Politecnico di Torino

Politecnico di Torino

Master's Degree course in Computer Engineering
A.Y. 2021/2022
December 2022 graduation session

Master's Degree Thesis

A reinforcement learning approach to the computational generation of biofabrication protocols

Supervisors

Prof. Stefano Di Carlo

Prof. Alessandro Savino

Dr. Roberta Bardini

Candidate

Alberto Castrignanò

Student ID n° 281689

Summary

Biofabrication is an emerging field that identifies the set of processes required for the generation of biologically functional products with structural organization and subsequent tissue maturation process. Biofabrication technologies have the potential to revolutionize the Regenerative Medicine (RM) domain, which aims to regenerate damaged tissues. Generally, RM techniques are costly and time-consuming, so computational approaches can tap into this potential, facilitating innovation in biofabrication and supporting process and product quality. This Thesis project developed a software framework capable of generating optimal protocols to improve biofabrication processes. The framework combines simulation techniques and Reinforcement Learning (RL) approaches to computationally generate optimal protocols for the simulated fabrication of epithelial sheets, while providing a customizable interface that can be set up with any simulator. The optimization engine uses a Deep Learning (DL)-based RL algorithm, the Advantage Actor Critic (A2C), which relies on a customizable neural network that adapts to the specific environment given to the engine through the interface. The potential of the framework is demonstrated through the optimization of a cell proliferation process in two different experiments: the maximization of the final number of obtained cells, and the obtaining of a defined target shape with cells at the end of the simulation. The experiments demonstrate both the easiness of implementation of the specific process and the potential of the RL approach to improve biofabrication processes in the future.

The Background Section introduces the field of biofabrication in order to provide the reader enough information to understand the importance of the novelty introduced by the proposed framework, together with basic notions about Artificial Neural Networks (ANN) and RL which are needed to understand the technical aspects about the engineered solution. In particular:

- The Convolutional Neural Networks Section shows briefly the evolution of neural networks, going from the basic structure of Multilayer Perceptron (MLP) to Convolutional Neural Network (CNN), in order to understand how the ANN model that is being used by the deep RL algorithm is built. Then,

it explains the ANN training process, in order to make the reader understand how the RL algorithm learns through the ANN model;

- The Reinforcement Learning Section explains the basic concepts of RL, introducing the first invented methods to make the reader understand how they are combined in the new A2C and Asynchronous Advantage Actor Critic (A3C) methods.

The highlighted motivations are related to the need of an automatic process that can run an optimization process on the searching of an optimal biofabrication protocol, without the drawbacks of a purely empirical approach, eventually having to rely on high costs of time and resources. For this reason, this Thesis work proposes a framework based on an Optimization via Simulation approach, in which an optimization engine searches the optimal biofabrication protocol through a specific simulator, which is made so that it can be adapted easily to future implementations of other simulators.

In the Methods Section the framework structure has been illustrated, going into the details of each framework component:

- the ANN model, that can be built with a different output for a different simulator. The proposed ANN is composed by a backbone (which learns the visual components) and two output heads, an Actor head (which outputs the action distributions) and a Critic head (which outputs the value of the state observed from the environment);

- the simulator interface, which contains the functions that are needed by the training process, so that it remains the same for different simulators, requiring only a different implementation of the functions;

- the exploration engine, which implements the A2C process that communicates with both the ANN model and the implemented simulator through its relative interface.

Finally, the Results Section proposes two experiments for a particular use case, describing the simulator used and how the framework interacts with it. This section exposes details of the implemented interface, exemplifying how an interface can be made. The two experiments are:

- Generating a protocol to maximize the total number of cells. This experiment relies on the optimization of epithelial cells proliferation, the target of this experiment is to increase the final number of obtained cells. The learning algorithm is controlled by how much the total number of cells is increasing between two simulation steps;

- Generating a protocol to maximize the total cell number within a defined target space. This experiment relies on the optimization of the position of the starting cell, the target of the experiment is to find the best position of the first cell which is being placed in the simulated space, in order to fill as much as possible a circular target space. The learning algorithm controlled by the fraction of cells inside the target space with respect to the total number of cells in the simulated space. This experiment uses the previous use case as the proliferation component, meaning that two different processes are learning at the same time by combining their operations.

The collected results show that the framework is useful to help the user in following the learning processes in order to obtain an optimal biofabrication protocol, and the experiments, which have been easily implemented, show that the algorithm is learning from the environment. As a final consideration, the Conclusions Section highlights future developments and improvements for the learning process, considering either improving the RL algorithm with a recent implementation of the A3C, or the neural network model, by trying different backbones or even completely different ANN models.

Ringraziamenti

Ringrazio innanzitutto i miei relatori, il Prof. di Carlo, il Prof. Savino, e la Dr. Bardini, per avermi offerto l'opportunità di lavorare ad un progetto che non avrei mai creduto essere in grado di portare avanti, e per avermi seguito fino in fondo credendo nel mio lavoro. Un ringraziamento speciale va a Roberta, che con molta pazienza mi ha guidato e consigliato nonostante la mia testardaggine, senza mai farmi dubitare delle mie capacità, e aiutandomi a perfezionare un progetto di tesi che altrimenti non sarebbe mai venuto così bene. Con le sue indicazioni fondamentali posso sentirmi più che orgoglioso del lavoro che ho svolto, e mi sento onorato e fortunato di aver potuto lavorare accanto a lei.

Ringrazio la mia famiglia, per avermi supportato in tutto e in tanti modi, nonostante le varie difficoltà e senza mai farmi sentire il peso delle mie necessità. In questi due anni la mia famiglia mi è stata molto accanto, con contributi che vanno da piccoli gesti ad appoggi essenziali, economici e non. Senza di loro, non avrei mai potuto farcela in tempo, senza dovermi preoccupare della mia situazione da fuori sede, e delle comuni preoccupazioni della vita. Mamma, papà, Valeria e la sua bellissima famiglia, sono parte integrante del mio percorso in quanto persone che, nonostante vedute diverse, incomprensioni e discussioni, le quali erano comunque per il mio bene e nel mio interesse, non hanno mai smesso di credere in me. Nell'anno di distanza, hanno sempre aspettato una mia chiamata, preoccupandosi del mio benessere e sopportando i miei momenti di intolleranza. Un grazie non basterà mai a ripagarvi, vi voglio bene.

Ringrazio i miei amici, tutti coloro che hanno continuato a gioire insieme a me dei miei successi, e a compatirmi quando mi lamentavo eccessivamente dei miei fallimenti. Grazie a Gabriele, che ha sempre avuto un occhio di riguardo nei miei confronti, trattandomi sempre come un fratello, il primo a vedere in me qualcosa che io non riuscivo a vedere, la prima spalla sulla quale poter contare sempre,

specialmente negli attimi più bui del mio percorso. Grazie agli sciroppi, che dopo anni continuano a volersi ritrovare, nonostante aver preso strade diverse. Siete una garanzia. Un ringraziamento speciale a Stefano, essenziale nel mio percorso, sia nello studio che nell'amicizia, sempre pronto ad aiutarmi e ascoltarmi, una compagnia fondamentale in un anno di solitudine da fuorisede. Stefano mi ha accolto in casa sua, mi ha accompagnato in ogni momento del mio percorso, e si è sempre fidato di me lavorando insieme, tutto questo volentieri e senza mai chiedermi nulla in cambio. Mi sento molto fortunato a poter concludere il mio percorso così come è iniziato: accanto a un vero amico. Grazie anche a Lorenzo, per avermi accolto insieme a Stefano, e loro due con Michele per aver lavorato con me nei progetti più belli che potessi realizzare.

Ringrazio i ragazzi del mu nu chapter di HKN, per avermi regalato un'esperienza indimenticabile, rendendo i miei anni da universitario dopo il covid meno vuoti, tra cene, eventi, feste, attività, risate ed emozioni. Non avrei mai immaginato di poter far parte di qualcosa di così grande, anche se piccolo all'apparenza, ed è anche grazie ai legami stretti con loro se sono sopravvissuto all'università.

Ringrazio infine Martina, non perchè è l'ultima persona da ringraziare, ma perchè il bello arriva alla fine. Sì, è una frase di circostanza, ma davvero lei è arrivata nel momento più critico della mia vita, ed è stata essenziale nella conclusione di questo percorso. Martina ha sentito e sopportato tutto, rimanendomi accanto quando non riuscivo a studiare con le lezioni a distanza e nulla sembrava avere senso; quando sentivo di subire ingiustizie tra esami falliti e situazioni scomode; quando credevo di non farcela e di aver buttato tempo prezioso, specialmente da fuorisede. Martina ha sopportato la mia incapacità di comunicare a distanza, la mia assenza, la mia noncuranza, venendone a capo con me e sempre portandomi nel cuore. Ci sono state discussioni, incomprensioni, difficoltà, ma è stato sempre nel suo interesse che io stessi bene, e ha fatto sì che non mi sentissi solo nemmeno quando ciò sembrava inevitabile, quando non riuscivo a non provare ansia e sconforto e le sembrava di non contribuire al mio benessere, ma la sua presenza in realtà era tutto. Ha cercato ogni giorno di distanza di farmi aprire, e non chiudermi nelle difficoltà di una nuova esperienza, e non ci ha pensato due volte a venire a stare con me il giorno del mio compleanno, quando ne avevo più bisogno, facendomi anche una bellissima sorpresa. Ha spronato la mia perseveranza, ha fatto sì che mi svegliassi ogni giorno consapevole che lei aspettava il mio buongiorno, e che avrebbe voluto sapere succesivamente come avessi passato la mia giornata, e mi ha fatto andare a letto ogni giorno senza sentirmi solo, pur vivendo in una stanza singola. Non si può riassumere nè quantificare quello che ha fatto per me, posso solo dire che le devo la mia salute mentale, il mio successo, il fatto che sono arrivato sin qui nel miglior modo possibile. A te dedico questa tesi, perchè di te c'è qua dentro più di quanto tu possa immaginare, ti amo.

Table of Contents

List of Tables

List of Figures

Acronyms

A2C
syncrhonous advantage actor-critic

A3C
asynchronous advantage actor-critic

AD
automatic differentiation

AI
artificial intelligence

AM
additive manufacturing

ANN
artificial neural network

CNN
convolutional neural network

DAG
directed acyclic graph

DL
deep learning

EM
extracellular matrix

LR

learning rate

MDP

markovian decision process

MLP

multilayer perceptron

MSE

mean squared error

OA

optimization algorithm

OvS

optimization via simulation

ReLU

rectified linear unit

RL

reinforcement learning

RM

regenerative medicine

SGD

stochastic gradient descent

TE

tissue engineering

Chapter 1

Introduction

Following the results and innovations given by Industry 4.0 and digitalization, such as 3D Printing, Additive Manufacturing or better simulations, the concept of biofabrication has evolved rapidly in the last decade, with a longer history of re-definitions. In [1] it is defined as "*the automated generation of biologically functional products with structural organization from living cells, bioactive molecules, biomaterials, cell aggregates such as micro-tissues, or hybrid cell-material constructs, through Bioprinting or Bioassembly and subsequent tissue maturation processes*". This definition clarifies the similarity and the confusion between biofabrication and bioprinting, which instead produces structures that can be used for other biological applications, by assembling living and non-living materials. Also in [2] the terminology used in biofabrication is re-conciliated for a deeper understanding of its concepts.

In the context of Tissue Engineering (TE), which is an interdisciplinary field that aims to produce biological substitutes, and Regenerative Medicine (RM), which aims to restore the structure and function of damaged tissues and organs [1], biofabrication is an important set of methodologies that can be applied to reach these goals, as they can generate constructs that more closely recapitulate the complexity and heterogeneity of tissue and organs than currently available therapies [2]. Given the potentiality of TE, this field has still many unanswered questions and the complexity of biological processes limits the ability to design optimal products and have successful application [3]. On this trail, biofabrication processes are complex both biologically and technologically, requiring a dynamic configuration of control parameters, which is called protocol [4]. Taking into account the need for expertise in the specific biofabrication process that is being followed, and the complexity of the biological processes (in particular the uncertainty on cells behavior which is relevant in many TE or RM approaches), advancements in the field are limited to human ability or luck in exploring the possibilities given from different process parameters, meaning that using traditional methodologies

is expensive and time-consuming. This is due to the fact that both the quantity and span of parameters can be large, making their combination a huge space to explore [3]. Trying each combination of parameters is called brute-force, a method that is well-known to not work in time-expensive tasks, other than in vast solution spaces. A brute-force approach is surely nowadays faster, but remains expensive and operator-depending, and still it's not fast enough to reach biofabrication goals.

For these reasons, Optimization Algorithms (OA) and Artificial Intelligence (AI) methods have been and are still being developed, revolutionizing common applications first and many more applications later. Although OAs and AI work very well in many cases, with biofabrication they are not sufficient alone. In order to explain why a certain biofabrication protocol works better than another, the complexity of biological processes requires not only to take into account the large ranges of process parameters, but also to explicitly model underlying biological behaviors and intrinsic characteristics of cells and tissues. In this perspective, two categories of approaches can be distinguished: black box and white box. With black box an algorithm acts without showing how it operates, while with white box there is an explanation about the internal processes. In the perspective of biofabrication, with white box approach, an algorithm can optimize its processes while giving information about how they work and why their best settings are better than the others. Even if it's evident that white box is preferable, such a method requires a complex model of the system, which is more difficult to obtain and handle with respect to black box models.

Following the idea proposed in [4], in this work an Optimization via Simulation (OvS) approach is proposed, introducing a framework that offers an optimization engine based on Reinforcement Learning (RL), which is an OA based on interacting with an environment and learning from its state evolutions. In addition, the framework offers a customizable interface that is designed to allow future different implementations by the plug-in of any simulator to the optimization engine. In this way, the user is allowed to use the framework with any type of interaction, only by creating the interface and giving it to the framework, so with minimum effort and programming knowledge required, creating different possible future implementations of the framework with simulators that are different from the one used in this project. In the following Sections, some state-of-the-art biofabrication techniques are presented (section 2.1), together with AI and RL concepts needed to design the framework (sections 2.2 and 2.3). Then, the adopted methodology is reported, by explaining the proposed framework and its components (3). Finally, an experiment is reported to show the potentiality of the framework and how easy is to adapt it to a specific use case (section 4).

Chapter 2

Background

2.1 State of the art

Biofabrication is deeply related to Additive Manufacturing (AM), which has evolved to be used for the production of high value parts with complex geometries [1]. In fact, in [2] it is shown how technologies born in Additive Manufacturing are now being commonly used in biofabrication, since they allow a more precise production with less waste production, even if materials can cost more than other manufacturing techniques (which is exactly the case of biofabrication). So it is clearer now how biofabrication and Bioprinting overlap, as they have the same basis, but biofabrication requires also a phase in which the built structure must be grown further to obtain the needed functional structure, also with approaches that are not restricted to Additive Manufacturing [1].

A common approach in biofabrication is the use of scaffolds, hierarchical prefabricated structures that support the proliferation of cells, and can be used directly in a defect site (*in situ*) [5], or to support the Extracellular Matrix (EM) formation while delivering bioactive factors in a controlled environment (*in vitro*) [1][3]. *In vivo*/*in situ* biological techniques are executed directly in the organisms where the tissue must be placed, while *in vitro* biological techniques are done in test tubes, Petri dishes or other environments outside the organism. The latter case allows more controllable experiments, without the difficulties of managing immunological responses, but has the drawback of having a biomaterial that does not behave like real cells *in situ*. *In vitro* environments can have many forms; an important examples are bioreactors, complex devices used to control the growth of 3D tissues [5]. Across developed methodologies, current researches aim at automatize the biofabrication processes. Controlled strategies have two possible approaches: bottom-up, based on the ability of cells to synthesize their own EM, or top-down, which uses scaffold or other supports to allow cells proliferation [5]. Both can be target of OA and AI methods.

Modern innovations in OA and AI allowed the introduction of new technologies (e.g. face recognition or self-driving cars) while allowing also improvements in existing processes, including biological applications. Optimization Algorithms [6] try to find an optimum solution in a given task, which can depend from a mathematical function, a cost function or other performance indexes. An important example of OA for AI is the Stochastic Gradient Descent, in which a function parameters are updated iteratively by a little fraction of the function Gradient. In the case of search-based algorithms, a solution space is explored by evaluating the performance obtained from different combinations of parameters, in an efficient way, thanks to heuristics, which are strategies based on common-sense or experience that are known to work in some cases. The most popular search-based algorithm is the Genetic Algorithm, in which a new set of solutions is generated by applying little random changes on the best previous solutions, or by combining them together. AI is instead an umbrella term, which collects different methodologies that can be used to fulfill a task mimicking human cognitive functions. Its meaning has been discussed for decades, but now common agreement follows the analogy of "acting rationally" described by Norving and Russell in [7], instead of the old "thinking humanly". Historically AI has been confused with Deep Learning (DL) [8], but actually it includes many other algorithms and technologies that can be combined together with a range of different goals. These algorithms can include also some OAs, or can work well with them. These approaches are versatile, allowing to improve the performance of many tasks that needed human effort, or expertise even with digitalized tools, until recently. For example, now AlphaFold has surpassed the limit on protein folding prediction [8]. Many more examples of application of OA and AI in biological processes can be found in [9] for drug discovery, in [10] for morphology-based cell culture, in [11] for droplet sorting and bright-field imaging through DL, and in [4] for GA for generating biofabrication protocols.

Many of the reported works improving biofabrication with an experimental approach need specialized technologies and procedures that, as it's now evident, are expensive in costs and time. In contrast to *in vivo* and *in vitro* approaches, *in silico* methodologies allow to exploit the power of modern computation, in addition to the possibility of integrating OA and AI to optimize the analysis of the biological processes, which is critical in biofabrication. However, one of the most important steps in switching from *in vivo*/*in vitro* to *in silico* is developing a validated model of the biological process that is being simulated, since the simulated process needs to behave with high fidelity with respect to the real process in order to obtain a functional product. A first relevant innovation that joins together *in silico* and *in vivo* or *in vitro* benefits are the Digital Twins, which are Digital Models that are validated with the real model first, and then are made able to both obtain information from and provide instructions to a real plant [12]. Digital twins rely on the combination of computational models of the physical process, and methods

to characterize, analyze and optimize the physical process.

Optimization via Simulation (OvS) is an example of this general scheme, which applies optimization methods on simulated environment. In this way, validated computational models can support the research for optimal process designs, without the drawbacks of required cost, time and resources from the experimental exploration of physical processes. OvS can be model-based, which applies an optimization algorithm on a simulator (this is a white-box approach where the model of biological processes explicitly represents underlying biological mechanisms), or metamodel-based, which uses a metamodel that estimates the input-output relations of the simulation model [13]. The advantage of using the simulation of a validated computational model of the physical system in OvS is that its results, that are directed by a simulator, can be validated on the real process, and explicitly represent its parts and organization, making it fully understandable. The optimization engine in OvS can also rely on a black-box model for the optimization part, in the cases in which it is more accurate, but thanks to the simulation models employed it still maintains the interpretability that is given by white-box.

Biological systems involved in biofabrication pose peculiar challenges to white-box computational modeling approaches due to their intrinsic complexity and their multi-level and multi-scale organization [14]. Existing approaches to model biological systems range from mathematical to computational models, relying on either continuous, discrete or hybrid formalisms [15, 16]. The diffusion of Systems Biology Markup Language (SBML) [17] determined the spread of Ordinary Differential Equations (ODEs) models in systems biology, while other approaches, such as Agent-Based Models (ABMs) and discrete models usage is less frequent, also due to their lack of reproducibility and standardization [18]. Yet, simulation approaches based on discrete models or ABMs have a great potential to enrich the toolkit of modelers in computational biology [19]. For example, Petri Nets models naturally express concurrency and encapsulation, and can be extended to represent hierarchies of multiple dimensional scales and levels of abstraction, proving very useful to model complex biological processes [20] such as ontogenesis [14, 21, 22], or the relations between an host organism and the microbiota[23, 24]. An high-level, domain-specific language to support accessible and user-friendly construction of hierarchical Petri Nets models is under development [25, 26].

OvS approaches thus can rely on a range of modeling formalisms and construction aids for white-box models of the biofabrication process of interest. Similarly, they can rely on a range of computational approaches for their optimization.

To exploit the potentiality of OvS, Deep RL algorithms are a viable way to obtain a learning model that acts following the chosen simulator rules. These algorithms combine the benefits of Artificial Neural Networks (ANN) with the ability of RL to learn from an environment.

2.2 Convolutional Neural Networks

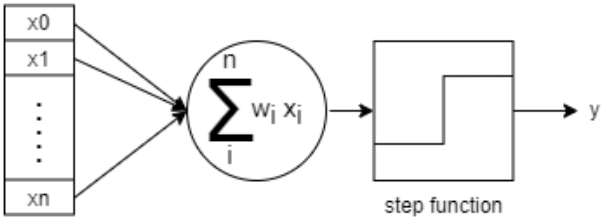

Figure 2.1: The perceptron.

2.2.1 Deep Artificial Neural Networks

ANN have a long history of evolution and research, but only in the last decade this field has encountered an exponential growth thanks to both smart ideas and, more important, advancements on computational power, GPUs and lowering of hardware costs [3].

The fundamental idea comes from Multi-Layer Perceptron (MLP), first introduced by Rosenblatt in 1958 [27], which is a chain of nodes, called perceptrons, which output is the input of another node: every node sums the output of previous connected nodes, by weighting each connection with a different value. Given x input sample, n size of input and w_i weight for input i:

$$\sum_{i}^{n}(w_i x_i) \tag{2.1}$$

This is inspired by how neurons work: each one propagates an electrochemical signal through the axons to other neurons thanks to synapses, where each structure (like the weights) defines what a neuron will perceive, and the neuron structure will define whether it will activate and fire or not [3].

In MLP the "neurons" are called perceptrons, and their weighted sum (equation 2.1) is applied on a step function (figure 2.1). This means that the perceptron is a linear classifier, so it can be trained by updating its separation surface vector with the difference of direction with each training sample (figure 2.2). Unfortunately, this works only with linearly-separable data [28], which isn't sufficient for real-world applications. However, perceptrons are still useful, as it's sufficient to add a non-linear activation function at its end to obtain a non-linear classifier. An example is using the sigmoid function to obtain a logistic regression classifier, but there are plenties of other activation functions: some of the most important are softmax [29], which acts like a probability and so is used at the output layers, and Rectified Linear Unit (ReLU) [30], which fires after an input equal to 0 and never saturates (figure 2.3).

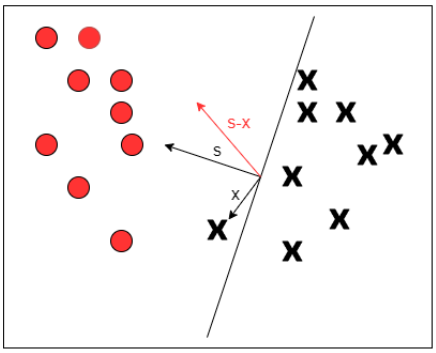

Figure 2.2: Update of separation surface vector through a data sample.

A further improvement is given from deep composition: putting together many layers of perceptron plus activation function allows to obtain a deep model (figure 2.4), which is highly non-linear, from where comes the definition of Deep Learning: the usage of a perceptron network with many non-linear layers, which is the MLP, obtaining a Deep Artificial Neural Network [31].

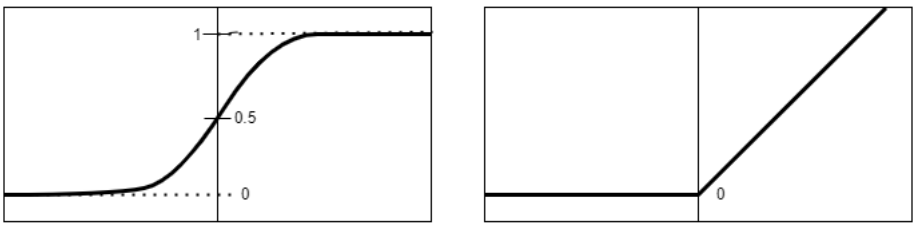

Figure 2.3: Softmax function (left), ReLU function (right).

MLP consist of layers of more perceptrons, where layers between input and output ones are called hidden layers, and each layer neuron has a connection to each of its next layer neurons (figure 2.5). Moreover, different layers can have different activation functions.

ANN can be trained by updating the neurons weights by comparing the output that the network gives when feeded with data with their truth value. This requires a labeled dataset, which has an associated category for each datapoint, and works by improving weights that are useful to produce the correct output, and reduce the ones that worsen the performance. This allows the network to adapt its layers so that only the needed data features are used, while in traditional Machine Learning models it is critical to decide which features are useful for a classification task, as dimensionality reduction improves accuracy and interpretability [32].

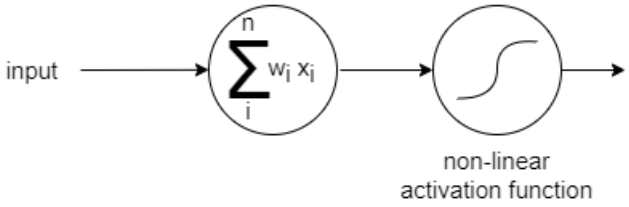

Figure 2.4: A perceptron linked to an activation function.

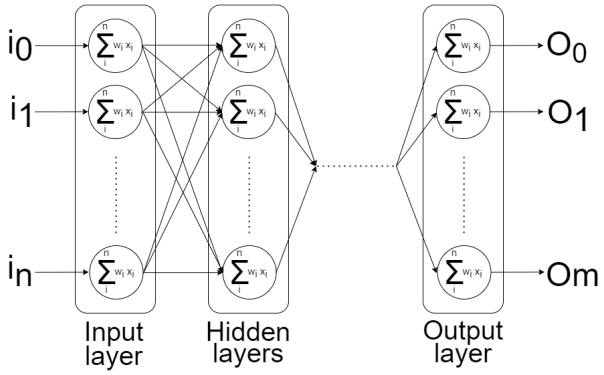

Figure 2.5: Deep Neural Network.

2.2.2 Convolutional layers

MLP are still not enough, as they don't reach good performance in complex applications. The first great improvement that brought DL to its current evolution is the convolutional layer [33]: a matrix (filter) applies a discrete convolution on an image color channels (RGB), producing a feature map (figure 2.6). The weights of the filter act at the same way of MLP weights, so during training they adapt to learn only important and useful features in the images. How much a convolutional layer slides on the image is called stride, while the filter size is called receptive field, and together they define the output feature map size (figure 2.7). In addition to convolutional layers, pooling layers (originally C-cells [33]) reduce the size of a feature map by combining together its subsections and thus reducing the computational power and memory required by the network (figure 2.8). Then, just as in MLP, activation functions can be added after the convolutional layer. Finally, more filters of the same type can be stacked together in a single layer, just like color channels are stacked in an RGB image, letting the network learn different features at the same time.

Thanks to Convolutional Layers, Convolutional Neural Networks (CNN) [34] are born, allowing new network architectures and the resolution on highly complex tasks that require images, videos, or even other time-sequence data such as audio

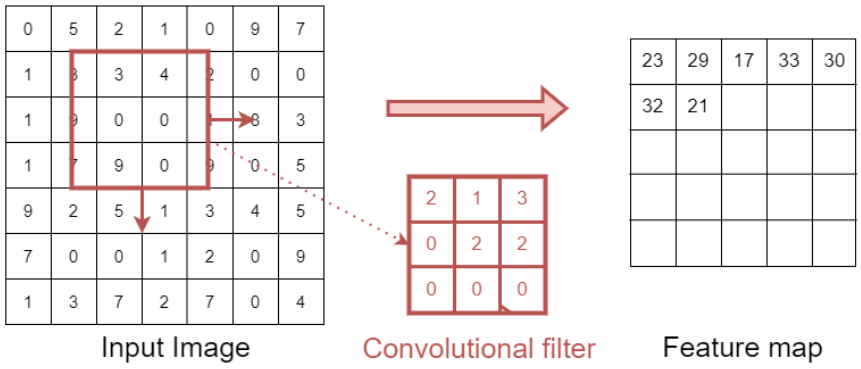

Figure 2.6: Convolutional filter applied on input to produce the feature map.

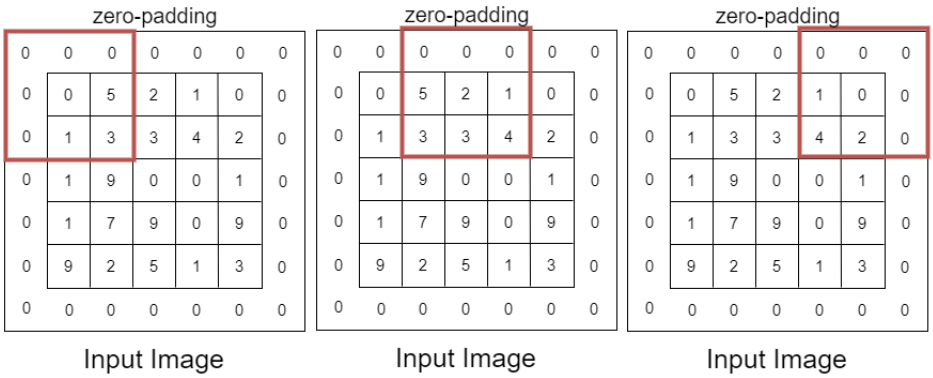

Figure 2.7: Example of filter sliding with zero-padding and stride=2.

or text. Classic perceptron layers (here called fully-connected layers) can still be used, and usually they are put at the end of the network, in order to collect what the convolutional layers have processed and to decide with softmax functions the output categories probabilities, at least in a classical classification task. For example, a network with two softmax can be trained to tell if, in an image from a dataset with images of cats and dogs, there is a cat or a dog. The first important simple CNN architectures are AlexNet [35], VGG [36], GoogleNet [37]. All of these architectures obtained one after the other a lower error on the ImageNet Large Scale Visual Recognition Challenge ILSVRC [38], but reached soon a limit due to the depth of the networks: a too deep network have a lower performance due to the "vanishing gradients" problem, which means that the gradients of the first layers becomes too small to let the network learn something, but also a deeper network means more memory usage and more computational time required.

To solve this problem, [39] introduces the Residual Network, a novel architecture that uses the Residual Layer, which contains more convolutional layers in sequence, and a skip connection from the start to the end of the layer. ResNet networks are

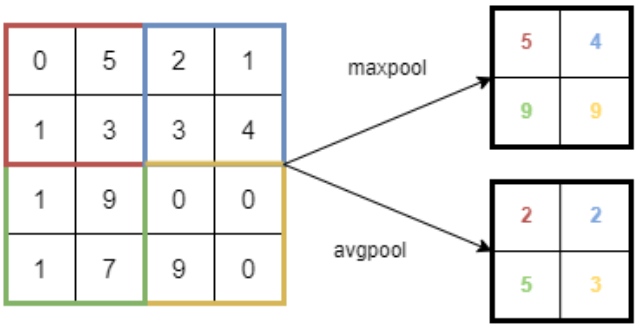

Figure 2.8: Example of possible pooling operations for pooling layers.

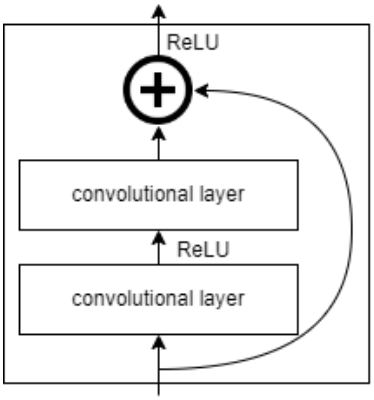

Figure 2.9: The residual layer.

proposed with more residual layers stacked together. This kind of connection allows the network to pass the gradient on, avoiding the vanishing gradient (figure 2.9). From now on, more architectures have been proposed, with new possible tasks, e.g.:

- object detection, detecting the position of something in the image;
- segmentation, assigning a category to each pixel of the image;
- instance segmentation, a combination of both the previous tasks;
- sequential data processing, videos, audios, texts
- image generation, generating images from other images or from textual description;
- description generation, generating textual description of images;
- domain generalization, recognizing objects across different domains (e.g. photos, drawings, cartoons).

and many more are under development.

2.2.3 Loss and Backpropagation

The training process in ML concerns the updating of a learning model parameters such that either an error is minimized or a performance index is maximized. In the setting of DL this update goes through an error function called loss function. Given a classifier C, its loss function is the measure of the error that C makes when predicting the label of input training data, by using its true known labels:

$$\begin{cases} L(C, x, y) = 1 & \text{if } C(x)! = y \\ 0 & otherwise \end{cases} \tag{2.2}$$

where L is the loss function, x is the predicted label and y is the true label. There are many possibilities for a loss function. Some common losses are Mean Square Error (MSE), Cross-Entropy, Huber [40] (and more in [41], [42]), but personalized functions can be used. In general, loss can be written as:

$$\sum_{x,y} L(C, x, y)P(x, y) \tag{2.3}$$

where $P(x, y)$ is the probability of predicting a label x when the true label is y. Training the classifier means minimizing the loss function, which translates into computing its gradient with respect to its parameters. To apply the gradient, the loss function needs to be differentiable at all points, so the gradient function needs to be continuous.

An optimization algorithm that can be used is Gradient Descent [43], an iterative algorithm:

- start from a random setting of parameters, then iteratively:
- calculate the gradient of the loss function;
- update the parameters with a small fraction of the gradient;
- stop at minimum.

Mathematically, the gradient of a function is orthogonal to level curves, which means that its direction goes to the steepest direction starting from the point in which it is evaluated. However, if the surface is too steep its value can be high, so subtracting it from the parameters could lead to surpass the loss point of minimum, so iteratively the algorithm could jump around it without never reaching it, or either reach a different local minimum which is not optimal. This is why the gradient needs to be multiplied with a small number λ:

$$\theta^{t+1} = \theta^t - \lambda \nabla L_{\theta^t} \tag{2.4}$$

where θ^t are the classifier C parameters at timestep t. The number λ is the Learning Rate (LR) [44], which must be set properly: a too high value causes overshooting,

which is jumping around the point of minimum, while a too small LR causes a slow convergence speed, or no learning at all. An hyperparameter optimization phase during training allows finding a good LR value. It can also follow an adaptive strategy, which makes it change during time: a decay parameter ρ [44] reduces the LR over time, and a momentum parameter v accumulates the gradient directions over time:

$$\lambda^{t+1} = \lambda^0 e^{-\rho t} \tag{2.5}$$

$$v^{t+1} = \lambda v^t - \alpha \nabla L(\theta^t) \tag{2.6}$$

$$\theta^{t+1} = \theta^t + v^{t+1} \tag{2.7}$$

Gradient descent can be applied even to non-convex functions, but in this case it's not guaranteed to reach a global optimum [45]. Luckily, in the case of DL it is not necessary: training too well the model leads to overfitting [46], which means that the model cannot generalize and so doesn't work well on new unseen data . In this case, training error is very low, but accuracy on test data (which is new data not used during training) is high. However, gradient descent is still not sufficient, as in ML tasks require usually a big number of training samples and parameters, making gradient descent impractical as it needs to be computed for each sample and parameter. Stochastic Gradient Descent (SGD) (from the idea of Robbins and Monro [47]) is an approximation of gradient descent, which applies the algorithm on a small batch of data at a time, called minibatch, which is much faster with an accuracy reduction as trade-off. As a further advantage, minibatches can be processed in parallel. Everytime the algorithm uses all the training samples it is called an epoch. To avoid bias, SGD needs a shuffled training dataset, which increases the algorithm performance [48].

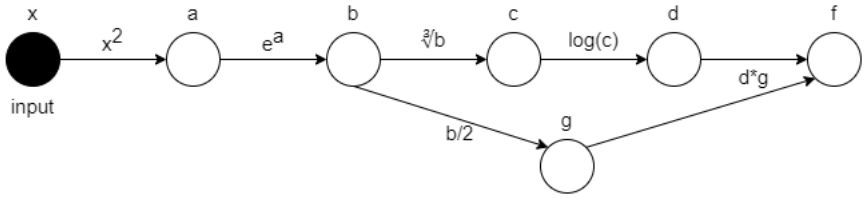

Figure 2.10: Computational graph of $f(x) = log(\sqrt[3]{e^{x^2}})(\frac{e^{x^2}}{2})$.

The gradient computation remains still a problem. Computing the gradient manually or with the simple chain rule is a long process, even computationally. Common loss functions are convex, but generally personalized losses aren't. As a work-around computational graphs are used [49], which consist in a Directed Acyclic Graph (DAG) where each intermediate node is a computation that could depend from more previous nodes, while more following nodes could depend from it. In Figure 2.10 the computational graph of $f(x) = log(\sqrt[3]{e^{x^2}})(\frac{e^{x^2}}{2})$ is shown as an example.

A fast way to compute the gradients without the drawbacks of long symbolic calculations is Automatic Differentiation (AD) [50], which keeps track of intermediate variables to exploit their dependencies, through computational graphs. It has two alternatives: Forward mode and Reverse mode. Starting from the example of Figure 2.10 the dependencies are:

$$f = d * g$$

$$g = b/2$$

$$d = log(c)$$

$$c = \sqrt[3]{b}$$

$$b = e^a$$

$$a = x^2$$

- forward mode: each variable gradient is computed starting from input nodes to output nodes, so that an already-computed derivative can be used subsequently in the computation of derivatives of nodes that depend on them. Given input x and i-th node n_i:

$$\frac{dn_i}{dx} = \frac{dn_i}{dn_{i-1}} \frac{dn_{i-1}}{dx}$$

so from the example:

$$\frac{da}{dx} = 2x$$

$$\frac{db}{dx} = \frac{db}{da}\frac{da}{dx} = (e^a)\frac{da}{dx}$$

$$\frac{dc}{dx} = \frac{dc}{db}\frac{db}{dx} = \frac{1}{3\sqrt[3]{b^2}}\frac{db}{dx}$$

$$\frac{dd}{dx} = \frac{dd}{dc}\frac{dc}{dx} = \frac{1}{c}\frac{dc}{dx}$$

$$\frac{dg}{dx} = \frac{dg}{db}\frac{db}{dx} = \frac{1}{2}\frac{db}{dx}$$

$$\frac{df}{dx} = \frac{df}{dd}\frac{dd}{dx} + \frac{df}{dg}\frac{dg}{dx} = g\frac{dd}{dx} + d\frac{dg}{dx}$$

- reverse mode: dependencies are stored like in the forward mode starting from input up to output nodes (forward pass); then, starting from output node back to input one, each node differentiation is computed by summing the product of following connected nodes derivative, each multiplied by their derivative with respect to the actual node (backward pass). Given input x and i-th node n_i:

$$\frac{df}{dn_i} = \frac{df}{dn_{i+1}} \frac{dn_{i+1}}{dn_i}$$

so from the example:

$$\frac{df}{dx} = \frac{df}{da}\frac{da}{dx}$$
$$\frac{df}{da} = \frac{df}{db}\frac{db}{da}$$
$$\frac{df}{db} = \frac{df}{dc}\frac{dc}{db} + \frac{df}{dg}\frac{dg}{db}$$
$$\frac{df}{dc} = \frac{df}{dd}\frac{dd}{dc}$$
$$\frac{df}{dd} = g$$
$$\frac{dd}{dc} = \frac{1}{c}$$
$$\frac{dc}{db} = \frac{1}{3\sqrt[3]{b^2}}$$
$$\frac{df}{dg} = d$$
$$\frac{dg}{db} = 1/2$$
$$\frac{db}{da} = e^a$$
$$\frac{da}{dx} = 2x$$

In the case of ANN, backward mode is called Backpropagation [51], and it can be seen in Figure 2.11. Performance of backpropagation depends on many factors: starting weights of the network, learning rate, training set. In any case, several problems can be faced by setting them wrongly: for example exploding gradients (too high values that can lead to loss values jumping around, or even numerical errors) or overfitting.

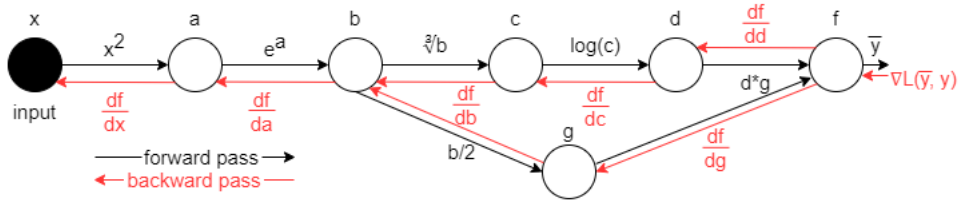

Figure 2.11: Backpropagation of the gradient of loss between real value y and output value $\bar{y}$.

Modern DL libraries have their own implementation of backpropagation: PyTorch uses its autograd engine [52], while Tensorflow has GradientTape [53]. Via backpropagation through the hidden layers, the network learns what features of the input data are relevant at each layer with no concept of science or prior knowledge required.

2.3 Reinforcement Learning

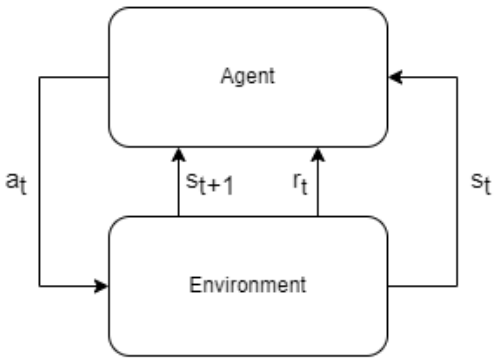

Figure 2.12: Agent and Environment interaction.

RL is a subset of OAs whose main components (agent, environment, reward) differentiates it from the other algorithms [54] (figure 2.12):

- Environment, can be interacted and has a state that changes with every interaction
- Agent, can act on the Environment and see how its state changes
- Reward, a score that is given from the environment when the agent interacts with it, and depends on the goal of the algorithm

The function of the reward is to tell how good is the taken action, so that the agent can take it into account while deciding if that action is suited to the environment state or not. Each environment state has a value, which can be seen as how much reward can be collected in the optimal case while being in that state. Both the reward and the state values can be used to decide if an action can be taken or not in the future: different strategies can vary in how much they are considered, but usually it's more important to take actions that maximize the accumulated reward, so going in the highest-value states is the common objective.

RL can be formalized through Markov Decision Process (MDP) [55], which consist in trajectories of state-action-reward, extending the Markov Chains [56]. Trajectories are collected by iteratively seeing environment state s, taking an action a and collecting a reward r, for each timestep t:

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, ..., s_t, a_t, r_t, ...$$

Following the trajectories, rewards are collected so that the return value can be estimated:

$$G_t = \sum_{i=0}^{t} r_i = r_0 + r_1 + r_2 + ... + r_t \tag{2.8}$$

A RL algorithm aims to maximize the return value G_t, which is the same as saying that environment states have the maximum values, respecting the reward hypothesis, which says that any goal can be translated in maximizing the return value. The true value of a reward is a difficult engineering problem, as it must be well suited to the goal and the environment, and must be able to better differentiate between good and bad action in a way that lets the algorithm learn [55]. Moreover, the process could go on indefinitely, so to avoid a divergence (reaching a too big value) a discount rate $0 < \gamma < 1$ is used:

$$G_t = \sum_{i=0}^{t} \gamma^i r_i = r_0 + \gamma r_1 + \gamma^2 r_2 + ... + \gamma^t r_t \tag{2.9}$$

This value is an hyperparameter as it defines how much the reward values are important on the short-run and on the long-run.

The environment state is completely described, but it could be not fully observable. The agent has its representation of the environment state, based on what it can see and what it needs. An important step here is the Markov Hypothesis, which says that the future states are independent from the past states (Markov Property) [55], so in probabilistic terms:

$$P[s_{t+1}|s_t] = P[s_{t+1}|s_0, s_1, s_2, ..., s_n] \tag{2.10}$$

or in simpler words, the current state is enough to completely know the environment, thus simplifying way more the mathematical treatment of the process. In MDP a

policy π is a map of the transition probabilities between the environment states, which is held by the agent, so in other terms it defines the agent behavior in the environment by telling it what to do given a state. The policy can be changed either considering the states transitions (deterministic policy), or also considering the possible actions in the current state (probabilistic policy). Given a policy, the state value function or V-function v_π defines the value of each state, and is evaluated by starting from a state s and following the policy, and then averaging the expected future rewards:

$$v_\pi(s) = E_\pi[\sum_{i=0}^{+inf} \gamma^i r_{t+i+1} | s_t = s] = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+1} + ... | s_t = s] \quad (2.11)$$

The action value function or Q-function Q_π defines the value of each action in each state, and is evaluated by starting from a state s, taking an action a and following the policy π, and then averaging the expected future rewards with respect to the policy:

$$Q_\pi(s,a) = E_\pi[\sum_{i=0}^{+inf} \gamma^i r_{t+i+1} | s_t = s, a_t = a] = [r_{t+1} + \gamma V(s_{t+1})] \quad (2.12)$$

which is equivalent to taking an action a in the state s and then evaluating the state value function of the next state. Each state-action values of the Q-function are called Q-values. Taking into account the policy $\pi(a|s)$, so the probability of taking action a in state s:

$$v_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a) \quad (2.13)$$

Thanks to the rewards, an RL agent can update the state value function and the Q-function to let the policy converge to the optimal one, but the exact way how this is done is a specific RL algorithm. The general idea is to maximize the expected rewards obtained in the future, which of course are not known in advance. However, we can simplify this mathematical treatment by rewriting the state value (equation 2.11) function in a recursive way:

$$v_\pi(s_t) = E_\pi[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + ...)] = E_\pi[r_{t+1} + \gamma v(s_{t+1}) | s_t = s] \quad (2.14)$$

then, taking into account transition probabilities $p(s'|s,a)$ of going into state s' by taking action a in state s, since the expectation E is the sum of outcomes multiplied by their probabilities, equations 2.13 and 2.14 can be rewritten as:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)(r + \gamma v_\pi(s')) \quad (2.15)$$

where r is the reward given by the transition from s to s'. This leads to a straightforward conclusion: to converge to an optimal policy, so a policy that makes the agent behave in the best possible way, it's enough to update the known state value function (by starting with a random one) with the obtained reward and the discounted next state value. Finally the discount value makes really sense, as its value can definitely change how much the algorithm can give more weight on either the already known value or the obtained reward, defining so how much it learns from the environment and how much it must follow the already known policy. This gives us an iterative approach as a starting point for RL algorithms.

Finally, if we want to optimize the policy, we need to get the best possible value for each state, obtaining from equation 2.15 the optimal state value function $v^*(s)$:

$$v^*(s) = \max_\pi v_\pi(s) = \max_\pi \sum_a \pi(a|s) Q_\pi(s,a) = max_a \sum_{s'} p(s'|s,a)(r + \gamma v^*(s')) \tag{2.16}$$

Even if until now these functions seem long and tricky, they all mean that we can start from our actual knowledge and update it by taking a discounted amount and choose the action with the best value.

2.3.1 Q-learning

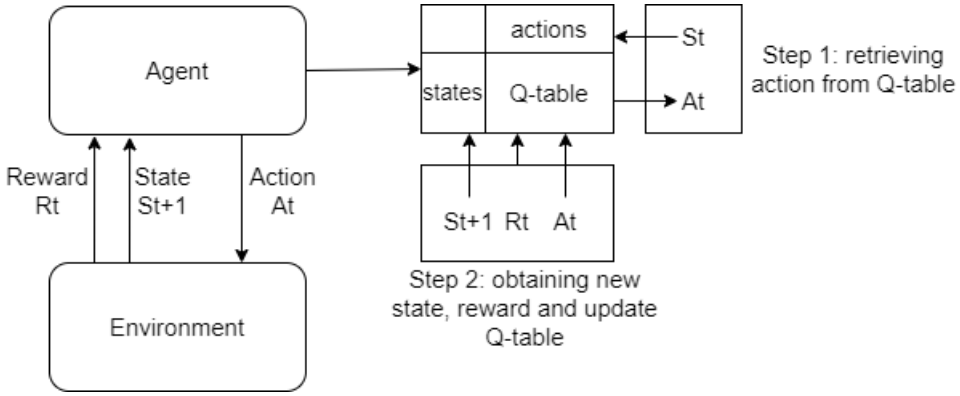

Figure 2.13: Q-learning representation of RL.

Thanks to the iterative computation of the state value function (equation 2.15), an analogue version of the Q-function can be written:

$$Q_\pi(s,a) = \sum_{s'} p(s'|s,a)(r + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s',a')) \tag{2.17}$$

where this time the outcomes are the Q-values of the next states and the probabilities are given by the policy itself, to compute the inner expectation. We can notice

that this value corresponds to the state value function of the considered next state $v_\pi(s')$, so we can rewrite again this function as:

$$Q_\pi(s,a) = \sum_{s'} p(s'|s,a)(r + \gamma v_\pi(s')) \tag{2.18}$$

Now, since the optimal state value function can depend on choosing the best actions through the Q-function, as shown at the end of previous Section, the strategy is now to optimize directly the Q-function:

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \tag{2.19}$$

which follows the iterative approach given from the Bellman equation. Starting from a random Q-function, we can choose the best action each time and update the function itself, obtaining the Q-learning algorithm (figure 2.13). This algorithm can be formulated in a model-based (or stochastic) method, through the transition probabilities:

$$Q^*(s,a) = r + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q^*(s',a') \tag{2.20}$$

or in a model-free (or deterministic) method, that requires only the Q-values:

$$Q^*(s,a) = (1-\alpha)Q^*(s,a) + \alpha(r + \gamma Q^*(s',a')) \tag{2.21}$$

where the parameter α defines how much value is taken from the current state, and how much from the reward and best next possible state, meaning that the Q-function can be updated more with the new information, or remain more or less the same. Both require simply to run the agent on the environment, collect the reward, and update with simple computations the Q-function. The deterministic method is even simpler, and since its Q-values are an association of state and action they can be collected in a matrix called Q-table. This process requires also a parameter ϵ that defines the probability to take a random action, or to take the best action according to the Q-table. In this way, if the probability of taking random actions is high, even taking wrong actions allows the algorithm to learn faster the optimal value for each action, and then it can be reduced to start doing more right actions, reaching the algorithm convergence. More important, it can be demonstrated that the Q-algorithm is able to converge to the optimal Q-function with a large amount of trials [55].

This algorithm is solid as it always converges when taking each state-action transition enough times [55], but has the drawback of having large Q-tables even for little complex problems, so it would require a long time to try every action many times for every state to let the model converge well to the optimal one. For this reason, a CNN can be used to approximate the Q-function. Still, this method is only able to work with discrete actions (e.g. up, down, left, right), so continuous

actions (e.g. a real number representing an acceleration) need to be discretized (divided into intervals) and to be limited to a minimum and a maximum value in order to work with Q-learning.

2.3.2 Policy Gradients

As already said, Q-function can be hard to learn for some applications, and it can be easier to map directly states to action. Moreover, Q-learning does not work with neither continue nor stochastic actions, and can be unstable other than suffering from states uncertainty. On the contrary, policy gradient can work with continuous or even stochastic actions, by relying on the optimization of a parameterized model that represents the policy [57]. In particular, policy gradient estimates the parameters of a learning model so that the policy $\pi_\theta(a|s)$ is optimal, by maximizing the expected reward as objective function:

$$J(\theta) = E_\pi[G_t] \tag{2.22}$$

$$\theta_* = \underset{\theta}{\text{argmax}}(J(\theta)) \tag{2.23}$$

Generally it follows the gradient update rule of Gradient Ascent, similar to the one of Gradient Descent already seen in Section 2.2.3 (equation 2.4):

$$\theta_{t+1} \leftarrow \theta_t + \lambda \nabla_\theta J(\theta_t) \tag{2.24}$$

but the gradient of J cannot be computed, so we need to rewrite it first:

$$\frac{\delta J(\theta)}{\delta\theta} = \frac{\delta}{\delta\theta} E_\pi[G(t)] = \frac{\delta}{\delta\theta} \int \pi_\theta(\tau) G(\tau) d\tau = \int G(\tau) \frac{\delta}{\delta\theta} \pi_\theta(\tau) d\tau = \tag{2.25}$$

$$= \int G(\tau) \pi_\theta(\tau) \frac{\delta}{\delta\theta} \log \pi_\theta(\tau) d\tau = E_\pi[G(t) \frac{\delta}{\delta\theta} \log \pi_\theta(t)] \tag{2.26}$$

where at the penultimate step the following identity has been used:

$$\frac{\delta}{\delta\theta} \log \pi_\theta(x) = \frac{1}{\pi_\theta(x)} \frac{\delta}{\delta\theta} \pi_\theta(x) \rightarrow \frac{\delta}{\delta\theta} \pi_\theta(x) = \pi_\theta(x) \frac{\delta}{\delta\theta} \log \pi_\theta(x) \tag{2.27}$$

So the derivative of the expected return is the expectation of the return weighted by the derivative of the log probability of actions in a given state. This result tells that it's enough to collect more trajectories and average them out, by considering for each timestep the probability of taking an action, sampling it and multiplying it with the return value. The defined objective function means nothing more than increasing the probability of taking actions that cause a good reward, and decreasing the probability of taking actions that cause a bad reward: this happens since if a sequence of actions lead to an improvement to the policy, then the gradient

will have a positive value on some parameters, which will be incremented in the update. This method requires only to act on the environment and then retrieve data, so it is completely model-free, and it introduces the REINFORCE algorithm [58]:

Algorithm 1 REINFORCE algorithm

start with random θ_0 parameters
for n episodes **do**
  start from initial state s_0
  while not done **do**
    observe state s_t and sample an action a_t from policy $\pi_\theta(a|s)$
    collect $s_t, s_{t+1}, r_t, log(\pi_\theta(a_t|s_t))$
  compute G_t for every t
  $\theta \leftarrow \theta + \lambda \nabla_\theta J(\theta)$

Reaching a final state is an episode, and the algorithm can be ran from an arbitrary number of episodes. Certainly more episodes means converging more to the optimal policy. To act with discrete actions, the policy can simply be a discrete distribution of actions that can be taken, and then the algorithm chooses an action by sampling from the distribution. To act with continuous action, a Gaussian distribution can be used by updating its mean and variance, and then the algorithm can sample a value by it.

2.3.3 Advantage Actor-Critic

Policy Gradients can suffer from high variance in the gradient computations, while Q-learning cannot represent continue actions, which could prevent reaching a global optimum solution [59]. An important advancement in RL are the Actor Critic algorithms [60], which join together both Q-learning and Policy Gradients algorithms: the Critic evaluates how good and bad are the taken actions, so the Q-value, or the states in which the environment falls after an action, so the state value, while the Actor defines how the agent will behave, so it contains the policy which is a probability distribution over actions that can be taken in a given state. In this way, the Actor objective function remains the same of the REINFORCE algorithm (left member), while the Critic actor is simply the Mean Square Error (MSE) between the current action value and the best possible action (right member):

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} [G_{t+1} \frac{\delta}{\delta\theta} \log \pi_\theta(t) + \frac{1}{2}(Q(s_t, a_t) - V(s_t))^2] \tag{2.28}$$

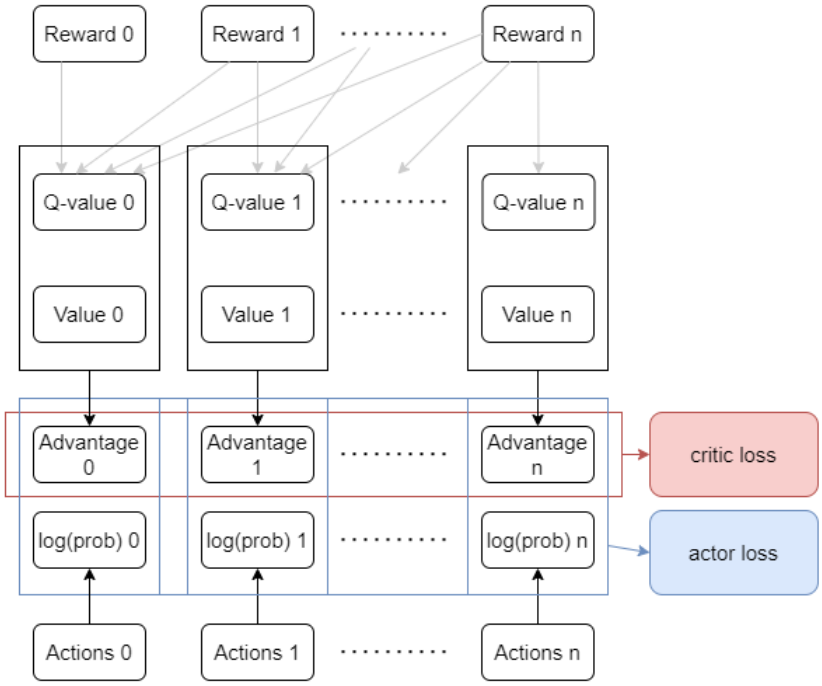

Figure 2.14: Data flow in A2C.

where T is the last timestep. The algorithm is almost the same of REINFORCE, while the learning model is composed of both probability distribution for Actor and Q-function (or Q-table) for critic. Sapling randomly from the distributions has however the drawback of increasing the variability in log probabilities, so the policy could converge to a sub-optimal one. Moreover, the fact that the expected return is multiplied by the log probabilities will worsen this issue. For these reasons, the log probabilities can be multiplied with a smaller value that still depends on the expected return, making so that the policy is updated with smaller steps thus becoming more stable. One of these approach is the Advantage Actor-Critic [61], in which the log probabilities are multiplied for the Advantage, which express how much the taken action is better with respect to the others:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.29}$$

from which follows the new objective function, schematized in Figure 2.14, where the member on the right is exactly the advantage:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} [A(s_t, a_t) \frac{\delta}{\delta\theta} \log \pi_\theta(t) + \frac{1}{2}(Q(s_t, a_t) - V(s_t))^2] = \tag{2.30}$$

$$= \sum_{t=0}^{T-1} [A(s_t, a_t) \frac{\delta}{\delta\theta} \log \pi_\theta(t) + \frac{1}{2} A(s_t, a_t)^2] \tag{2.31}$$

Advantage Actor-Critic can be proposed in two forms:

- Asyncronous Advantage Actor-Critic (A3C) [61], able to perform more parallel trainings to improve the performance in a shorter time, by syncing their results;
- Synchronous Advantage Actor-Critic (A2C) [62], one or more agents act at a time and their result are joined together by resolving their inconsistencies.

A3C is surely more performing as it explores more the search space, and is more reliable, while A2C is simpler to implement. In any case, both are simple to implement by using Deep ANN, where each agent is a different neural network model, comprised of an Actor head and a Critic head:

- the Actor head can use a layer with softmax activation functions as output layer, to approximate the distributions of discrete actions, or two parallel layers that represent both mean and variance that will be used to sample a continue action from a consequent Gaussian distribution;
- the Critic head uses its output layer to generate the state value.

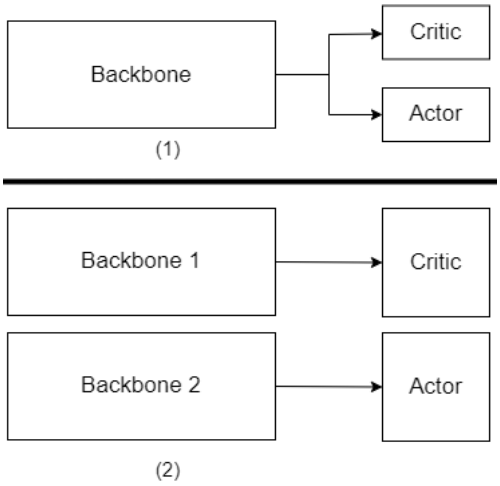

Figure 2.15: Actor and Critic heads with common backbone (1) and individual backbone (2).

There are many ways to create an ANN for Actor-Critic, and all can either use a common part (backbone) to process data and then pass the output to both heads, or there can be two separated processing layers for both heads from the beginning (figure 2.15).

Chapter 3

Methods

The framework proposed in this project is designed to respond to requirements to overcome the limits of traditional strategies that optimize biofabrication processes, highlightened in Section 2.1:

- need of human expertise and intervention; the framework is designed to autonomously act on a simulator and validated model;

- problem-specificity of the biofabrication solution; the framework has an easy-to-implement interface, which requires a little knowledge of Python language, supporting accessibility and flexibility, with the possibility to generalize its use to different simulators;

- expensiveness of traditional experimental methods; the combination of a simulator with an exploration engine based on RL allows an optimal exploration of the solution space without both the time and cost required by the empirical processes;

- interpretability of the solution; using an ANN to generate biofabrication protocols together with a simulator that represents the biological process gives an hybrid between black-box and white-box approaches, leveraging on both interpretability of the biofabrication protocol and a model capturing biological complexity that is required to learn the optimal one.

To tackle these challenges, this Thesis proposes an Optimization via Simulation methodology where the optimization process relies on RL models and the A2C algorithm to learn optimal biofabrication protocols, interfacing with simulations of white-box models of the growth of epithelial cellular sheets in culture. To

support these functionalities, this work also provides a framework to implement the RL model, optimization process and the interface to the simulator, featuring the components shown in Figure 3.1:

- ***Learning model***, a CNN that takes images as input and provides actions as output;
- ***Exploration engine***, implements the A2C algorithm to learn the optimal biofabrication protocol;
- ***Simulator interface***, contains the functions that must be implemented to attach a simulator to the framework.

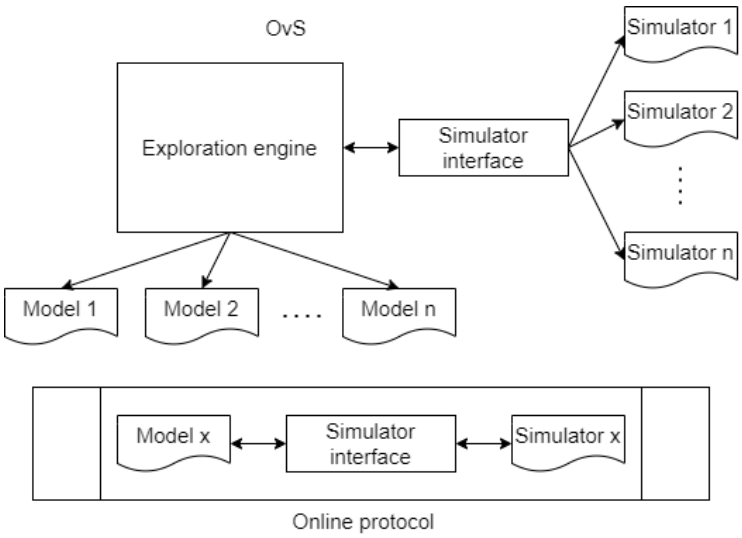

Figure 3.1: The proposed framework components.

The proposed framework executes the following flow:

- the *exploration engine* takes observations from the simulator and gives them to the *learning model*;
- the *learning model* receives an observation of the simulator state (which is an image) via the *simulator interface*, and produces an action as output to the exploration engine;
- the *exploration engine* executes the action on the simulator via the *simulator interface*, and collects the mentioned data together with other data needed to learn.

The simulator is accessed through a common *simulator interface*, which needs to be implemented for the specific employed simulator. Since the simulator communicates with the model during execution, the produced action sequence is defined as an ***Online protocol***.

Considering this general functioning, the detailed framework design is as follows.

The *exploration engine* implements a RL agent, by using the A2C algorithm (section 2.3.2) together with the ANN model (section 2.2), while the simulator part represents the RL environment, whose states are the observations, and that provides the exploration engine with the reward values for each action-state transition.

On the other hand, the *learning model* provides the probability distributions of actions. By running the *exploration engine* for a number of episodes, which are epochs for the ANN model, the latter can learn the best policy for a specific application, through the loss defined by the A2C algorithm. Each specific application defines the employed simulator and the target of the simulation.

This work implements a A2C algorithm that is a variant of the existing one. The proposed A2C uses a single isolated Agent that does not share its data. This stems from the intention of balancing the trade-off which inherently underlies model-based OvS: the interpretability of white-box simulation models has high computational cost [13]. To focus on the framework adaptivity potentialities, this work relies on the simplification of the RL model. In fact, using a single-agent A2C allows to have a simple RL model, that still can reach adequate performance without the drawback of synchronizing multiple agents. Moreover, the proposed approach allows running multiple learning experiments as parallel agents, both to perform larger experimental campaigns over fixed parameters, and to systematically explore hyperparameters to experimentally define the best ones for a given application in a shorter time (section 4.2.3). In the following Sections, each framework component will be explored with more details to better understand the data flow across the framework, the learning process functioning and how they support the optimization of biofabrication processes.

3.1 Artificial Neural Network model

This Section elucidates ANN model structure and operation in relation to the other framework components.

3.1.1 Model structure

The ANN model is built to be general and flexible, that is, to allow any simulator to be attached and any application to be implemented in the future. In particular, it can provide the actions required by the environment, whichever their number or their type is (both in the case they act over a continue and a discrete space). The

ANN model implementation relies on the tensorflow library (version 2.9.1) [63]. The model structure is shown in Figure 3.2 and is based on the A2C model which has been illustrated in Section 2.3.3:

- a ***backbone*** processes the environment state as observations to learn the visual components which are important to represent it. In this way, the Critic and the Actor can estimate their values. This implementation relies on the ResNet18 model [39] as a backbone.
- a ***Critic head*** processes the backbone output to generate the current state value;
- an ***Actor head*** processes the backbone output to generate:
  - one value through a softmax function for each discrete action;
  - two values (μ, mean and σ, variance) to generate a Gaussian distribution, from which sample a value, for each continuous action.

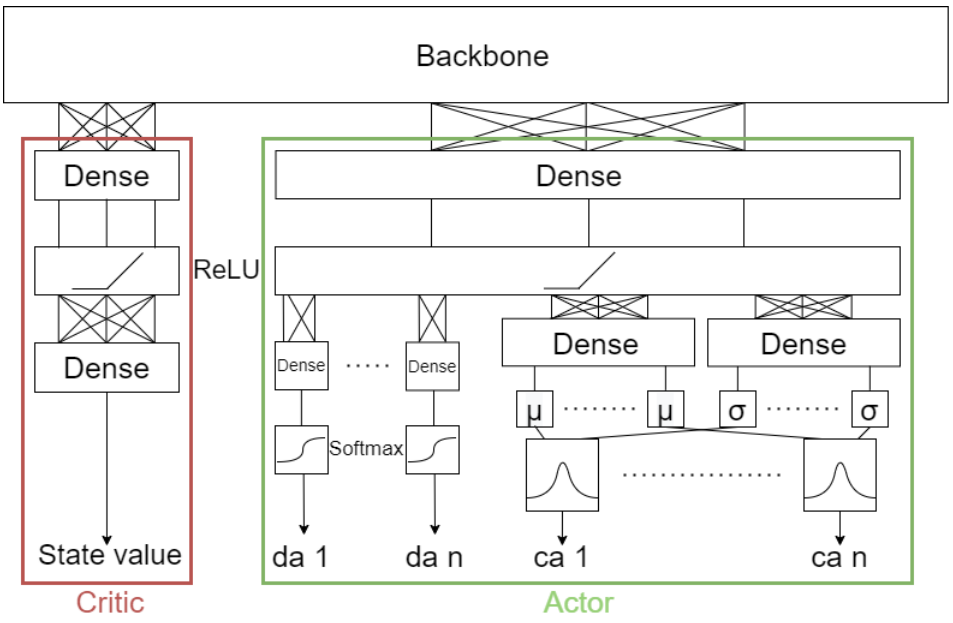

Figure 3.2: ANN model.

The *Actor head* and *Critic head* both have as an input layer a fully connected layer (a simple MLP layer), defined as 'dense' in tensorflow, and their number of neurons can be decided as hyperparameters. The ANN model outputs' size depends on the implemented environment, since different applications may require different set of actions for the respective simulators. For this reason, having a model whose

output size can be adapted is crucial to allow flexibility in using different simulators as for the *Actor head*:

- the number of output Gaussian distributions (and so number of neurons in both mean and variance layers) is equal to the number of continue actions;
- the number of output (parallel) layers with softmax activations is equal to the number of discrete actions; each layer's output has a size corresponding to the number of possible values of its corresponding discrete action.

The *Actor head* clips the output values of variance layers and of discrete action layers between 10^{-2} and 10^{3}, to avoid numerical issues in computation during the training phase. More details on the heads and on the backbone can be found in Appendix A.

3.1.2 Model operation

The first operation step is building the model. This requires to specify the size and shape of data that will be given to the ANN as an input. In fact, this determines the size and shape of its convolutional layers' output. The framework operates the ANN model with an input shape of *(1, width, height, channels)*, where *width* and *height* are, respectively, the width and the height of the image that constitutes an observation of the environment, while *channels* is the number of color channels from the image (e.g. it equals 3 for RGB images).

The model passes the generated values and distributions to the *exploration engine*, which uses them to act on the environment, and to generate the values needed for computing the objective function used in A2C, which in turn reflects in the update of the ANN weights through backpropagation (section 2.2.3).

3.2 Simulator interface

In order to support generality and flexibility in the framework, its design devises from its first implementation the use of different simulators in the future. For this reason, the training process needs to rely on fixed functions and format for data exchange, regardless of the implemented application and the relative simulator chosen. For this reason, the framework features a *simulator interface*, that is, a model containing the list of functions that need to be implemented, so that they can be called in a piece of code regardless of their implementation. This allows the framework to potentially work with any application, and to track data in a personalized way through the learning process. This allows the user to check in real-time how the learning process is progressing, decide what data to save in order

to analyze the generated protocols and the training progression, and also when to save this data.

In this work, an *environment* is defined as a specific implementation of the *simulator interface*. In order to attach the chosen simulator, it is sufficient to implement a personalized version of the environment file (Appendix B) whose structure is inspired by the one proposed in [64]. Its main attributes are required to set the ANN model (as already specified in Section 3.1), and for parameters specific to the training process, such as the learning rate, number of epochs, maximum number of steps for each epoch, the possibility to restore the training from a previous state, and finally the name of the directory that will contain the output data. The main functions allowing the *simulator interface* to support the advancement of the training process follow. In order to define a specific implementation of the environment, it is necessary to maintain the external signature of these function and to implement their internal functioning according to the specific simulator attached. Comments in the code refer to the general functionalities to be implemented below them (the insertion points are indicate by `< insert code here >`).

- `reset` prepares the environment for the next episode starting from an initial state;

```
def reset(self):
    observation = None

    # environment reset to initial state
    # < insert code here >

    # creation of the observation of the initial state
    # < insert code here >

    return observation

```

- `render` returns an image showing the environment state;

```
def render(self):
    image = None

    #creation of the observation of the current state
    # < insert code here >

    return image

```

- `adapt_action` transforms the actions provided by the ANN in a format that can be used by the step method;

```
def adapt_actions(self, discrete_actions=None, continue_actions=
    None):

    actions = None

    #prepare the needed actions for the step function
    # < insert code here >

    return actions

```

- `step` acts on the environment with the provided actions and provides to the training process an observation of the environment, the reward value, and a boolean value that tells whether the environment has reached a terminal state or not;

```
def step(self,action):

    observation = None
    #create observation of the new state
    # < insert code here >

    reward = 1
    #define the value of the reward
    # < insert code here >

    done = False
    #decide if a final state has been reached
    # < insert code here >

    #execute the specified action on the simulator
    # < insert code here >

    #None is always returned since usage of the fourth returned
    value is still not implemented
    return observation, reward, done, None

```

These methods must interact with the class attributes and with the simulator to implement the environment. In this way, the *exploration engine* is able to call them, avoiding the need to recode it for each different simulator and application (figure 3.3).

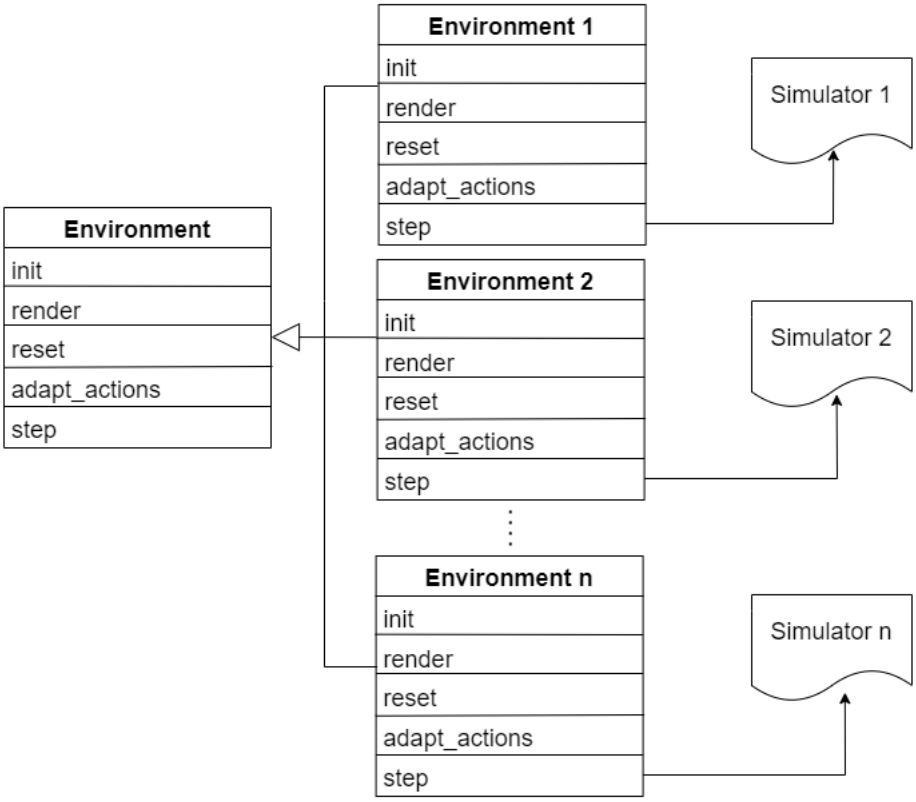

Figure 3.3: Environment implementations for different simulators.

Other methods in the environment allow to keep track of the training performance, to define the data that needs to be saved in a file, and to restore the training with that data:

- `save_performance` is called during the training to decide whether or not to collect performance indexes related to the environment into environment variables;

```
def save_performance(self, values):

    #save environment data needed as performance indexes into
    specific variables
    # < insert code here >

    #the checks on whether or not to save this data depend on the
     environment
    #values is a list that contains in order first epoch Q-value,
     elapsed epoch time, last epoch loss value, current epoch
    number
```

- `get_performance` is called during the training to save the performance indexes in a checkpoint file;

```
def get_performance(self):

    #return saved performance indexes
    # < insert code here >

    return self.performance_indexes
```

- `load_performance` is called to restore the performance indexes in the respective environment variables when restoring the training from a previous state through checkpoint files;

```
def load_performance(self, values):

    #restore the saved performance indexes
    #'values' has the same structure of the performance indexes
    returned from the get_performance function
    # < insert code here >
```

- `check_performance` called during the training to decide whether or not to save the collected performance indexes in a checkpoint file;

```
def check_performance(self, values):

    #add checks to decide whether to save data in a file (and
    return True) or not (and return False)
    #values has the same structure of the values parameter in
    save_performance function
    # < insert code here >

    return False
```

- `data_to_save` is called during the training to save data in a checkpoint file, which have been collected during the training inside the environment, and that can be analyzed successively after the training process;

```
def data_to_save(self):

    #return environment data to be saved in a checkpoint file
    # < insert code here >

    return self.data_to_save
```

- `load_data_to_save` is called to restore the collected data in the respective variables inside the environment, when restoring the training from a previous state through checkpoint files;

```
def load_data_to_save(self, data):

    #restore the data returned with the data_to_save functions in
     the respective variables
    # < insert code here >
```

3.3 Training phase

The heart of the exploration engine is the `train` class with its `train` function, which interacts with both the environment and the model, implementing the A2C algorithm. They can be found in the `train.py` file (see Appendix C):

```
def train(self, save_every=10, stating_epoch=0):
    ...
```

The class contains relevant attributes for the training, such as the learning rate `lr` and the discount rate `gamma`. Its functions are `get_infos` and `train`. The first returns and prints strings with useful information in real-time, while the second starts the training, allowing to specify for how many epochs to save the needed data (`save_every` parameter, default `10`) and from which epoch to start (starting_epoch parameter, default `0`) The `starting_epoch` parameter is useful if there is the need to restore the training from a previous state through the saved data.

The `train` function executes the following operations:

- **Initial check**: it checks the environment for the presence of all the necessary methods;
- **ANN optimizer creation**: it creates an optimizer for the ANN training;
- **ANN building**: it builds the ANN with the specific input size;
- **Checkpoint loading**: it loads checkpoint files (if a previous state is being restored);
- **Training loop**: it starts the training iterations from `starting_epoch + 1`, and continues until the maximum epoch number specified is reached;
- **Data collection start**: opening a `GradientTape` tensorflow section, so that the flow of data inside the model is tracked.

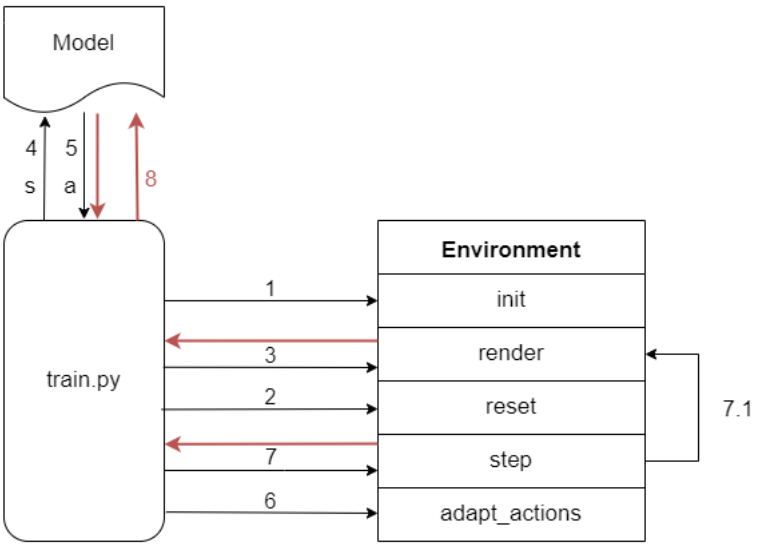

Figure 3.4: Training process. The black arrows represent the interactions, the red ones represent the data needed for backpropagation.

During every epoch, it is possible to show information in real time. Figure 3.4 provides an overview of the training process, which, during each epoch, performs the following steps:

- **Reset**: inside the `GradientTape` section the environment is reset (Figure 3.4.2), obtaining the initial state observation (Figure 3.4.3);
- **Observation**: the observation is given to the model (Figure 3.4.4), which will provide the first actions (Figure 3.4.5); the model provides also the state value, which will be saved, together with the logarithm of the probability of getting the obtained action, in order to compute the loss at the end of the epoch;
- **Iteration**: iteratively, the obtained actions are given to the environment (Figure 3.4.6, Figure 3.4.7), which will generate an observation (7,1) and will provide it to the training process;
- **Loss estimation**: once reached a terminal state, the Q-values and advantages are estimated, converted to tensors, and used together with the saved data to compute the objective functions defined by A2C, for both *Actor* and *Critic* losses;
- **Backpropagation**: the two losses are summed together and used by the *GradientTape* section as loss to compute the gradients, which are then back-propagated in the ANN through the optimizer (Figure 3.4.8);
- **Data collection**: before starting the next epoch, the function checks if there is the need to save data or not. This check depends both on condition specified

inside the environment and on the epoch number as specified through the `save_every` parameter, and saves the required data in files if needed.

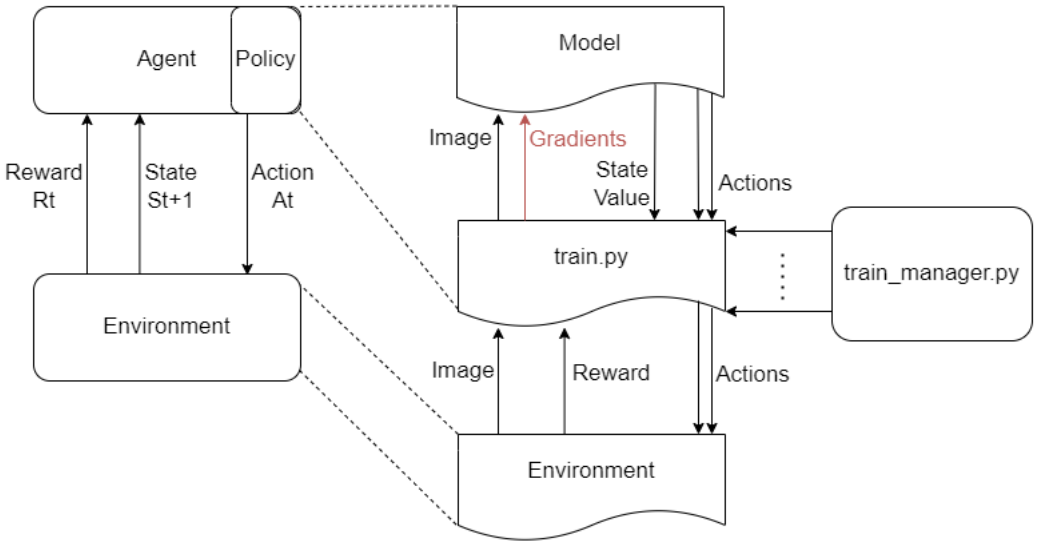

Figure 3.5: A2C implementation. The train manager executes more agents in parallel.

As an important improvement to performance, a `train_manager` (see Appendix D) allows to specify an environment for each combination of hyperparameters, so that the `train` function can run multiple times in parallel over them (Figure 3.5). The `train_manager` consists in a single function, `parallel_train`, which runs more training processes with different hyerparameters, executing the following operations:

- **Hyperparameters definition**: different lists of hyperparameters are defined;
- **Environments creation**: for each combination of values of the different lists, a different environment is created, specifying the names of the checkpoint files needed to restore the training from a previous state;
- **Subprocesses creation**: for each environment, a subprocess that executes the training is run through the `process` class from the Python module `multiprocessing`. A pipe (a communication channel) is passed to the subprocess;
- **Subprocesses storing**: each training subprocess is stored, so that it is possible to request real time information on the training through the pipe, and also to wait for all the subprocesses to end their training.

Chapter 4

Results

This Thesis work applies the proposed methodology on a specific use case, that is, the biofabrication of epithelial sheets. In particular, it relies on the `PalaCell2D` simulator and model provided in [65]. After describing the set-up of the *PalaCell2D environment* (Section 4.1), this Section illustrates the validation strategy and the experiments performed to demonstrate the functionality of the framework (Section 4.2).

4.1 Palacell set-up

The `PalaCell2D` framework simulates the proliferation of epithelial cells oriented to tissue morphogenesis [65], and acts as the environment for the framework. It relies on a vertex model, representing cells through their membrane, which is modeled as a set of vertices (figure 4.1). In this vertex model variant, cells don't share vertices, thus allowing the presence of interstitial gaps, and considering cells in contact when their distance is under a threshold.

The number of vertices of a cell changes dynamically so that the density of vertices remains uniform along the membrane. This allows mechanisms such as apoptosis (programmed death of the cell) or proliferation (division of a cell into two daughter cells). Both mechanical properties and impact of inner and outer fluids of the cells are simulated through internal and external forces. Apoptosis and proliferation are controlled through the cells' internal pressure, depending from the cell mass m, the cell area A, the pressure sensitivity η and the target density ρ_0:

$$p_{int} = \eta(\frac{m}{A} - \rho_0) \tag{4.1}$$

This pressure controls how much the cell can grow, as the mass changes with a rate that depends on the external pressure p_{ext} applied on the cell, the target area A_0,

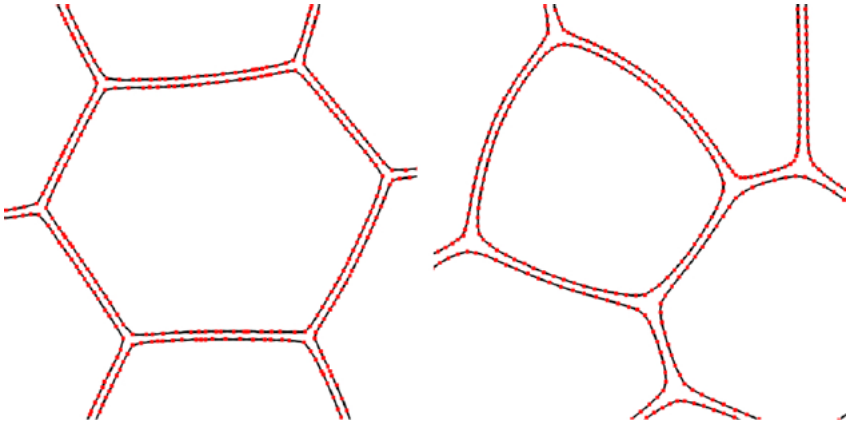

Figure 4.1: Representation of cells membranes through vertices (the red dots) in the PalaCell2D vertex model (taken from Conradin et al., 2021 [65]).

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 is stopped:

$$dm = \begin{cases} \nu A(1 - \frac{p_{ext}}{p_{max}}) & \text{if proliferating} \\ -\nu_{relax}(A - A_0)(1 - \frac{p_{ext}}{p_{max}})dt & \text{otherwise} \end{cases} \tag{4.2}$$

The external pressure p_{ext} depends on both the pressure resulting from the contact with other cells and the external force F_{ext}, which can be either local (interaction with a wall) or global (other external constraints). The proposed framework validation relies on the simulation of two well-characterized tissue behaviors: the proliferation of cells in an unbounded domain, and in a domain where it is controlled through an external force. In the second case, a probability is used to control the switch from the relax state to the proliferation state, through a parameter a_{prolif} that re-scales the probability:

$$p_{switch} = \begin{cases} a_{prolif}(1 - \frac{p_{ext}}{p_{max}}) & \text{if } p_{ext} <= p_{max} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

In the experiment proposed in [65], F_{ext} models a deformable capsule that acts on the cells vertices, depending on the distance from the capsule center and on the capsule radius.

The simulations consider iteratively the vertices positions for each cell and their exerted forces. They are controlled through a configuration file that contains parameters related to simulation, physical and numerical constraints. In addition, the configuration file allows to specify the name of an input file, used to restore the simulation from a previous state, and that of an output file that saves the

simulation's current state after a specified number of iterations. These interactions are summarized in Figure 4.2.

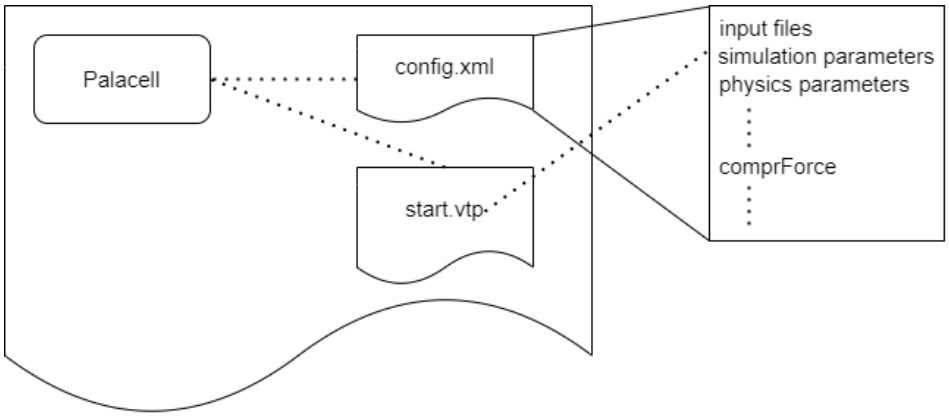

Figure 4.2: `PalaCell2D` interactions with configuration, input and output files.

The configuration file is an `.xml` file (see Appendix E for its detailed structure). Both input and output files are `.vtp` files from the Visualization Toolkit `VTK` [66], which models the cells as arrays of points. These files can be used to retrieve information about the simulation state, and they can be converted to images thanks to the `VTK` library.

For the experiments proposed in this Thesis work, the `PalaCell2D` simulator source code has been modified to consider another parameter, `initialPos`, which is the position of the first cell that is generated in the simulation space. This parameter is written in the configuration file and it is used by the simulator at the beginning of the simulation. Among all the parameters in the configuration file, the following ones are needed to control the simulation in the experiments:

- `initialVTK`, the name of the input `.vtp` file from which to restore the simulation (`string`), if missing the simulation will start from the beginning;
- `finalVTK`, the name of the output `.vtp` file that will be generated (`string`);
- `numIter`, the number of simulation iterations (`integer`);
- `stopAt`, the iteration number at which the simulation will be interrupted (`integer`);
- `initialPos`, the position of the initial cell (two value `x` and `y`, `integer`);
- `compressionAxis`, the axis on which the external force is applied (either `'x'` or `'y'`, `string`);
- `comprForce`, the amplitude of the applied external force (`float`).

This Thesis work included building an environment to interact with the `PalaCell2D` simulator (see Section 3). The environment implements the functions provided by the interface, and another function, `configure`, which is used by the other functions of the environment to create the configuration file needed by the simulator *ad hoc*:

```
def configure(self, file_path, num_iter, axis, compr_force,
    initial_path, initial_position, final_path):
    #file_path is the name of the final .xml file
    #num_iter will be set to the numIter parameter
    #axis will be set to the compressionAxis parameter
    #compr_force will be set to the comprForce parameter
    #initial_path will be set to the initialVTK parameter
    #initial_position will be set to the initialPos parameter
    #final_path will be set to the finalVTK parameter

```

The parameters for the environment that are common to all experiments are:

- `width` and `height`: the width and height in pixels of the observations that will be given to the model, default is `300`;
- `iters`: the number of iterations of each simulation step, default is `20`;
- `max_iterations`: the maximum number of iterations of the simulation, default is `4200`;
- `mode`: the specific experiment to run, default is `'prolif'`.

In order to read PalaCell2D output files, the following step was to build specific functions inside an helper file 'vtkInterface.py':

- `read_cell_num` reads the number of cells in the simulation space from the specified simulation output file;

```
def read_cell_num(filename):
    #filename is the name of the .vtp file, without the extension
```

- `create_png_from_vtk` reads the specified simulation output file and produces a `.png` image of the simulation space, which is centered on the cells population, not on the simulation space;

```
def create_png_from_vtk(filename):
    #filename is the name of the .vtp file, without the extension
```

- `create_decentered_pil_image` reads the specified simulation output file and returns an image of the simulation space, centered on the the simulation space;

```
def create_decentered_pil_image(filename):
    #filename is the name of the .vtp file, without the extension
```

- `add_target` adds a circular target, at the specified coordinates and with the specified radius, to the provided image;

```
def add_target(array, target_center, target_radius, width=400,
    height=400):
    #array is the numpy array containing the RGB pixel values
    #target_center is a couple of integers representing the
    target x,y coordinates
    #target_radius is the radius of the circular target
    #width, height are the dimensions of the array
```

- `count_target_points` reads the specified simulation output files and returns the number of vertices inside and outside the specified target.

```
def count_target_points(filename, target_center, target_radius):
    #filename is the name of the .vtp file, without the extension
    # target_center is a couple of integers representing the
    target x,y coordinates
    #target_radius is the radius of the circular target
```

4.2 Experiments

This Section illustrates the experimental designs to validate the proposed framework. The general validation strategy aims at showing how the proposed OvS methodology and the implemented framework are able to computationally generate biofabrication protocols with increased performance for the defined applications, supporting the effective optimization of the stimuli provided to the cells during simulations towards defined objectives. The validation strategy aims at showing that the model is actually learning from the environment, by improving the final number of cells during the advancement of the learning process. Validation of the proposed framework includes two objectives, corresponding to two sets of experiments: the optimization of the number of cells (section 4.2.1) and of their geometrical conformation (section 4.2.2), respectively, at the end of a simulation. The capability

of a biofabrication protocol to get to the objective is measured by the evaluation of objective-specific metrics, as detailed in the following paragraphs, that describe the experiments and and discuss the results obtained.

4.2.1 Optimization of the final number of cells

This experiment aims to show that the proposed methodology and framework support the effective optimization of stimuli provided to the cells during simulations, towards the maximization of the final number of cells obtained. In this case, performance evaluation considers, as metrics for validation, the *mean* and *variance* of the final number of cells across simulations. This metrics are evaluated at intervals corresponding to a fixed number of epochs in the learning process, starting from the first up to the last ones. This allows to understand whether the model is learning. In fact, it allows to pinpoint which actions are leading to an increment of the number of cells, which corresponds to an increase of the *mean*, and to verify whether the learning model is actually exploiting these action, which implies a decrease of the *variance* over time.

The simulated space in the *PalaCell2D environment* is a `400x400x1` grid. The framework operates the simulator to execute two main phases:

- **Initialization**: before the simulation starts, a single cell is located at the center of the simulated space;
- **Simulation**: at each simulation step, the parameters `compressionAxis` and `comprForce` are set to correspond to the values defined by the actions provided by the learning model;
- **Stop**: simulation continues until the number of learning epochs defined by the `numIter` parameter is complete.

At initialization, during the execution of `reset`, the parameter `initialPos` is set to `200 200` and the parameter `numIter` is set to `0` so that after the initial cell placing the simulation has not started yet. During simulation, the `numIter` parameter is set to `20`, that is, each learning process has `20` epochs. For each epoch in the learning process, a number of `20` simulation steps is performed, so that the total number of simulation steps in a simulation are `3400`.

The goal of the optimization process is to learn the optimal protocol to maximize the number of cells at the end of a simulation. In other words, the goal of the learning process is to find the values of the `compressionAxis` and `comprForce` parameters at every step that allow the simulator to reach the maximum number of cells at the simulation end. Thus, in the learning process, the reward value computed at each iteration is the increment of cells with respect to the previous iteration.

To keep track of the training results the following data is being saved:

- the highest number of cells obtained since the first epoch, which is the environment performance index;
- the number of cells obtained at the end of each epoch.

As detailed in Section 3, the proposed Thesis work organizes experiments based on combinations of relevant hyperparameters. Inside the `training_manager`, two lists of relevant hyperparameters for the learning process are defined:

- three values of learning rate (`lr`): `0.001`, `0.0001` and `0.00001`;
- two values of discount rate (`gamma`): `0.95` and `0.99`.

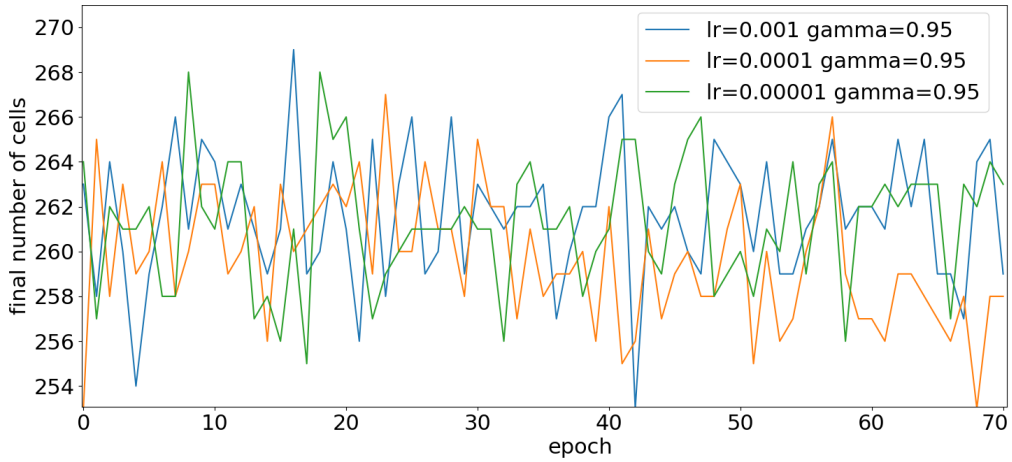

Figure 4.3: Final number of cells for each epoch.

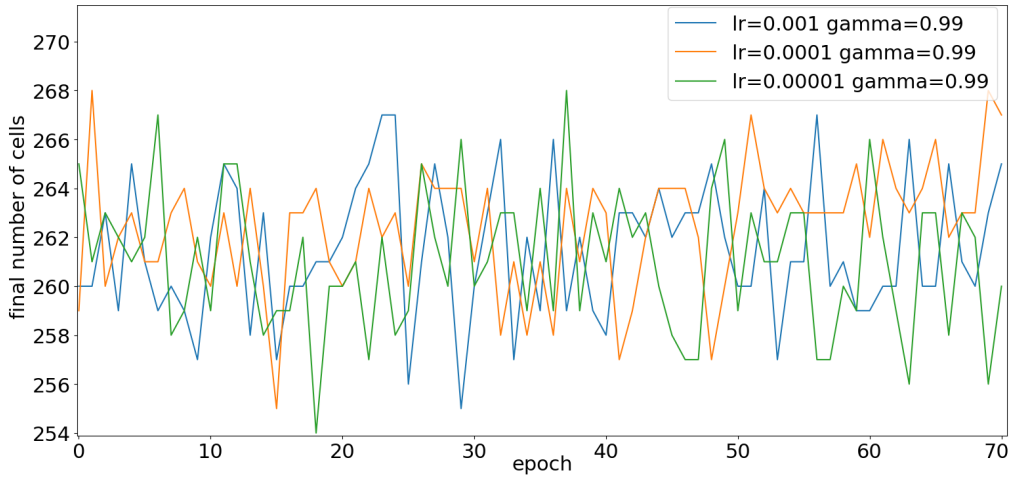

Figure 4.4: Final number of cells during simulations for each learning epoch.

In total, experiments considered six environments, created by combining the two lists, and six different training processes. Each different training process runs for `70` epochs, and data is saved every `5` epochs, in order to help the easy recapitulation of the process if a restart was needed.

Figures 4.3 and 4.4 show the trends of the obtained cell number at the end of each epoch for each combination of `lr` and `gamma` values. Comparing the last phase of the training processes, the worst performance is obtained with `lr=0.0001` and `gamma=0.95`, while the best performance is obtained with `lr=0.0001` and `gamma=0.99`.

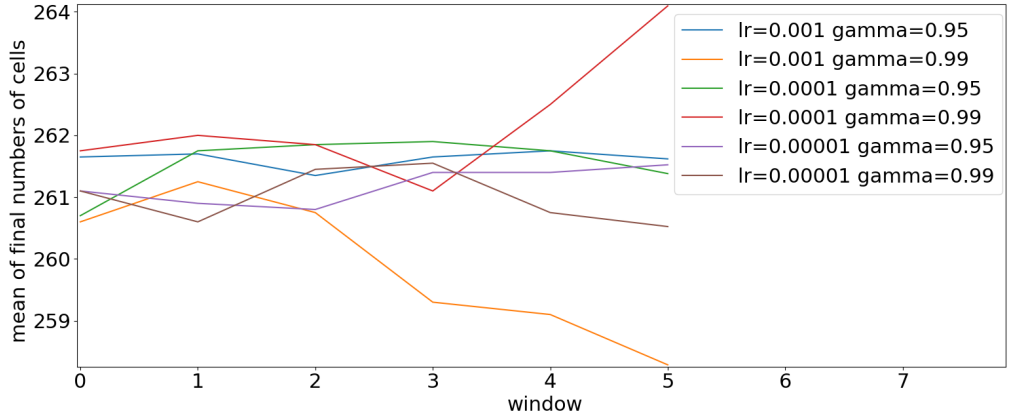

Figure 4.5: Mean of the final number of cells in simulations over six windows of 20 learning epochs.

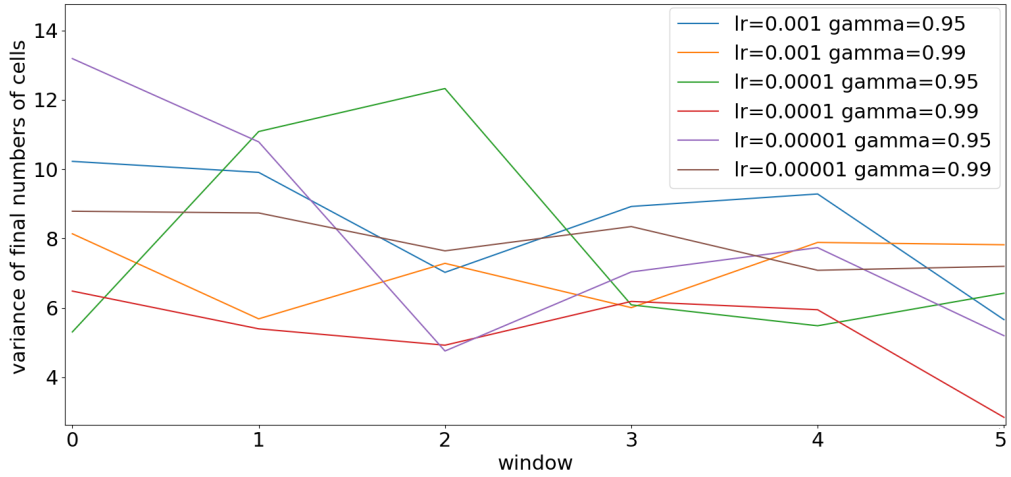

Figure 4.6: Variance of the final number of cells in simulations over six windows of 20 learning epochs.

To better evaluate the results, the *mean* and *variance* of the final number of cells are considered in six windows of 20 epochs, respectively at epochs: 0-20, 10-30,

20-40, 30-50, 40-60, 50-70. In this way it is possible to understand whether the training is still in an *exploration* phase (trying different solutions to learn more) or in the *exploitation* phase (using as much as possible the knowledge of the best obtained results). In the first case the *mean* increases in the successive windows, while the *variance* remains high, in the second case the *mean* remains almost stable between windows while the *variance* decreases. Figures 4.5 and 4.6 support the evaluation of this.

The best case is confirmed to be `lr=0.0001` and `gamma=0.99`, where the *mean* value increases more than in the other settings. In particular, it starts with a lower *variance* with respect to the other settings, meaning that it already starts in a more stable situation where it exploits its knowledge, and then furhter improves its performance during the last epochs, where the *mean* increases and the *variance* lowers. On the other hand, the other settings have a *mean* that grows a little during the last epochs, proving to indulge more in exploration during the first ones, passing to exploitation only at the end. On the contrary, the particular case of `lr=0.001` and `gamma=0.99` exhibits a decreasing *mean* and an increasing *variance* over time, meaning that it is not learning at all. These results show that a systematic exploration of the hyperparameters space is really useful to improve the performance, as different combinations lead to different behaviors of the training process.

		lr		
		0.001	0.0001	0.00001
gamma	**0.95**	**269**	267	267
	0.99	268	268	268

Table 4.1: Highest number of cells obtained for each training setting.

A final evaluation is given by the best obtained performance for every setting, which can be observed in table 4.1. The best performance is given by the setting `lr=0.001` and `gamma=0.95`. The protocol obtained through the learning process based on this setting devises the application of very low compression force stimuli on the cells along the entireity of the simulation, which is consistent with the inverse relationship between compression force at a cell and the probability it proliferates (see Appendix E). Figure 4.8 details the values of the parameters `comprForce` and `compressionAxis` at each simulation step within the optimal generated protocol. Is it possible to observe that the protocol keeps a constant stimulation of `comprForce=0.005` over the sole X axis (`compressionAxis='X'`). For assessing the impact of the optimization process over the generated protocols, 4.7 and 4.8 show the structure of generated protocols at epochs 0 and 16 respectively. It can be seen that, at the beginning of the learning process, the learning model tries random values as exploration, and at the end it uses a value that on average works better than the others.

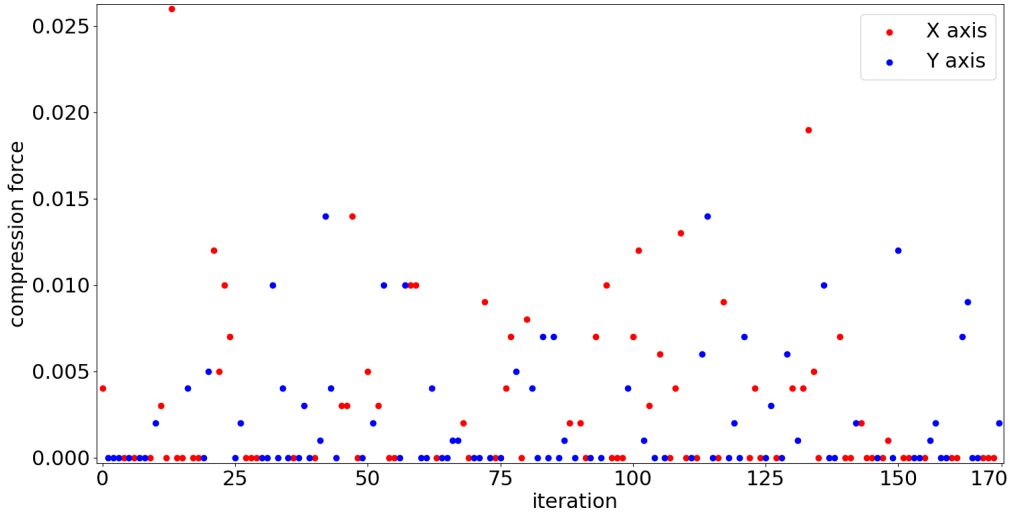

Figure 4.7: Values of parameters `comprForce` and `compressionAxis` at each iteration for epoch `0`.

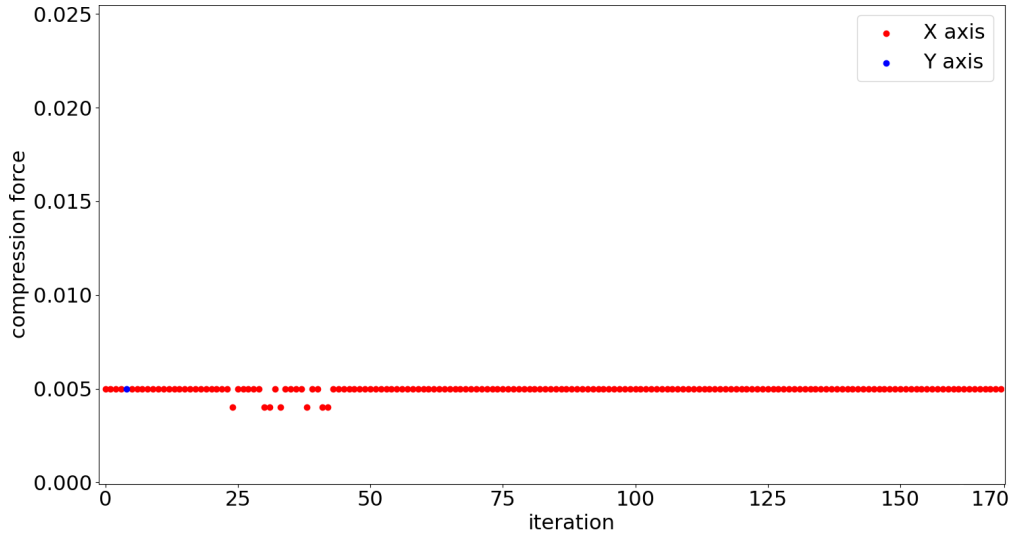

Figure 4.8: Values of parameters `comprForce` and `compressionAxis` at each iteration for epoch `16`.

Although the best overall performance is obtained through the setting `lr=0.001` and `gamma=0.95`, the goal is to obtain a model that can be reliable in generating an optimal protocol, so the case `lr=0.0001` and `gamma=0.99` is considered the most successful one as it is the more stable one in the terms of *mean* and *variance*. This is relevant for protocol generation, since it makes it the best setting for which the generation of the protocol can be easily interpreted even continuing the learning process. In fact, the training can't be carried on until convergence, and these

results serve as a confirmation of the framework potentiality to generate a stable result, supporting the assumption that if the training continued for more epochs this trend would be confirmed.

As a further trial, a new set of hyperparameters is investigated: the value of the `numIter` parameter. Starting from the best setting of the last experiment (`lr=0.0001` and `gamma=0.99`), the `numIter` parameter is set to the values `40` and `50` in order to understand how changing its value affects the simulations. In Figure 4.9 the trends of the obtained cells at the end of each epoch is shown. The best performance is obtained with the case `iters=50`, which is clearly predominant on the other two, showing that incrementing the number of simulator iterations executed during each environment step can improve the performance.

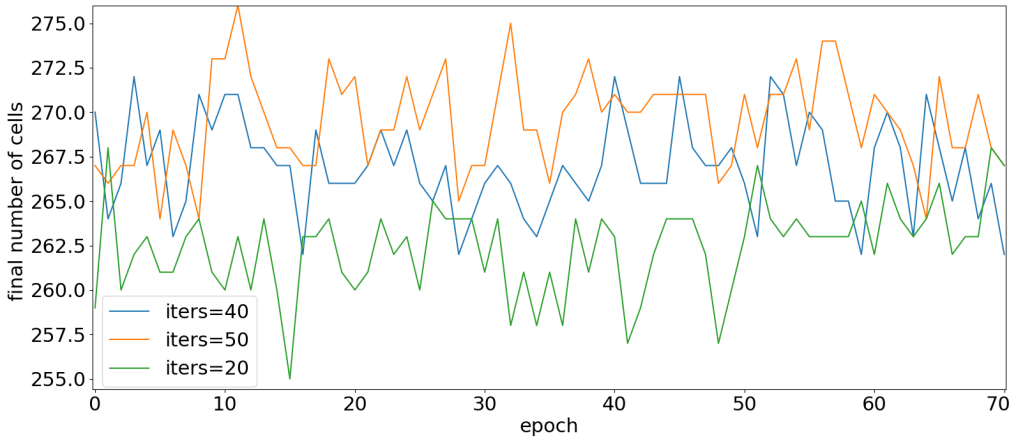

Figure 4.9: Final number of cells for each epoch.

For confirmation, in figures 4.10 and 4.11 the *mean* and *variance* are shown for six windows of `20` epochs, going from epoch `0` to `50` every `10` epochs. It can be seen that the setting with `iters=50` has the best *mean* in all windows, with a decreasing *variance*. In general, both new settings `iters=40` and `iters=50` learn well at the beginning, *exploiting* soon what they learn, and then they start *exploring* again at the end of the experiment, contrary to the original case `iters=20` which is slower, as it *explores* for almost the entire experiment, starting *exploiting* what it learns only at the end.

The best obtained performance for every setting can be observed in table 4.2. The best one is given by the setting `iters=50`, at epoch `11`. In figures 4.12 and 4.13 the values of the generated protocols for both epoch `0` and `11` are shown for comparison. Also here the learning model at the beginning is *exploring* with random values, while later it remains in an interval of values for which the simulation behaves better.

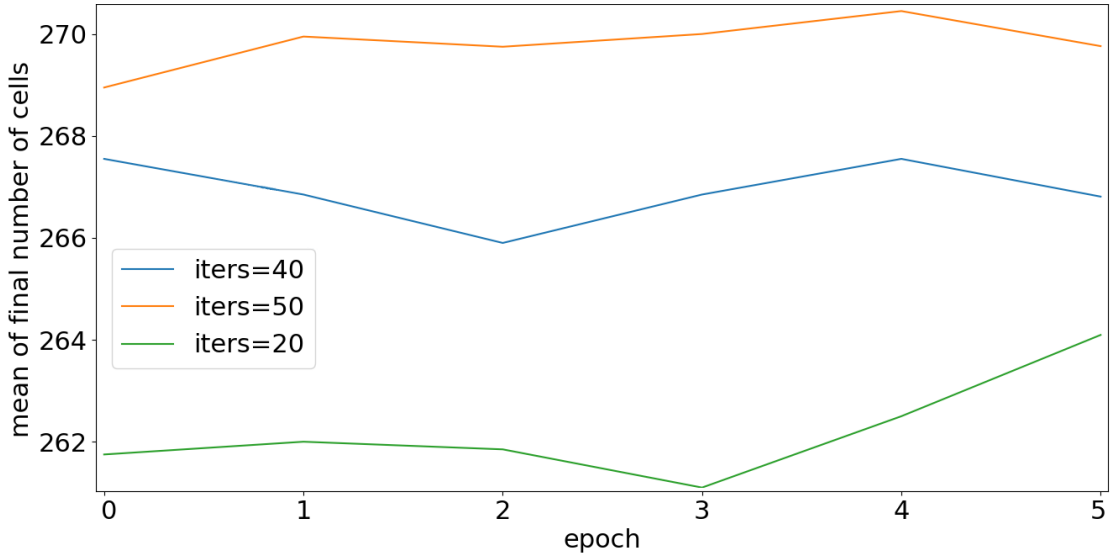

Figure 4.10: Mean value of the six windows.

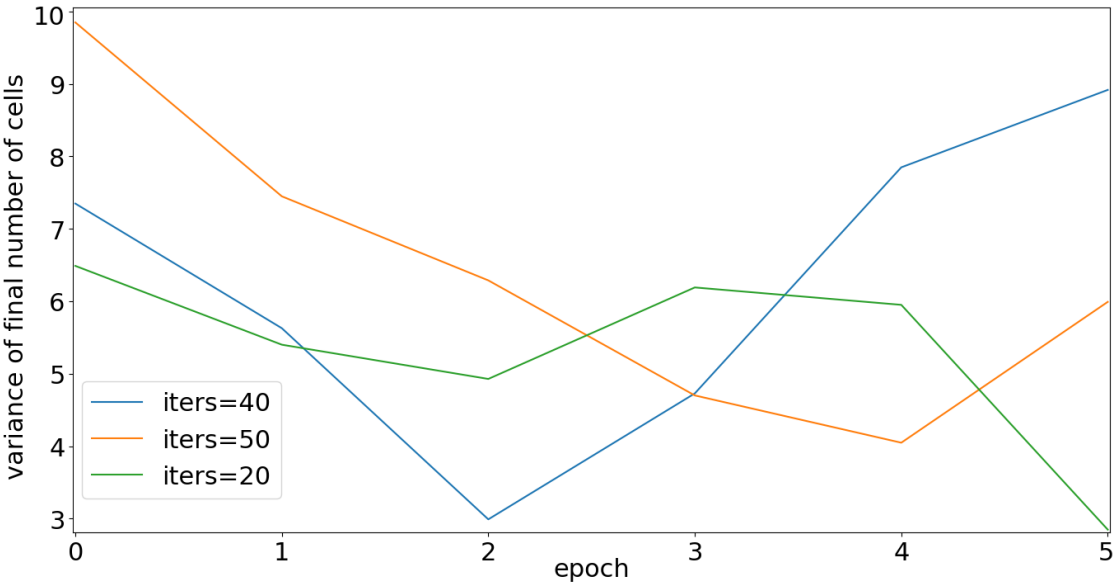

Figure 4.11: Variance of the six windows.

iters	20	40	50
final cells	268	272	276

Table 4.2: Highest number of cells obtained for each training setting.

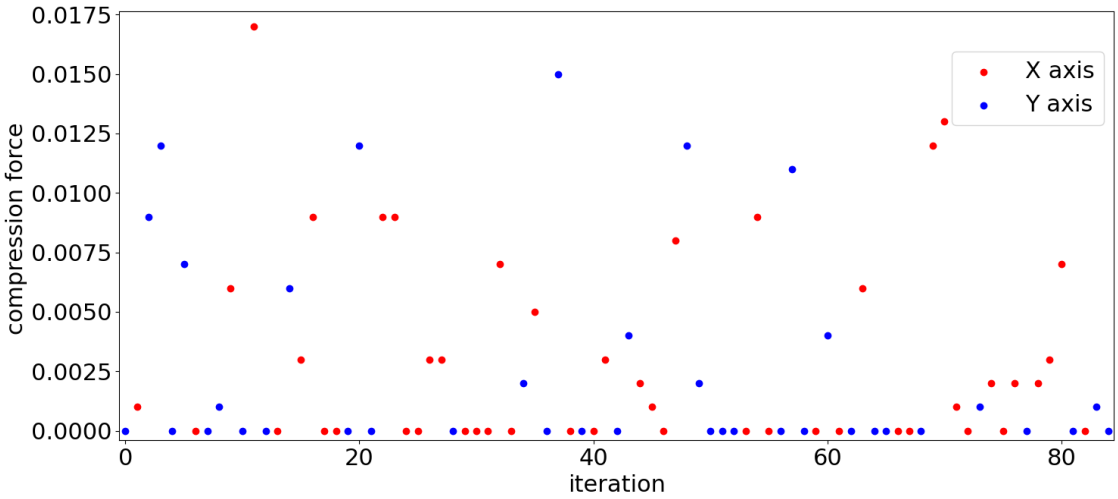

Figure 4.12: Values of parameter `comprForce` and `compressionAxis` at each iteration for epoch `0`.

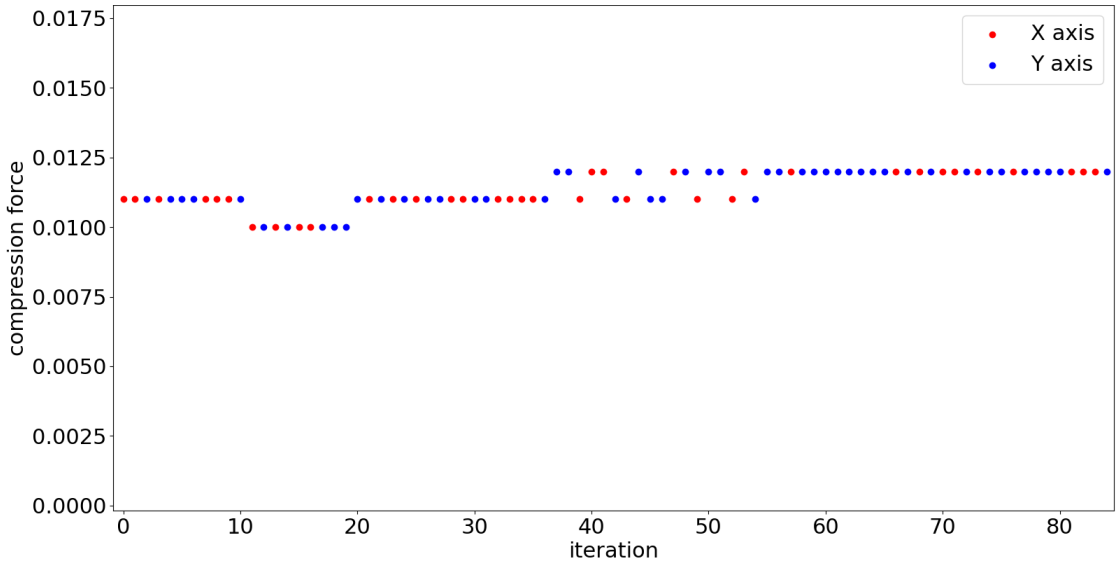

Figure 4.13: Values of parameter `comprForce` and `compressionAxis` at each iteration for epoch `11`.

4.2.2 Optimization of the starting position

The goal of the experiment is to show that the framework is able to learn the best position where to put the first cell at the beginning of the simulation, in order to maximize the number of cells within a defined circular target space, and minimize the number of cells outside. The validation strategy aims at showing that the starting position indicated by the learning model becomes closer to the target space over time. The validation metrics are the *mean* and the *variance* of the ratio of cells inside the target space with respect to the total number of cells at the end of the simulation, evaluated in intervals of a fixes number of epochs which are considered starting from the first epochs up to the last ones. In this way, it can be shown if the model is learning to approach the target space (increasing *mean* over time) and if it is *exploiting* the best learnt positions (decreasing *variance* over time). The considered target space is located at the position `x=200`, `y=250` with a radius of `80`. Moreover, after locating the starting cell, the same setting of the previous experiment is replicated, so to control the proliferation of cells as well, for filling the target space.

This requires two learning models to work together during the training process. Since `tensorflow` creates a single global `session` for training, in order to train two separate models at the same time these need to be run in two separated `subprocesses`. Thus, during the environment creation (the main environment) a second environment is created. This second environment has the same settings as the one from the previous experiment, except the maximum number of iterations, which is `numIter=2500`, a value that is sufficient to fill the target. A second training `subprocess` (inner training process) is run with it. In order to synchronize the two `subprocesses`, an helper training class is created, with the same structure of the one already presented in Section 3.3, but with a `pipe` in addition that is used to communicate with the first `subprocess`. During the training process the two environments performs the following steps:

- during its execution, `reset` creates an observation of the empty space including the indication of the target space;
- during its execution, `step` creates a simulation configuration file with the same values as in the previous experiment, except for the `initialPos` parameter, which in this case is set to the position indicated by the learning model as action;
- the `pipe` mediates the sending of the path of the configuration file to the inner training process;
- the inner training process receives the name of the configuration file through the `pipe` and uses it as starting configuration file, then runs the inner training process;

- when it reaches a final state, the inner training process sends the name of the final output file to the main environment through the `pipe`;
- the main environment reads the final output file to count cells inside and outside the target space, and to produce an observation.

The action space of the model is defined, for both starting position coordinates, between `90` and `310`. Starting from the goal of obtaining a precise target shape, the considered reward is the difference between the number of cells inside and outside the target space, divided by the number of obtained cells. To keep track of the results, the training saves the following data:

- the highest fraction of cells inside the target space with respect to total number of cells, which is the environment performance index;
- for each epoch, the fraction of cells inside the target space with respect to total number of cells.

Inside the training manager, two lists of hyperparameters are defined:

- two values of learning rate (lr): `0.001`, `0.0001`;
- two values of discount rate `gamma`: `0.95`, `0.99`.

In total, four environments are being created by combining the two `lists`, obtaining four different training processes. Each different training process runs for `70` epochs, and data is saved every `5` epochs.

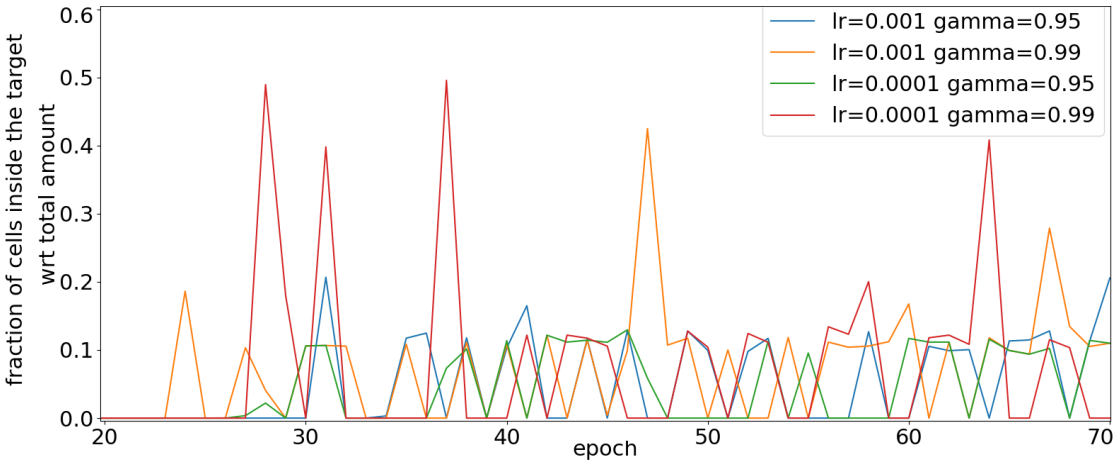

Figure 4.14: Fraction of cells inside the target area for each epoch.

The results trend can be observed in Figure 4.14. Until epoch `20` the target space is still not being reached, but afterwards all the processes show better results, meaning that in the first part of the learning the space is *explored* until a position

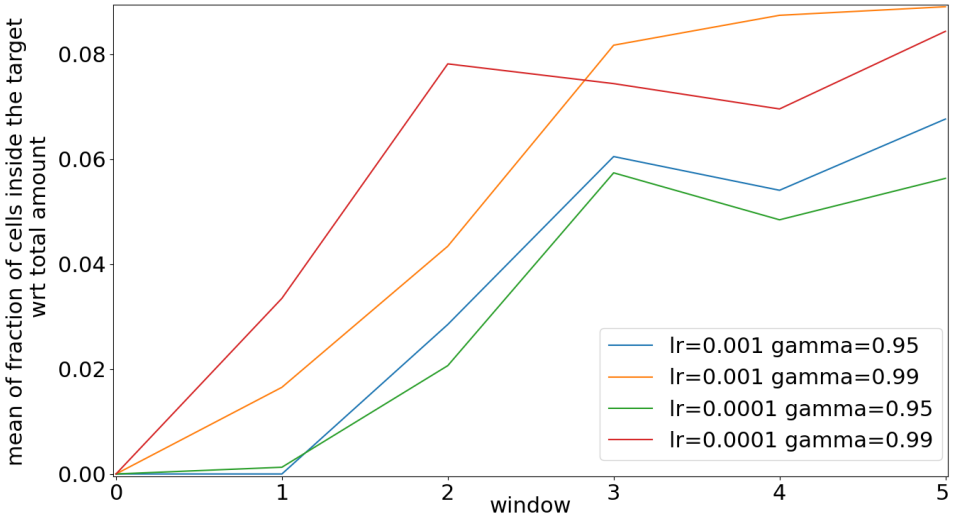

Figure 4.15: Mean value of the six windows.

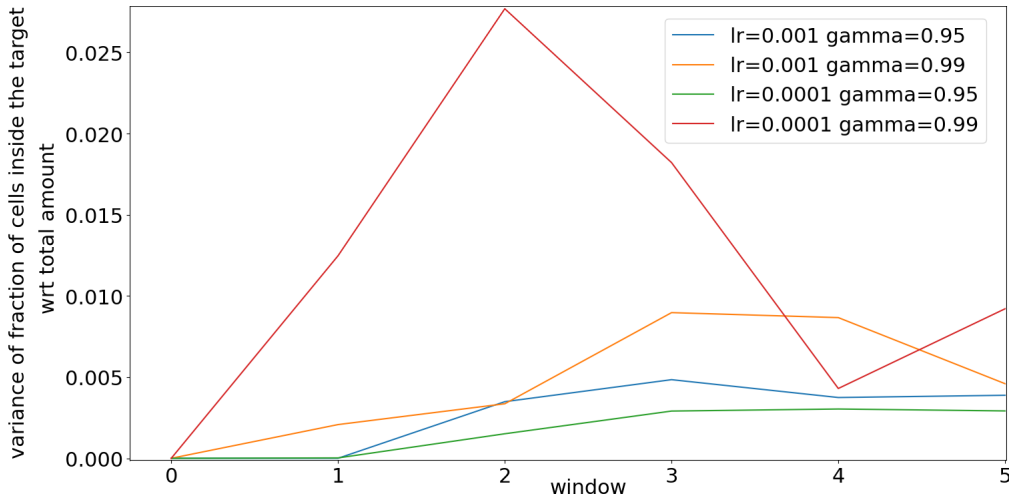

Figure 4.16: Variance of the six windows.

near the target space is found. In particular, later all the processes seem to remain around the target space, and at the last `15` epochs the target space is filled even more. Again the *mean* and *variance* of these values are considered in windows of `20` epochs, starting from epoch `0` up to epoch `50` every `10` epochs (so six windows in total). This evaluation can be observed in Figures 4.15 and 4.16. The best case is the setting `lr=0.001` and `gamma=0.99` as it reaches the best results with almost the same *variance* as the others. In general, all the settings become more stable over time, as the *mean* increases, but the *variance* is still growing meaning that the training process is still *exploring* the solution space.

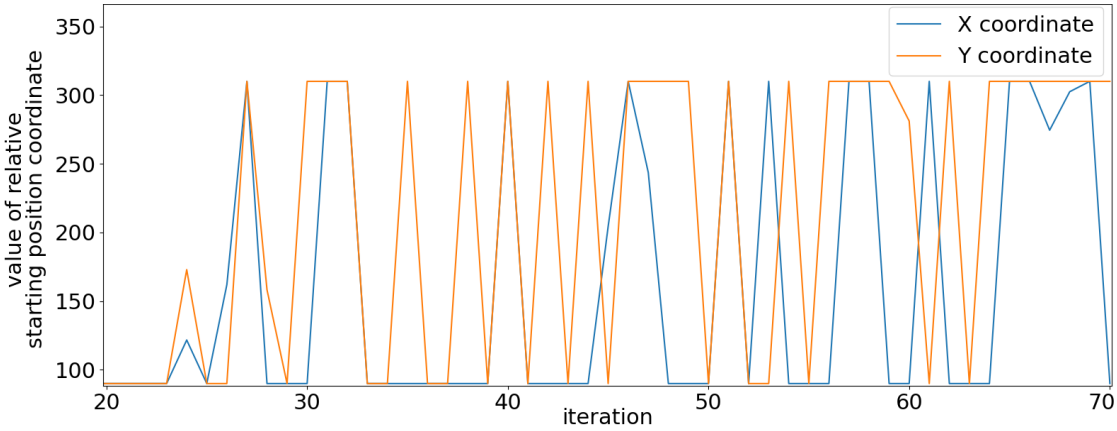

Figure 4.17: Values of both coordinates `X` and `Y` of the starting cell position for each epoch.

Figure 4.17 shows the values of both starting cell coordinates `x` and `y` for each epoch. Each couple of values corresponds to the generated protocol for the relative epoch. Until epoch `20` the network remains at `(90, 90)`, due to the fact that when the models gives values outside the action space the actual action provided to the environment is set to the lower bound for the specific axis, if the original action is lower, or to the upper bound if the original action is higher. After epoch `20`, the network starts to alternate between `90` and `310` on both coordinates for the same reason, meaning that it is starting to try different values inside the action space, but still it is trying to *explore* the space, so more training is still necessary.

The best obtained performance for each setting can be observed in table 4.3. The best performance is given by the setting `lr=0.0001` and `gamma=0.99`, where the obtained protocol is the application of the parameter `initialPos=262.56 224.76`, at epoch `37`.

		lr	
		0.001	**0.0001**
`gamma`	0.95	20.68	12.95
	0.99	42.48	**49.56**

Table 4.3: Highest fraction of cells inside the target space with respect to total number of cells for each training setting.

As in the previous experiment, the best overall case is considered the one that reaches more stability, which is the setting `lr=0.001` and `gamma=0.99`. In figures 4.18 and 4.19 the values of the generated protocols for both epoch `0` and `37` are shown for comparison: also in this experiment the model tries random values for *exploration*, but then converges to the application of a very low compression force stimuli.

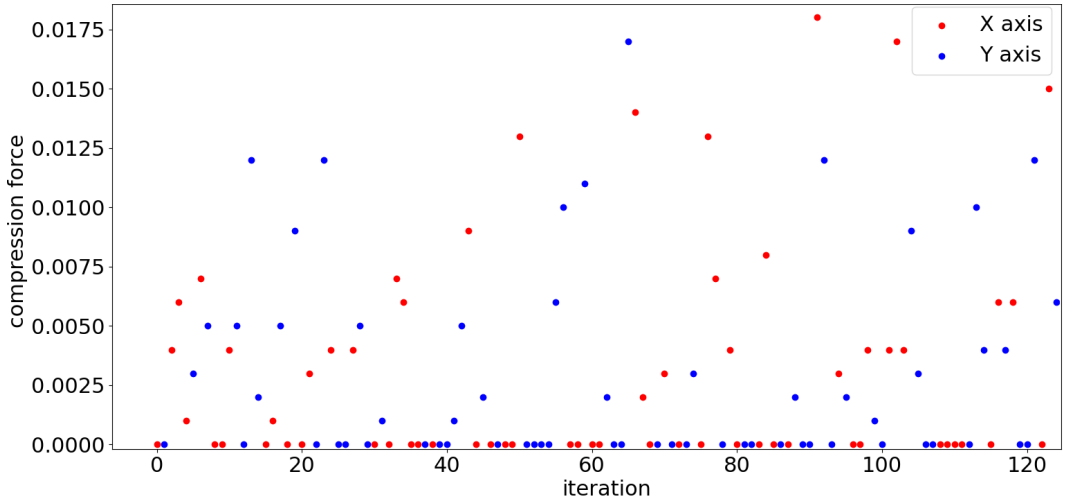

Figure 4.18: Values of parameter `comprForce` and `compressionAxis` at each iteration for epoch `0`.

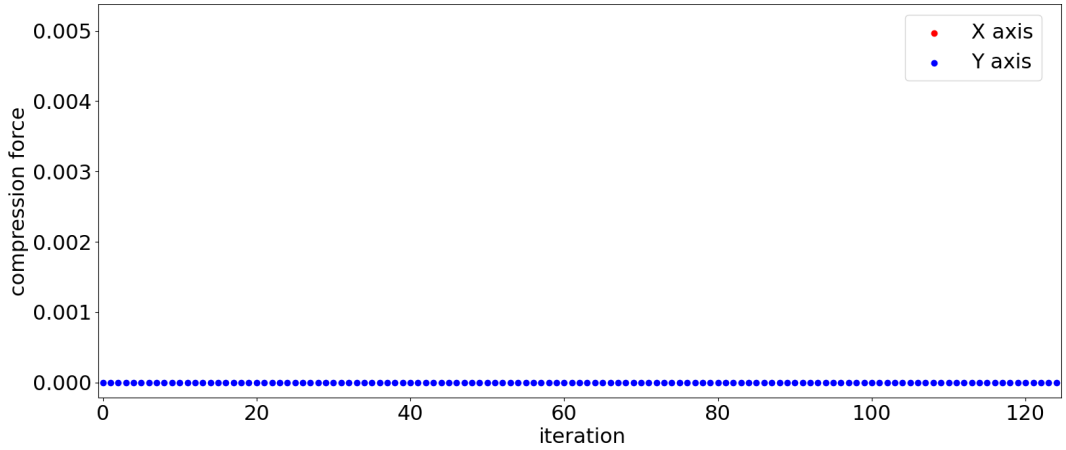

Figure 4.19: Values of parameter `comprForce` and `compressionAxis` at each iteration for epoch `37`.

Since this experiment is faster, this work extended it to `70` additional epochs, leveraging the capability of the training process to restart from the checkpoint files obtained after the last epoch. This yielded a training process of `140` epochs while requiring almost the same total experimental time as the first experiment, which lasted `70` epochs only. Figure 4.20 shows the results. All shown processes seem to be *exploring* more around the target, in particular for the settings `lr=0.001` and `gamma=0.99`, and `lr=0.0001` and `gamma=0.95` between epoch`70` and `110`. Then, all the four processes seem to be *exploring* more toward the target. To better evaluate

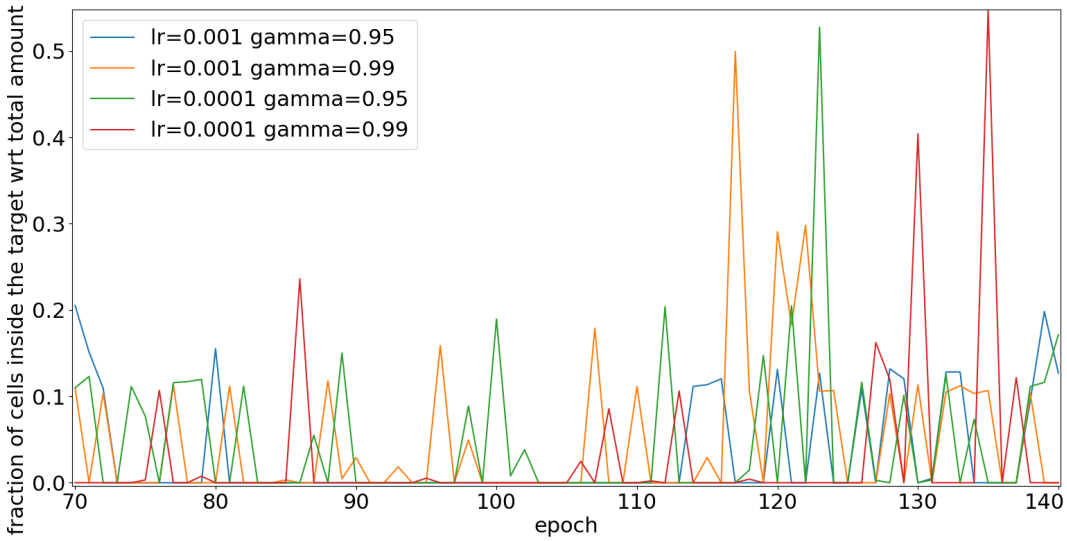

Figure 4.20: Fraction of cells inside the target space with respect to total number of cells for each epoch.

the trends, the *mean* and the *variance* in the windows are again analyzed (figures 4.21 and 4.22). All the processes show to behave similarly, with the case `lr=0.001` and `gamma=0.99` that is confirmed to be the best setting for the experiment. In these `70` epochs, the *mean* continues increasing with a small *variance*, than drop suddenly with an increasing `variance`, adding to the previous observation that the process is still *exploring* the solution space. This can be seen also at the end of the experiment at which the *mean* is starting again to grow while the *variance* is starting to decrease, resulting in more *exploitation*. Figure 4.23 shows the values for both starting position coordinates for each epoch of the best setting.

The best obtained performance for every setting can be observed in table 4.4. Among those, the best performance corresponds to the setting `lr=0.0001` and `gamma=0.99`, generating a protocol that consists in the application of the parameter `initialPos=193.52 310.0`, at epoch `135`. In figures 4.24 and 4.25 the values of the generated protocols for both epochs `0` and `135` are shown for comparison, confirming the same results of the best performance in the first `70` epochs.

		lr	
		0.001	**0.0001**
`gamma`	0.95	20.68	52.78
	0.99	49.96	**54.72**

Table 4.4: Highest fraction of cells inside the target space with respect to total number of cells for each training setting.

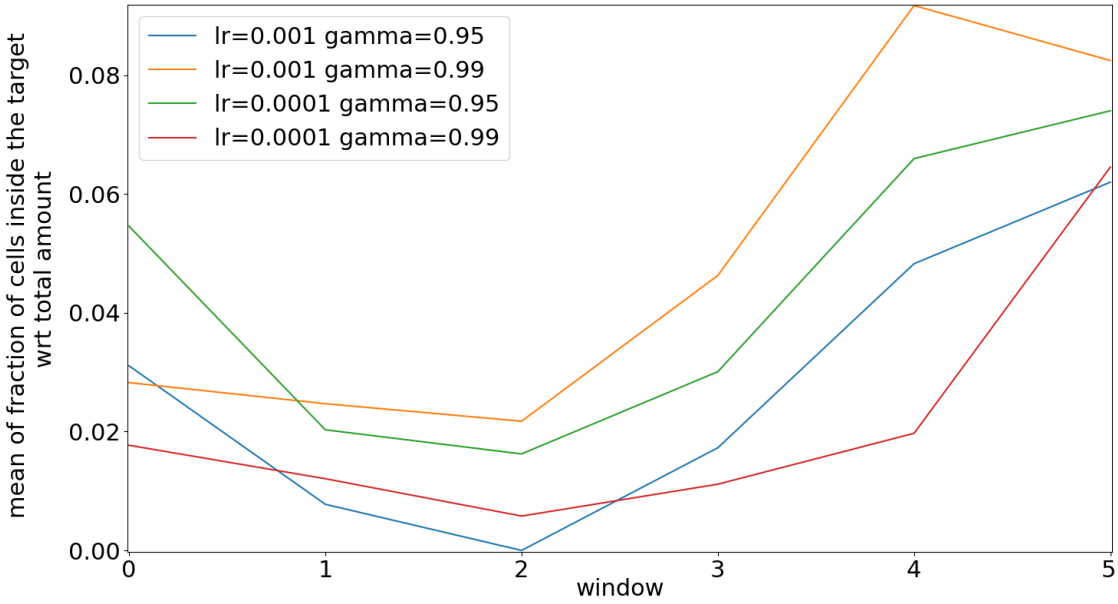

Figure 4.21: *Mean* value of the six windows for epochs `70-140`.

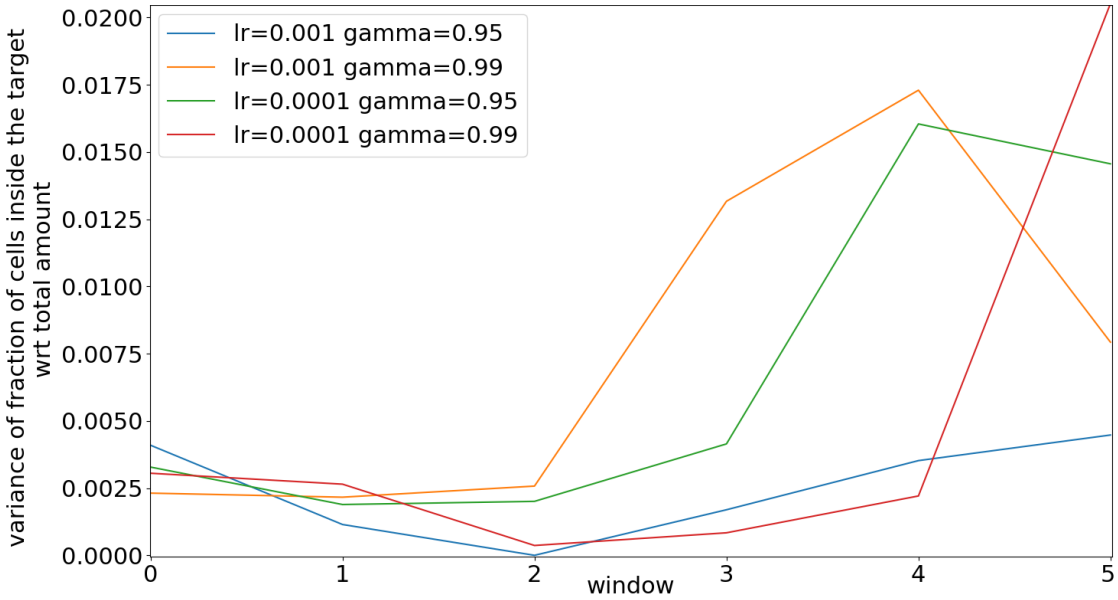

Figure 4.22: *Variance* of the six windows for epochs `70-140`.

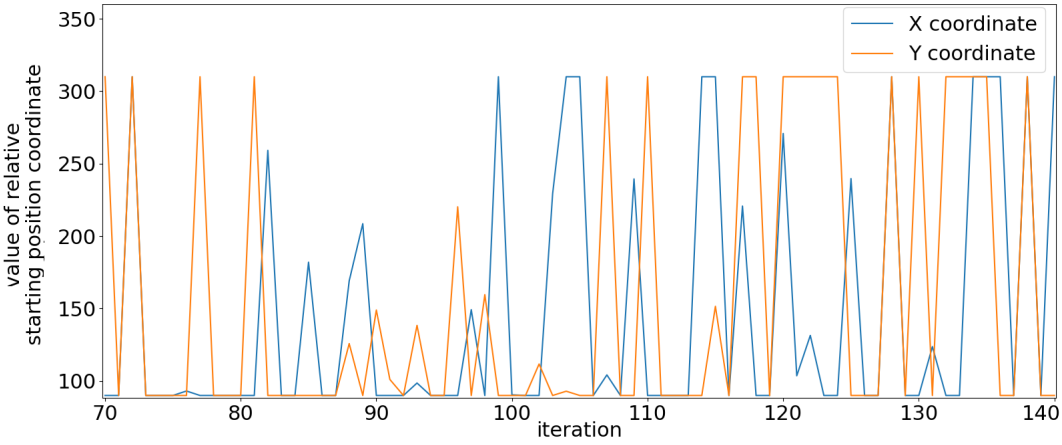

Figure 4.23: Values of both coordinates `x` and `y` of the starting cell position for epochs `70-140`.

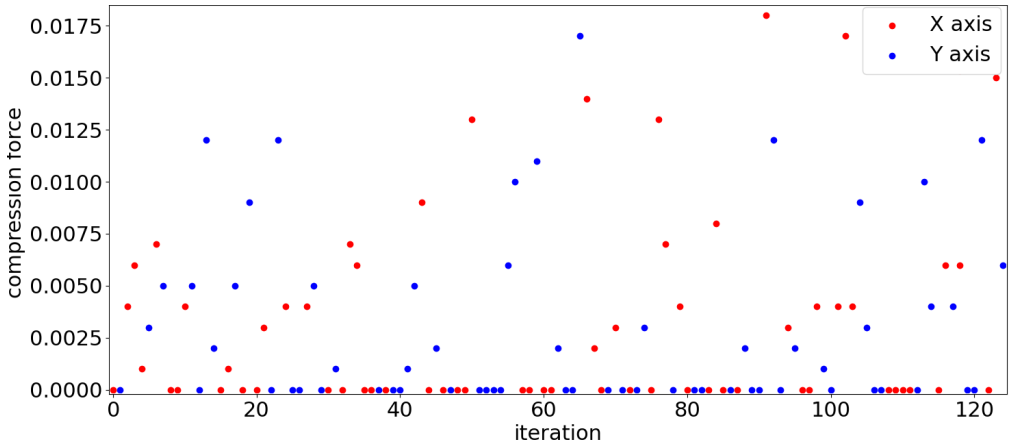

Figure 4.24: Values of parameter `comprForce` and `compressionAxis` at each iteration for epoch `0`.

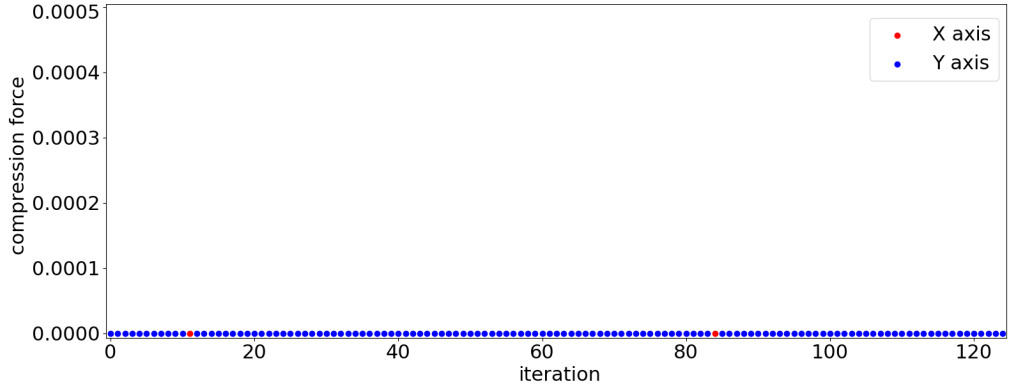

Figure 4.25: Values of parameter `comprForce` and `compressionAxis` at each iteration for epoch `135`.

4.2.3 Computational performance

The reproduced experiments runs relied on an AMD Ryzen 9 5950X 16-Core 2.2GHz processor with 64GB ram. The execution environment leveraged Singularity [67], a container platform that allows to run an environment with necessary packages already installed, without having to install them on the server. The container can support even OSes different from the one installed on the server. A container can be created through a 'definition file', in which must be specified the commands to execute pre-installations, or scripts at the container launch. Then, the container can be executed at any time on any server that supports singularity, through the platform itself. The container definition file can be found in Appendix F.

For the first experiment, a singularly ran training process takes between about `1800` and `2000` seconds to complete an epoch, while running more training processes in parallel takes between about `2000` and `2100` seconds to complete an epoch for each. The speeding up in both the worst case and best case scenarios is:

$$\frac{1800 * 6}{2100} <= sp <= \frac{2000 * 6}{2000} \rightarrow 5.14 <= sp <= 6$$

For the second experiment, a singularly ran training process takes between about `1500` and `1600` seconds to complete an epoch, while running more training processes in parallel takes between about `1680` and `1730` seconds to complete an epoch for each, obtaining a speeding up in both the worst case and best case scenarios of:

$$\frac{1500 * 4}{1730} <= sp <= \frac{1600 * 4}{1680} \rightarrow 3.47 <= sp <= 3.81$$

These results show that the parallelization capability of the single-agent variant of A2C proposed in this thesis project (see Section 3) allows to perform experiments in an efficient way, as respectively six and four processes are being finalized simultaneously within a comparable time span as a single one.

Chapter 5

Conclusions

This work presents a framework for learning how to generate optimal biofabrication protocols through Optimization via Simulation, which relies on a reinforcement learning approach for the optimization part, which is the Advantage Actor-Critic algorithm. The framework is made to be easily adaptable to any simulator in the future thanks to an interface, and to the specific use case implemented by the user. Moreover, the framework allows to run more parallel training processes in order to find the best hyperparameter settings within computational times comparable to those of a single experiment. To show the framework capabilities, the proliferation of epithelial cells oriented to tissue morphogenesis is proposed as use case, through a specific simulator. Two experiments have been proposed: the generation of protocols for maximizing the final number of obtained cells, and for obtaining a defined target shape with cells at the end of a simulation. The first one relies on the optimization of the number of obtained cells, the second one on the optimization of both proliferation and the starting cell position choice in order to fill a specific target space. The results show that the framework is able to explore the solution space defined by the user, which can be easily implemented and modified, and that the ability of running multiple parallel training processes is useful in speeding up the training by trying different hyperparameter settings at the same time, which is crucial for a learning algorithm. For each experimental setting, the framework provides an optimal biofabrication protocol, whose relevance to the real-world application is based on the expressiveness and prediction capabilities of the simulative model it leverages. Considering this, results show that the framework is capable to impact the structure of generated protocols through the learning process, implying the underlying learning model is learning how to optimize the process. Future works could improve the implemented optimization component, both on the learning algorithm and on the neural network model sides, aiming to obtain a reinforcement learning model that works better with images as data provided as input. In particular, the learning algorithm could be improved by

trying different variants of A2C and A3C and considering their advantages and drawbacks. On the other hand, the neural network model could be improved both in its heads' structure and in the chosen backbone, as different backbones could learn the visual information in different ways, and a new backbone can be easily be implemented as long as its output size will be the same of the current ReLU backbone. Finally, further research should target how to choose better reward values for simulators in the biofabrication context.

Appendix A

ANN model

model.py

```
'''
Credits: Alberto Castrignanò, s281689, Politecnico di Torino
'''
import tensorflow as tf
import tensorflow_probability as tfp

'''
Basic residual block.
'filters' is the number of filters applied in the convolutional layer.
'conv1x1' allow to add a 1x1 convolutional block instead of a simple skip connection
'''
class ResidualBlock(tf.keras.Model):
    def __init__(self, filters, kernel_size=3, strides=1, conv1x1=False):
        super().__init__()
        self.conv1 = tf.keras.layers.Conv2D(filters, padding="same", kernel_size=kernel_size, strides=strides, kernel_initializer = "glorot_uniform")
        self.conv2 = tf.keras.layers.Conv2D(filters, padding="same", kernel_size=kernel_size, kernel_initializer = "glorot_uniform")
        if conv1x1:
            self.conv1x1 = tf.keras.layers.Conv2D(filters, kernel_size=1, strides=strides, kernel_initializer = "glorot_uniform")
        else:
            self.conv1x1 = None
        self.bn1 = tf.keras.layers.BatchNormalization()
        self.bn2 = tf.keras.layers.BatchNormalization()

    def call(self, x):
```

```
25        out = self.conv1(x)
26        out = self.bn1(out)
27        out = tf.keras.activations.relu(out)
28        out = self.conv2(out)
29        out = self.bn2(out)
30        if self.conv1x1:
31          out = tf.keras.layers.Add()([out,self.conv1x1(x)])
32        else:
33          out = tf.keras.layers.Add()([out,x])
34        return tf.keras.activations.relu(out)
35
36  '''
37  The Resnet block: each is composed of an arbitrary number of residual
        blocks
38  "filters" is the number of filters of convolutional layers in each
      residual block
39  (in this implementation, you can't obtain a resnet block made of
      residual blocks with a different number of filters)
40  "blocks_number" is the number of residual blocks composing the resnet
        block.
41  '''
42  class ResnetBlock(tf.keras.layers.Layer):
43      def __init__(self, filters, blocks_number, downsample=True, **
      kwargs):
44        super(ResnetBlock, self).__init__(**kwargs)
45        self.residual_blocks = []
46        for i in range(blocks_number):
47          if i == 0 and downsample:
48            self.residual_blocks.append(ResidualBlock(filters, strides
      =2, conv1x1=True))
49          else:
50            self.residual_blocks.append(ResidualBlock(filters))
51
52      def call(self, x):
53        for l in self.residual_blocks.layers:
54          x = l(x)
55        return x
56
57  '''
58  ResNet18: accepts an arbitrary-sized image, outputs a 'latent'-sized
      fully connected layer with relu activations.
59  Traditionally, it should output a number of softmax activations
      corresponding to the number of binary classes, given a
      classification task.
60  Here, the last layer is a dense so that we can get a latent
      representation of the input image, and elaborate it further, thus
      using the ResNet18 as a backbone.
61  '''
62  class ResNet18(tf.keras.Model):
```

```
63      def __init__(self, latent=1000): #latent can be an hyperparameter
64        super(ResNet18, self).__init__()
65        self.block_a = tf.keras.Sequential([tf.keras.layers.Conv2D(32,
    kernel_size=7,strides=2,padding="same",kernel_initializer = "
    glorot_uniform"),
66                                            tf.keras.layers.
    BatchNormalization(),
67                                            tf.keras.layers.MaxPool2D(
    pool_size=3, strides=2, padding="same")])
68        self.block_b = ResnetBlock(32, 2, downsample=False)
69        self.block_c = ResnetBlock(64, 2)
70        self.block_d = ResnetBlock(128, 2)
71        self.block_e = ResnetBlock(256, 2)
72        self.dense = tf.keras.layers.Dense(latent,activation='relu',
    kernel_initializer = "glorot_uniform")
73
74      def call(self, x):
75        out = self.block_a(x)
76        out = self.block_b(out)
77        out = self.block_c(out)
78        out = self.block_d(out)
79        out = self.block_e(out)
80        out = tf.keras.layers.GlobalAveragePooling2D()(out)
81        out = self.dense(out)
82        return out
83
84  '''
85  The Actor part: it can be composed of multiple discrete or continue
        action layers. Each share a fully connected layer.
86  Discrete actions: represented with a fully connected layer with
        softmax activations, representing the probability of doing an
        action. Its output size is the number of actions that can be made.
87  Continue actions: represented with two fully connected layer, which
        output dimensions are the number of continuous actions that can be
        made.
88                    One layer represents the mean, the other the
        variance, that will be used to sample a value in the normal
        distribution.
89                    The normal distribution itself will be used to
        compute the loss for that action.
90                    This approach corresponds to using a single
        multivariate normal distribution with covariances=0 (diagonal
        matrix)
91  '''
92  class PolicyHead(tf.keras.Model):
93    def __init__(self, num_continue=0, num_discrete=0, range_continue=
        None, dim_discrete=None, hidden=256): #hidden is an hyperparameter
94      super().__init__()
```

```
95      self.dense1 = tf.keras.layers.Dense(hidden,activation="relu",
        kernel_initializer = "glorot_uniform")
96      if num_continue>0:
97        self.num_continue = num_continue
98        self.range_continue = range_continue
99        self.mu_dense = tf.keras.layers.Dense(num_continue, None,
        kernel_initializer = "glorot_uniform")
100       self.sigma_dense = tf.keras.layers.Dense(num_continue, None,
        kernel_initializer = "glorot_uniform")
101     else:
102       self.mu_dense = None
103       self.sigma = None
104     if num_discrete>0:
105       self.discrete_actions = []
106       for i in range(num_discrete):
107         self.discrete_actions.append(tf.keras.layers.Dense(
        dim_discrete[i], activation="softmax", kernel_initializer = "
        glorot_uniform"))
108     else:
109       self.discrete_actions = None
110
111   def call(self, x):
112     out = self.dense1(x)
113     if self.mu_dense:
114       mu = self.mu_dense(out)
115       sigma = self.sigma_dense(out)
116       sigma = tf.keras.activations.softplus(sigma)+1e-5
117       sigma = tf.clip_by_value(sigma, 1e-2, 1e3)
118       continue_actions = []
119       norm_functions = []
120       for i in range(self.num_continue):
121         norm = tfp.distributions.Normal(mu[0][i], sigma[0][i])
122         action = tf.squeeze(norm.sample(1), axis=0)
123         norm_functions.append(norm)
124         continue_actions.append(action)
125     else:
126       norm_functions = None
127       continue_actions = None
128     if self.discrete_actions:
129       discrete_actions = []
130       for dense in self.discrete_actions:
131         out = tf.clip_by_value(dense(out),1e-2,1e3)
132         discrete_actions.append(out)
133     else:
134       discrete_actions = None
135
136     return discrete_actions, continue_actions, norm_functions
137
138 '''
```

```
139 The Critic part: it is composed of two fully connected layers, the
    first with relu activations.
140 The second layer is taken as it is, in order to output a value
    representing the Q-value of the state.
141 '''
142 class ValueHead(tf.keras.Model):
143   def __init__(self, hidden=256): #hidden is an hyperparameter
144     super().__init__()
145     self.dense1 = tf.keras.layers.Dense(hidden,activation="relu",
    kernel_initializer = "glorot_uniform")
146     self.dense2 = tf.keras.layers.Dense(1,None, kernel_initializer =
    "glorot_uniform")
147
148   def call(self,x):
149     out = self.dense1(x)
150     out = self.dense2(out)
151     return out
152
153 '''
154 The ActorCrit network: backbone can be chosen (resnet, encoder, or a
    single fully connected layer).
155 '''
156 class ActorCritic(tf.keras.Model):
157   def __init__(self, latent=1000, backbone="resnet", num_continue=0,
    num_discrete=0, range_continue=None, dim_discrete=None,
    hidden_actor=256, hidden_critic=256):
158     super().__init__()
159     self.backbone = ResNet18(latent)
160     self.actor = PolicyHead(num_continue, num_discrete,
    range_continue, dim_discrete, hidden_actor)
161     self.critic = ValueHead(hidden_critic)
162
163   def call(self,x):
164     repr = self.backbone(x)
165     discrete_actions, continue_actions, norm_functions = self.actor(
    repr)
166     value = self.critic(repr)
167     return discrete_actions, continue_actions, norm_functions, value
```

```
tensorflow version 2.9.1
tensorflow_probability version 0.17.0
```

Appendix B

Environment Blueprint

EnvironmentBlueprint.py

```
'''
Credits: Alberto Castrignanò, s281689, Politecnico di Torino
'''
class EnvironmentName():
    '''
    -   width and height will be necessary to generate images for the neural network
    -   lr will be used for the neural network
    -   gamma is the gamma parameter in the a2c algorithm
    -   preload_ parameters must be set together to restore a previous condition, each with its filename
        (preload_observations = True is the only exception but for now is not implemented)
        N.B. x and y must be multiple of 200(check if this number is coherent with the one in train.py)
    -   output_dir will be the name where data mentioned above will be saved
    '''
    def __init__(self, width=300, height=300, lr=0.001, gamma=0.99, preload_model_weights=None, preload_losses=None, preload_observations=None, preload_performance=None, preload_data_to_save=None, output_dir="output_dir_name"):
        self.num_continue = 0 #change with number of needed continue
        self.num_discrete = 0 #change with number of needed discrete actions
        self.range_continue = None #put here a list of bounds for the continue actions
        self.dim_discrete = None #put here a list of sizes for the discrete actions
        self.epochs = 300 #put here number of epochs to run
```

```
22          self.iterations = 1001 #put here number of max iterations for
     a single epoch
23          self.width = width
24          self.height = height
25          self.channels = 3 #for RGB images
26          self.lr = lr
27          self.gamma = gamma
28          self.preload_model_weights = preload_model_weights
29          self.preload_observations = preload_observations
30          self.preload_losses = preload_losses
31          self.preload_performance = preload_performance
32          self.preload_data_to_save = preload_data_to_save
33          self.output_dir = output_dir
34
35      def reset(self):
36          observation = None #must be a numpy array that complies with
     the shape (1,self.width,self.height,self.channels)
37          return observation
38
39      def render(self):
40          image = None #must be a PIL.Image object
41          return image
42
43      def adapt_actions(self, discrete_actions=None, continue_actions=
     None):
44          #the training process will provide this function two tensors:
45          #discrete_actions will contain a list of size (num_discrete
     ,1)
46          #continue_actions will contain a list of size (num_continue
     ,1)
47          #here you can put together them in a way that will comply
     with how the step function accepts the actions
48          return None
49
50      def save_performance(self, values):
51          #use this function to save some performance indexes from
     values or from environment
52          #train.py will call this function at every epoch, so any
     check to decide if save or not it's up to who writes the function
     body
53          #values is a list that contains in order: first Q-value,
     epoch elapsed time, loss, epoch number
54          return
55      def load_performance(self, values):
56          #after loading from file the performance, pass its content to
      this function
57          #then load the saved performance indexes in the respective
     vars
58          return
```

```
59      def check_performance(self, values):
60          #use this function to decide whether to add data to save to
        file (return True) or not (return False)
61          #values has the same structure of the one in save_performance
         function
62          return False
63      def get_performance(self):
64          #use this function to return saved performance indexes
65          return None
66
67      def _get_info(self): #infos that can be printed during training,
        must return a string
68          return None
69
70      def data_to_save(self): #return data that can be saved in a numpy
         file
71          return None
72      def load_data_to_save(self,data): #loads previous saved data
73          return
74
75      def step(self, action):
76          #action will be structured in the way defined in
        adapt_actions
77          observation = None #must be a numpy array that complies with
        the shape (1,self.width,self.height,self.channels)
78          reward = 1 #depends on the environment
79          done = False #whether the environment has reached a final
        state or not
80          return observation, reward, done, None
81          #last thing returned can be useful infos. until now infos use
         has not been implemented
```

```
pygame version 2.1.2
```

Appendix C

Training process

train.py

```
'''
Credits: Alberto Castrignanò, s281689, Politecnico di Torino
'''

import numpy as np
import time
import tensorflow as tf
import os
import checks
from model import ActorCritic

class Train:
    def __init__(self, env, lr, gamma, id):
        self.env = env
        self.lr = lr
        self.gamma = gamma
        self.losses = []
        self.loss = 0
        self.epoch = 0
        self.id = id

    def get_infos(self):
        strings = [""]
        strings.append("train id: "+str(self.id))
        strings.append("losses: "+str(self.loss))
        strings.append("epoch: "+str(self.epoch))
        strings.append("env infos: "+self.env._get_info())
        strings.append("")
        for s in strings:
            print(s)
        return strings
```

```
32
33      def train(self, save_every=10, verbose=True, recv=None,
    starting_epoch=0):
34          lr = self.lr
35          gamma = self.gamma
36          tf.get_logger().setLevel('ERROR')
37          os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
38          num_continue = self.env.num_continue
39          num_discrete = self.env.num_discrete
40          range_continue = self.env.range_continue
41          dim_discrete = self.env.dim_discrete
42          model = ActorCritic(num_continue=num_continue,num_discrete=
    num_discrete,range_continue=range_continue,dim_discrete=
    dim_discrete)
43          self.model = model
44          #check for env integrity
45          checks.check_env(self.env)
46          '''setup the environment'''
47          num_continue = self.env.num_continue
48          num_discrete = self.env.num_discrete
49          self.env.save_performance([])
50          '''
51          setup training
52          '''
53          GAMMA = gamma
54          optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
55          epochs = self.env.epochs
56          iterations = self.env.iterations
57          output_dir = self.env.output_dir+"_"+str(lr)+"_"+str(gamma)
58          if not os.path.exists("out/"+output_dir):
59              os.makedirs("out/"+output_dir)
60
61          self.model.build((1,self.env.width,self.env.height,self.env.
    channels))
62          self.model.summary()
63          if self.env.preload_model_weights:
64              model.load_weights(self.env.preload_model_weights)
65          if self.env.preload_losses:
66              self.losses = np.load(self.env.preload_losses+".npy").
    tolist()
67              self.losses.append(0)
68          if self.env.preload_performance:
69              performance_indexes = np.load(self.env.
    preload_performance+".npy", allow_pickle=True)
70              self.env.load_performance(performance_indexes)
71          if self.env.preload_data_to_save:
72              self.env.load_data_to_save(np.load(self.env.
    preload_data_to_save+".npy", allow_pickle=True).item())
73
```

```
74          print("GO: ", self.id)
75          if starting_epoch>0:
76              starting_epoch = starting_epoch+1
77          for j in range (starting_epoch,epochs):
78              if recv!=None and recv.poll():
79                  msg = recv.recv()
80                  if msg=='info':
81                      self.get_infos()
82              elapsed_time = time.time()
83              self.epoch = j
84              with tf.GradientTape() as tape:
85                  done = False
86                  rewards = []
87                  values = []
88                  log_probs = []
89                  discrete_log_probs = []
90                  observation = self.env.reset()
91                  observation = observation/255
92                  observation = tf.convert_to_tensor(observation)
93                  for iter in range(iterations):
94                      if recv!=None and recv.poll():
95                          msg = recv.recv()
96                          if msg=='info':
97                              self.get_infos()
98                      '''
99                      obtain actions and generate log probs
100                     '''
101                     discrete_actions, continue_actions, normals,
    value = self.model(observation)
102                     values.append(value[0][0])
103
104                     #discrete actions
105                     d_acts = []
106                     if discrete_actions:
107                         for da in discrete_actions:
108                             probs = da[0].numpy().astype('float64')
109                             action = np.random.choice(len(da[0]),
    size=1, p=(probs/sum(probs))) #obtain a random action based on the
     probability given by each discrete action
110                             discrete_log_prob = tf.math.log(da[0][
    action[0]]) #log of probability of given action
111                             discrete_log_probs.append(
    discrete_log_prob)
112                             d_acts.append(action[0])
113                     #continue actions
114                     if continue_actions:
115                         temp_cont = []
116                         for (i,nd) in enumerate(normals):
```

```
117                             log_prob = nd.log_prob(continue_actions[i
    ]) #log of probability of given action
118                             log_probs.append(log_prob)
119                             temp_cont.append(tf.clip_by_value(
    continue_actions[i],self.env.range_continue[i][0],self.env.
    range_continue[i][1]))
120                         continue_actions = tf.convert_to_tensor(
    temp_cont)
121                     '''
122                     act on the environment
123                     '''
124                     observation, reward, done, info = self.env.step(
    self.env.adapt_actions(d_acts, continue_actions))
125                     rewards.append(reward)
126                     observation = observation/255
127                     observation = tf.convert_to_tensor(observation)
128                     if done:
129                         break
130                 '''
131                 compute loss and backpropagte
132                 '''
133                 #compute Q-values
134                 Qval = 0
135                 Qvals = np.zeros_like(values)
136                 for t in reversed(range(len(rewards))):
137                     Qval = rewards[t] + GAMMA * Qval
138                     Qvals[t] = Qval
139                 ##transform values, Qvals into keras tensors
140                 Qvals = tf.convert_to_tensor(Qvals)
141                 values = tf.convert_to_tensor(values)
142                 #compute advantage
143                 advantage = Qvals - values
144                 #compute actor loss
145                 if num_continue>0:
146                     log_probs = tf.convert_to_tensor(log_probs)
147                     actor_continue_loss = 0
148                     for i in range(num_continue):
149                         temp_log_probs = [-log_probs[j] for j in
    range(len(log_probs)) if (j+i)%num_continue==0]
150                         actor_continue_loss += tf.math.reduce_mean(
    temp_log_probs*advantage)
151                 if num_discrete>0:
152                     discrete_log_probs = tf.convert_to_tensor(
    discrete_log_probs)
153                     actor_discrete_loss = tf.math.reduce_mean([-
    discrete_log_probs[i]*advantage[int(i/num_discrete)] for i in
    range(len(discrete_log_probs))])
154                 #compute critic loss and sum up everything
```

```
155                 critic_loss = 0.5 * tf.math.reduce_mean(advantage**2)
    ##MEAN SQUARE ERROR
156                 ac_loss = critic_loss
157                 if num_continue>0:
158                     ac_loss += actor_continue_loss
159                 if num_discrete>0:
160                     ac_loss += actor_discrete_loss
161                 ac_loss = tf.convert_to_tensor(ac_loss)
162                 #compute gradients and backpropagate
163                 grads = tape.gradient(ac_loss, self.model.
    trainable_variables)
164                 optimizer.apply_gradients(zip(grads, self.model.
    trainable_variables))
165                 '''
166                 print epoch stats and save weights, scores,
    observations
167                 '''
168                 print("train id: ", self.id)
169                 print("epoch: ", j, ", loss: ", ac_loss.numpy(), " lr
    : ", lr, " gamma: ", gamma)
170                 print("Elapsed epoch time: ",str(time.time()-
    elapsed_time))
171                 self.losses.append(ac_loss)
172                 if self.env.check_performance([Qvals[0].numpy(),time.
    time()-elapsed_time,ac_loss,j]):
173                     self.model.save_weights("out/"+output_dir+"/
    model_at_epoch_"+str(j)+"(best).h5")
174                     np.save("out/"+output_dir+"/
    data_to_save_at_epoch_"+str(j)+"(best)",self.env.data_to_save())
175                 self.env.save_performance([Qvals[0].numpy(),time.time
    ()-elapsed_time,ac_loss,j])
176                 if verbose:
177                     print(len([i for i in grads if i==None]))
178             '''
179             save weights, scores, observations
180             '''
181             if j%save_every==0:
182                 self.model.save_weights("out/"+output_dir+"/
    model_at_epoch_"+str(j)+".h5")
183                 np.save("out/"+output_dir+"/data_to_save_at_epoch_"+
    str(j),self.env.data_to_save())
184                 np.save("out/"+output_dir+"/performance_at_epoch_"+
    str(j),self.env.get_performance())
185                 np.save("out/"+output_dir+"/losses_at_epoch_"+str(j),
    self.losses)
186         self.model.save_weights("out/"+output_dir+"/last_model.h5")
187         #np.save("out/"+output_dir+"/full_scores",self.scores)
188         np.save("out/"+output_dir+"/full_losses",self.losses)
```

```
189         np.save("out/"+output_dir+"/last_data_to_save",self.env.
    data_to_save())
190         np.save("out/"+output_dir+"/last_performance",self.env.
    get_performance())
```

`numpy version 1.20.1`

Appendix D

Train manager

train_manager.py

```
'''
Credits: Alberto Castrignanò, s281689, Politecnico di Torino
'''
import time
from train import Train
import itertools
from multiprocessing import Process
from multiprocessing import Pipe

lr_list = [0.001, 0.0001]
gamma_list = [0.99, 0.95]

def parallel_train():
    #set when loading previous state
    starting_epoch = 0
    envs = []
    trains = []
    combs = itertools.product(lr_list, gamma_list)
    for i, (lr, gamma) in enumerate(combs):
        #choose the environment!
        env = EnvironmentName()
        env.epochs = 101
        #set to load previous state
        #env.preload_model_weights = "out/"+env.output_dir+"/model_at_epoch_"+str(starting_epoch)+".h5"
        #env.preload_data_to_save = "out/"+env.output_dir+"/data_to_save_at_epoch_"+str(starting_epoch)
        #env.preload_losses = "out/"+env.output_dir+"/losses_at_epoch_"+str(starting_epoch)
        #env.preload_performance = "out/"+env.output_dir+"/performance_at_epoch_"+str(starting_epoch)
```

```
28          #or either:
29          #env.preload_model_weights = "out/"+env.output_dir+"/
    last_model.h5"
30          #env.preload_losses = "out/"+env.output_dir+"/full_losses"
31          #env.preload_data_to_save = "out/"+env.output_dir+"/
    last_data_to_save"
32          #env.preload_performance = "out/"+env.output_dir+"/
    last_performance"
33          envs.append(env)
34      processes = []
35      senders = []
36      for i, (lr, gamma) in enumerate(combs):
37          recv, send = Pipe()
38          train = Train(envs[i], lr, gamma, i)
39          proc = Process(target=train.train, args=[5, False, recv,
    starting_epoch])
40          proc.start()
41          trains.append(train)
42          processes.append(proc)
43          senders.append(send)
44      start_time = time.time()
45      while True:
46          print("Select a number between 0 and ",len(trains)-1," to get
     infos: ")
47          try:
48              ind = input()
49              if ind=="exit":
50                  for proc in processes:
51                      proc.terminate()
52                      proc.kill()
53                      exit()
54              elif ind=="time":
55                  print(str(time.time()-start_time))
56              else:
57                  senders[int(ind)].send('info')
58          except Exception as e:
59              print("insert a valid index!")
60              print(e)
```

Appendix E

PalaCell2D configuration file

compr_example.xml

```
<?xml version="1.0" ?>
<parameters>
   <geometry>
      <initialVTK>output/prolif_0.001_0.95-_final_cell.vtp</initialVTK> <!-- Name of initial vtk file-->
      <finalVTK>prolif_0.001_0.95-</finalVTK> <!-- Name of output vtk file-->
   </geometry>
   <simulation>
      <type>1</type>
      <exportStep>20</exportStep> <!-- Number of steps every which the data will be written in an output file -->
      <initStep>0</initStep> <!-- Number of initial step -->
      <verbose>0</verbose> <!-- Verbosity level of the simulation, 0 means no printing-->
      <!-- Whether to write relative informations in an output file (true) or not (false) -->
      <exportCells>true</exportCells>
      <exportForces>false</exportForces>
      <exportField>false</exportField>
      <exportSpecies>false</exportSpecies>
      <exportDBG>false</exportDBG>
      <exportCSV>false</exportCSV>
      <seed>40</seed> <!-- Seed of the noise applied on points, 0 is no noise, <0 is random seed-->
      <exit>iter</exit> <!-- Stop the simulation when reaching respectively: iter(numIter), time(numTime), cell(numCell)-->
      <numIter>20</numIter>
      <numTime>7200</numTime>
      <numCell>300</numCell>
      <startAt>0</startAt> <!-- Starting iteration -->
```

```
25      <stopAt>20</stopAt> <!— Final iteration —>
26      <initialPos>0 0</initialPos> <!— X,Y coordinates of initial
    cell —>
27    </simulation>
28    <physics>
29      <diffusivity>2.0</diffusivity> <!— Chemical diffusivity —>
30      <reactingCell>0</reactingCell> <!— Number of first cell —>
31      <reactionRate>0.001</reactionRate>
32      <dissipationRate>0.0001</dissipationRate>
33      <growthThreshold>0.025</growthThreshold>
34      <zeta>0.7</zeta>
35      <rho0>1.05</rho0> <!— Target density —>
36      <d0>0.5</d0>
37      <dmax>1.0</dmax>
38      <n0>123</n0> <!— Number of vertices for each cell—>
39      <numCells>1</numCells> <!— Number of different types of cells
    —>
40      <cell type="default">
41        <divisionThreshold>300.0</divisionThreshold>
42        <pressureSensitivity>2.5</pressureSensitivity>
43        <nu>0.0025</nu>
44        <nuRelax>0.01</nuRelax>
45        <A0>300.0</A0> <!— Target area of cells —>
46        <k4>0.01</k4> <!— Spring constant on cells —>
47        <probToProlif>0.001</probToProlif> <!— Parameter that
    controls probability of proliferation —>
48        <maxPressureLevel>0.05</maxPressureLevel>
49        <zetaCC>0.4</zetaCC>
50      </cell>
51      <numVertex>3</numVertex> <!— Number of different types of
    vertices —>
52      <edgeVertex>-1</edgeVertex>
53      <vertex type="default">
54        <k1>0.2</k1>
55        <k3>0.2</k3>
56      </vertex>
57      <vertex type="1">
58        <k1>0.001</k1>
59      </vertex>
60      <vertex type="2">
61        <k3>0.2</k3>
62      </vertex>
63      <extern>
64        <compressionAxis>X</compressionAxis><!— Axis on which
    external compression force is applied —>
65        <comprForce>0.0</comprForce> <!— External compression force
     amplitude —>
66        <kExtern>0.01</kExtern> <!— Constraint constant —>
```

```
67          <center>200 200</center> <!— Center of the simulated sphere
    —>
68          <rMin>100</rMin> <!— min radius of the simulated sphere —>
69        </extern>
70    </physics>
71    <numerics>
72        <dx>1.0</dx> <!— Lattice resolution in micometers —>
73        <dt>1.0</dt> <!— Time resolution in seconds —>
74        <domain>0 0 400. 400.</domain> <!— Domain size in micrometers
    (x0 y0 xmax ymax)—>
75    </numerics>
76 </parameters>
```

Appendix F

Singularity definition file

palacell.def

```
Bootstrap: docker
From: ubuntu:20.04
Stage: build

%files

%post
    apt update
    apt upgrade -y
    apt install git -y
    apt install python3-pip -y

    pip3 install numpy
    pip3 install gym
    pip3 install tensorflow
    pip3 install vtk
    pip3 install tensorflow-probability
    pip3 install pygame

    ln -fs /usr/share/zoneinfo/Europe/Rome /etc/localtime
    export DEBIAN_FRONTEND=noninteractive
    echo "export DEBIAN_FRONTEND=noninteractive" >>
    $SINGULARITY_ENVIRONMENT

    apt-get update -y
    apt-get install libopenmpi-dev -y

    apt-get install -y python3-opencv
    pip3 install opencv-python
```

Bibliography

[1] Groll J., Boland T., Blunk T., Burdick J. A., Cho D. W., et al. «Biofabrication: reappraising the definition of an evolving field». In: *Biofabrication* 8.1 (Jan. 2016), p. 013001. DOI: 10.1088/1758-5090/8/1/013001. URL: https://dx.doi.org/10.1088/1758-5090/8/1/013001 (cit. on pp. 1, 3).

[2] Moroni L., Boland T., Burdick J. A., et al. «Biofabrication: A Guide to Technology and Terminology». In: *Trends in Biotechnology* 36.4 (Nov. 2017), pp. 384–402. DOI: 10.1016/j.tibtech.2017.10.015. URL: https://doi.org/10.1016/j.tibtech.2017.10.015 (cit. on pp. 1, 3).

[3] Mackay S. B., Marshall K.., A Grant-Jacob J., et al. «The future of bone regeneration: integrating AI into tissue engineering». In: *Biomedical Physics & Engineering Express* 7.5 (July 2021), p. 052002. DOI: 10.1088/2057-1976/ac154f. URL: https://dx.doi.org/10.1088/2057-1976/ac154f (cit. on pp. 1–3, 6).

[4] Giannantoni L., Bardini R., and Di Carlo S. «A methodology for co-simulation-based optimization of biofabrication protocols». In: *bioRxiv* (2022). DOI: 10.1101/2022.01.28.478198. eprint: https://www.biorxiv.org/content/early/2022/04/28/2022.01.28.478198.full.pdf. URL: https://www.biorxiv.org/content/early/2022/04/28/2022.01.28.478198 (cit. on pp. 1, 2, 4).

[5] Pereira R., Freitas D., Tojeira A., Almeida H., Alves N., and Bártolo P. «Computer modelling and simulation of a bioreactor for tissue engineering». In: *International Journal of Computer Integrated Manufacturing* 27 (July 2013), pp. 1–14. DOI: 10.1080/0951192X.2013.812244 (cit. on p. 3).

[6] Yang X. and Koziel S. «Computational Optimization: An Overview». In: *Computational Optimization, Methods and Algorithms*. Ed. by Koziel S. and Yang X. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–11. ISBN: 978-3-642-20859-1. DOI: 10.1007/978-3-642-20859-1_1. URL: https://doi.org/10.1007/978-3-642-20859-1_1 (cit. on p. 4).

[7] Norvig P. and Russel S. J. *Artificial Intelligence: A Modern Approach*. Denver, CO: Prentice Hall, 1995 (cit. on p. 4).

[8] Kim H. «AI, big data, and robots for the evolution of biotechnology». In: *Genomics Inform* 17.4 (2019), e44–. DOI: 10.5808/GI.2019.17.4.e44. eprint: http://genominfo.org/journal/view.php?number=579. URL: http://genominfo.org/journal/view.php?number=579 (cit. on p. 4).

[9] Von Stosch M., Portela MC R., and Varsakelis C. «A roadmap to AI-driven in silico process development: bioprocessing 4.0 in practice». In: *Current Opinion in Chemical Engineering* 33 (2021), p. 100692. ISSN: 2211-3398. DOI: https://doi.org/10.1016/j.coche.2021.100692. URL: https://www.sciencedirect.com/science/article/pii/S2211339821000241 (cit. on p. 4).

[10] Grexa I., Diosdi A., Harmati M., et al. «SpheroidPicker for automated 3D cell culture manipulation using deep learning». In: *Scientific Reports* 11 (2021), p. 14183. DOI: 10.1038/s41598-021-94217-1. URL: https://doi.org/10.1038/s41598-021-94217-1 (cit. on p. 4).

[11] Anagnostidis V., Sherlock B., Metz J., Mair P., Hollfelder F., and Gielen F. «Deep learning guided image-based droplet sorting for on-demand selection and analysis of single cells and 3D cell cultures». In: *Lab Chip* 20 (5 2020), pp. 889–900. DOI: 10.1039/D0LC00055H. URL: http://dx.doi.org/10.1039/D0LC00055H (cit. on p. 4).

[12] Udugama I.A., Lopez P.C., Gargalo C.L., et al. «Digital Twin in biomanufacturing: challenges and opportunities towards its implementation». In: *Lab Chip* 1 (2020), pp. 257–274. DOI: 10.1039/D0LC00055H. URL: http://dx.doi.org/10.1039/D0LC00055H (cit. on p. 4).

[13] Soares do Amaral J. V., Barra Montevechi J. A., de Carvalho Miranda R., and Trigueiro de Sousa Junior W. «Metamodel-based simulation optimization: A systematic literature review». In: *Simulation Modelling Practice and Theory* 114 (2002), p. 102403. DOI: https://doi.org/10.1016/j.simpat.2021.102403. URL: https://www.sciencedirect.com/science/article/pii/S1569190X21001040 (cit. on pp. 5, 26).

[14] Bardini R. «A diversity-aware computational framework for systems biology». PhD thesis. Politecnico di Torino, 2019 (cit. on p. 5).

[15] Bartocci E. and Lió P. «Computational modeling, formal analysis, and tools for systems biology». In: *PLoS computational biology* 12.1 (2016), e1004591 (cit. on p. 5).

[16] Bardini R., Politano G., Benso A., and Di Carlo S. «Multi-level and hybrid modelling approaches for systems biology». In: *Computational and structural biotechnology journal* 15 (2017), pp. 396–402 (cit. on p. 5).

[17] Keating S. M., Waltemath D., König M., Zhang F., Dräger Andreas, Chaouiya C., et al. «SBML Level 3: an extensible format for the exchange and reuse of biological models». In: *Molecular systems biology* 16.8 (2020), e9110 (cit. on p. 5).

[18] Vieira L. S. and Laubenbacher R. C. «Computational models in systems biology: standards, dissemination, and best practices». In: *Current Opinion in Biotechnology* 75 (2022), p. 102702 (cit. on p. 5).

[19] Bardini R., Politano G., Benso A., and Di Carlo S. «Computational tools for applying multi-level models to synthetic biology». In: *Synthetic Biology*. Springer, Singapore, 2018, pp. 95–112 (cit. on p. 5).

[20] Heiner M., Donaldson R., and Gilbert D. «Petri nets for systems biology». In: *Symbolic Systems Biology: Theory and Methods. Jones and Bartlett Publishers, Inc., USA (in Press, 2010)* (2010) (cit. on p. 5).

[21] Bardini R., Benso A., Politano G., and Di Carlo S. «Nets-within-nets for modeling emergent patterns in ontogenetic processes». In: *Computational and Structural Biotechnology Journal* 19 (2021), pp. 5701–5721 (cit. on p. 5).

[22] Bardini R., Benso A., Di Carlo S., Politano G., and Savino A. «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 (cit. on p. 5).

[23] Bardini R., Di Carlo S., Politano G., and Benso A. «Modeling antibiotic resistance in the microbiota using multi-level Petri Nets». In: *BMC systems biology* 12.6 (2018), pp. 59–79 (cit. on p. 5).

[24] Bardini R., Politano G., Benso A., and Di Carlo S. «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 (cit. on p. 5).

[25] Muggianu F., Benso A., Bardini R., Hu E., Politano G., and Di Carlo S. «Modeling biological complexity using biology system description language (bisdl)». In: *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE. 2018, pp. 713–717 (cit. on p. 5).

[26] Benso A., Bardini R., Di Carlo S., Politano G., Muggianu F., and Hu E. «Modeling biological complexity using Biology System Description Language (BiSDL)». In: (2020) (cit. on p. 5).

[27] Rosenblatt F. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65 6 (1958), pp. 386–408. DOI: 10.1037/h0042519. URL: https://psycnet.apa.org/doiLanding?doi=10.1037%5C%2Fh0042519 (cit. on p. 6).

[28] Minsky M. and Papert S. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969 (cit. on p. 6).

[29] Bolin G. and Lacra P. *On the Properties of the Softmax Function with Application in Game Theory and Reinforcement Learning*. 2017. DOI: 10.48550/ARXIV.1704.00805. URL: https://arxiv.org/abs/1704.00805 (cit. on p. 6).

[30] Nair V. and Hinton E. G. «Rectified Linear Units Improve Restricted Boltzmann Machines». In: *International Conference on Machine Learning*. Vol. 27. June 2010, pp. 807–814 (cit. on p. 6).

[31] LeCun Y., Bengio Y., and Hinton G. «Deep learning». In: *Nature* 521 (May 2015), pp. 1476–4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539 (cit. on p. 7).

[32] Bouchlaghem Y., Akhiat Y., and Amjad S. «Feature Selection: A Review and Comparative Study». In: *E3S Web Conf.* 351 (2022), p. 01046. DOI: 10.1051/e3sconf/202235101046. URL: https://doi.org/10.1051/e3sconf/202235101046 (cit. on p. 7).

[33] Fukushima K. «Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position». In: *Biol. Cybernetics* 36 (Apr. 1980), pp. 193–202. DOI: 10.1007/BF00344251. URL: https://doi.org/10.1007/BF00344251 (cit. on p. 8).

[34] LeCun Y., Boser B., Denker J. S., Henderson D., Howard R. E., Hubbard W., and Jackel L. D. «Backpropagation Applied to Handwritten Zip Code Recognition». In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541 (cit. on p. 8).

[35] Krizhevsky A., Sutskever I., and Hinton G.E. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (cit. on p. 9).

[36] Simonyan K. and Zisserman A. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: 10.48550/ARXIV.1409.1556. URL: https://arxiv.org/abs/1409.1556 (cit. on p. 9).

[37] Szegedy C., Wei L., Yangqing J., Sermanet P., et al. «Going deeper with convolutions». In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594 (cit. on p. 9).

[38] *Imagenet Large Scale Visual Recognition Challenge (ILSVRC)*. https://image-net.org/challenges/LSVRC/ (cit. on p. 9).

[39] He K., Zhang X., Ren S., and Sun J. «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 (cit. on pp. 9, 27).

[40] Huber J. P. «Robust Estimation of a Location Parameter». In: *The Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101. DOI: 10.1214/aoms/1177703732. URL: https://doi.org/10.1214/aoms/1177703732 (cit. on p. 11).

[41] Janocha K. and Czarnecki W. M. *On Loss Functions for Deep Neural Networks in Classification*. 2017. DOI: 10.48550/ARXIV.1702.05659. URL: https://arxiv.org/abs/1702.05659 (cit. on p. 11).

[42] Jadon A., Patil A., and Jadon S. *A Comprehensive Survey of Regression Based Loss Functions for Time Series Forecasting*. 2022. DOI: 10.48550/ARXIV.2211.02989. URL: https://arxiv.org/abs/2211.02989 (cit. on p. 11).

[43] Cauchy A. L. *Méthode générale pour la résolution des systèmes d'équations simultanées*. Vol. 10. Ouvres complètes 1. Oct. 1847, pp. 399–402 (cit. on p. 11).

[44] Senior A., Heigold G., Ranzato M. A., and Yang K. «An empirical study of learning rates in deep neural networks for speech recognition». In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 6724–6728. DOI: 10.1109/ICASSP.2013.6638963 (cit. on pp. 11, 12).

[45] Lee J., Simchowitz M., I. Jordan M. I., and Recht B. «Gradient Descent Only Converges to Minimizers». In: *COLT*. 2016 (cit. on p. 12).

[46] Dietterich T. «Overfitting and Undercomputing in Machine Learning». In: *ACM Comput. Surv.* 27.3 (Sept. 1995), pp. 326–327. ISSN: 0360-0300. DOI: 10.1145/212094.212114. URL: https://doi.org/10.1145/212094.212114 (cit. on p. 12).

[47] Robbins H. and Monro S. «A Stochastic Approximation Method». In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586 (cit. on p. 12).

[48] Ke Z., Cheng H., Yang C., and Huang H. «Analyzing the Interplay Between Random Shuffling and Storage Devices for Efficient Machine Learning». In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 276–287. DOI: 10.1109/ISPASS51385.2021.00050 (cit. on p. 12).

[49] van Merrienboer B., Breuleux O., Bergeron A., and Lamblin P. «Automatic differentiation in ML: Where we are and where we should be going». In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/770f8e448d07586afbf77bb59f698587-Paper.pdf (cit. on p. 12).

[50] Wengert R. E. «A Simple Automatic Derivative Evaluation Program». In: *Commun. ACM* 7.8 (Aug. 1964), pp. 463–464. ISSN: 0001-0782. DOI: 10.1145/355586.364791. URL: https://doi.org/10.1145/355586.364791 (cit. on p. 13).

[51] Baydin A. G., Pearlmutter B. A., Radul A. A., and Siskind J. M. «Automatic Differentiation in Machine Learning: A Survey». In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 5595–5637. ISSN: 1532-4435 (cit. on p. 14).

[52] *Pytorch autograd documentation*. https://pytorch.org/docs/stable/autograd.html (cit. on p. 15).

[53] *Tensorflow GradientTape documentation*. https://www.tensorflow.org/api_docs/python/tf/GradientTape (cit. on p. 15).

[54] L. P. Kaelbling, M. L. Littman, and A. W. Moore. «Reinforcement Learning: A Survey». In: (1996). DOI: 10.48550/ARXIV.CS/9605103. URL: https://arxiv.org/abs/cs/9605103 (cit. on p. 15).

[55] van Otterlo M. and Wiering M. «Reinforcement Learning and Markov Decision Processes». In: *Reinforcement Learning: State-of-the-Art*. Ed. by Wiering M. and van Otterlo M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1. URL: https://doi.org/10.1007/978-3-642-27645-3_1 (cit. on pp. 16, 19).

[56] Chan K., Lenard C., and Mills T. «An Introduction to Markov Chains». In: *MAV 49th Annual Conference*. Dec. 2012. DOI: 10.13140/2.1.1833.8248 (cit. on p. 16).

[57] Sutton R. S., McAllester D., Singh S., and Mansour Y. «Policy Gradient Methods for Reinforcement Learning with Function Approximation». In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf (cit. on p. 20).

[58] Williams R. J. «Simple statistical gradient-following algorithms for connectionist reinforcement learning». In: *Machine Learning* 8 (1992), pp. 229–256. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696 (cit. on p. 21).

[59] Grondman I., Busoniu L., Lopes G. A. D., and Babuska R. «A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595 (cit. on p. 21).

[60] Konda V. and Tsitsiklis J. «Actor-Critic Algorithms». In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf (cit. on p. 21).

[61] Mnih V., Badia A. P., Mirza M., Graves A., Lillicrap T. P., et al. «Asynchronous Methods for Deep Reinforcement Learning». In: *ICML 2016* (2016). DOI: 10.48550/ARXIV.1602.01783. URL: https://arxiv.org/abs/1602.01783 (cit. on pp. 22, 23).

[62] *OpenAI Baselines: ACKTR & A2C*. URL: https://openai.com/blog/baselines-acktr-a2c/ (cit. on p. 23).

[63] *Tensorflow documentation v2.9*. https://www.tensorflow.org/versions/r2.9/api_docs/python/tf (cit. on p. 27).

[64] *OpenAI Gym documentation*. https://www.gymlibrary.dev/ (cit. on p. 29).

[65] Conradin R., Coreixas C., Latt J., and Chopard B. *PalaCell2D: A framework for detailed tissue morphogenesis*. 2021. DOI: 10.48550/ARXIV.2102.00248. URL: https://arxiv.org/abs/2102.00248 (cit. on pp. 36, 37).

[66] *VTK documentation*. https://vtk.org/documentation/ (cit. on p. 38).

[67] *Singularity documentation*. https://docs.sylabs.io/guides/3.5/user-guide/index.html (cit. on p. 57).