A methodology combining reinforcement learning and simulation to optimize biofabrication protocols applied to the simulated culture of epithelial sheets.

Alberto Castrignanò[a], Roberta Bardini[a], Alessandro Savino[a], Stefano Di Carlo[a]

[a]*Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy., Corso Duca degli Abruzzi, 24, Turin, 10129, Italy*

Abstract

Tissue Engineering and Regenerative Medicine aim to replicate and replace tissues for curing disease. However, full tissue integration and homeostasis are still far from reach. Biofabrication is an emerging field that identifies the processes required for generating biologically functional products with the desired structural organization and functionality and can potentially revolutionize the regenerative medicine domain, which aims to use patients' cells to restore the structure and function of damaged tissues and organs. However, biofabrication still has limitations in the quality of processes and products. Biofabrication processes are often improved empirically, but this is slow, costly, and provides partial results. Computational approaches can tap into biofabrication underused potential, supporting analysis, modeling, design, and optimization of biofabrication processes, speeding up their improvement towards a higher quality of products and subsequent higher clinical relevance. This work proposes a reinforcement learning-based computational design

space exploration methodology to generate optimal protocols for the simulated fabrication of epithelial sheets. The optimization strategy relies on a deep reinforcement learning algorithm, the Advantage-Actor Critic, which relies on a neural network model for learning. In contrast, simulations rely on the `PalaCell2D` simulation framework. Validation demonstrates the proposed approach on two protocol generation targets: maximizing the final number of obtained cells and optimizing the spatial organization of the cell aggregate.
Keywords: reinforcement learning, optimization, simulation, biofabrication, computational biology

---

1. Introduction

Biofabrication is the automated generation of biologically functional products with a structural organization from living cells and biomaterials through bioprinting or bioassembly and subsequent tissue maturation processes [1, 2]. Biofabrication includes a rich set of methodologies to recapitulate the structural and functional organization and heterogeneity of biological parts, from cell aggregates to tissues, with the challenging goal of fabricating organs [2]. Tissue Engineering (TE) applications to Regenerative Medicine (RM) aim to use patients' cells to restore the structure and function of damaged tissues and organs [1]. While few products undergo clinical translation, full tissue integration and homeostasis are still far from reach [3]. In these domains, the complexity of target biological systems and the intrinsic uncertainty in their behavior limits the capability to design optimal products and reliably con-

trol their structure and function during biofabrication [4], limiting products' clinical relevance. The biofabrication processes' design space is a multidimensional region where input variables and process parameters determine the final product quality [5]. Hence, implementing an effective Design Space Exploration (DSE) of a biofabrication process is challenging and time-consuming [4]. Empirical DSE has high costs in terms of time and resources. Nevertheless, most often, advancements in the field are demanded by human effort or luck. Automation and digitalization increase the yield of experimental campaigns [6] and thus of DSE. However, they still need to implement an empirical approach, which fails to account for a good exploration even if experiments are parallel, faster, and more reliable. Besides the inefficient brute-force screenings, experimental designs of biological protocols can rely on established methodologies to optimize the usage of resources [7], such as One Factor At a Time (OFAT) [8, 9], factorial experimental design [10] and Design of Experiment (DoE) computational techniques [11, 9]. The latter supports research design by enabling efficient, systematic exploration and exploitation of complex design spaces and can tackle multi-factorial problems [12, 13, 14]. Nevertheless, DoE approaches show limitations in supporting complex processes such as biofabrication due to the enormous number of experiments required to map the design space and to build predictive models from data. Computational approaches and Artificial Intelligence (AI) can support the design and optimization of biofabrication processes beyond the limitations of the empirical approach [15, 16]. In particular, computational

modeling and optimization exploit the power of computation to explore system behavior and to conduct systematic DSE, accounting for the relations between complex parameter configurations and process outputs. To impact the biofabrication domain, computational approaches must rely on white-box models of the physical process to explicitly represent the relations between the generated protocols and the biological mechanisms modeled [17]. Nevertheless, most computational approaches to biofabrication overlook biological complexity and primarily focus on the non-living components of the process. Models that express the complexity of the physical process have huge state spaces [18]. Model complexity requires computational DSE to rely on metaheuristics for computational feasibility [19]. Few approaches combine computational DSE and white-box modeling of biological complexity. At the same time, the potential combinations of different DSE and white-box simulation approaches are countless. Considering this, an essential first step is to assess the computational feasibility and the relevance of results for different combinations.

Previous work provided a model-based Optimization via Simulation (OvS) approach [20], where optimization relied on a Genetic Algorithm (GA) [21] and orchestrated the white-box simulation of a discrete model of the biofabrication of epithelial sheets [22]. This work continues the exploration of different DSE and white-box simulation approaches, with the application of Machine Learning (ML) and the simulation of white-box models to optimize biofabrication. In particular, optimization relies on a variant of

the Synchronous Advantage Actor-Critic (A2C) Deep Reinforcement Learning (DRL) algorithm [23] and orchestrates the simulation of vertex models *via* `PalaCell2D`, a simulator of epithelial sheets growth [24, 25]. Validation of the proposed methodology demonstrates its ability to maximize the final number of cells obtained and optimize their geometrical configuration. This work further assesses the computational feasibility and biological relevance of different combinations of computational DSE and white-box simulation approaches to optimize biofabrication. This computational approach aims to overcome limitations posed by empirical DSE by providing rational guidance to experimental campaigns, supporting innovation and quality standards in biofabrication processes, and increasing clinical and industrial translatability of the resulting products.

The remaining of the paper illustrates state of the art (subsection 1.1, the proposed methodology (section 2), the validation strategy and experimental results (section 3), and future directions for advancing this research (section 4).

1.1. Background

1.1.1. Models of biological complexity

The relations between the generated protocols and the biological mechanisms modeled must be explicit to improve its performance and provide an explanation rooted in biological knowledge for provided results. Thus, a white-box modeling approach is preferable but has the drawback of higher model complexity, which implies long computational times for simulations.

Several white-box modeling approaches for biological complexity exist, including continuous, discrete, or hybrid formalisms [26, 27]. Ordinary Differential Expressions (ODEs) usage is frequent in systems biology. In contrast, other approaches, such as Agent-Based Models (ABMs) and discrete models, have a great potential to enrich the toolkit of modelers in computational biology [28, 29], yet are less frequent, also due to their lack of standardization [30]. High-level Petri Nets (PN) models are accessible, formally complete, and naturally support modeling of complex biological processes [31] such as ontogenesis [32, 33] or the relations between a host organism and the microbiota [34, 35]. Other approaches, such as vertex models [36], allow capturing epithelial sheets' dynamic behavior.

1.1.2. Model-based Design Space Exploration

In particular, OvS approaches combine simulation and optimization to systematically explore behavioral trajectories of a model after different configurations of parameters [20]. Considering that each trajectory corresponds to a biofabrication process design, OvS supports the computational generation of simulated biofabrication protocols to test in the laboratory to prioritize more informative experiments to optimize the process.

Two main categories of OvS approaches [20]. On the one hand, metamodel-based OvS uses a metamodel to estimate the input-output relations of the simulation model [37]. The substitution of simulation models with metamodels in OvS has the advantage of reducing the computational times [38].

However, this has a cost in terms of accuracy [39] and explainability of results. Metamodel-based OvS of biofabrication mainly targets the non-living parts of the process. For example, ML approaches proved helpful in optimizing bioprinting [40, 41], also by finding the best combination of bio-inks and scaffolds through multi-objective optimization [42]. Deep Learning (DL) proved able to optimize a bioprinting process, predicting optimal configurations of printing parameters for specific objectives [43, 44, 45]. Fewer metamodel-based OvS approaches account explicitly for the living parts in biofabrication. For example, DL predicted functional activations from different environmental stimuli [45]. Also, an Artificial Neural Network (ANN) predicted the induction of osteogenesis from biomechanical loading parameters [46]. However, these approaches rely on black-box models of the simulated system, which fail to provide the desired explainability.

On the other hand, model-based OvS relies on the simulation of a white-box model of the process, which provides higher explainability but has high computational costs [20]. To support real-world applications, OvS must provide informative and fully explainable results. Model-based OvS techniques tackle the complexity of the corresponding design space with optimization strategies based on heuristics and meta-heuristics, combining stochastic exploration with local exploitation of optimal solutions [47]. Nevertheless, few model-based OvS approaches exist in biomanufacturing. Most of them focus on the non-living parts of the biofabricated construct and the structural setup before product maturation. For example, Finite Elements Model (FEM)-

based modeling and optimization are primarily employed in biomechanical research [48, 49, 50, 51, 52], where, for simulations, Computational Fluid Dynamics (CFD) modeling unravels hydrodynamic patterns and study the influence of fluid dynamics on cell behavior during tissue maturation [53, 54, 55]. Model-based OvS approaches that explicitly model and simulate the complexity of its biological components can leverage several simulation and optimization approaches. Independently of the modeling formalism chosen, comprehensive models of biological systems are complex and have huge state spaces [18], requiring model-based OvS approaches to rely on metaheuristic algorithms [19] with a search component [56] for optimization. Metaheuristic algorithms can be either based on single solutions or populations of solutions. On the one hand, single-solution algorithms generate and improve a single solution by local search, which implies the risk of converging to local optima. This category includes several algorithms, such as simulated annealing [57] and tabu search [58]. On the other hand, population-based metaheuristics work with a population of candidate solutions during the search process. By fostering the diversity of solutions within the population, these algorithms reduce the risk of converging to local optima. This category includes, among others, GAs [59], Particle Swarm Optimization (PSO) [60] and Ant Colony Optimization (ACO) [61].

A previous work [22] exemplifies a model-based OvS approach combining evolutionary computation and discrete models simulation to generate protocols for the biofabrication of human epithelial monolayers computation-

ally. Simulation relies on discrete models whose high abstraction level can express more detailed intracellular dynamics while still simulating hundreds of cells. Still, the computational complexity of such simulations is high and requires a metaheuristic approach to optimization to reach optimality. The latter relies on a GA that iteratively generates new protocols to simulate, evolves them, and evaluates their performance by simulation. After that, the best-performing ones pass to the next generation until reaching optimality. This approach has the advantage of ensuring convergence within feasible computational times.

However, this sticks to an offline computational DSE paradigm, performing protocol generation and their simulation in separate moments. This shows limitations in a wetware-in-the-loop scenario, where protocols should dynamically adapt to the current system state given a final target. In particular, the evolutionary process only learns from the outcome of simulations. It selects the protocols that generate the best products but does not learn from any intermediate point along the simulation, discarding valuable information for the optimization process. Also, protocol generation occurs offline from simulation, which would fail to adapt to the dynamic responses of a biofabricated system in real time.

Another approach for the computational DSE of processes with large state spaces is the combination of DRL with white-box simulations, which overcomes this limitation by learning continuously by interacting with the system itself. Reinforcement Learning (RL) is a ML approach, whose core is

an agent capable of self-learning by interacting with an environment, where numerical rewards guide the interactions and the training process [62, 63]. Given this considerable advantage, RL has broad application for optimizing real-world manufacturing processes [64]. In DRL, the capability of DL to work as feature extractors allows the application of RL to problems with high-dimensional state spaces, such as biofabrication processes [18].

2. Materials and Methods

This work proposes a computational DSE methodology where a RL model and the A2C algorithm learn optimal biofabrication protocols to optimize simulated biofabrication. Fig 1 presents the implementation of the proposed methodology.

The exploration engine (Fig 1.A) relies on a single-agent variation of the A2C algorithm and a ANNs as a learning model; the training process occurs through the interaction between Actor and Critic, and learning rules and rewards adapt to the particular application (Fig 1.D). The simulator interface (Fig 1.B) allows the exploration engine to interact with the simulator of the process model. The simulator of the process model (Fig 1.C) works as a learning environment (Fig 1.E) for the training process.

The training process consists of epochs. Each learning epoch aims to generate the best possible action to take the learning environment closer to the target and contains the three following steps (numbered in Fig 1).

1. **Observation collection**: the exploration engine reads the simulation

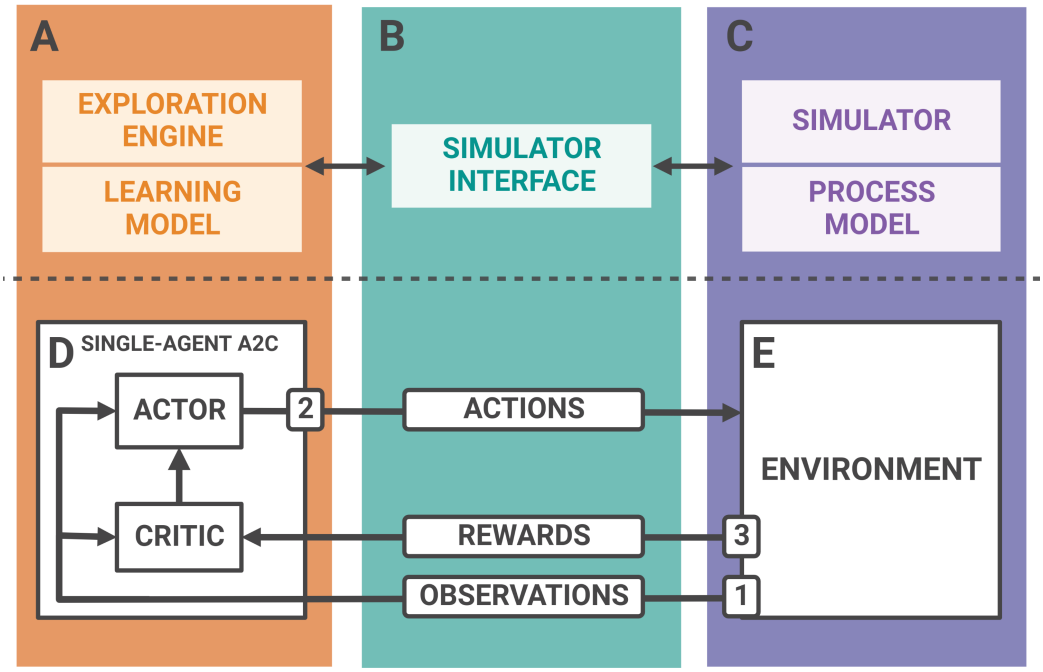


Figure 1: The components of the proposed framework and their interactions during the training process. (A) The exploration engine interrogates the learning model at each epoch. (B) The simulator interface allows the exploration engine to use the simulation performed by the simulator on the process model as a learning environment. (C) The simulator simulates the process model to simulate the application. (D) The proposed single-agent variation of the A2C algorithm interrogates the underlying ANNs, which generates distributions of actions to learn the best policy for a specific application based on the target of the optimization process through the interaction between the Actor and Critic. During each epoch, the optimization process flow includes these three steps, mediated by the simulator interface: (1) the communication of simulation states as observations from the learning environment to the learning model through the exploration engine; (2) the communication of a chosen action generated by the A2C algorithm by interrogating the underlying learning model; (3) the communication of rewards from the learning environment to the Critic in the learning model to evaluate the chosen action to further the training process.

state as an observation from the learning environment, passing it to the learning model.

2. **Action generation**: the learning model, based on the received observation, generates an action that the exploration engine sends to the simulator for it to perform in the learning environment.
3. **Reward collection**: the exploration engine reads the reward associated with the performed action from the learning environment. It passes it to the Critic in the learning model to evaluate the chosen action's effect for approaching the optimization process's target Fig 1.C.

Each epoch results in a generated action. At the end of the training process, the sequence of generated actions to reach the optimal target defines an *optimal biofabrication protocol*.

The following subsections provide a detailed description of the exploration engine (subsection 2.1), the learning model (subsection 2.2), and the simulator interface (Appendix B. Simulator interface), which are the constitutive components of the proposed methodology. Since they are application-specific, the Results and Discussion provides a more detailed description of the simulator and the process model. Finally, Training process illustrates the training process.

2.1. Exploration engine

This work relies on a RL model for learning. RL aims to find an optimal agent behavior to get maximal rewards. In particular, the exploration engine

relies on Actor-Critic algorithms [65], which combine Q-learning [66] and Policy Gradients [67]. Policy gradient methods directly model and optimize a policy, representing an optimal agent behavior. In Actor-Critic, the Actor component generates an action to perform over the learning environment by random sampling from the probability distribution of all performable actions in a given state, relying on the same objective function as the *REINFORCE* algorithm [68].

The policy π is usually modeled with a parameterized function for the parameter vector θ: $\pi_\theta(a|s)$. The value of the reward $J(\theta)$, which works as the objective function, depends on the chosen policy (see Equation 1). In Equation 1, $d^\pi(s)$ is the stationary distribution of Markov chain [69] for π_θ, $a \in \mathcal{A}$ are actions, $s \in \mathcal{S}$ are states, $V^\pi(s)$ and $Q^\pi(s,a)$ are, respectively, the value of state s and the value of a (s,a) pair, when following a policy π [70].

$$J(\theta) = \sum_{s \in \mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \tag{1}$$

The goodness of the performed action a is evaluated based on the reward $J(\theta)$ generated by its execution in the learning environment. According to the *Policy Gradient Theorem* [71], the derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy π_θ. This allows us to compute the objective function without considering the derivative of the state distribution $d^\pi(s)$, untying calculations from the double dependency on action selection and states distribution [70]. In other

words, the state distribution depends on the policy parameters, but there is no dependency of the policy gradient on the gradient of the state distribution [72]. This simplifies the calculation of the gradient $\nabla_\theta J(\theta)$ dramatically, as in Equation 2.

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a) \pi_\theta(a|s) \\ &\propto \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) \end{aligned} \tag{2}$$

In Actor-Critic algorithms, the Critic component evaluates the goodness of the performed action a computing the reward $J(\theta)$ as the Mean Squared Error (MSE) between the current value and the best possible action [72], transforming Equation 1 into Equation 3, where T is the last timestep.

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \left[G_{t+1} \frac{\delta}{\delta \theta} \log \pi_\theta(t) + \frac{1}{2} \left(Q(s_t, a_t) - V(s_t) \right)^2 \right] \tag{3}$$

Since random sampling from action distributions can determine high variability in the log probabilities generated, letting the policy converge to a sub-optimal choice, the A2C algorithm [23] adjusts the log probabilities by multiplying them by the advantage. The advantage is a factor that quantifies how better the chosen action compares to other possible actions (Equation 4).

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{4}$$

By applying Equation 4 within Equation 3, we obtain a more stable

training process, as depicted in Equation 5, where $A(s_t, a_t)$ is the advantage.

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{t=0}^{T-1} \left[A(s_t, a_t) \frac{\delta}{\delta\theta} \log \pi_\theta(t) + \frac{1}{2} \left(Q(s_t, a_t) - V(s_t) \right)^2 \right] = \\ &= \sum_{t=0}^{T-1} \left[A(s_t, a_t) \frac{\delta}{\delta\theta} \log \pi_\theta(t) + \frac{1}{2} A(s_t, a_t)^2 \right]\end{aligned} \tag{5}$$

The exploration engine takes the actions generated by the ANN model to use them in the environment. Then, after the effect of such actions, it generates the values to compute the A2C objective function, which in turn guides the update of the ANN weights through backpropagation [73].

Synchronous A2C [73] devises more agents acting simultaneously and combines the results they independently obtain, resolving inconsistencies. Each agent is a different ANN model, comprising an Actor and a Critic component as detailed in Introduction. To balance the vast capabilities of DRL with the computational feasibility of simulation-based DSE, this work proposes a simplified variant of the A2C algorithm where learning relies on a single agent. Overcoming the drawback of synchronizing multiple agents has a cost in accuracy. However, it allows to speed up the interaction with the environment while still expressing the capability to exploit response information and dynamically adapt to the system evolution.

2.2. Learning model

The learning model leverages the general ANN structure devised by the A2C algorithm [73]. The ANN structure includes a backbone (Fig 2.A) that processes the simulation states as observations and provides its outputs as an input to both the Critic and the Actor components. The ANN model implementation relies on the `tensorflow` library (version 2.9.1) [74], while the backbone relies on the ResNet18 model [75].

2.2.1. Model structure

The Actor (Fig 2.B) processes the backbone output to generate actions. It can produce both discrete and continuous actions. In particular, for each discrete one, the Actor generates a single value among the possible ones, leveraging a softmax function. The Actor component clips the output values of variance layers and discrete action layers between 10^{-2} and 10^{3} to avoid numerical issues in computation during the training phase. On the other hand, for each continuous one, it generates two values: (μ, the mean, and σ, the variance of a Gaussian distribution from which it randomly samples a value for generating the action. The Critic subnetwork (Fig 2.C) processes the backbone output to generate the current state value.

2.3. Model adaptability

The ANN model adapts to the application and training process. The Actor and Critic components have a fully connected Multi-Layer Perceptron (MLP) as an input layer. The number of input neurons in such layers is a

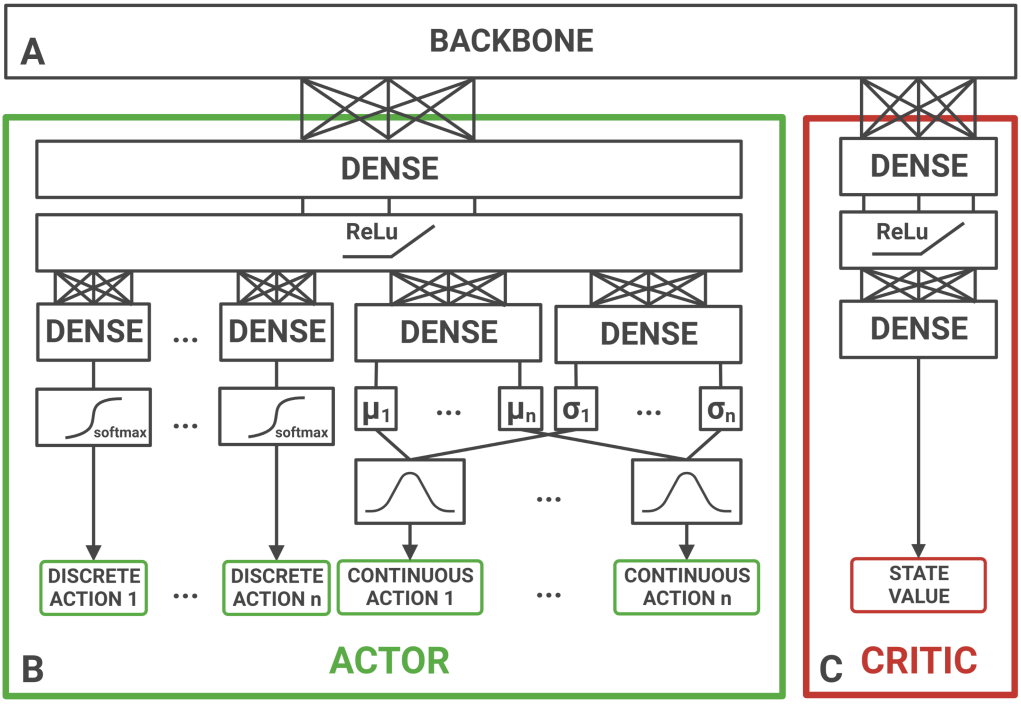


Figure 2: Actor-Critic ANN model. (A) A single backbone processes observations from the learning environment to feed the Critic and the Actor. (B) The Actor processes the backbone output to generate discrete and continuous actions over the learning environment. The architecture ANN adapts to the type and number of actions to generate. For discrete actions, the number of parallel output layers (based on softmax activations) equals the number of discrete actions to be generated (in the Figure, ranging from 1 to n). Each layer has an output size corresponding to the number of possible values for the discrete action it handles. For continuous actions, the number of action distributions in output and the number of neurons in both mean (μ) and variance (σ) layers equals the number of continuous actions to be generated (in Figure, ranging from 1 to n). Distributions support sampling of the continuous value for each of the actions considered. (C) The Critic processes the backbone output to generate the current state value. The ANN model implementation relies on the `tensorflow` library (version 2.9.1) [74], whilte the backbone relies on the ResNet18 model [75].

hyperparameter for the training process. The size and shape of data provided to the ANN as an input determine the size and shape of its output layers. The width and height of the image-like observation of the environment, and channels, which is the number of color-like channels, set the input shape of the ANN.

The ANN output size adapts to the set of actions to generate, ranging from 1 to n in Fig 2, for both continuous and discrete actions. The number of continuous actions to generate sets the number of action distributions in output and the number of neurons in both the mean (μ) and the variance (σ) layers. Distributions support sampling of the continuous value for each of the actions considered. The set of possible values for discrete actions sets the number of parallel output layers with softmax activations.

2.4. Training process

The training process, described in Training process, relies on the `train` class, which interacts both with the environment and the model, implementing the A2C algorithm. The class manages training parameters, such as the learning rate (`lr`) and the discount rate (`gamma`), and supports getting relevant information in real-time and setting training length and starting epoch, allowing to restore of the training from a previous state through the saved data if needed.

At first, the training creates the ANN model with the required input size and its optimizer (Algorithm 1, line 1). If restoring a previous state, it loads

Algorithm 1 Training process

```
1: learningModel = ActorCritic(learningParameters)   ▷ Instantiate ANN model and optimizer
2: e = PalacellEnv()                                 ▷ Instantiate environment
3: while current_epoch <= epochs do                  ▷ Training loop
4:     e.reset()                                     ▷ Reset environment
5:     initialObservation = e.getObservation()       ▷ Get initial observation
6:     actions = learningModel(initialObservation)   ▷ Generate actions
7:     e.step(actions)                               ▷ Apply actions over the environment
8:     finalObservation = e.getObservation()         ▷ Get environment observation
9:     reward = e.getReward(finalObservation)        ▷ Get corresponding reward
10:    learningModel.applyReward(reward)             ▷ Apply reward to ANN model
```

checkpoint files. Then, it instantiates the learning environment (Algorithm 1, line 2) and checks its integrity. The training loop continues until it reaches the maximum epoch number (Algorithm 1, line 3). The loop includes data collection through a `GradientTape` tensorflow section to track the data flow in the model. It is possible to show information in real-time during the training loop. Each epoch in the training loop consists of six main steps, operated by the `train` method (Algorithm 1, line 4-10).

The first step (Algorithm 1, line 4) prepares the environment for the next learning episode, starting from the new initial state, by calling the `reset` method (Algorithm 1, line 4), and getting the first observation (Algorithm 1, line 5). The learning model uses the collected observation to generate actions (Algorithm 1, line 6). It also generates the probability of getting the obtained

action and the state value that will support the loss computation at the end of the epoch. The `step` method (Algorithm 1, line 7) applies the generated actions on the learning environment. After that, the process gets a new observation of the learning environment, capturing its new state after the execution of the actions, and collects the corresponding reward (Algorithm 1, line 8-9). The training feeds the rewards into the learning model for learning (Algorithm 1, line 10). The Q-values and advantages, the saved state value, and the probability of getting the obtained action allow computing the objective functions defined by A2C for both Actor and Critic losses. The two losses are summed together and used by the `GradientTape` section as the loss to compute the gradients for backpropagation in the ANN through the optimizer.

The `train_manager` class allows implementing different environments for different sets of hyperparameters to perform distinct parallel training processes (Fig 3.A) *via* its `parallel_train` function, relying on the `process` class from the Python module `multiprocessing`, and a `pipe`, that is, a communication channel, passed to each subprocess. Each subprocess contains a whole and independent training process, based on the interactions between the `train` class (Fig 3.B), the `environment` file implementation (Fig 3.C), and the learning model (Fig 3.D), that allow the individual A2C agent (Fig 3.E) to learn by iteratively interacting with the learning environment (Fig 3.F). Finally, each training subprocess is stored so that it is possible to request real-time information through the pipe and wait for all the subprocesses to

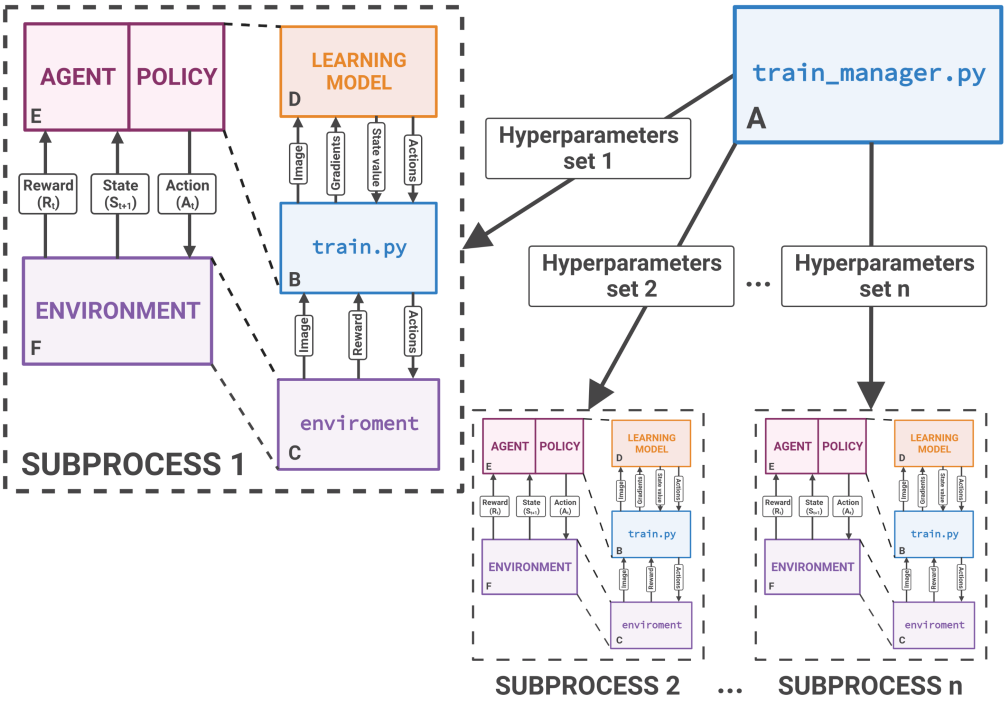


Figure 3: The proposed implementation of the A2C learning algorithm to support multiple training processes running in parallel. The `train_manager` (A), which consists of a single function, `parallel_train`, runs multiple training processes, each one with a different hyperparameter set, in parallel. Each subprocess contains a whole and independent training process based on the interactions between (B) the `train` class, (C) the `environment` file implementation, and the (D) learning model, that allows the individual A2C agent (E) to learn by iteratively interacting with the learning environment (F).

end their training.

3. Results and Discussion

The proposed methodology is intended to perform DSE of protocol generation for real-world applications and physical biofabrication processes. In perspective, the optimization engine will interact with an automatic culture system, whose specific effectors define tunable parameters forming a biofab-

rication protocol. A first step towards real-world applications is optimizing protocols for simulated biofabrication.

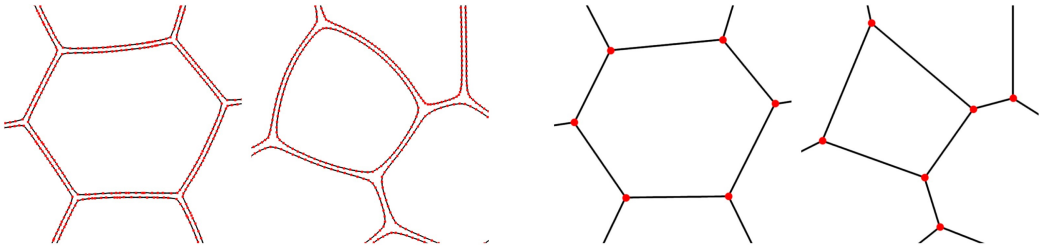
In particular, this work relies on `PalaCell2D`, a simulator of the proliferation of epithelial cells oriented to tissue morphogenesis [24, 25]. The `PalaCell2D` setup allows simulating tissue growth under different compression stimuli (see `PalaCell2D` setup). The optimization engine must communicate with the biofabrication process to control such stimuli. The latter will imply to interface with a bioreactor for real-world applications. To this aim, an implementation of a simulator interface (see Appendix B. Simulator interface) supports the interaction between the optimization engine and the `PalaCell2D` simulator. Consequently, a protocol consists of a sequence of values for the `PalaCell2D` tunable parameters. Then, optimized protocol generation (see Optimized protocol generation) aims to find the parameter values at each simulation step for optimizing the final product obtained at the end of the simulation.

Experiments relied on an AMD Ryzen 9 5950X 16-Core 2.2GHz processor with 64GB RAM. Training processes took between `1500` and `2000` seconds execution time to complete an epoch. The execution environment leveraged the container platform Singularity [76].

3.1. `PalaCell2D` setup

`PalaCell2D` relies on a vertex model [36] that represents cells through their membrane, modeled as a set of vertices (Fig 4). `PalaCell2D` simulations

iteratively evaluate the vertices positions for each cell and compute the forces exerted on each vertex.



(a) PalaCell (b) Vertex model

Figure 4: `PalaCell2D` models cell membranes as sets of vertices (the red dots) (a), basing on the vertex model approach (b) (taken from Conradin et al., 2021 [24] and Conradin et al., 2022 [25]).

In the `PalaCell2D` model, the number of vertices modeling a cell changes dynamically with its size so that the density of vertices remains uniform along the cell membrane. It supports the simulation of (i) mechanical properties, (ii) internal and external forces from the cellular, and (iii) extracellular compartments over the membrane. `PalaCell2D` supports simulating cellular processes such as apoptosis and proliferation. Simulation controls them based on the internal cellular pressure. The latter depends on defined model parameters set for the simulation (as in Eq (6)): the cell mass m, the cell area A, the pressure sensitivity η, and the target density ρ_0.

$$p_{int} = \eta(\frac{m}{A} - \rho_0) \tag{6}$$

Internal pressure controls the growth in the mass of the cell. The mass changes with a rate that depends on the external pressure p_{ext} applied on

the cell, the target area A_0, the mass growth rates during proliferation ν and relax ν_{relax}, the simulation time step dt and the pressure threshold p_{max} above which the proliferation stops as inEq (7).

$$dm = \begin{cases} \nu A(1 - \frac{p_{ext}}{p_{max}}) & \text{if proliferating} \\ -\nu_{relax}(A - A_0)(1 - \frac{p_{ext}}{p_{max}})dt & \text{otherwise} \end{cases} \tag{7}$$

External pressure p_{ext} depends on the contact with other cells and the external force F_{ext}. The latter is either local (i.e., the interaction of a cell with a wall of the culture environment) or global (i.e., the force exerted over the whole culture environment).

The simulation scenario used for validation controls cell proliferation by applying a global external force (in Eq (8)). The parameter a_{prolif} indicates the probability of controlling the switch to the proliferation state in cells.

$$p_{switch} = \begin{cases} a_{prolif}(1 - \frac{p_{ext}}{p_{max}}) & \text{if } p_{ext} <= p_{max} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

In the experiment proposed in [24], F_{ext} models an external pressure exerted on a deformable capsule that acts on the cellular vertices with an intensity that depends on their distance from the capsule center and the capsule radius.

3.2. Optimized protocol generation

The goodness of generated protocols depends on their capability to generate the target products at the end of a simulation. Protocols control parameter values representing the initial positioning of cells and the compression stimuli along the simulation (see Appendix A. Framework setup parameters). Thus, they have two sections:

1. the initial section sets the value of the `initialPos` parameter once;
2. the second section sets the value for the `comprForce` and `compressionAxis` parameters at each simulation step.

Within this structure, the framework aims to learn the values for each parameter in an optimal protocol. Objective-specific metrics measure the similarity of the protocol-based final product to the target. Such similarity is the measure of protocol performance. The training process evaluates metrics every five epochs to identify actions leading the simulation state closer to the target.

The capability of the DSE is assessed by demonstrating the framework capabilities to generate optimized protocols to obtain two different targets: (1) the maximal number of cells (see Target 1: maximal final number of cells) and (2) a circular patch of cells (see Target 2: a circular patch of cells). Both targets recapitulate relevant ones for the biofabrication of epithelial sheets. The first includes maximizing the number of cells obtained, supporting the cell density required by a biomimetic Tissue Engineering and Regenerative

Medicine (TERM) product to replace epithelia [77]. The second introduces the element of spatial control, aiming to optimize cell density and the epithelial sheet's position and spatial organization within the culture system, still maximizing the number of cells. The latter is a first step towards control of supracellular architecture, one of the significant open challenges in TERM biofabrication [78].

The following Sections describe the performed experiments and their results, including the learning performance along the training process and the generated optimal protocols, showing the impact of different hyperparameters combinations over learning performance. Since an exhaustive DSE of the simulated biofabrication processes targets is unfeasible in finite computational times [4], results aim to analyze the evolution of the learning processes reaching optimality rather than its convergence to a global optimum.

3.3. Target 1: maximal final number of cells

The first target aims at the maximal number of cells within the epithelial sheet at the end of a simulation. Tunable parameters in the protocol are `compressionAxis` and `comprForce`, and the reward value at the end of each epoch is the increment in the number of cells compared to the previous epoch. An increase in the mean number of cells across epochs indicates the training is moving toward optimality. A decrease in the variance of the number of cells indicates that the training process needs to explore new solutions. Instead, it exploits effective actions to get closer to the target.

3.4. Framework setup for Target 1

In the framework setup (see Appendix A. Framework setup parameters), the simulated space in the environment is a `400x400x1` grid. Learning targets compression stimuli and not the initial position of the cell, which holds constant (`initialPos=(x=200, y=200)`). During training, the framework operates the simulator to execute two main phases: initial cell positioning before the simulation starts, and `compressionAxis` and `comprForce` values setting at each training epoch. The training includes `70` learning epochs, each corresponding to `numIter=20` simulation steps. Then, the total number of simulation steps is `3400`. Performance evaluation occurs every `5` epochs.

3.5. Experimental results for Target 1

Each experiment runs an independent training process based on a different combination of two learning hyperparameters: the *learning rate* (`lr`) and the *discount rate* (`gamma`). Tab 1) illustrates the six combinations explored, summarizing the highest absolute final number of cells obtained as a performance measure for every setting.

Fig 5 visualizes learning performance as the distribution of the final number of cells obtained within sliding windows of epochs, considering six windows of 20 learning epochs (0-20, 10-30, 20-40, 30-50, 40-60, and 50-70, respectively). The Figure depicts the distributions' mean, median, and variance at each window with a dashed green line, solid green line, and error bars, respectively.

Exp3 (`lr=0.0001`, `gamma=0.99`) exhibits a steady increase in the mean

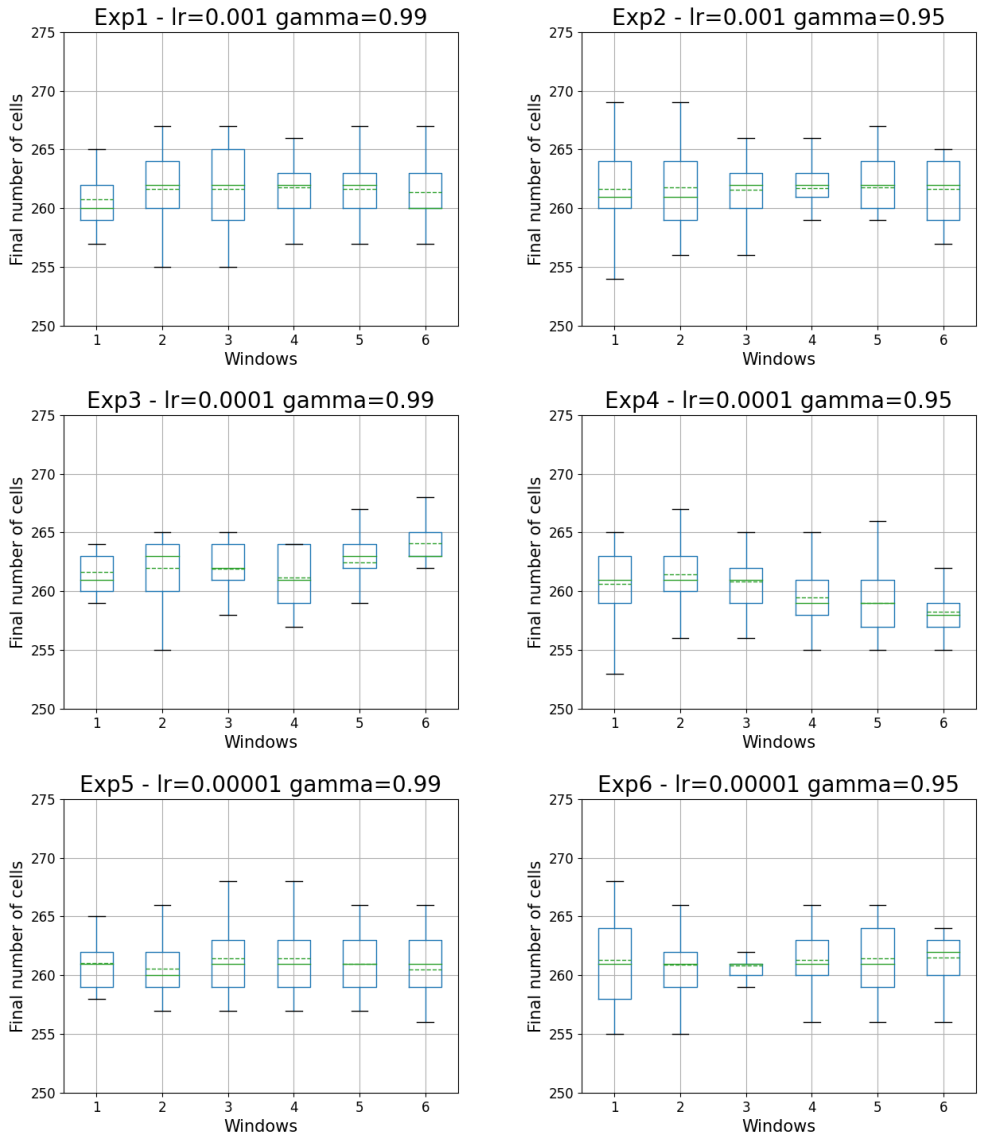


Figure 5: Target 1 - learning performances. Boxplots report distributions of the final number of cells obtained within each epoch during windows of 20 learning epochs. Each panel shows the performance of one training process over six windows, based on the following combinations of hyperparameters values (from the top left to the bottom right): `lr=0.001` and `gamma=0.99` (*Exp1*), `lr=0.001` and `gamma=0.95` (*Exp2*), `lr=0.0001` and `gamma=0.99` (*Exp3*), `lr=0.0001` and `gamma=0.95` (*Exp4*), `lr=0.00001` and `gamma=0.99` (*Exp5*), `lr=0.00001` and `gamma=0.95` (*Exp6*). Each boxplot reports the absolute value's median (solid green line) and the mean (dashed green line).

Experiments	Hyperparameters		Highest final number of cells
	`lr`	`gamma`	
Exp1	`0.001`	`0.99`	268
Exp2	`0.001`	`0.95`	269
Exp3	`0.0001`	`0.99`	268
Exp4	`0.0001`	`0.95`	267
Exp5	`0.00001`	`0.99`	268
Exp6	`0.00001`	`0.95`	267

Table 1: Experiments for Target 1 - maximal number of cells. Experiments correspond to a combination of the `lr` and `gamma` parameters each. For each experiment, the last column reports the highest final number of cells obtained across training.

value starting at window four and a low starting variance: this training process begins in a stable situation that allows exploiting knowledge and then further improves its performance during the last epochs, until reaching, at window six, the highest mean among the six training processes, and very low variance. On the contrary, *Exp2* (`lr=0.001`, `gamma=0.95`) and *Exp6* (`lr=0.00001`, `gamma=0.95`) present lower mean and higher variance during the first windows, indicating they indulge in exploration, and only later switch to exploitation. Moreover, *Exp2* has a decreasing variance, meaning it reached a form of optimality.*Exp4* (`lr=0.0001`, `gamma=0.95`) has decreasing variance and decreasing mean, reaching the lowest mean value across processes at window six. *Exp5* (`lr=0.00001`, `gamma=0.99`) shows a decreasing mean and an increasing variance: it is not exploring nor exploiting, corresponding to low learning performance. Similarly, *Exp1* (`lr=0.001`, `gamma=0.99`) shows a stable mean value and an increased variance at the last window compared to the first, indicating low learning performance.

Fig 6 shows the impact of the optimization process, visualizing the structure of generated protocols at the beginning (epoch `0`, upper panel) and at the end (epoch `16`, lower panel) of *Exp2*. Protocols consist of stimulation parameters `comprForce` and `compressionAxis` values at each simulation step. The x-axis represents the sequence of simulation steps along a simulation. The y-axis illustrates the intensity of the compression force (`comprForce`) exerted over the culture system. In contrast, the color represents the compression axis (`compressionAxis`, either `'X'`, orange dots, or `'Y'`, blue dots). The protocol generated at epoch `0`, when no optimization has occurred yet, exhibits a very disorganized structure (Fig 6, upper panel), where compression stimuli lie on a wide values range. Instead, the protocol from epoch `16` (Fig 6, lower panel) keeps a constant stimulation of `comprForce=0.005` over a single compression axis (`compressionAxis='X'`). In the model, compression inhibits proliferation. A low-compression protocol is consistent with the target of maximizing the final cell number, according to the knowledge of the process and `PalaCell2D` modeling, that assumes the inverse relationship between external force at a cell and the probability the cell proliferates [24, 25].

3.6. Target 2: a circular patch of cells

In the second experiment setup, the DSE aims at fulfilling a circular cell patch with a defined position and dimension at the end of a simulation. This requires maximizing the number of cells within the designated circular target space and minimizing the number outside at the end of a simulation. The

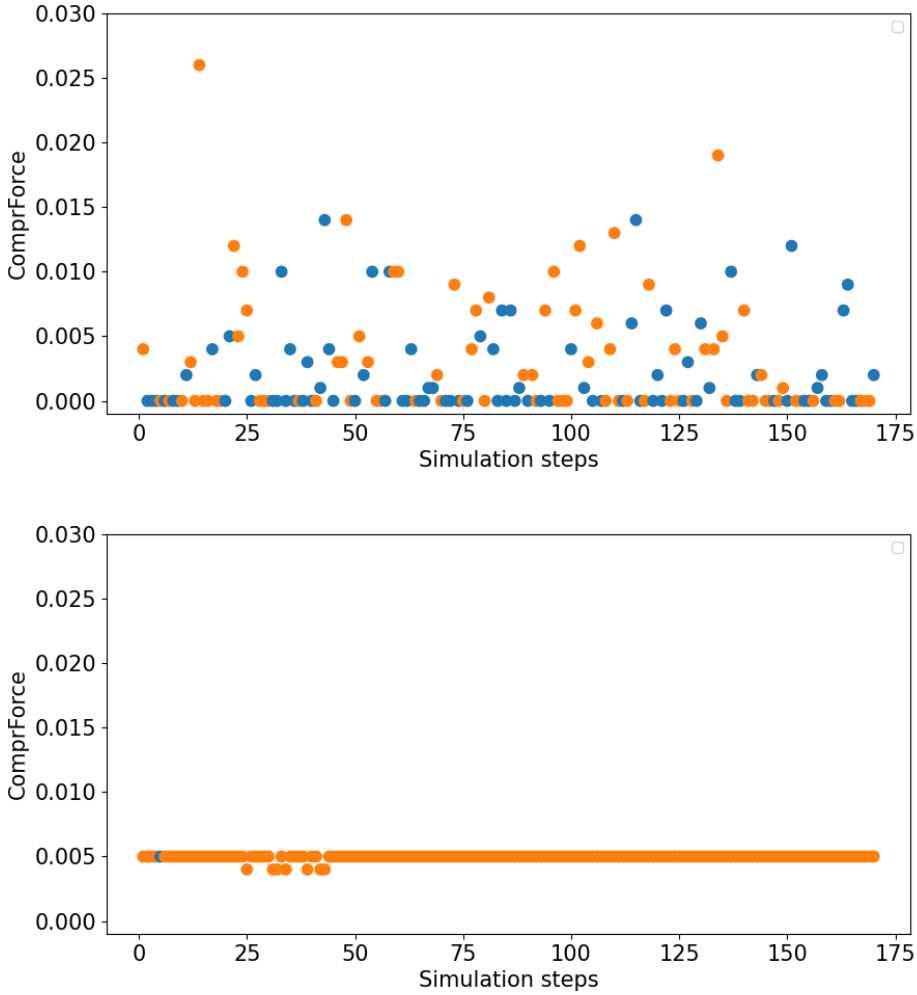


Figure 6: Protocols generated for Target 1 - maximal number of cells, based on *Exp2* (`lr=0.001`, `gamma=0.95`). The protocol obtained at epoch `0` (upper panel), when no optimization has occurred yet, exhibits a very disorganized structure; the protocol obtained at epoch `16` (lower panel), after training advanced, keeps a constant stimulation of `comprForce=0.005` over a single compression axis (`compressionAxis='X'`). Stimulation parameters `comprForce` and `compressionAxis` mark each simulation step in the protocol. The x-axis represents the sequence of simulation steps along a simulation. Dots represent the stimulation at each simulation step: their position over the y-axis illustrates `comprForce` values, and their color represents `compressionAxis` values, either `'X'` (orange dots) or `'Y'` (blue dots).

reward is the fraction of cells within the target ($fraction_{inside}$), as in Eq (9) where $cells_{inside}$ and $cells_{outside}$ are the number of cells inside and outside the target space, and $cells_{final}$ is the total number of obtained cells.

$$fraction_{inside} = \frac{cells_{inside} - cells_{outside}}{cells_{final}} \tag{9}$$

Increasing the mean $fraction_{inside}$ along training indicates improving performance. In contrast, the variance of this value indicates whether learning is in exploration (low variance) versus exploitation (high variance) mode. Protocols for this target control both the initial position for seeding the first before the simulation starts and the compression stimuli along the simulation.

3.7. Framework setup for Target 2

The framework setup (see Appendix A. Framework setup parameters) still includes a simulated space in the environment that is a `400x400x1` grid, as in Framework setup for Target 1, with the addition of a circular target centered in `x=200` and `y=250`, with a radius length of `80`. The learning process targets compression stimuli and the initial position of the first cell in the simulation. For all experiments, the exploration range for `initialPos` lies between `90` and `310` for both starting position coordinates. This requires two learning models to work together during the training process. The primary training process learns compression stimuli; a secondary, inner training process learns initial positions. This requires nested learning environments leveraging a hierarchy of `subprocesses` (see Appendix C. Nested training process for

technical details).

The inner environment recapitulates the setup from Target 1: maximal final number of cells, except for a longer total simulation duration, designed to allow cells to fill the target if the initial position chosen allows it. Thus, the aimed protocols cover the initial positioning of cells and the compression stimuli along the simulation, following two steps:

1. `initialPos` parameter value setting (once, before the simulation starts);
2. `comprForce` and `compressionAxis` parameters values setting (at each simulation step).

Consequently, each epoch sees the generation of a single action that sets the initial position of the cell (`initialPos`). Then, the main training process launches the inner training and waits for it to complete. During the main process' epoch, the inner process acts on the cell at the initial position set by the primary process, learning over textttcompressionAxis and `comprForce` and performing `numIter=20` iterations per epoch for a total of `2500` simulation steps. Each training process runs for `140` epochs, and performance evaluation occurs every `5` epochs.

3.8. Experimental results for Target 2

Each experiment runs an independent training process based on a different combination of two learning hyperparameters: the *learning rate* (`lr`) and the *discount rate* (`gamma`). Tab 2 illustrates the four combinations explored,

summarizing the highest absolute final fraction of cells obtained during training.

Experiments	Hyperparameters		Highest final fraction of cells in target
	`lr`	`gamma`	
Exp7	`0.001`	`0.99`	49.96
Exp8	`0.001`	`0.95`	20.68
Exp9	`0.0001`	`0.99`	54.72
Exp10	`0.0001`	`0.95`	52.78

Table 2: Experimental sets for Target 2 - a circular patch of cells. Experiments correspond to a combination of the `lr` and `gamma` parameters each. The last column reports the highest absolute fraction of cells inside the target across training for each experiment.

Fig 7 visualizes the learning performance as the distribution of the final number of cells obtained within sliding windows of epochs, considering twelve windows of 20 learning epochs (0-20, 10-30, 20-40, 30-50, 40-60, 50-70, 60-80, 70-90, 80-100, 90-110, 100-120, 110-130 and 120-140, respectively). The Figure depicts the distributions' mean, median, and variance at each window with a dashed green line, solid green line, and error bars, respectively.

Until epoch `20`, the cells never overlap the target area in any training processes. Afterward, all four training processes exhibit exploration and exploitation phases, indicating that the main training process generated a `initialPos` value that allows the epithelial patch to overlap the target.

Across all experiments, in the first half of the training (windows 1-6), all processes show high mean and decreasing variance, indicating a dominant exploitation mode. However, around window 9, mean values reach a low and grow for the second time. In contrast, variance increases, indicating

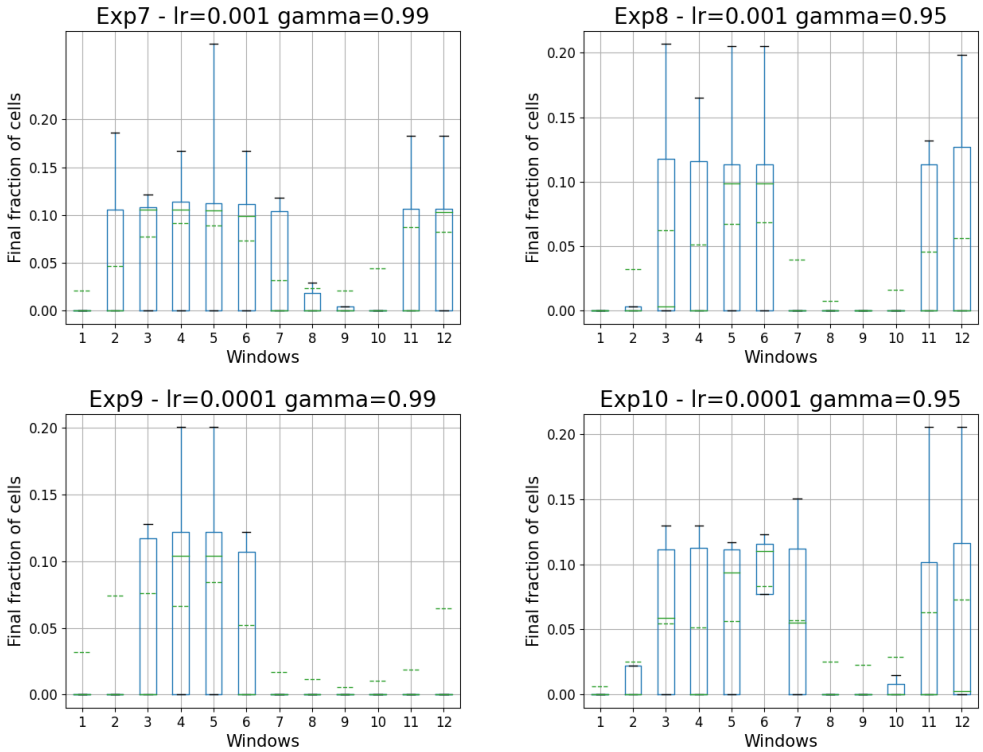


Figure 7: Target 2 - learning performances. Boxplots reporting distributions of the final fraction of cells obtained at each epoch during windows of 20 learning epochs. Each panel shows the performance of one training process over twelve windows, based on the following combinations of hyperparameters values (from the top left to the bottom right): `lr=0.001` and `gamma=0.99` (*Exp7*), `lr=0.001` and `gamma=0.95` (*Exp8*), `lr=0.0001` and `gamma=0.99` (*Exp9*), `lr=0.0001` and `gamma=0.95` (*Exp10*). Each boxplot reports the absolute value's median (solid green line) and the mean (dashed green line).

that the processes enter exploration mode. In particular, *Exp7* (`lr=0.001`, `gamma=0.99`), and *Exp10* (`lr=0.0001`, `gamma=0.95` explore between windows 6 and 9, only to get back to exploitation during the last part of the training. *Exp7* (`lr=0.001`, `gamma=0.99`) has the higher mean values among all training processes for this target, while its variance values are similar to the others.

Fig 8 illustrates the simulated space with a bi-dimensional histogram of the `X` and `Y` values of the starting cell position generated by the training process based on the `lr=0.001` and `gamma=0.99` setting (*Exp7*). It shows a sequence of panels corresponding to the same learning windows as in Fig 7. The action space subspaces color intensity is directly proportional to the total number of times the training chose a position within it during the window. The red circle indicates the target area, and its color intensity is directly proportional to the mean $fraction_{inside}$ across the epochs belonging to the window.

On a closed loop using Fig 8, it can be noticed that in the first window, the models give values outside the action space, and the actual action provided to the environment relocates to its closest boundaries. After epoch `20`, the exploration considers different values inside the action space.

To appreciate the difference in the generated actions due to the learning process, Fig 9 shows the values of the generated compression stimuli at epoch `0` (upper panel) and `135` (lower panel), respectively, for the training process based on the *Exp7* settings. As in the experiments of Experimental results for Target 1, the training process generates more organized and low-compression

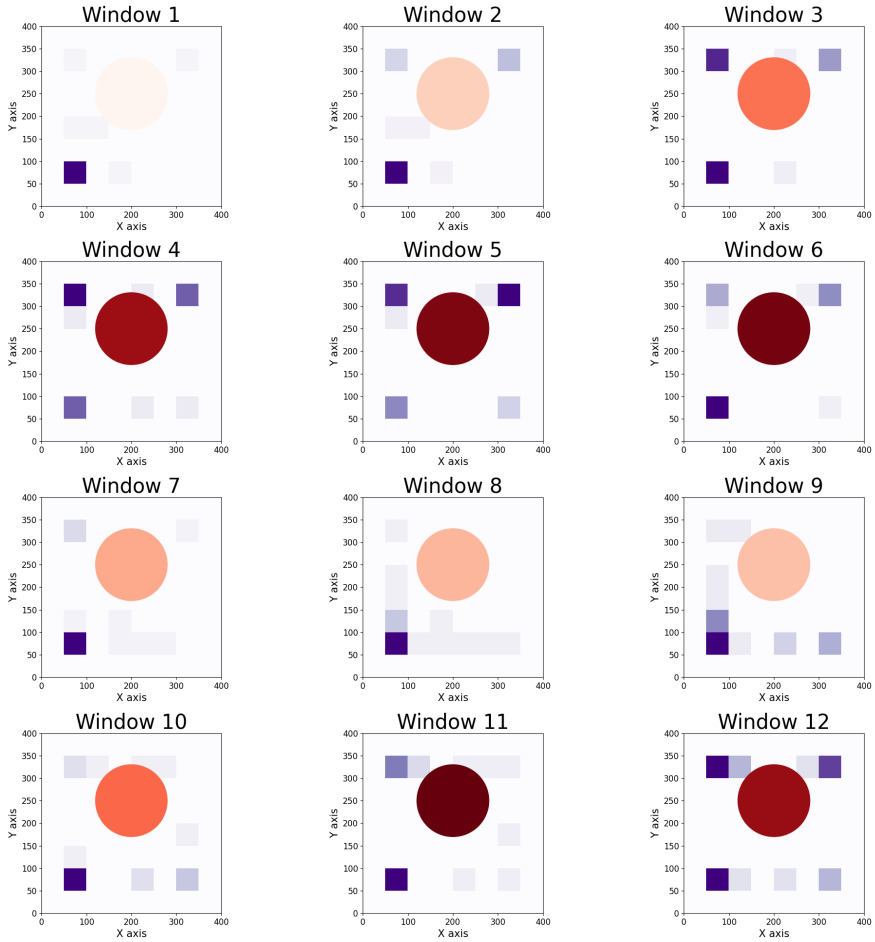


Figure 8: A synthetic representation of the starting cell positions generated based on the `lr=0.001` and `gamma=0.99` (*Exp7*) setting. Each panel is a bi-dimensional histogram of the `X` and `Y` values of the starting cell position generated by the training process during a window of 20 learning epochs. The twelve windows cover epochs 10-30, 20-40, 30-50, 40-60, 50-70, 60-80, 70-90, 80-100, 90-110, 100-120, 110-130, and 120-140. The action space subspaces color intensity is directly proportional to the total number of times the training chose a position within it during the window. The red circle indicates the target area, and its color intensity is directly proportional to the mean $fraction_{inside}$ across the epochs belonging to the window.

stimuli throughout the simulation.

3.9. Hyperparameters exploration: `numIter`

While experiments presented in Target 1: maximal final number of cells and Target 2: a circular patch of cells explore different combinations of learning hyperparameters, this experimental set explores the impact of the `numIter` hyperparameter (see Appendix A. Framework setup parameters), a training hyperparameter. The exploration is the same as Target 1: maximal final number of cells, making the RL target maximize the number of cells obtained at the end of each epoch. Since the goal is to explore different `numIter` values to evaluate the impact of the number of simulation steps performed during each learning epoch, all other hyperparameters hold constant values for `lr` (0.0001) and `gamma` (0.99), as they were a high-performance combination from Target 1: maximal final number of cells.

3.9.1. Experimental results for `numIter` exploration

Table 3 illustrates experimental results exploring five values of `numIter`: `20` (the value used for Target 1: maximal final number of cells and Target 2: a circular patch of cells), `40`, `50`, `100` and `200`, together with the highest final number of cells obtained for every setting.

Figure 10 visualizes learning performance as the evolving distribution of the values of the final number of cells obtained along training processes. As in Experimental results for Target 1, performance is evaluated within six windows of 20 learning epochs (0-20, 10-30, 20-40, 30-50, 40-60, and 50-70,

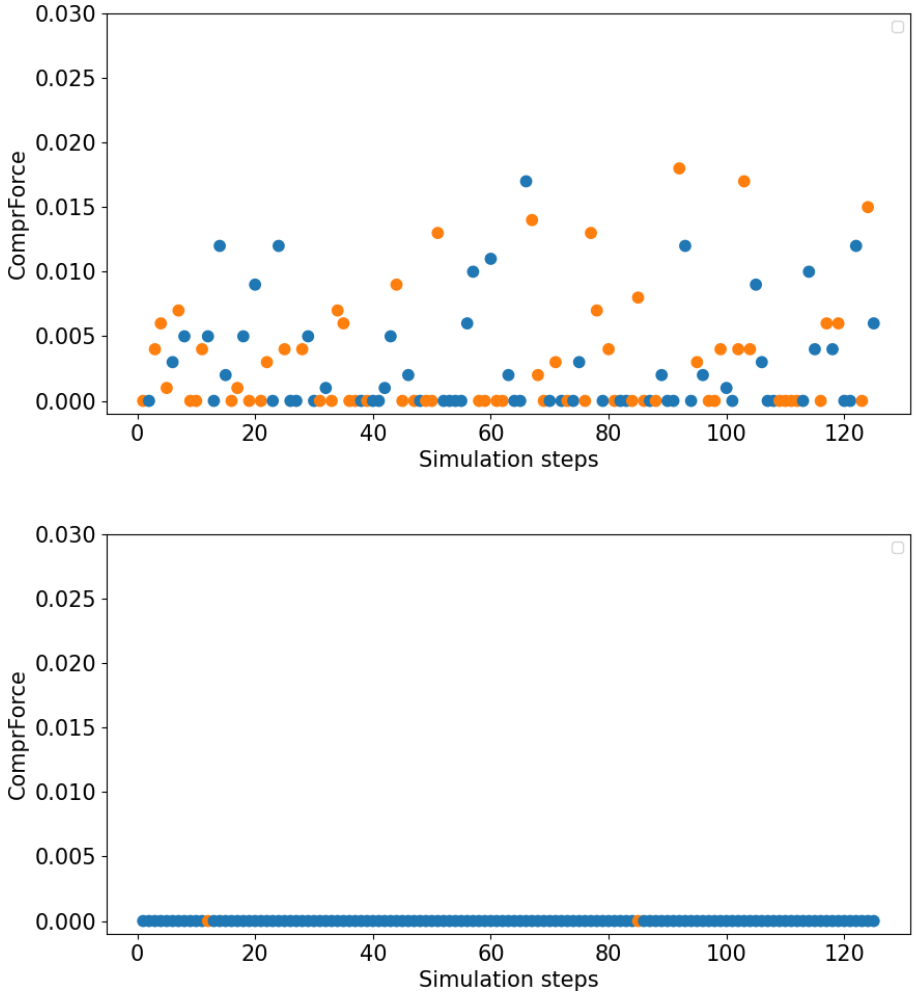


Figure 9: Protocols generated for Target 2 - a circular patch of cells, based on *Exp7* (`lr=0.001`, `gamma=0.99`). The protocol obtained by the training process at epoch `0` (upper panel), when no optimization has occurred yet, exhibits a very disorganized structure; the protocol obtained at epoch `135` (lower panel), after training advanced, keeps a constant stimulation of `comprForce=0.005` over a single compression axis (`compressionAxis='X'`). Each generated protocol contains stimulation parameters `comprForce` and `compressionAxis` values at each simulation step. The x-axis represents the sequence of simulation steps along a simulation. Dots represent the stimulation at each simulation step: their position over the y-axis illustrates `comprForce` values, and their color represents `compressionAxis` values, either `'X'` (orange dots) or `'Y'` (blue dots).

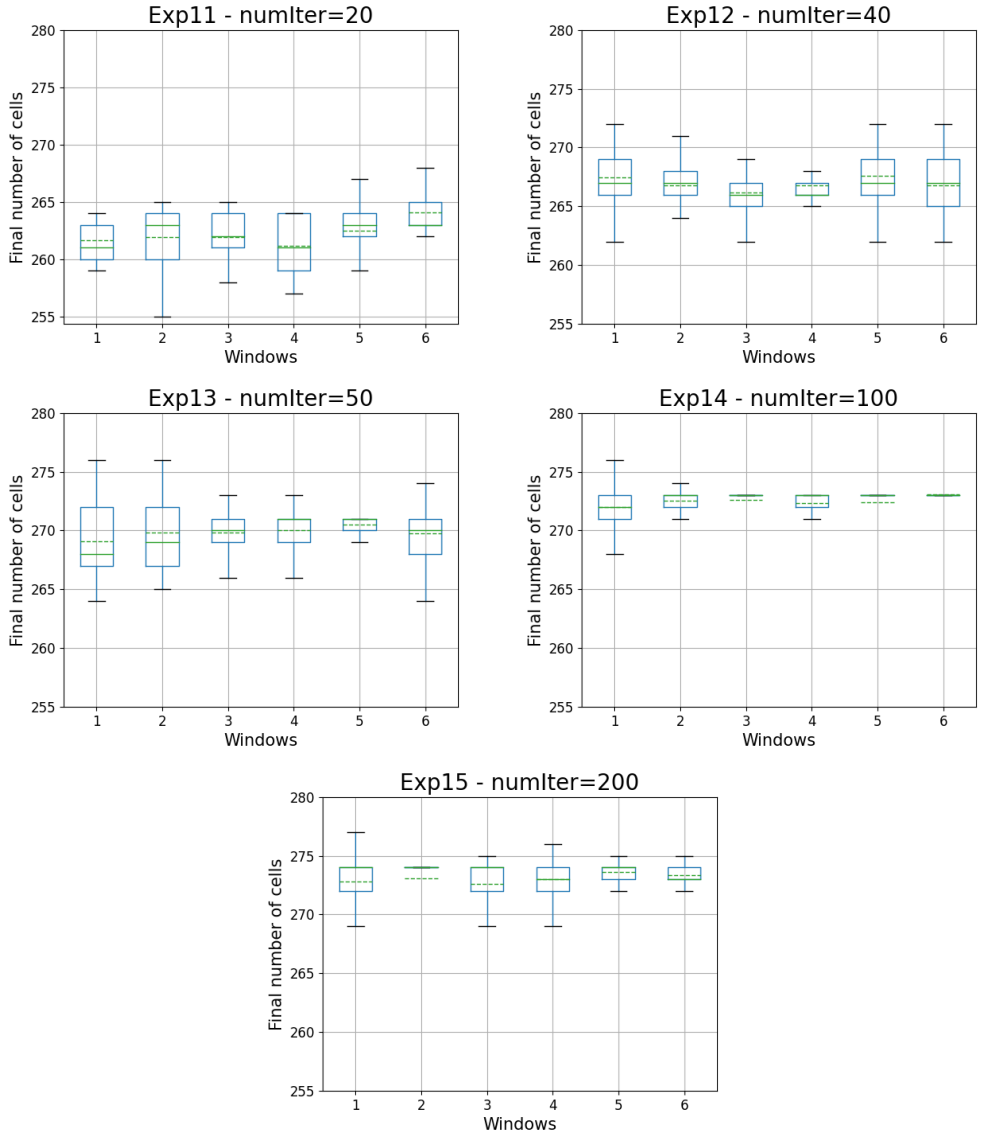


Figure 10: Target 1 - `numIter` hyperparameter exploration. Boxplots reporting distributions of the final fraction of cells obtained at each epoch during windows of 20 learning epochs. Each panel shows the performance of one training process over six windows, based on the `lr=0.0001` and `gamma=0.99` combination of learning hyperparameters values and different `numIter` values (from the top left to the bottom right): `numIter=20` (*Exp11*), `numIter=40` (*Exp12*), `numIter=50` (*Exp13*), `numIter=100` (*Exp14*) and `numIter=200` (*Exp15*). Each boxplot reports the absolute value's median (solid green line) and the mean (dashed green line).

Experiments	Hyperparameters			Highest final number of cells
	`lr`	`gamma`	`numIter`	
Exp11	`0.0001`	`0.99`	`20`	268
Exp12	`0.0001`	`0.99`	`40`	272
Exp13	`0.0001`	`0.99`	`50`	274
Exp14	`0.0001`	`0.99`	`100`	277
Exp15	`0.0001`	`0.99`	`200`	276

Table 3: `numIter` hyperparameter exploration over Target 1 - maximal number of cells. Each experiment corresponds to a fixed combination of the `lr` and `gamma` parameters and a different `numIter` value. The last column reports the highest absolute number of cells obtained during each experiment.

respectively).

Increasing values for `numIter` improves the overall learning performance across experiments. Indeed, *Exp12* (`numIter=40`) exhibits consistently higher mean values than *Exp11* along training. Similarly, *Exp13* (`numIter=50`), *Exp14* (`numIter=100`), and *Exp15* (`numIter=200`) exhibit different patterns of exploration and exploitation, but a uniform increase in the mean values along the training. In particular, *Exp14* and *Exp15* mark a boundary in enlarging the number of simulation steps per epoch. In fact, *Exp14* shows very steady increase in the mean and dramatic decrease in the variance, until reaching full exploitation mode at window six. In contrast, *Exp15* starts losing control of the generated values, reaching high mean values, but way higher variance. A more in-depth analysis can determine if such behavior is due to common overfitting issues.

4. Conclusions

This work presents a computational DSE approach to optimize the simulated biofabrication of epithelial sheets and to generate optimal biofabrication protocols based on a variant of the A2C algorithm. Starting from the idea proposed in [22], it aims to generate biofabrication protocols for the simulated growth of epithelial cells under compression stimuli to validate its capabilities.

Results show that the framework can explore the solution space defined by the specific application, providing a comparative performance analysis of different hyperparameters settings and showing that the framework can impact the structure of generated protocols through the training process.

Future works will target analyzing and testing other synchronous A2C variants that involve multiple agents acting simultaneously and joining their independent results, resolving their inconsistencies. Moreover, we plan to systematically explore hyperparameter combinations and tackle more complex optimization targets, such as different geometries for target areas to fill and the coexistence of several cell types within a construct.

5. Data Availability Statement

The source code and data used to produce the results and analyses presented in this manuscript are available from the public GitHub repository `https://github.com/smilies-polito/rLotos`.

6. Acknowledgements

We acknowledge the precious collaboration and support provided by `PalaCell2D` team, Dr. Raphaël Conradin, and Prof. Bastien Chopard from the Scientific and Parallel Computing (SPC) Group at the University of Geneva, Geneva, Switzerland.

7. Appendices

Appendix A. Framework setup parameters. This section overviews relevant parameters to the framework setup, including simulation and learning environment configuration and the management of output files for visualization and performance metrics calculations.

Simulation setup The `PalaCell2D` simulation configuration file holds different types of simulation parameters. Some parameters control the simulation duration:

- `numIter` controls the number of simulation steps (or iterations) to perform.
- `stopAt` indicates the simulation step (or iteration) when to stop the simulation.

Another set of parameters is the ones controlling the simulation evolution. The biofabrication protocol sets their values according to the actions generated by the RL algorithm along the training. This work includes a new parameter

to the ones provided in `PalaCell2D`: `initialPos`, the position of the first cell generated in the simulation space.

- `initialPos` sets the position of the initial cell, indicating its `x` and `y` coordinate values.
- `compressionAxis` sets the axis for external force application, with either `'X'` or `'Y'` as values.
- `comprForce` indicates the intensity of the external force applied.

Another set of parameters in the configuration supports the management of input and output files. Both types are `.vtp` files from the Visualization Toolkit `VTK` [79].

- `initialVTK` indicates the name of the input `.vtp` file from which to restore the simulation (if this value is missing, the simulation starts from the beginning).
- `finalVTK` indicates the name of the generated output `.vtp` file.

Learning environment setup This work implements the simulator interface (see Materials and Methods) to use `PalaCell2D` simulations as a learning environment. Moreover, it relies on the `configure` method, called by the implemented `environment`, to create the configuration file needed by the `PalaCell2D` simulator. The implemented `PalaCell2D environment` has the following parameters to interface the learning process.

- `width` and `height`: the width and height in pixels of the observations provided to the learning model, with a default value of `300`.
- `numIter`: the number of simulation steps per learning epoch, with a default value of `20`.
- `max_iterations`: the maximum number of total simulation steps, with a default value of `4200`.

Output management Performance evaluation relies on the computation of defined metrics over the generated output. Data saving occurs every five training epochs. The helper file `vtkInterface.py` contains the following methods to read `PalaCell2D` outputs.

- `read_cell_num`: reads the number of cells in the simulation space from the simulation output file.
- `create_png_from_vtk`: reads the simulation output file and produces a visualization of the simulation space. This visualization centers on the cellular aggregate and not on the simulation space.
- `create_decentered_pil_image`: reads the simulation output file and produces a visualization of the simulation space. This visualization centers on the simulation space.
- `add_target`: adds a circular target with given center coordinates and radius to a chosen image.

- `count_target_points`: reads the simulation output file and returns the number of cellular vertices inside and outside a specified target.

Appendix B. Simulator interface. The simulator interface contains the set of required functions and a format for data exchange to fulfill the specific objectives and requirements of different applications. Each specific application requires an implementation of the simulator interface, where the necessary functions are implemented to interact with the chosen simulator. Each implementation of the simulator interface creates an environment based on the *environment* file, whose structure follows the one proposed in [80]. The *environment* file sets the ANN model structure (see Learning model) and the training process parameters for the target application.

The simulator interface includes the following set of required functions to advance the training process.

- `reset`: prepares the environment for the next episode starting from an initial state;
- `render`: returns an image showing the environment state;
- `adapt_action`: transforms the actions provided by the ANN in a format that can be used by the `step` method;
- `step`: acts on the environment with the generated actions and provides an observation of the environment, the reward value, and a boolean

value that tells whether the environment has reached a terminal state or not to the training process.

The simulator interface also includes the following set of required functions to manage data produced by the training process and keep track of performance.

- `save_performance`: indicates whether or not to collect performance indexes related to the environment into environment variables.
- `get_performance`: saves the performance indexes in a checkpoint file.
- `load_performance`: restores the performance indexes in the respective environment variables when restoring the training from a previous state through checkpoint files.
- `check_performance`: indicates whether or not to save the collected performance indexes in a checkpoint file.
- `data_to_save`: indicates which data to save in a checkpoint file that is accessible after the training.
- `load_data_to_save`: restores the collected data in the respective variables inside the environment when restoring the training from a previous state through checkpoint files.

Appendix C. Nested training process. To tackle Target 2 (see Target 2: a circular patch of cells), a helper training class allows synchronizing the two nested `subprocesses`. This class has the same structure as the one already presented in Training process, with the addition of a `pipe` allowing communication between the two `subprocesses`. The two environments perform the steps described in Training process during the training process.

Algorithm 2 Nested training loop

```
1: e_main = PalacellEnv()
2: e_inner = PalacellEnv()
3: while current_epoch_main <= epochs_main do
4:    e_main.reset()
5:    initialObservation_main = e_main.getObservation()
6:    actions_main = learningModel(initialObservation_main)
7:    e_main.step(actions_main)
8:    while current_epoch_inner <= epochs_inner do
9:       initialObservation_inner = e_inner.getObservation()
10:      actions_inner = learningModel(initialObservation_inner)
11:      e_inner.step(actions_inner)
12:      finalObservation_inner = e_inner.getObservation()
13:      reward_inner = e_inner.getReward(finalObservation_inner)
14:      learningModel.applyReward(reward_inner)
15:   finalObservation_main = e_main.getObservation()
16:   reward_main = e_main.getReward(finalObservation_main)
17:   learningModel.applyReward(reward_main)
```

This flow leverages communication mechanisms between `subprocesses`, which supports the steps illustrated in Alg. 2. After instantiation of the main and inner learning environments (Alg. 2, lines 1-2), the training creates a simulation configuration file for the inner training, with the same values as in the previous experiment (see Target 1: maximal final number of cells).

But, the `initialPos` parameter it is not constant, but rather set by actions generated by the learning model. The main `subprocess` runs its training process (Alg. 2, lines 3-7), and sends the simulation configuration to the second training process through the `pipe`. The second `subprocess` uses it as the starting configuration file, then runs its training process (Alg. 2, lines 8-14). This training process runs the same number of simulation iterations per epoch as Target 1: maximal final number of cells (`numIter=20`) for a total of `2500` simulation iterations. When the second training process terminates, its `subprocess` sends the generated output to the main `subprocess` through the `pipe`. The main `subprocess` reads the output, computes performance metrics and produces an observation for the main training process (Alg. 2, line 15). The main training process learns using such observation and the corrresponding reward (Alg. 2, lines 15-17). Then, it proceeds to the next epoch.

References

[1] J. Groll, T. Boland, T. Blunk, J. A. Burdick, D. W. Cho, et al., Biofabrication: reappraising the definition of an evolving field, Biofabrication 8 (1) (2016) 013001. `doi:10.1088/1758-5090/8/1/013001`. URL `https://dx.doi.org/10.1088/1758-5090/8/1/013001`

[2] L. Moroni, T. Boland, J. A. Burdick, et al., Biofabrication: A guide to technology and terminology, Trends in Biotechnology 36 (4) (2017)

384–402. doi:10.1016/j.tibtech.2017.10.015.
URL https://doi.org/10.1016/j.tibtech.2017.10.015

[3] L. Moroni, J. A. Burdick, C. Highley, S. J. Lee, Y. Morimoto, S. Takeuchi, J. J. Yoo, Biofabrication strategies for 3d in vitro models and regenerative medicine, Nature Reviews Materials 3 (5) (2018) 21–37.

[4] S. B. Mackay, K. Marshall, J. A Grant-Jacob, et al., The future of bone regeneration: integrating ai into tissue engineering, Biomedical Physics & Engineering Express 7 (5) (2021) 052002. doi:10.1088/2057-1976/ac154f.
URL https://dx.doi.org/10.1088/2057-1976/ac154f

[5] M. I. Sadowski, C. Grant, T. S. Fell, Harnessing qbd, programming languages, and automation for reproducible biology, Trends in biotechnology 34 (3) (2016) 214–227.

[6] S. A. Gbadegeshin, The effect of digitalization on the commercialization process of high-technology companies in the life sciences industry, Technology Innovation Management Review 9 (1) (2019).

[7] R. Yasui, K. Sekine, H. Taniguchi, Clever experimental designs: Shortcuts for better ipsc differentiation, Cells 10 (12) (2021) 3540.

[8] J. Gilman, L. Walls, L. Bandiera, F. Menolascina, Statistical design of experiments for synthetic biology, ACS Synthetic Biology 10 (1) (2021) 1–18.

[9] S. Zobel-Roos, A. Schmidt, L. Uhlenbrock, R. Ditz, D. Köster, J. Strube, Digital twins in biomanufacturing, in: Digital Twins, Springer, 2020, pp. 181–262.

[10] K. H. Chang, P. W. Zandstra, Quantitative screening of embryonic stem cell differentiation: endoderm formation as a model, Biotechnology and bioengineering 88 (3) (2004) 287–298.

[11] I. Walsh, M. Myint, T. Nguyen-Khuong, Y. S. Ho, S. K. Ng, M. Lakshmanan, Harnessing the potential of machine learning for advancing "quality by design" in biomanufacturing, in: Mabs, Vol. 14 (1), Taylor & Francis, 2022, p. 2013593.

[12] M. A. Bukys, A. Mihas, K. Finney, K. Sears, D. Trivedi, Y. Wang, J. Oberholzer, J. Jensen, High-dimensional design-of-experiments extracts small-molecule-only induction conditions for dorsal pancreatic endoderm from pluripotency, Iscience 23 (8) (2020) 101346.

[13] M. Kuterbekov, P. Machillot, F. Baillet, A. M. Jonas, K. Glinel, C. Picart, Design of experiments to assess the effect of culture parameters on the osteogenic differentiation of human adipose stromal cells, Stem cell research & therapy 10 (1) (2019) 1–9.

[14] S. Zhang, S. Vijayavenkataraman, W. F. Lu, J. Y. Fuh, A review on the use of computational methods to characterize, design, and optimize tissue engineering scaffolds, with a potential in 3d printing fabrication,

Journal of Biomedical Materials Research Part B: Applied Biomaterials 107 (5) (2019) 1329–1351.

[15] G. Lyu, R. W. Brennan, Towards iec 61499 based distributed intelligent automation: Design and computing perspectives, in: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), Vol. 1, IEEE, 2019, pp. 160–163.

[16] K. K. Ng, C.-H. Chen, C. K. Lee, J. R. Jiao, Z.-X. Yang, A systematic literature review on intelligent automation: Aligning concepts from theory, practice, and future perspectives, Advanced Engineering Informatics 47 (2021) 101246.

[17] R. Bardini, S. Di Carlo, Computational modeling and optimization of biofabrication in tissue engineering and regenerative medicine—a literature review, bioRxiv (2023) 2023–03.

[18] J. F. Groote, T. W. Kouters, A. Osaiweran, Specification guidelines to avoid the state space explosion problem, Software Testing, Verification and Reliability 25 (1) (2015) 4–33.

[19] M. Abdel-Basset, L. Abdel-Fatah, A. K. Sangaiah, Metaheuristic algorithms: A comprehensive review, Computational intelligence for multimedia big data on the cloud with engineering applications (2018) 185–231.

[20] J. V. S. Do Amaral, J. A. B. Montevechi, R. de Carvalho Miranda, W. T. de Sousa Junior, Metamodel-based simulation optimization: A

systematic literature review, Simulation Modelling Practice and Theory 114 (2022) 102403.

[21] G. Squillero, Microgp—an evolutionary assembly program generator, Genetic programming and evolvable machines 6 (2005) 247–263.

[22] L. Giannantoni, R. Bardini, S. Di Carlo, A methodology for co-simulation-based optimization of biofabrication protocols, in: Bioinformatics and Biomedical Engineering: 9th International Work-Conference, IWBBIO 2022, Maspalomas, Gran Canaria, Spain, June 27–30, 2022, Proceedings, Part II, Springer Nature, 2022, pp. 179–192.

[23] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, et al., Asynchronous methods for deep reinforcement learning, ICML 2016 (2016). doi:10.48550/ARXIV.1602.01783.
URL https://arxiv.org/abs/1602.01783

[24] R. Conradin, C. Coreixas, J. Latt, B. Chopard, Palacell2d: A framework for detailed tissue morphogenesis, Journal of Computational Science 53 (2021) 101353.

[25] R. E. Y. Conradin, Cell-based simulation of tissue morphogenesis, Ph.d. thesis, University of Geneva (2022).

[26] E. Bartocci, P. Lió, Computational modeling, formal analysis, and tools for systems biology, PLoS computational biology 12 (1) (2016) e1004591.

[27] R. Bardini, G. Politano, A. Benso, S. Di Carlo, Multi-level and hybrid modelling approaches for systems biology, Computational and structural biotechnology journal 15 (2017) 396–402.

[28] R. Bardini, G. Politano, A. Benso, S. Di Carlo, Computational tools for applying multi-level models to synthetic biology, in: Synthetic Biology, Springer, Singapore, 2018, pp. 95–112.

[29] F. Muggianu, A. Benso, R. Bardini, E. Hu, G. Politano, S. Di Carlo, Modeling biological complexity using biology system description language (bisdl), in: 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), IEEE, 2018, pp. 713–717.

[30] L. S. Vieira, R. C. Laubenbacher, Computational models in systems biology: standards, dissemination, and best practices, Current Opinion in Biotechnology 75 (2022) 102702.

[31] M. Heiner, R. Donaldson, D. Gilbert, Petri nets for systems biology, Symbolic Systems Biology: Theory and Methods. Jones and Bartlett Publishers, Inc., USA (in Press, 2010) (2010).

[32] R. Bardini, A. Benso, G. Politano, S. Di Carlo, Nets-within-nets for modeling emergent patterns in ontogenetic processes, Computational and Structural Biotechnology Journal 19 (2021) 5701–5721.

[33] R. Bardini, A. Benso, S. Di Carlo, G. Politano, A. Savino, Using nets-within-nets for modeling differentiating cells in the epigenetic landscape,

in: International Conference on Bioinformatics and Biomedical Engineering, Springer, Cham, 2016, pp. 315–321.

[34] R. Bardini, S. Di Carlo, G. Politano, A. Benso, Modeling antibiotic resistance in the microbiota using multi-level petri nets, BMC systems biology 12 (6) (2018) 59–79.

[35] R. Bardini, G. Politano, A. Benso, S. Di Carlo, Using multi-level petri nets models to simulate microbiota resistance to antibiotics, in: 2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), IEEE, 2017, pp. 128–133.

[36] A. G. Fletcher, M. Osterfield, R. E. Baker, S. Y. Shvartsman, Vertex models of epithelial morphogenesis, Biophysical journal 106 (11) (2014) 2291–2304.

[37] G. G. Wang, S. Shan, Review of metamodeling techniques in support of engineering design optimization, J. Mech. Des. (2007).

[38] A. Parnianifard, A. Azfanizam, M. K. A. M. Ariffin, M. I. S. Ismail, N. Ale Ebrahim, Recent developments in metamodel based robust black-box simulation optimization: An overview, Recent Developments in Metamodel Based Robust Black-Box Simulation Optimization: An Overview (May 23, 2018). Decision Science Letters 8 (1) (2018) 17–44.

[39] J. P. Kleijnen, Simulation-optimization via kriging and bootstrapping: a survey, Journal of Simulation 8 (2014) 241–250.

[40] G. D. Goh, S. L. Sing, W. Y. Yeong, A review on machine learning in 3d printing: Applications, potential, and challenges, Artificial Intelligence Review 54 (1) (2021) 63–94.

[41] M. H. Ali, S. Batai, D. Sarbassov, 3d printing: A critical review of current development and future prospects, Rapid Prototyping Journal (2019).

[42] A. Menon, B. Póczos, A. W. Feinberg, N. R. Washburn, Optimization of silicone 3d printing with hierarchical machine learning, 3D Printing and Additive Manufacturing 6 (4) (2019) 181–189.

[43] I. Buj-Corral, A. Bagheri, M. Sivatte-Adroer, Effect of printing parameters on dimensional error, surface roughness and porosity of fff printed parts with grid structure, Polymers 13 (8) (2021) 1213.

[44] I. N. Aguilar, L. J. Smith, D. J. Olivos III, T.-M. G. Chu, M. A. Kacena, D. R. Wagner, Scaffold-free bioprinting of mesenchymal stem cells with the regenova printer: Optimization of printing parameters, Bioprinting 15 (2019) e00048.

[45] W. L. Ng, A. Chan, Y. S. Ong, C. K. Chua, Deep learning for fabrication and maturation of 3d bioprinted tissues and organs, Virtual and Physical Prototyping 15 (3) (2020) 340–358.

[46] M. Dattatrey, A. K. Tiwari, B. Ghoshal, J. Singh, Predicting bone

modeling parameters in response to mechanical loading, IEEE Access 7 (2019) 122561–122572.

[47] A. H. Halim, I. Ismail, S. Das, Performance assessment of the metaheuristic optimization algorithms: an exhaustive review, Artificial Intelligence Review 54 (3) (2021) 2323–2409.

[48] P. Mc Donnell, N. Harrison, M. Liebschner, P. Mc Hugh, Simulation of vertebral trabecular bone loss using voxel finite element analysis, Journal of biomechanics 42 (16) (2009) 2789–2796.

[49] G. Iori, L. Peralta, A. Reisinger, F. Heyer, C. Wyers, J. van den Bergh, D. Pahr, K. Raum, Femur strength predictions by nonlinear homogenized voxel finite element models reflect the microarchitecture of the femoral neck, Medical Engineering & Physics 79 (2020) 60–66.

[50] R. N. Mishra, M. K. Singh, V. Kumar, Biomechanical analysis of human femur using finite element method: A review study, Materials Today: Proceedings (2022).

[51] M. Ramu, M. Ananthasubramanian, T. Kumaresan, R. Gandhinathan, S. Jothi, Optimization of the configuration of porous bone scaffolds made of polyamide/hydroxyapatite composites using selective laser sintering for tissue engineering applications, Bio-Medical Materials and Engineering 29 (6) (2018) 739–755.

[52] L. Zheng, J. Yang, X. Hu, J. Luo, Three dimensional finite element analysis of a novel osteointegrated dental implant designed to reduce stress peak of cortical bone, Acta of Bioengineering and Biomechanics 16 (3) (2014).

[53] R. Santoro, A. L. Olivares, G. Brans, D. Wirz, C. Longinotti, D. Lacroix, I. Martin, D. Wendt, Bioreactor based engineering of large-scale human cartilage grafts for joint resurfacing, Biomaterials 31 (34) (2010) 8946–8952.

[54] T. Nguyen, O. Carpentier, F. Monchau, F. Chai, J.-C. Hornez, P. Hivart, Numerical optimization of cell colonization modelling inside scaffold for perfusion bioreactor: A multiscale model, Medical engineering & physics 57 (2018) 40–50.

[55] M. Mehrian, L. Geris, Optimizing neotissue growth inside perfusion bioreactors with respect to culture and labor cost: a multi-objective optimization study using evolutionary algorithms, Computer methods in biomechanics and biomedical engineering 23 (7) (2020) 285–294.

[56] S. Amaran, N. V. Sahinidis, B. Sharda, S. J. Bury, Simulation optimization: a review of algorithms and applications, Annals of Operations Research 240 (2016) 351–380.

[57] P. J. Van Laarhoven, E. H. Aarts, P. J. van Laarhoven, E. H. Aarts, Simulated annealing, Springer, 1987.

[58] F. Glover, Tabu search: A tutorial, Interfaces 20 (4) (1990) 74–94.

[59] S. Katoch, S. S. Chauhan, V. Kumar, A review on genetic algorithm: past, present, and future, Multimedia Tools and Applications 80 (2021) 8091–8126.

[60] R. Poli, J. Kennedy, T. Blackwell, Particle swarm optimization: An overview, Swarm intelligence 1 (2007) 33–57.

[61] M. Dorigo, T. Stützle, Ant colony optimization: overview and recent advances, Springer, 2019.

[62] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[63] K. Aggarwal, M. M. Mijwil, A.-H. Al-Mistarehi, S. Alomari, M. Gök, A. M. Z. Alaabdin, S. H. Abdulrhman, et al., Has the future started? the current growth of artificial intelligence, machine learning, and deep learning, Iraqi Journal for Computer Science and Mathematics 3 (1) (2022) 115–123.

[64] R. d. R. Faria, B. D. O. Capron, A. R. Secchi, M. B. de Souza Jr, Where reinforcement learning meets process control: Review and guidelines, Processes 10 (11) (2022) 2311.

[65] V. Konda, J. Tsitsiklis, Actor-critic algorithms, in: S. Solla, T. Leen, K. Muller (Eds.), Advances in Neural Information Processing Systems,

Vol. 12, MIT Press, 1999, p. .
URL https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[66] M. Van Otterlo, M. Wiering, Reinforcement Learning and Markov Decision Processes, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, Ch. Reinforcement Learning and Markov Decision Processes, pp. 3–42. doi:10.1007/978-3-642-27645-3_1.
URL https://doi.org/10.1007/978-3-642-27645-3_1

[67] R. S. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: S. Solla, T. Leen, K. Müller (Eds.), Advances in Neural Information Processing Systems, Vol. 12, MIT Press, 1999, p. .
URL https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf

[68] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine Learning 8 (1992) 229–256. doi:10.1007/BF00992696.
URL https://doi.org/10.1007/BF00992696

[69] P. A. Gagniuc, Markov chains: from theory to implementation and experimentation, John Wiley & Sons, 2017.

[70] L. Weng, Policy gradient algorithms, lilianweng.github.io (2018).

URL https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

[71] R. S. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, Advances in neural information processing systems 12 (1999).

[72] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, M. Riedmiller, Deterministic policy gradient algorithms, in: International conference on machine learning, Pmlr, 2014, pp. 387–395.

[73] , Openai baselines: Acktr & a2c, https://openai.com/blog/baselines-acktr-a2c/, [Online; accessed 7-February-2023] (February 2023).

[74] , Tensorflow documentation v2.9, https://www.tensorflow.org/versions/r2.9/api_docs/python/t [Online; accessed 7-February-2023] (February 2023).

[75] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778. doi:10.1109/CVPR.2016.90.

[76] , Singularity documentation, https://docs.sylabs.io/guides/3.5/user-guide/index.html, [Online; accessed 7-February-2023] (February 2023).

[77] T. Weng, W. Zhang, Y. Xia, P. Wu, M. Yang, R. Jin, S. Xia, J. Wang, C. You, C. Han, et al., 3d bioprinting for skin tissue engineering: Current status and perspectives, Journal of tissue engineering 12 (2021) 20417314211028574.

[78] G. Bao, Biofabrication in tissue engineering, Racing for the Surface: Antimicrobial and Interface Tissue Engineering (2020) 289–312.

[79] , Vtk documentation, https://vtk.org/documentation/, [Online; accessed 7-February-2023] (February 2023).

[80] , Openai gym documentation, https://www.gymlibrary.dev/, [Online; accessed 7-February-2023] (February 2023).