry $\rightarrow$ interaction between nearest cells in an Euclidean sense.
- **Transition Function:** $f(s)$ is given by the rules of the 2D Game Of Life.
- **Interaction:** Local in a 2-dimensional sense, where nearby cells are considered as cells that touch each other in an Euclidean 2D geometry.
- **Input Space:** $\mathscr{X} = \{0,1,2,3,4,5,6,7,8,9,|\}$ (a symbol).
- **Output Space:** $\mathscr{Y} = [0,1]^{\{0,1,2,3,4,5,6,7,8,9,|\}}$ (a probability distribution over a set of symbols).
- **Encoding:** One-hot encoding: the nearby cells to the input symbol are activated with value 1 (alive) while all other input cells are kept deactivated (value 0, dead). For better propagation of the input, 3 cells around the input value are activated.
- **Halting:** Count how many times a cell on the right boundary is active, and accumulate a quantity ("count") indicating how many alive cells touched the right boundary. When the count exceeds a threshold, the computation halts. This can be considered as adding a special state space layer that accumulates counts and then applying a decoding function on it (omitted for simplicity).
- **Decoding:** Distribution decoding: accumulate the likelihood of the next symbol by counting how many 'alive' cells touched the right border at each position during the entire inference session (until halting). Specifically, count how many cells touch the right border at a certain height, then sum/average the activations around each symbol (using a convolution), collect the values corresponding to each output symbol, and finally apply a softmax:
$$y = \text{softmax}(\text{convolve1d}(\text{kernel}, \text{counts}))$$
with kernel defined as $[1,1,1]/3$ for averaging.
- **Reward Injection:** Similar to one-hot encoding, applied to the lower and upper boundaries; more cells are activated if the reward is higher.

3.2.2 Multi Token Inference

The model allows the inference of multiple tokens in the following manner:

- Initialize with state S_0.
- Feed the first token as input.
- Wait for halting and generate the next token (sampling from the output distribution) $\rightarrow y$.
- (Optional, for meta-learning) Compute and feed the reward, and wait for halting again.
- Feed back the last generated token y_t as input x_{t+1}, while keeping the state unchanged.
- Generate the next token.
- (Optional) Feed reward.
- Feed the generated token as input and retain the state.
- Continue until the generated token is an end-of-text token (|).

This process allows the generation of sequences of arbitrary length, assuming the model is able to retain memory correctly. Since the model has a finite memory state space, there is a limited amount of information that can be memorized and a limited maximum computational complexity. This issue can be addressed by designing a sufficiently large state space for the task at hand or by choosing adaptive memory strategies (for example, allowing read/write on an external memory or employing a dynamic memory state space). `https://github.com/BoccheseGiacomo/gol_lm/`

4 Optimization

The core idea behind Emergent Models (EMs) is that their learned behaviors are encoded entirely within the initial state (S_0^0) rather than through explicit trainable parameters like in neural networks.

The optimization goal is to discover the optimal initial configuration of the state space that maximizes performance metrics, such as accumulated reward in reinforcement learning or minimization of predictive loss in supervised scenarios. Due to the non-differentiable and iterative nature of EM computations, traditional gradient-based optimization methods like backpropagation are not applicable under the current settings.

Instead, our optimization strategy requires black-box methods capable of efficiently exploring large, sparse, and highly nonlinear search spaces. Among these methods, genetic algorithms [1] are particularly suitable as they iteratively select, mutate, and recombine candidate solutions, promoting the emergence of desired behaviors through maximization of a fitness function.

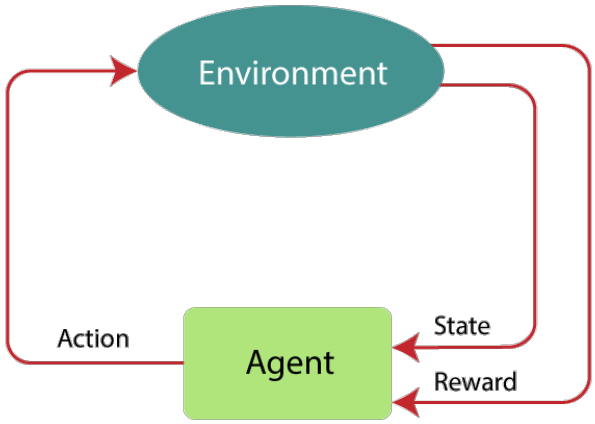


Figure 3: Reinforcement Learning settings illustration

Optionally, Bayesian optimization [12] provides a complementary approach that employs surrogate models to intelligently guide the genetic mutations towards the most promising configurations, reducing the computational cost of expensive evaluations. These strategies together allow for the efficient discovery of initial states that enable sophisticated emergent behaviors.

These optimization techniques may require higher computational resources than standard methods, but are essential for leveraging the full flexibility, generality, and adaptive capabilities inherent in EMs.

As a practical problem, let us consider an agent guided by an emergent model under reinforcement learning settings. The agent receives as input an observation x^i, and returns an action a^i at each step i. The agent also receives a reward r^i, and its goal is to maximize the expected cumulative reward $E(R)$, where $R = \sum r^i$ over an episode. (see figure 3)

The agent's policy is represented internally by the emergent model, which maps an observation to an action (or probability distribution over actions). By controlling S_0^0, we can represent an arbitrary policy. The aim is to train S_0^0 to its optimal value, which maximizes $E(R)$.

The interactions between the agent and the environment, that repeatedly occur along the entire episode, are (for a single iteration):

1. the emergent model (agent) receives observation x^i from the environment;
2. the agent takes an action a^i, $S_0^{i+1} = \phi'(S_0^i, x^i)$;
3. the environment returns a reward r^i to the agent.

The training process, addressed by genetic algorithms, starts by initializing a population of n agents with random initial states $S_{0,j}^0$ (for each agent j). Then, we run an episode and accumulate rewards for each agent by summing them for each iteration ($R_j = \sum_i r_{i,j}$). Then we select the best k performing agents with highest cumulative rewards R, we apply random mutations and crossover and reproduce them by cloning their mutated state up to reaching the initial population of n agents. We continue to a new episode and so on until convergence. (see figure 4). This workflow is assumed to be without meta learning. Under meta-learning settings, the last reward r^i is fed as input along with the next input x^{i+1}, letting the model exploit reward information as an input.

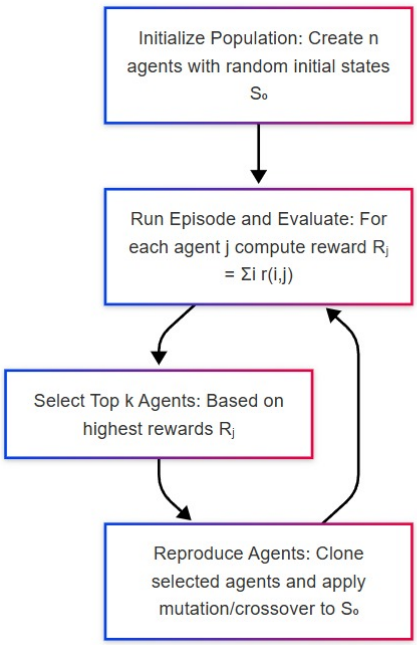


Figure 4: Flow chart of the whole training process addressed by a genetic algorithm optimization.

5 Meta-learning & higher-order behaviours

The flexibility of EMs allows for the potential emergence of higher-order behaviours like: meta-learning [9], antifragility [6, 13], self-learned inductive biases [8]. To intuitively understand higher-order behaviours, it helps to draw parallels with Deep Learning (DL) and Turing Machines (TMs).

In Deep Learning, a neural network's behavior is defined by parameters (weights and biases) that shape how inputs are transformed into outputs. These parameters are fixed and remain separate from the intermediate computations (activations) that represent the data as it flows through the network's layers.

In Turing Machines, behavior (the *program*) is encoded directly into the initial configuration of the tape, separately from input symbols, which occupy another portion of the tape. The machine executes instructions based on the program region, manipulating inputs and intermediate states to produce an output at halting.

EMs integrate these concepts. In particular, in EMs, the initial state of the system, the non-perturbed state S_0, encodes the *program*, similarly to a Turing machine. This state determines how the model processes inputs. Unlike neural networks, EMs do not maintain a clear separation between parameters (*program*) and activations (intermediate computations). Instead, both are mixed into a unified state space that evolves over time.

The input data, injected through an encoding function $E(x)$, directly influences the entire state space. This means that inputs can dynamically alter the program (parameters), because the state at any given moment incorporates both parameters and activations. During sequential tasks, the final state at the end of one iteration becomes the initial state for the next ($S_0^{i+1} = S_T^i$); consequently, inputs or rewards from previous iterations can directly modify the parameters that control the model's behavior in future iterations.

Term	Emergent Models	Neural Networks	Turing Machines
p	Program region of state	Weights and biases	Program on tape
x	Input region of state	Input layer	Input symbols on tape
S_t	State (program + activations)	Activations	Entire tape configuration
f	Transition function	Entire network	Transition rules

Due to their theoretical Turing completeness, EMs can virtually implement any computational mechanism internally. This includes advanced capabilities like meta-learning, where the model develops strategies to optimize its own parameters in response to external performance signals (rewards). In other words, EMs can dynamically *self-program* based on feedback from the environment.

This is fundamentally different from traditional machine learning models, such as neural networks, which keep their parameters separate from activations, making direct parameter adjustments by inputs or outputs impossible during normal operation. This structural separation restricts traditional models from spontaneously developing higher-order behaviors such as meta-learning or inductive biases.

Inductive biases are characteristics that allow an optimization algorithm to exploit more efficiently hidden patterns or structures in data, leading to faster learning and better generalization. The human brain is rich in inductive biases that greatly enhance its ability to quickly learn and adapt, significantly outperforming current artificial neural networks in terms of generalization, learning speed, and data efficiency.

5.1 Theoretical explanation

Consider a traditional **Turing Machine (TM)**. Its initial state (the tape) can conceptually be divided into two distinct regions:

- An **input region** (x), containing symbols representing the specific input provided for computation.
- A **program region** (p), containing symbols encoding the algorithm or instructions that govern computation.

In a TM, the program p is explicitly defined externally by humans, while x changes according to the single input value. **Emergent Models (EMs)** adopt a similar conceptual division:

- The perturbed state $\tilde{S}$ of an EM encodes both the **program** (p) and the **input** (x).
- Unlike TMs, where p is explicitly defined, EMs determine the optimal program through optimization (e.g., genetic algorithms). Thus, p is implicitly *searched* rather than explicitly programmed.

Formally, at any iteration step i of a sequential operation task, an EM's initial perturbed state $\tilde{S}_0^i$ can abstractly be decomposed as:

$$\tilde{S}_0^i = (p^i, x^i)$$

where:

- p^i is the region encoding the program or parameters governing computational behavior.
- x^i denotes the current input encoded for processing.

p governs how the model behaves, and assuming Turing completeness, by varying p we can model any computable function g on every input x.

Influence of Input on the Program Region

Since an EM evolves by repeatedly applying a fixed transition function f across its entire state, changes in the input region x^i at timestep i can propagate and alter the program region p^{i+1} at the subsequent timestep, assuming state retention:

$$S_0^{i+1} = \tilde{S}_T^i$$

Assuming Turing Completeness, by controlling the initial state on the first iteration S_0^0, we can induce an arbitrary modification policy of p^{i+1} in response to x^i (or reward, that is fed as an input). We name the ability of an emergent model to build arbitrary meta-programs to modify its own internal parameters in response to inputs **Universal Meta-Learning**.

This grants the existence of an initial state configuration S_0^0 such that we can implement an arbitrary optimization and adaptation algorithm inside an EM.

Emergence of Higher-Order Behaviors

Assuming that:

- Inputs x include performance signals (rewards r) explicitly encoded into the state.
- The EM architecture has sufficient expressiveness, tending to Turing Completeness.
- We run an external optimization loop (genetic algorithm) to optimize S_0^0.

It logically follows that EMs can exhibit emergent higher-order behaviors, such as:

- **Meta-learning**: Developing strategies that modify internal programs in response to rewards, enabling rapid adaptation.
- **Inductive biases**: Evolving inherent structural assumptions to efficiently exploit hidden data regularities that speed up the learning process.
- **Self-healing (Antifragility)**: Spontaneously acquiring mechanisms to counteract disruptions, maintaining stability through adaptive responses.
- **Memory management**: Self-organizing strategies to dynamically retain or discard past information in such a way that grants best performances.

These behaviors can emerge within the EM's evolutionary optimization process: if beneficial behaviors arise randomly in one agent within the population, such agent will outperform its competitors (surviving more or learning faster), and become prevalent in subsequent generations.

5.2 A note on instability

In general, there will always exist a sequence of inputs $[x_0, ..., x_i]$ such that the model could 'unlearn' abilities and get out of the stability region. This will cause the model to collapse, and this is the reason why each model 'lives' for a limited time and then diverges ('dies'), similarly to living systems.

For this reason, a single emergent model can't be used for long-duration tasks, but it's always needed to keep a population of EMs with online genetic algorithms that continue to select the best-performing ones and prevent divergence by selecting the best agents and eliminating the divergent ones. This also makes the model continuously evolve and improve in time, similar to life in nature, where there's no distinction between a 'training' phase and an 'inference' phase. Stochastic transition functions may help in keeping this stability and contribute to exploration.

5.3 Learning process

Simplifying the landscape and analysing a stable emergent model (assume the best model of the population) during the learning process (red line), comparing it with the standard training of a neural network with backpropagation (blue line), the chart of the learning curve should look something like Figure 5.

The following chart is built to explain intuitively what happens when an EM gets trained. The process is highly simplified as it assumes there is no noise, instabilities, or points where the system gets stuck on local maxima, and it assumes that learning goes on linearly forever. A real chart would look much messier, with peaks going up and down, discontinuities, instabilities and growth towards a final saturation point.

The key point from the chart is that inductive biases and meta-learning allow learning at a higher speed as $t \to \infty$. The slope of the emergent model's learning curve for $t > T3$ is higher than the slope of the neural network. This describes the ability of inductive biases to exploit a higher fraction of the information coming from inputs, if compared with traditional forms of learning, which waste most of the information content they get exposed to.

For various periods of training time:

- $\mathbf{0 < t \leq T1}$: The neural network with backpropagation learns fast, until a saturation point (learns all the *easy* information). This is due to the fast convergence of backpropagation. The emergent model learns slower, since it is guided by genetic algorithms.
- $\mathbf{T1 < t \leq T2}$: The NN saturates (has learnt all the 'easy' information), continuing to learn but slower at a linear regime. Having no inductive biases prevents it from extracting information more efficiently from training data. First inductive biases start to emerge in the EM for evolutionary reasons. This allows extraction of information from data in a faster way, peaking in the meta-learning region.
- $\mathbf{T2 < t \leq T3}$: The NN continues to learn in the linear regime. The EM goes to full emergence of meta learning, where it learns at an exponential rate, until a further saturation point. The point where learning curves cross is denoted as the break-even point. From this point on, EMs are in advantage over NNs.
- $\mathbf{t > T3}$: NN continues in the linear regime. The EM has fully developed meta-learning and inductive biases, and continues learning at a higher rate than the NN. This advantage should continue indefinitely until final saturation or until divergence for instability of the EM.

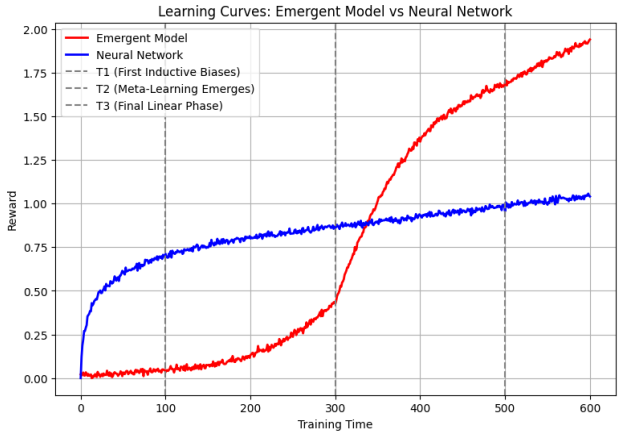


Figure 5: Simplified learning curves

6 Discussion

In the following subsections, the expected advantages and disadvantages of EMs are presented and discussed.

6.1 Advantages

We believe EMs hold significant potential to be a game changer, offering (at least theoretically) solutions to several limitations of current machine learning models. EMs, in contrast to current technologies, may exhibit the following advantages.

- **Spontaneous inductive biases**: they facilitate an accelerated learning process and reduce data requirements.
- **Meta-Learning**: extending inductive biases to enable the model to autonomously optimize its reprogramming strategy based on reward signals.
- **Increased expressivity with fewer parameters**: through recursion and adaptive computation, we can model more complex systems with fewer parameters, potentially reaching Turing completeness with adaptable external memory.
- **Better generalization and reasoning performance**: Emergent models are based on simple transition functions applied recursively over large state spaces. This gives EMs the tendency to model external functions in a highly compressed way, that minimizes Kolmogorov complexity. This induces generalization and is at the core of abstraction capabilities.
- **Higher flexibility**: leveraging genetic algorithms or Bayesian optimization to train for non-differentiable tasks, including reinforcement learning and interactions with unknown, black-box systems.
- **Life-like**: these models resemble natural processes, which could make them better at modeling natural behaviors and physics.
- **Dynamic optimization**: demonstrating superior performance in dynamic optimization scenarios, where objective functions evolve over time, through meta-learning capabilities that enable the model to learn and adapt fast.

6.2 Disadvatages

While EMs look promising, several challenges may hinder their training for complex tasks:

- **Stochastic optimization**: the reliance on stochastic genetic optimization, rather than deterministic methods like gradient descent, significantly increases the computational resources required for training.

- **Instability**: since the model's parameters are dynamic - the program evolves over time and is encoded in the state space- it is susceptible to spontaneous collapse during inference due to perturbation effects.
- **Combinatorial explosion**: the large state space necessitates an extensive search for optimal configurations, potentially leading to a combinatorial explosion.

7 Research Roadmap

The following steps outline the research roadmap for Emergent Models:

1. **Formalization and Verification**: Establish rigorous mathematical definitions and clear intuitive explanations to verify the theoretical foundations of EMs, ensuring they are well-defined at each point.
2. **Integration into RL Tasks**: Implement EMs in a simplified reinforcement learning scenario, such as navigating a dummy car within a 2D environment.
3. **Training and Evaluation**: Utilize black-box optimization techniques (e.g., genetic algorithms, Bayesian optimization) to identify optimal initial states for emergent behaviors. Evaluation criteria will include:
   - Convergence speed;
   - Stability;
   - Emergence of meta-learning and higher-order behaviors (assessed via learning curves);
   - Comparative analysis against neural networks regarding learning speed, efficiency, generalization capability, stability, robustness, and emergent behaviors.
4. **Publishing**: Publish a detailed paper on Emergent Models with rigorous definitions and experimental findings, either in a peer-reviewed journal or as a preprint on arXiv. Additionally, release an open-source repository to facilitate building and training EMs on reinforcement learning tasks.

8 Dummy 2-D Car Environment

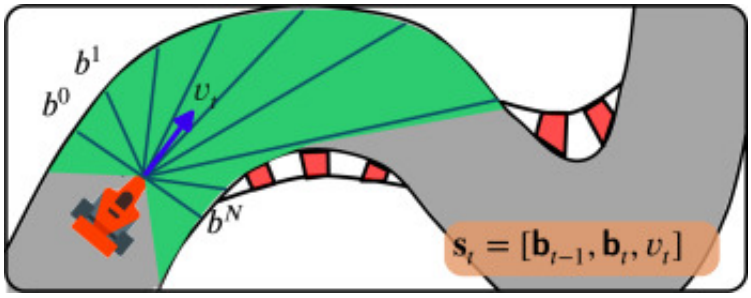


Figure 6: Dummy 2D car environment.

This environment represents a simplified 2-dimensional racing scenario for reinforcement learning. A car moves along a predefined track and can perform actions such as steering, accelerating, and braking. The primary objective is to complete the track in the shortest possible time without exiting the track boundaries or colliding with obstacles.

The emergent model serves as the policy that decides the car's actions based on sensory inputs (black visual rays in Figure 6). The interaction follows standard reinforcement learning principles as illustrated in Figure 7.

Observation Variables (Model Inputs)

- **Speed**: Current speed of the car.
- **LIDAR Sensors**: Distances from the car to nearby obstacles (visual rays shown in black).

Action Variables (Model Outputs)

- **Acceleration**: Controls the car's speed (accelerate/brake).
- **Steering**: Direction control (left/right).

Reward Structure

The reward encourages efficient and safe driving behavior by accounting for:

- Maintaining the car on track (penalties for going off-track).
- Total distance traveled along the intended track direction.
- Average speed maintained along the correct path.
- Computational efficiency (penalty proportional to the computation time to encourage faster reasoning).

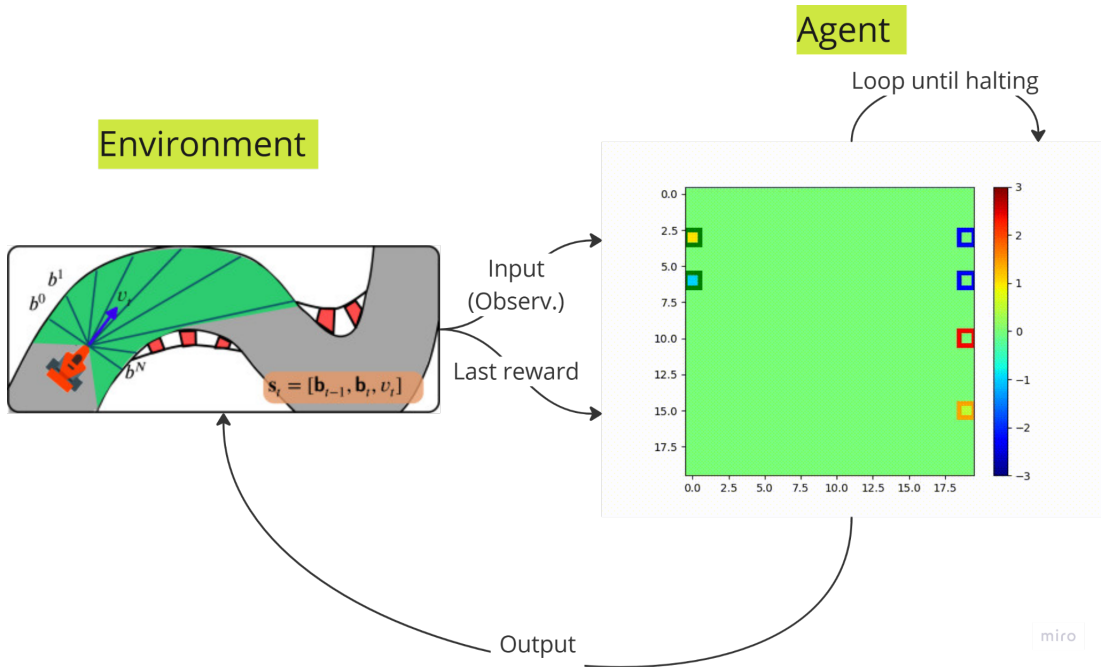


Figure 7: Agent-environment interaction in our dummy 2D car scenario.

9 Further Research Steps

To further advance and thoroughly evaluate Emergent Models, we propose the following research steps:

- **Increase Robustness and Generalization**: Train and validate EMs on increasingly complex tasks, including arithmetic reasoning and natural language processing, to enhance their capability to generalize to more challenging inputs.
- **Expand Linguistic Understanding**: Evaluate and refine the inherent language modeling abilities of EMs, assessing their performance in linguistic tasks.
- **Evaluate Transfer Learning**: Investigate the potential of EMs to transfer learned knowledge effectively across different domains and tasks.

- **Develop Advanced Capabilities**: Deeply explore meta-learning techniques and the spontaneous emergence of higher-order behaviors, aiming for better stability and sophisticated adaptability in EMs.
- **Optimize Performance and Efficiency**: Improve the core components of EMs—including transition, encoding, decoding, and halting functions—to achieve higher performance levels while simultaneously reducing computational requirements.

References

[1] Tanweer Alam et al. "Genetic Algorithm: Reviews, Implementations, and Applications". In: *International Journal of Engineering Pedagogy (iJEP)* (2020). arXiv: 2007.12673 [cs.NE]. URL: https://arxiv.org/pdf/2007.12673 (visited on 04/05/2025).

[2] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[3] Bert Wang-Chak Chan. "Lenia-biology of artificial life". In: *arXiv preprint arXiv:1812.05433* (2018).

[4] Martin Gardner. "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game "Life"". In: *Scientific American* 223 (Oct. 1970), pp. 120–123. DOI: 10.1038/scientificamerican1070-120. URL: http://dx.doi.org/10.1038/scientificamerican1070-120 (visited on 04/05/2025).

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Third. Boston: Pearson/Addison Wesley, 2007. ISBN: 0321455363 9780321455369 0321462254 9780321462251 0321455371 9780321455376 0321476174 9780321476173. URL: http://infolab.stanford.edu/~ullman/ialc.html.

[6] Ming Jin. "Preparing for black swans: The antifragility imperative for machine learning". In: *arXiv preprint arXiv:2405.11397* (2024).

[7] Kai Malcolm and Josue Casco-Rodriguez. "A comprehensive review of spiking neural networks: Interpretation, optimization, efficiency, and best practices". In: *arXiv preprint arXiv:2303.10780* (2023).

[8] Tom M Mitchell. "The need for biases in learning generalizations". In: ().

[9] Huimin Peng. "A comprehensive overview and survey of recent advances in meta-learning". In: *arXiv preprint arXiv:2004.11149* (2020).

[10] Paul Rendell. *Turing Machine Universality of the Game of Life*. 1st ed. Emergence, Complexity and Computation. Springer Cham, 2015, pp. XV, 177. ISBN: 978-3-319-19841-5. DOI: https://doi.org/10.1007/978-3-319-19842-2. published.

[11] Robin M Schmidt. "Recurrent neural networks (rnns): A gentle introduction and overview". In: *arXiv preprint arXiv:1912.05911* (2019).

[12] Jasper Snoek et al. "Scalable Bayesian optimization using deep neural networks". In: *International conference on machine learning*. PMLR. 2015, pp. 2171–2180.

[13] Nassim Nicholas Taleb and Raphaël Douady. "Mathematical definition, mapping, and detection of (anti) fragility". In: *Quantitative Finance* 13.11 (2013), pp. 1677–1689.

[14] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[15] Eric W. Weisstein. *Universal Cellular Automaton*. *MathWorld*–A Wolfram Web Resource. URL: https://mathworld.wolfram.com/UniversalCellularAutomaton.html (visited on 04/05/2025).

[16] Wikipedia contributors. *Turing completeness — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Turing_completeness&oldid=1279841424. [Online; accessed 5-April-2025]. 2025.

[17] Stephen Wolfram. *Cellular Automata and Complexity: Collected Papers*. https://www.stephenwolfram.com/publications/cellular-automata-complexity/. Accessed on: [Date you accessed the page].