Cellular Automata Control with Deep Reinforcement Learning

Emanuel Becerra Soto

2020-07-23

Contents

List of Tables

List of Figures

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam non enim id arcu hendrerit pulvinar. Suspendisse vehicula nibh sed leo porttitor cursus. Cras vel quam interdum, suscipit tortor in, tempus ipsum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec dictum velit non sapien vestibulum efficitur finibus ac ipsum. Proin eget diam condimentum, aliquam orci vel, auctor sapien. Nulla ac magna suscipit, tincidunt mi a, gravida quam. Proin eros leo, dapibus id luctus at, interdum et nunc. Morbi dapibus dictum ex ut volutpat. Donec consectetur eget urna facilisis hendrerit. Ut sodales cursus magna. Vivamus ipsum ipsum, congue vitae maximus eu, pellentesque vel erat.

Curabitur ex orci, venenatis ut nulla id, varius dignissim orci. Donec nec blandit nisi. Vivamus ex nibh, accumsan imperdiet maximus at, viverra id libero. Quisque magna dolor, facilisis eget nisi ut, vehicula luctus turpis. Sed in eros in lectus mattis auctor. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Nam sit amet pellentesque lectus, blandit porta nisl. Duis vulputate faucibus erat, non dapibus enim congue non. Curabitur bibendum nibh sit amet congue congue. Mauris eleifend a justo rutrum sagittis.

Acknowledgments

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam non enim id arcu hendrerit pulvinar. Suspendisse vehicula nibh sed leo porttitor cursus. Cras vel quam interdum, suscipit tortor in, tempus ipsum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec dictum velit non sapien vestibulum efficitur finibus ac ipsum. Proin eget diam condimentum, aliquam orci vel, auctor sapien. Nulla ac magna suscipit, tincidunt mi a, gravida quam. Proin eros leo, dapibus id luctus at, interdum et nunc. Morbi dapibus dictum ex ut volutpat. Donec consectetur eget urna facilisis hendrerit. Ut sodales cursus magna. Vivamus ipsum ipsum, congue vitae maximus eu, pellentesque vel erat.

Curabitur ex orci, venenatis ut nulla id, varius dignissim orci. Donec nec blandit nisi. Vivamus ex nibh, accumsan imperdiet maximus at, viverra id libero. Quisque magna dolor, facilisis eget nisi ut, vehicula luctus turpis. Sed in eros in lectus mattis auctor. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Nam sit amet pellentesque lectus, blandit porta nisl. Duis vulputate faucibus erat, non dapibus enim congue non. Curabitur bibendum nibh sit amet congue congue. Mauris eleifend a justo rutrum sagittis.

Chapter 1

Introduction

1.1 Motivations

Since antiquity the idea of building a thinking machine has always been in the minds of philosophers, artists, sciencemen, kings and commoners alike, filling us with wonder, terror and contemplation. A human creation capable of human feats, would turn us, at least in an allegorical sense, into gods.

Talk about why automata are a good benchmark.

- Easy to program
- Model complex systems
- Are models of universal computation

1.2 Problem Statement

In this thesis, we consider the advantages of using Cellular Automata as underlying environments for RL tasks. To fully incorporate a CA into the RL framework a Markov Decision Process (MDP) should be defined. So we need to be able to find appropiate sets of *Actions* ($\mathcal{A}$), *States* ($\mathcal{S}$) and *Rewards* ($\mathcal{R}$), also specify an "Agent" that interacts with the CA and explicitly or implicitly define the transitions of the MDP $p(s', r|s, a) : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$.

Thus our first challange is engineering a RL task that succesfully integrates the Forest Fire CA with the semantics of an "Agent" or "Helicopter" flying over the forest trying to extinguish the wild fire.

The optimaztion problem that we want to solve is: Minimize the

To Do: Describe the RL task and state the optimization problem.

1.3 Objectives

1.3.1 Main Objectives

- Propose a novel environment for Reinforcement Learning tasks, based on Cellular Automata, that could be used as an alternative benchmark instead of Atari games.
- Characterize the environment by solving it by state of the art methods.

1.3.2 Specific Objectives

- Select a Cellular Automaton model for the environment, in this case the Forest Fire Cellular Automaton.
- Design a RL task incorporating the CA dynamics.
- Implementation of the RL environment, following the Open AI gym API.
- Apply DQN to solve the proposed task.

Chapter 2

Cellular Automata

2.1 Introduction

Cellular Automata (sg. Cellular Automaton) are computational and mathematical systems with two essential characteristics: a discrete structure and a local interaction between its parts, but its killer feature is that they are simple, yet capable of producing complex behaviour.

A quick *Google Scholar* search for the term "Cellular Automata", retrieves roughly 13,000 research articles (during 2019-2020 period). Despite not being as popular (by the same metric) as other topics, like "Cancer" (~128,000) or "Deep Learning" (~104,000), they are, arguably, still popular.

The topic of CA dates back to the 1950's spanning a history of 70 years. Since then the mathematical and practical properties of CA have been studied and its applications have been explored in different branches of science including physical and social.

Perhaps the reason behind this popularity is their simplicity. CA are composed of decentralized interactions of their individual parts (cells), these cells are usually arranged in a very regular grid and are updated following the same rules for all cells. The rules only take into account the vicinity of a given cell. From this design specifications one could naively anticipate that a very homogeneous state is reached after some iterations of the CA. But this is not always the case, surprisingly for some simple configurations and rules, complex behavior emerges. This behavior is rich enough to be used to model natural systems. The structures that arises from local interactions where not designed *a priori* and their nature is capricious as they could be oscillating, chaotic, ordered, random, transient, stable. Their scale is also far from the initial size of the cell neighborhoods.

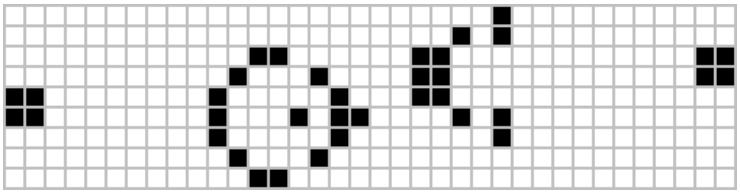

Figure 2.1: A glider gun, a famous pattern, from Conway's Cellular Automaton "Game of Life". Image taken from: *https://en.wikipedia.org/wiki/File:Game_of_life_glider_gun.svg*.

2.2 Main Characteristics

Cellular Automata are mathematical objects that are mainly characterized by (Ilachinski 2001):

1. A discrete lattice of cells: A n-dimensional arrangement of cells, usually 1-D, 2-D or 3-D.
2. Homogeneity: Cells are equivalent in the sense that they share an update function and a set of possible states.
3. Discrete states: Each cell is in one state from a finite set of possible states.
4. Local Interactions: Cell interactions are local, this is given by the update function being dependent on neighboring cells.
5. Discrete Dynamics: The system evolves in discrete time steps. At each step the update function is applied simultaneously (synchronously) to all cells.

2.3 Mathematical Definition

The following is adapted from the book Probabilistic Cellular Automata (Louis and Nardi 2018).

Cellular Automata (CA) are dynamical systems of interconnected finite-state automata (cells). The automata evolution is through discrete time steps and it is dictated by a function dependent on a neighborhood of interacting automata.

The main mathematical aspects of a CA are:

- The network G: A graph G.

  $$G = (V(G), E(G))$$

  The set of vertices $V(G)$ represents the location of the automata (cells). The set of edges $E(G)$ describes the spatial relations between automata.

- The alphabet S: Defines the states that each automata can take. In common CA settings S is a finite set. It is also called *local space* or *spin space*.
- The configuration space $S^{V(G)}$: This is the set of all possible states of the CA. A specific configuration is denoted as:

$$\sigma = \{\sigma_k \in V(G)\}$$

  σ_k is the configuration of the automaton at position k.
- The neighborhoods V_k:

$$V_k \subset V(G)$$

  The subset of nodes that can influence or interact with the automaton at $k \in V(G)$ (ordinarily it includes itself). A typical configuration for V_k is: $G = \mathbb{Z}^2$, and $V_k = \{k, k \pm e_1, k \pm e_2\}$, where (e_1, e_2) is the canonical basis of $\mathbb{Z}^2$, (north/south, east/west). This is known as the *von Neumman neighborhood*.
- The global update F:

$$F : S^{V(G)} \to S^{V(G)}$$

$$(F(\sigma))_k = f_k(\sigma_{V_k})$$

  The global update F is calculated by applying a local function f_k per automata at k. In the classical setting f_k is the same for all the automata.

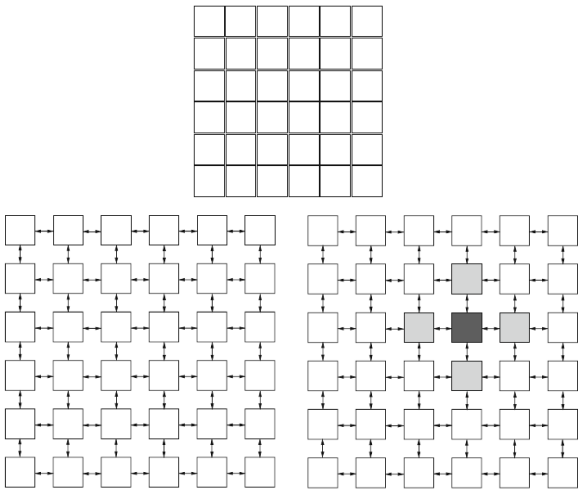

Figure 2.2: The underlying graph structure of a Cellular Automaton. A highlighted cell and its neighbors. Image adapted from (Hoekstra, Kroc, and Sloot 2010).

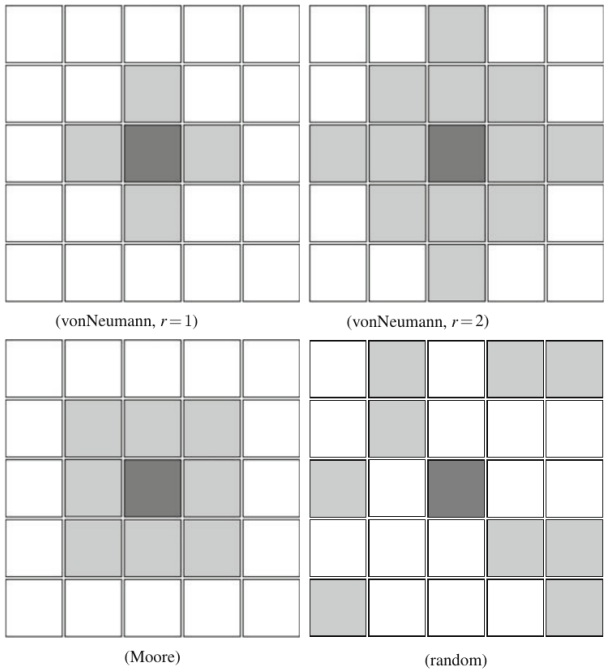

Figure 2.3: Different types of neighborhoods. Image adapted from (Hoekstra, Kroc, and Sloot 2010).

2.4 Classification

Wolfram from extensive simulations of 1-D CA grouped the general behavior of CA in four informally defined classes (Wolfram 2002). The classification is done by simulation from a variety of lattice initializations to broadly characterize the CA.

1. Class 1: A quick evolution towards a homogeneous global state is observed. All cells stop changing and all the randomness in the initial configuration disappears.
2. Class 2: The CA evolution leads to isolated patterns that could be periodic or stable.
3. Class 3: Pseudo-random or chaotic patterns emerge. Any stable structure is quickly destroyed by the surrounding noise.
4. Class 4: This is the most interesting type of behavior. Patterns that interact in a complex way emerge. These complex patterns are stable for long periods of time. Eventually the complex patterns can settle into a global state like Class 2 behavior but this can take a vast amount of time. Wolfram has conjectured that many Class 4 CA are capable of universal computation (Wolfram 2002).

This classification was inspired by the behavior observed in continuum dynamical systems. For example the homogeneous states of class 1 CA are analogous to fixed-point attracting states, or the repeating structures of class 2 CA are analogous to continuous limit cycles and class 3 chaotic patterns are analogous to strange attractors, while class 4 behavior does not have an obvious continuum analogy (Ilachinski 2001).

Other classification schemes can be found in the literature like: based in the structure of their attractors (Kůrka 1997) or by the structure of their "Garden of Eden" states (non-reachable global states) (Kari 1994).

2.5 Modeling of Complex Systems

Since the consolidation of the scientific method in the XVII century two methodologies emerged for generating and evaluating scientific knowledge. Them being the "experimental" and the "theoretical" paradigms. The experimental paradigm is concerned with observing, measuring and quantifying natural phenomena in order to test an hypothesis. Experimentation also can be made in a playful manner to collect and organize data. The theoretical paradigm seeks logical and mathematical explanations of natural phenomena. Both paradigms are complementary in the sense that predictions can be made by the theoretical paradigm and be tested using the experimental one, then if the experimental

findings support the predictions the theory is kept otherwise is rejected. In other words theory can be supported or falsified through experimentation.

A third scientific paradigm recently appeared, namely the "computational" paradigm. In this approach the study of nature is done through computer simulations. The computational paradigm works in partnership with the "experimental" and "theoretical" ones, so where observed phenomena are not easily tractable by analytical descriptions or direct experimentation is not allowed, computational simulation still permits further inquiry. Additionally when possible the outputs from the computations can be validated against experimental data and predictions from the theory, thus establishing helpful feedback between paradigms. The invention of the digital computer enabled the numerical solution of analytical models by means of discretization of quantities. Finally the computational paradigm can sometimes be used as a shortcut to the theoretical or experimental paradigms, for example if the problem at hand is well described, data obtained from a simulation could be employed as a proxy for real world data or by the contrary if the studied phenomena is poorly characterized a first computational approach could be performed to gain further understanding that may enable an experimental or a theoretical approach.

The theoretical and computational paradigms force us to formally disclose our assumptions in the form of variables, processes and relationships among them, thus forming what is known as an "abstract model" or simply a "model". Mathematical and computational models can be grouped according on how *state*, *space* and *time* variables are abstracted into the model (Hoekstra, Kroc, and Sloot 2010).

Table 2.1: How different models handle *state*, *space* and *time*. *C* stands for continuous and *D* for discrete. The discrete nature of CA is highlighted. Table adapted from the book "Simulating complex systems by cellular automata" (Hoekstra, Kroc, and Sloot 2010).

Type of model	State	Space	Time
Partial differential equations (PDEs)	C	C	C
Integro-difference equations	C	C	D
Coupled ordinary differential equations (ODEs)	C	D	C
Interacting particle systems	D	D	C
Coupled map lattices (CMLs)	C	D	D
Systems of difference equations	C	D	D
Lattice Boltzmann equations (LBEs)	C	D	D
Cellular Automata (CA)	**D**	**D**	**D**
Lattice gas automata (LGAs)	D	D	D

Complex systems is a broadly defined term to encompass dynamical systems with more than a few interacting parts commonly in a non-linear way, in other

words systems that have emergent properties from individual interactions. This kind of systems are common in the the natural and social sciences. One of the classical examples is the formation of ant colonies and the organization that comes with it (Hölldobler, Wilson, and others 1994). Each ant is sensing external stimuli and acting upon them, the cumulative reactions of all the ants culminates in the building of the ant-hill, feeding it, defending it and attacking other ant colonies. Everything from following 20 to 40 responses to a particular stimuli.

As seen by the ant-colony example, a correctly chosen set of interacting rules for a group of identical generic entities can create self-organization and/or emergent behavior (Bak 2013). However a general algorithm to find a correct set of local rules to produce a target global behavior is not know (Hoekstra, Kroc, and Sloot 2010). Regardless going in the opposite way is as easy as running a simulation using a proposed set of rules and checking if they yield the target behavior. In summary the question, What set of rules yields this behavior? Is extremely difficult, yet asking, Does this set of particular rules yields this behavior? Is rather easy.

In CA the same dichotomy arises. Going from local rules to global behavior (local to global mapping) is as easy as to just perform the computations defined by the CA but going in the other direction (global mapping to local) is extremely difficult. This issue is known as the "inverse problem". Regardless of its difficulty numerous attempts have been tried, with limited success. A common approach is the usage of optimization techniques like evolutionary algorithms or simulated annealing to find rules driving the system to desired attractor basins (Ganguly et al. 2003).

The semantics of CA made them readily available to model complex systems. In fact Ilachinski mentions that *"CA are, fundamentally the simplest mathematical representations of a much broader class of complex systems."* (Ilachinski 2001). The modeling of a complex system using CA often proceeds in a bottom-up manner. Initially essential properties of the interacting parts are abstracted and then codified into the updating rules of CA cells. Secondly the simulation is run in order to learn the *mesoscopic laws* that emerge from the individual interactions. With domain knowledge this process could be iterated, first proposing essential rules and see if they produce reasonable emergent behavior and then improving the rules from this feedback. Finally the system could be enlarged to a gigantic simulation (mainly in space) to get the *macroscopic* final behavior.

However Toffoli points out that this is not the best allocation of computational budget and recommends to use the learned *mesoscopic laws* as input for higher-level analytical or numerical models (Hoekstra, Kroc, and Sloot 2010). To give an example he imagines a CA that predicts the formation of water droplets from simple interactions of particles. Then if the computational resources are available the model could be escalated millions and millions of times to model fog, clouds, all the way up to global weather. However scaling in this way is not really necessary as once the bulk properties of a water droplet have been found they could easily be feed as numerical parameters of a higher-level model

(e.g. differential equation), so in the end the CA had help us to get something like droplets per cubic meters or temperature an so forth. In Toffoli words *"A successful CA model is the seed of its own demise!"*, nonetheless at the end of the day the CA had help us taming the complex system and clearly expressing the essential microscopic dynamics of the system.

2.6 Computing

An analogy between CA and conventional computers can be made. The initial configuration of a CA could be though as input data to be computed over by the CA rules, producing results several time steps ahead and displayed on whatever configuration reached by the lattice.

This analogy is not coincidence at all and it is further exposed by the history of CA. In the early 1950's von Neumann was trying to build a machine that not only should be self-replicating but also capable of universal computability. Von Neumann's endeavors were successful and produced the first two-dimensional automaton formally shown to be Turing-complete (Von Neumann, Burks, and others 1966). Twenty years later John Conway's "Game of Life" was introduced and later was also found to be computationally universal (Elwin, Conway, and Guy 1982)(Poundstone 2013). More recently 1-D CA "Rule 110" has been proved to be universal and is one of the simplest known systems with such property (Cook 2004).

The usual strategy to prove that a given CA is universal is to show its equivalence with other systems known to be universal. Other strategy is to directly build on the lattice all the primitive elements of computing, namely *storage* (memory), *transmission* (internal clock and wires) and *processing* (AND, OR and NOT gates) (Ilachinski 2001). Once a given system supports these computational primitives building an universal machine becomes a clerical work of assembling modules. The "Game of Life" is proven to be universal in this fashion.

On the other hand possessing the same power as a conventional digital computer plays an important role on our mathematical ability to make predictions on the behavior of CA, because all universal computers require resources in the same order of magnitude to process a particular algorithm, thus in general a computational shortcut to the simulation of any universal CA does not exist (Toffoli 1977). This implies that even if an analytical expression for exactly capturing the evolution of a universal CA is obtained, evaluating such expression would take asymptotically the same time as just running the CA and observing its own evolution. Thus remarkably the most efficient way to characterize a universal CA is through its own simulation (Ilachinski 2001).

A different explanation of why, in general, there is not an analytical expression to predict, for any time step t the grid configuration of a universal CA, is by means of the *Halting Problem*, which it is known to be *undeciadable* (Turing

1936). Thus if a CA it is taken as a running program there is no general way of knowing if it would *halt*, with the meaning of *halting*, for this case, being that the CA would reach any configuration that signals the results of the computations.

Furthermore locality being one of the main ingredients of CA imposes an almost independent update on each cell that is only influenced by its neighbors. As the execution of the program by the CA is being carried out individually by its cells the computations are being executed in a fully parallel manner. Consequently simulations on CA allow for efficient parallel implementations of any real world system that can be codified into the CA formalism. For example CA based machines CAMs (CA Machines) have been proposed by Toffoli and others (Toffoli 1984). A hardware implementation of a CAM was developed at MIT (Margolus 1995) that for the modeling of complex systems it could achieve a performance of several orders of magnitude higher that a traditional computer at a comparable cost.

2.7 Generalized Cellular Automata

Generalizations to the classical attributes of CA can be conceived (Ilachinski 2001) enabling extensions like:

- Asynchronous CA: Allows asynchronous updating of the CA.
- Coupled-map Lattices: Allows real valued cell states. These systems are simpler than partial differential equations but more complex that standard CA.
- Probabilistic CA: The rules are allowed to be stochastic, assigning probabilities to cell state transitions.
- Non-homogeneous CA: Updating functions are allowed to vary from cell to cell. A simple example is a CA with two rules distributed randomly throughout the lattice. On the other extreme case, simulations have been performed with random assignment of all Boolean functions with small number of inputs (Kauffman 1984).
- Boolean Networks: A type of non-homogeneous CA with emphasis on variable inputs per node (Alonso-Sanz 2011).
- Mobile CA: In this model some cells can move through the lattice. The mobile parts of the CA can be thought as robots or agents and their movement is dictated from an internal state that reflects the features of the local environment.
- Structurally Dynamic CA: Considers the possibility of evolving cell arrangement. In standard CA the lattice is only a substrate for the ongoing computation, but what happens when this passivity is removed.

2.8 History

Precursor ideas about Cellular Automata (CA) can be traced back to 1946 cybernetics models of excitable media of Wiener and Rosenbluth (Weiner and Rosenblunth 1946), however their usual agreed upon inception was when in 1948 John von Neumann following a suggestion from mathematician Stanislaw Ulam introduced CA to study self replicating systems, particularly biological ones (Von Neumann and others 1951)(Von Neumann, Burks, and others 1966).

Von Neumann's basic idea was to build a lattice in $\mathbb{Z}^2$ capable of copying itself, to another location in $\mathbb{Z}^2$. The solution, in spite of being elaborate and involving 29 different cell states, was modular and intuitive. Since then more constructions capable of the same feat have been found with a lesser number of states (Codd 1968).

In the 1960's theoretical studies of CA were made, especially as instances of dynamical systems and their relation to the field of symbolic dynamics. A notable result from the epoch is the Curtis-Hedlund-Lyndon theorem (Hedlund 1969), which characterizes translation-invariant CA transformations.

In 1969 Konrad Zuse published the book *Calculating Space* (Zuse 1970) with the thesis that the universe is fundamentally discrete as a result of the computations of a CA-like machinery. Likewise during 1960's computer scientist Alvy Ray Smith demonstrated that 1-D CA are capable of universal computation and showed equivalences between Moore and von Neumann neighborhoods, reducing the first to the second (Smith III 1971).

A key moment came with the invention of 2-D CA Game of Life. Pure mathematician J.H. Conway created "Life" as a solitaire and simulation type game. To play "Life" a checkerboard was needed, then counters or chips were put on top of some squares. This represented an initial alive population of organisms and the initial configuration would evolve following reproduction and dying rules. The rules were tweaked by Conway to produce unpredictable and mesmerizing patterns. The game was made popular when was published as recreational mathematics by Martin Gardner in 1970 (Gardner 1970). Despite its name and interesting properties "Life" has little biological meaning and should be only interpreted as a metaphor (Ermentrout and Edelstein-Keshet 1993).

During the 80s the notoriety of CA was boosted to the current status as CA became quintessential examples of complex systems. The focus of research was shifted towards CA as modeling tools. Is in this decade that the first CA conference was held at MIT (Ilachinski 2001) and that a seminal review article of Stephen Wolfram was published (Wolfram 1983).

Since then applications have been coming in a variety of domains. In the biological sciences models of excitable media, developmental biology, ecology, shell pattern formation and immunology, to name a few, have been proposed (Ermentrout and Edelstein-Keshet 1993). CA can be applied in image processing for noise removal

Table 2.2: Key events in the history of Cellular Automata. Table adapted from the book Cellular Automata A Discrete Universe (Ilachinski 2001).

Year	Researcher	Discovery
1936	Turing	Formalized the concept of computability.
1948	von Neumann	Introduced self-reproducing automata.
1950	Ulam	Realistic models for complex extended systems.
1966	Burks	Extended von Neumann's work.
1967	von Bertalanffy, et al	Applied System Theory to human systems.
1969	Zuse	Introduced the concept of "computing spaces".
1970	Conway	Introduced the CA "Game of Life".
1977	Toffoli	Applied CA to modeling physical laws.
1983	Wolfram	Authored a seminal review article about CA.
1984	Cowan, et al	The Santa Fe Institute is founded.
1987	Toffoli, Wolfram	First CA conference held at MIT.
1992	Varela, et al	First European conference on artificial life.

and border detection (Popovici and Popovici 2002). For physical systems fluid and gas dynamics are well suited for CA modeling (Margolus 1984). Also they have been proposed as a discrete approach to expressing physical laws (Vichniac 1984).

2.9 Forest Fire Models

Forest fire models are a type of Probabilistic Cellular Automata (PCA). They try to capture the dynamics and general patterns of tree clusters that emerge from an evolving forest subject to perturbations.

They trace their origins to statistical physics and are closely related with percolation phenomena and dissipative structures. However they have been proved a valuable tool for ecological and natural hazard sciences as simple but powerful modeling tools (Zinck, Johst, and Grimm 2010).

Forest fire models help to tackle questions like: Will the tree population eventually dies out?, What is the general shape of tree clusters?, What is the shape of the boundary between the forest and the fire?

At first glance forest fire models seem similar to epidemiological cellular automata models though they place emphasis on finite population and the persistence of a pathology over time in contrast to the infinite forest population and the emphasis on the spatial extension of the fire.

They broadly have the following characteristics (Louis and Nardi 2018):

1. Cells of at least three types:

- Non-burning tree
- Burning tree
- No tree (empty)

2. A rule for fire initiation:

- A starting configuration with fire cells, usually randomly chosen fire positions.
- Accident simulation, like with a small probability self-ignition of a tree cell.
- Space-time distributed ignition instances (e.g. Poisson distributed).

3. A rule for fire propagation. It involves a stochastic rule for fire spreading between neighborhoods that can be based on actual terrain conditions.

2.10 The Drossel and Schwabl forest fire model

The forest fire model that will be used through this document is the Drossel and Schwabl model (DSM) (Drossel and Schwabl 1992).

DSM was born from research on statistical physics about phase transitions and self-organized criticality (Bak, Chen, and Tang 1990)(Drossel and Schwabl 1992). For this reason the CA was not intended as a modeling tool for real wildfires and was only a metaphor.

Nevertheless data from real wildfires was compared against DSM predictions with the, no so surprising, observation that the model did not perfectly match real world datasets, as it was build with the only concern of generating fire sizes following a power law. DSM was overestimating the frequency of large fires (Millington, Perry, and Malamud 2006). Even though its origins and pitfalls DMS is still valuable, as it has a strong advantage against other wildfire models due to its simplicity and analytical tractability (Zinck, Johst, and Grimm 2010). Likewise it provides a set of starting assumptions that can be augmented to the required complexity along with the usually seen trade-off between increasing a model's predictive capabilities and its generalizing power.

Consequently success have been achieved using this simple model, for example fire shape patterns have been obtained that closely resemble actual wildfires (Zinck and Grimm 2008)(Zinck, Johst, and Grimm 2010).

The Drossel and Schwabl model consist of a lattice in $\mathbb{Z}^2$ populated with three type of cells: 0 (no tree), 1 (burning tree), and 2 (non-burning tree). All cells are synchronously updated according to the following rules: For each state $\sigma_k(n)$ at site k and time step n.

- A burning tree is consumed at next time step:

$$\sigma_k(n) = 1 \mapsto \sigma_k(n+1) = 0 \quad \text{With probability } 1$$

- A new tree grows from an empty position k, dictated by parameter $p \in [0,1]$:

$$\sigma_k(n) = 0 \mapsto \sigma_k(n+1) = 2 \quad \text{With probability } p$$

- The fire is propagated through the vicinity or a lightning event occurs, which ignites a tree and is tuned by $f \in [0,1]$:

$$\sigma_k(n) = 2 \mapsto \sigma_k(n+1) = 1$$

$$\text{With probability} \begin{cases} 1, & \text{if at least one neighboring tree is burning} \\ f, & \text{if no neighboring tree is burning} \end{cases}$$

Chapter 3

Reinforcement Learning

3.1 Machine Learning

Machine Learning is a subfield of Computer Science and Artificial Intelligence (AI). It aims at creating entities capable of learning from examples. What distinguishes it from other AI approaches is the special emphasis on avoiding explicit programming for the task at hand, instead relying only on examples (data) to solve it and from there generalizing to previously unseen cases. So at the heart of this approach lies data. In the classical machine learning setting the algorithm is provided with a set of questions and answers. Then after a period of computations over the pairs questions-answers, the algorithm is presented with new questions, some of them never seen before, that must be answered well enough.

The formalization of this setting leads to an approach known as Supervised Learning, where the questions-answers pairs are codified into mathematical objects, usually numeric vectors for questions and real numbers or categories for answers and the proposed solution comes in the form of a function that maps questions to answers. The name supervised comes from the fact that the algorithm is provided with the answers to the questions, so in a figurative way it is being supervised by a teacher.

Deviation from this classical setting leads to the other broad categories of machine learning. Unsupervised Learning algorithms are provided with just data (questions and not answers) and aim to uncover some structure in such data. Semisupervised Learning is halfway between Unsupervised and Supervised methods as only some answers are given, thus the algorithm must first find some structure on the questions (Unsupervised) to extend the answers to similar questions and then perform Supervised Learning. Finally, Reinforcement Learning algorithms get data and answers, but answers have only a grade on

how favorable they are. The data and grades are obtained through interaction with an environment and the task here is to find answers that maximize the obtained grades.

3.2 Reinforcement Learning

The contents of this section have mostly been adapted from Sutton and Barto's Introduction to Reinforcement Learning book (Sutton and Barto 2018).

Reinforcement Learning (RL) aims to develop algorithms that can satisfactorily inform an agent which decisions to make in order to achieve a goal. The world in which the agent acts is called the environment. The decisions made by the agent could affect the environment and hopefully drive it closer to the goal. A signal of how well the agent is performing is received at each decision step. The signal is called reward and it is an abstraction to represent great or poor sequences of actions. Plenty of this reward signal mean a successful agent closer to its objective.

The Reinforcement Learning Problem could be formalized with Markov Decision Processes (MDPs). The MDP framework models the interaction between an agent and an environment. The agent takes actions and the environment responds with observations and a reward signal, this process is repeated until termination or forever. The non-terminating case is known as a continuing task and the terminating one as an episodic task.

So at each time step $t = 0, 1, 2, 3, ..., T$ the agent using the information of state $S_t \in S$ of the environment must choose an action $A_t \in \mathcal{A}(S_t)$ from the available ones, then in the next time step $t + 1$ the environment transits to a new state $S_{t+1} \in \mathcal{S}$ and returns a reward $R_t \in \mathcal{R} \subset \mathbb{R}$. The dynamics between the agent and environment produces a trajectory of states, actions and rewards indexed by time.

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

Of special importance is the case of a finite MDP, in which the cardinality of states, actions and rewards ($\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$) is finite. In a finite MDP the random variables S_t, R_t have well defined discrete probability distributions that depend only on the previous action A_{t-1} and state S_{t-1}. This is known as the Markov property:

$$p(s', r|s, a) \doteq \Pr(S_t = s',\ R_t = r \mid S_{t-1} = s,\ A_{t-1} = a)$$

This $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ function is known as the *dynamics of the MDP*. The "|" (bar) symbol instead of a "," (comma) in the function arguments is just a reminder that p is calculating a conditional probability given s and a. From here on the assumed MDP would be a finite one.

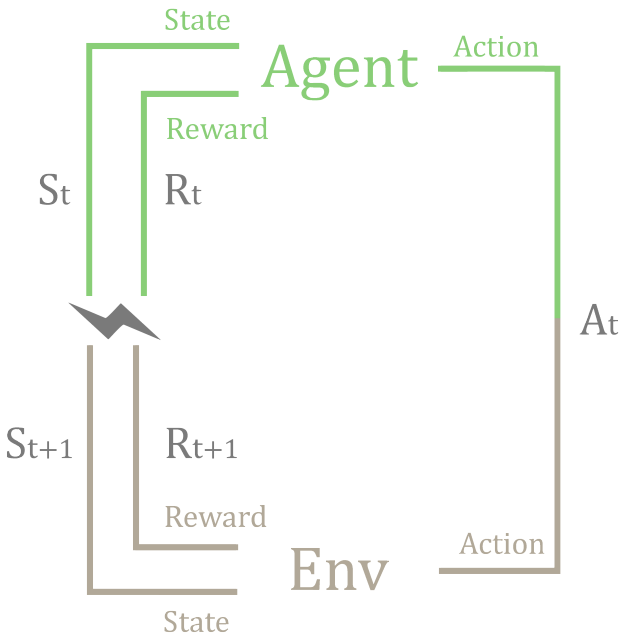

Figure 3.1: Diagram of the *Agent-Environment system* formalized as a MDP. At time step t the "Agent" with knowledge of observation S_t and reward R_t performs action A_t to which the "Environment" responds with a new observation S_{t+1} and reward R_{t+1}, the cycle is iterated. Figure adapted from (Sutton and Barto 2018).

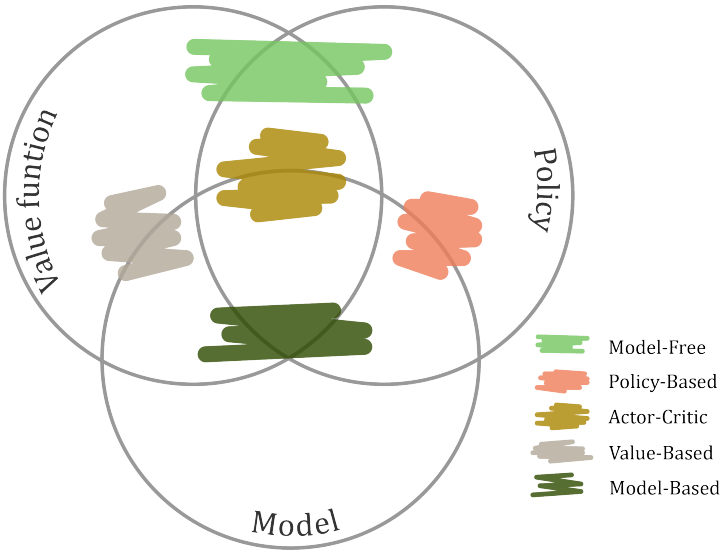

Figure 3.2: Broad classification of Reinforcement Learning algorithms. Adapted from David Silver's Lecture *https://www.youtube.com/watch?v=2pWv7GOvuf0&t=1163s*.

3.2.1 Goals and Return

The reward signal must guide the agent into achieving a goal. The hint that the sequence of rewards provides is such that if the the agent maximizes the total amount of received rewards, the goal would be reached. Thus the agent objective turns to, not maximizing immediate reward, but instead cumulative reward in the long run. This informal idea is known as the *reward hypothesis* and it is stated as follows:

> "All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal." (Sutton and Barto 2018)

In practice, when designing a RL task rewards can be thought of being of two types:

1. **Sparse:** Rewards are only given when the goal is reached.
2. **Shaped:** Rewards are also given when reaching subgoals.

Sparse rewards are preferred since the reward signal should only communicate the *what* we want to accomplish and not the *how*. Similarly the RL framework provides better ways of inserting domain knowledge into the system, like in the initial policy or the initial value function. So it is not surprising that shaped rewards can bias learning when the subgoals are not aligned with the final goal or

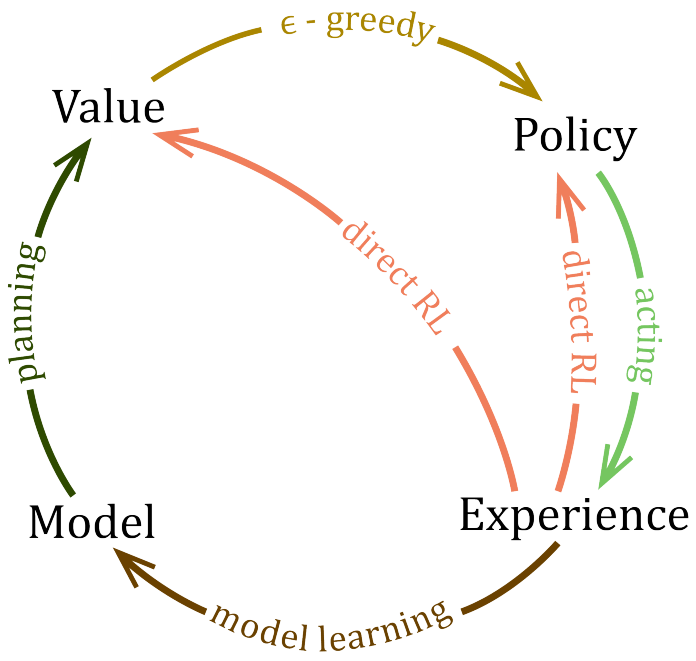

Figure 3.3: Broad relations between different types of strategies to solve the Reinforcement Learning Problem. Adapted from David Silver's Lecture *https://www.youtube.com/watch?v=2pWv7GOvuf0&t=1163s*.

when accomplishing them is given more importance by the agent. Nonetheless, if well designed, shaped rewards can accelerate learning and are often much easier to learn from.

To formalize the previous discussion, the random variable G_t *return* is defined, for the time step t in an episodic task with final step T:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$$

To handle the continuing task case, where $T = \infty$, the notion of *return* can be extended by the inclusion of the discounting parameter $\gamma \in [0 - 1]$.

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$$

This *return* formulation converges to a finite value if $\gamma < 1$ and the reward sequence R_k is bounded. The parameter γ represents how much weight far away rewards must be given, specifically a discount of γ^{k-1} for the reward R_{t+k}, k steps into the future. If $\gamma = 0$, the agent is "myopic" and is only concerned with maximizing immediate rewards. On the other hand as γ approaches 1, more and more distant rewards are being taken into account, thus the agent becomes more farsighted.

A generalization of *return* that combines both, episodic and continuing setting is as follows:

$$G_t \doteq \sum_{k=0}^{T} \gamma^k R_{t+1+k}$$

Where T is turned into a parameter of future steps to take into account by the convention that if the episode terminates, all the futures rewards would be 0. So in this formulation T could be unbounded ($T = \infty$) or $\gamma = 1$, but not both at the same time.

The return G_t at time t follows and important recursive relation:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

3.2.2 Policy

A *policy* π is a mapping from states to probabilities of selecting each particular action. Formally a function of 2 arguments, where the symbol "|" is just to remind us that the action depends on the state.

$$\pi(a|s) \doteq \Pr[A_t = a|S_t = s], \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

Policy codifies the behavior of the agent in the face of environment states. A well performing policy is one that generates behavior that gets the agent closer

to its goal. An optimal policy is one that accomplishes the goal. Reinforcement Learning algorithms must find policies that maximize the *expected return*, since by the *reward hypothesis* this is the same as accomplishing the goal. In other words the problem that RL is trying to solve is to find a policy π that maximizes the *expected return* $\max_{\pi} \mathbb{E}_\pi G_t$.

3.2.3 Value Functions

Value functions capture the idea of how desirable each state is. They are always discussed in the context of a particular policy π. A whole family of methods uses them to solve the RL problem.

The *value function* $v_\pi(s)$ of a state s under a policy π is defined as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s], \quad \forall s \in \mathcal{S}$$

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \middle| S_t = s\right], \quad \forall s \in \mathcal{S}$$

Similarly for action-state pairs ($a \in \mathcal{A}$, $s \in \mathcal{S}$) the *action-value* function for policy π, $q_\pi(s, a)$ is defined:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a], \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

$$q_\pi(s, a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \middle| S_t = s, A_t = a\right], \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

The function q_π represents the value of forcing the agent into taking action a and then following the policy π.

The functions v_π and q_π can be estimated from interaction with the environment. The average, per state s, of rewards obtained following the policy π will converge to $v_\pi(s)$. Similarly, if keeping track of state-action pairs, $q_\pi(s, a)$ can be estimated by the same method.

A fundamental property of value functions is that they satisfy a recursive relationship known as the *Bellman's Equation*:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_\pi(s')\big], \quad \forall s \in \mathcal{S}$$

If the dynamics of the environment are completely known, the *Bellman's Equation* can be used to obtain the value function v_π for all states since a system of linear $|\mathcal{S}|$ equations with $|\mathcal{S}|$ unknowns arises.

Similarly the *Bellman's Equation* for value-action functions is:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)\big[r + \gamma \sum_{a'} \pi(s'|a')\ q_\pi(s', a')\big], \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

3.2.4 Optimal Policies

Solving the Reinforcement Learning problem can be seen as finding a policy that achieves a high expected return. For finite MDPs the concept of an *optimal policy* can be defined. Value functions can be used to order policies in the following way: A policy π is said to be better than other policy π' if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. Thus value functions define a partial ordering over policies, that is to say:

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s), \quad \forall s \in \mathcal{S}$$

There is always at least one policy that is better than or equal to all other. The policies with this property are named *optimal policies* and are denoted by π_*. The value function under an *optimal policy* (*optimal state-value function*) is written v_* and can be expressed as:

$$v_*(s) \doteq \max_\pi v_\pi(s), \quad \forall s \in \mathcal{S}$$

In the same fashion, an *optimal action-value function* is defined as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

Bellman's equation for value functions can be rewritten for the *optimal state-value function*. Intuitively it represents the agent perfectly responding with the best action to any situation of the environment. For all states $s \in \mathcal{S}$:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s,a)$$

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma v_*(s')\big]$$

The above equalities are also present on they *action-value* $q_*(s,a)$ form:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1},a') \;\middle|\; S_t = s, A_t = a\right]$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$$

For finite MDPs *Bellman's optimality equation*, either for v_* or q_*, has an unique solution. If v_* or q_* are known for all states and actions, it is possible to solve the Reinforcement Learning problem and derive optimal behavior from them.

$$a = \operatorname*{argmax}_{a'} \sum_{s',r} p(s',r|s,a')\big[r + \gamma\, v_*(s')\big]$$

$$a = \operatorname*{argmax}_{a'} q_*(s,a')$$

3.3 Deep Q Networks

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam non enim id arcu hendrerit pulvinar. Suspendisse vehicula nibh sed leo porttitor cursus. Cras vel quam interdum, suscipit tortor in, tempus ipsum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec dictum velit non sapien vestibulum efficitur finibus ac ipsum. Proin eget diam condimentum, aliquam orci vel, auctor sapien. Nulla ac magna suscipit, tincidunt mi a, gravida quam. Proin eros leo, dapibus id luctus at, interdum et nunc. Morbi dapibus dictum ex ut volutpat. Donec consectetur eget urna facilisis hendrerit. Ut sodales cursus magna. Vivamus ipsum ipsum, congue vitae maximus eu, pellentesque vel erat.

3.4 The Atari benchmark

The Arcade Learning Environment (ALE) was introduced in 2013 (Bellemare et al. 2013). It was quickly adopted as benchmark by the RL community. ALE consits on a wrapper around the Stella emulator[1] for the Atari 2600 console. This wrapper provides an interface to more than 50 Atari games. ALE was used by the Google's DeepMind team in the DQN paper (Mnih et al. 2015), arguibly setting the current trend of deep reinforcement learning.

The desirable characteristics of Atari games for RL evaluation are their variety of tasks, their short episodic nature and scores that can be directly interpreted as rewards. The agents interact with ALE thorugh a set of 18 actions derived from 6 simple actions, namely four directions (up, down, left, right) and fire and NO-OP actions. The final 18 actions are combinations of the basic actions. The state information is raw pixel data with dimensions 210×160, with each pixel on 7-bit color value (0-127). An important parameter

[1] https://stella-emu.github.io/

Chapter 4

Experiments

4.1 Materials and Methods

4.1.1 Environment

A general tool for creating RL tasks was design (*Forest Fire Environment Maker (FFEM)*). The tool generates environments based on the forest fire CA, specifically the Drossel and Schwabl model (DSM) (Drossel and Schwabl 1992). We developed the tool from an initial idea of an "helicopter" flying over the symbolic wild fire, represented by the DMS, trying to extinguish the fire by means of allowing the "helicopter" to change fire cells to empty cells. During the implementation the environment was naturally generalized. The basic idea is to have an agent on top of the CA lattice. The agent has its own state and parameters. Particularly it has an specified position at one of the nodes of the CA, where it can act by changing the cell on that position to another of its cell states. The agent navigates the CA changing cells states. When the agent reads its next instruction it moves to a new cell (or stays in place) and then affects, if applicable, the destination. After some agent movements the CA is updated and the full process is repeated.

The ability of allowing an agent to change the CA configurations express the attempt to drive the CA to a desired global state. In the RL framework this is done by giving a reward to the agent at each emitted action. This was done by mere counting all cell states indicating unwanted behavior and then multiplying by some weight (usually negative). This was generalized into getting the frequency of any cell type and then multiplying by a weight to generate a score per cell type. Then all the scores were combined into a global score. In summary the used reward function was the dot product of the cell-states frequencies by some weights, with the semantics that weights for desirable states are greater than the unwanted ones.

The particular environment that was used for this thesis is one that has the semantics of a "helicopter" trying to extinguish a forest fire. This is represented by an agent influencing the dynamics of a DSM and is as follows: The first component is a DSM with a lattice of size 3x3 with parameters $p = 0.33$ (tree growth probability) and $f = 0.066$ (tree spontaneous burning probability). The used boundary conditions were set to invariant using an extra exogenous cell type that it is not involved in any CA dynamics, we just called them "rocks". The starting lattice configuration was made via random sampling by cell with probability of choosing "tree" of 0.75 and "fire" of 0.25. The second component, the agent ("helicopter") has a starting position in the middle of the lattice (*2nd row and column*), from there it can move in any direction of a Moore's neighborhood, thus the agent can choose from 9 actions (*"Left-Up", "Up", "Right-Up", "Right", "Stay", "Left", "Left-Down", "Down", "Right-Down"*), then after arriving to its new destination the helicopter extinguishes the fire, represented by changing, if applicable, a "fire" cell to "empty". After some movements of the "helicopter" the forest fire CA is updated following the dynamics dictated by DSM. The specific sequence of agent movements and CA updates is: *move, update, move, move, update, move, move, update, move, move, ...* (see 4.1). In accordance to the RL paradigm, at each performed action the environment must respond with a reward signal, in our setting this was $+1$ per "tree" cell and -1 per "fire" cell, in other words a balance between trees and fires. Finally the RL task was continuous.

We selected the parameters of the DMS by seeking a separation between the scales of f and p ($f \ll p$), since a steady state is reached when $\lim \frac{f}{p} \to 0$ (Drossel and Schwabl 1992) and we wanted to see if the agent would significantly alter the dynamics of the CA. In our case $\frac{f}{p} = 0.2$. Alongside, empirical tuning was conducted from running the environment with a random policy to select the exact values of f and p that would make the grid relatively active at each step. It is important to note that the steady state only appears on large grids, as smaller ones are subject to finite size effects (Drossel and Schwabl 1992), thus we paid more attention to the empirical second criterion. Nevertheless balancing the two parameters would be important in future grid scaling experiments.

Moreover the FFEM tool adds two new cells types. A "rock" type that does not interact with anything and is used as a barrier to the fire dynamics, it is always constant. A "lake" type that behaves exactly like a "rock" but it marks special positions in the grid that can alter the "helicopter" internal state. For example an idea (yet to be implemented) is to make the "power" of the "helicopter" limited so it would need to use it "wisely" and then would go to a "lake" to recharge it, akin to refilling its tank to keep fighting the wild fire. Also the FFEM tool generalizes the "powers" of the "helicopter" allowing for the exchange of any cell type for another, this lets for the creation of environments with a diversity of semantics like: An environment where the agent has to put barriers or one where it has to strategically deforest to prevent the spread of fire or even one where it has to completely reorganize the CA lattice. Even

more functionality was included like a deterministic mode, generalized reward functions and several termination conditions, to check all the FFEM features consult: *https://github.com/elbecerrasoto/gym-forest-fire*.

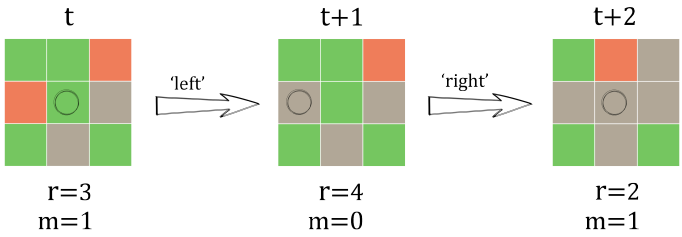

Figure 4.1: Illustration of two transitions in the proposed environment. On time t the helicopter is at 2nd row and column, then it moves to the left, on arriving to its new position it changes the "fire" cell to "empty". Then, at the next time step, it returns to its original position, however the CA was updated before its arrival, anyway the "fire" cell at the middle (not shown) is promptly eliminated and replaced by an "empty" cell. The parameter m is an internal state of the environment, that is decreased by 1 at each step, when it reaches 0 the CA is updated at the next step, before the action takes place, and then m is restored to it max value ($m = 1$). The current reward r is $+1$ per "tree" and -1 per "fire".

4.1.2 Code Availability

The source code for this thesis is openly available at:

1. *https://github.com/elbecerrasoto/gym-forest-fire* for the RL environments.
2. *https://github.com/elbecerrasoto/CARL* for the DQN implementations.

4.1.3 Deep Q Networks

We tried to solve the proposed environment using model-free, value-function approximation RL. This approach is justified by the lack of general analytical models for CA dynamics (Ilachinski 2001) and their combinatorial explosion of states. For illustration in our small 3x3 environment the combinatorics are 3^9 grid states multiplied by 3^2 "helicopter" positions and 2 internal states for the agent-environment synchronization, giving a grand total of $2 \times 3^2 \times 3^9 =$

3.54294×10^5, a still manageable quantity by tabular RL standards, however merely scaling to a 16x16 grid leads to an untenable $\approx 10^{124}$ states.

Deep Q Networks (DQN) (Mnih et al. 2013)(Mnih et al. 2015) meets the previously stated conditions and is well suited for tasks with a discrete set of actions and when sampling from the environment is low-cost. We implemented DQN with n-step unrolling of the Bellman optimality equation for Q values (Sutton 1988). The implementation was made in Python 3.x, the *lingua franca* of AI and RL. The non-linear Q function approximator was an Artificial Neural Network (ANN). We implemented the *experience replay* memory as a *double-ended queue* (deque) data structure, which behaves as a list but allows for efficient "appends" and "pops" from either side ($\mathcal{O}(1)$). (see *https://docs.python.org/3.8/library/collections.html?highlight=collections#collections.deque*). Thus for a fixed size of an *experience replay*, incoming observations are appended to one end and old observations are "popped" from the other.

A high level explanation of our DQN implementation (see Apendix 1) is as follows: The logic of the e-greedy policy was abstracted into an "Agent class" which also holds a memory of past observations (*experience replay*), during the training loop the agent performs an action and advances the environment one step, saving the transition tuple (s, a, r, s') into its own memory. Then the agent samples a minibatch of transitions from the memory buffer and feeds it to the ANN model. A forward prediction of Q values for all 9 actions is computed followed by a backward optimization step. These sequence of agent acts and ANN train steps is iterated during T steps. Each C training steps the weights of the policy and target networks are synchronized.

For the full code details refer to: *https://github.com/elbecerrasoto/CARL*

4.1.4 Preprocessing

The FFEM tool generates environments that follow the guidelines of the *Open AI Gym API* (OAGA) (Brockman et al. 2016). The API specifies that the observation should be a numerical vector, that is commonly returned as a *numpy* n-dimensional array (Walt, Colbert, and Varoquaux 2011). In our implementation we instead returned a tuple of three *numpy arrays* representing: the grid data (cell states), the position of the helicopter and the remaining moves (m) to CA updating. We made the knowledge of m available to the agent to guarantee the Markov property. A more difficult RL task can be made by not knowing m (just dropping its value from the tuple), thus shifting the system to a *Partially Observable Markov Decision Process* (POMDP), where the hidden state m should be inferred by the agent. Strictly speaking our implementation is departing from the *Open AI Gym API* observation specification, however it does it in favor of facilitating the data processing and if purity is insisted on, merely concatenating the data and reshaping it into a single *numpy array* would fix it.

The input to the ANN models was a one-hot encoding of the CA grid concatenated

with the "helicopter" position and the m remaining moves parameter. This was supported by a FFEM feature to indicate the output format of the CA lattice, that can be returned in plain numeric representation, one-hot encoding or by channels as the CA lattice can be interpreted as an image.

4.1.5 ANN Architectures

Model-free, value-function approximation RL tries to tackle the intractability of directly estimating $Q(s, a)$ for a high number of state-action pairs with a parameterized version $Q(s, a; \theta)$. In the case of Deep Reinforcement Learning Q is approximated by a ANN with parameters θ.

A naive, nonetheless intuitive, approach is to directly translate $Q(s, a) : S \times A \rightarrow \mathbb{R}$ to a network architecture that predicts a Q-value for each state-action pair. However this approach incurs on $|A|$ forward computations of the ANN since in order to get the next action following the *greedy policy*, we need to check for all values of the available actions $\forall a \in A(S_t) : Q(s, a; \theta)$. One of the key insights of the DQN algorithm (Mnih et al. 2013)(Mnih et al. 2015) was to change to an architecture of the form $Q_{network} : S \rightarrow \mathbb{R}^{|A|}$, predicting all Q-values in a single forward pass and also with the extra justification that perhaps the model would learn features about state s that can be reused to predict all Q-values for a given s. This architecture is know as the $Q_{network}$.

Three different $Q_{network}$ architectures were used. The models were build using the *Pytorch* library (Paszke et al. 2019). They were Multilayer Perceptrons (MLP) with four or three hidden layers and ReLU activations. The weights and biases were initialized under the *Pytorch* defaults (*Kaiming uniform* (He et al. 2015)). The tensors transformations of the layers are described below:

- Architecture 1 (A1):
  - Number of learnable parameters: 7.4313×10^4

    $$(30 \times 1) \rightarrow (128 \times 1) \rightarrow (256 \times 1) \rightarrow (128 \times 1) \rightarrow (32 \times 1) \rightarrow (9 \times 1)$$

- Architecture 2 (A2):
  - Number of learnable parameters: 4.9673×10^4

    $$(30 \times 1) \rightarrow (256 \times 1) \rightarrow (128 \times 1) \rightarrow (64 \times 1) \rightarrow (9 \times 1)$$

- Architecture 3 (A3):
  - Number of learnable parameters: 2.87881×10^5

    $$(30 \times 1) \rightarrow (256 \times 1) \rightarrow (512 \times 1) \rightarrow (256 \times 1) \rightarrow (64 \times 1) \rightarrow (9 \times 1)$$

4.1.6 Training Details

A $Q_{network}$ capable of predicting the values of the actions would solve the Reinforcement Learning problem (the agent would act greedily over the Q-

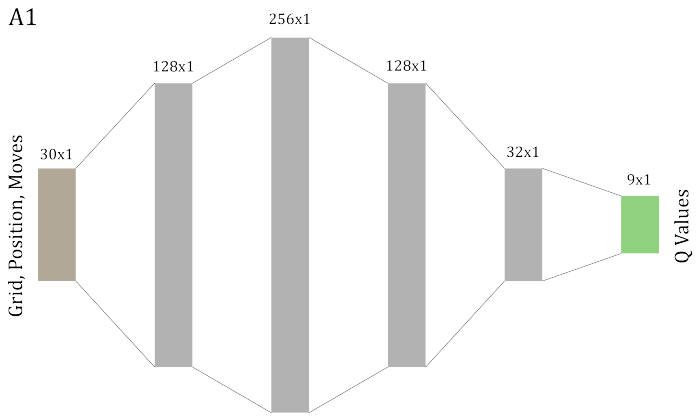

Figure 4.2: Diagram of Architecture 1 (A1).

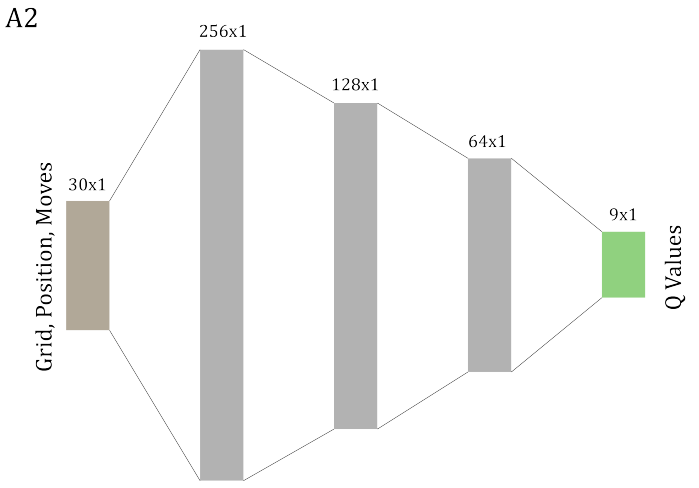

Figure 4.3: Diagram of Architecture 2 (A2).

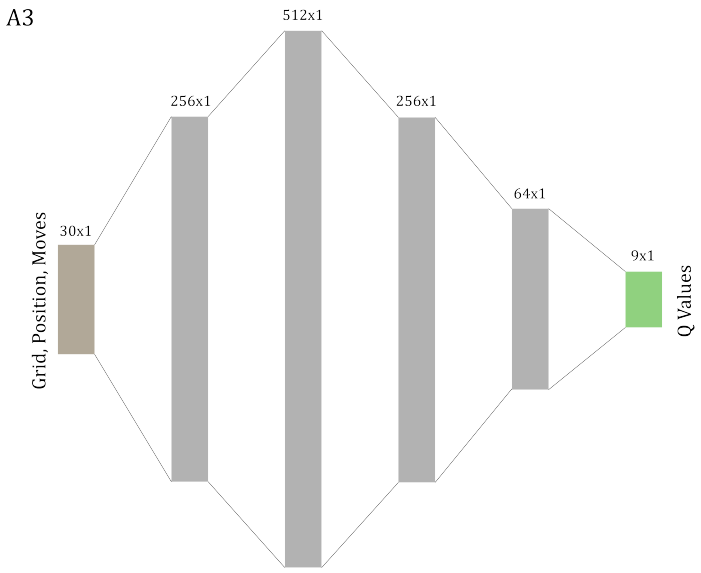

Figure 4.4: Diagram of Architecture 3 (A3).

values). Unluckily training a $Q_{network}$ could be tricky as theoretical convergence results are only available for linear approximators (Tsitsiklis and Van Roy 1997). Additionally *Supervised Learning* techniques assume that the data was generated from the same distribution and its instances are *i.i.d*, both assumptions are violated because the training distribution is constantly changing as the implicit policy defined by the network is changing and the generated observations are highly correlated. To combat the instability that arises DQN uses a replay memory and a target network.

Nine experiments were run using the described environment, between experiments some hyperparameters were allowed to vary, namely the type of initial exploration, the unrolling of the Bellman's update, the network architecture, learning rate, batch size, and γ discount parameter. The overall values for the hyperparameters were taken from (Mnih et al. 2015) and then informally scaled to observe its effects on performance. We did not perform a systematic grid search due to high computational cost. The hyperparameters that were varied are shown in Table 4.2, while the constant hyperparameters can be seen in Table 4.1.

Each experiment was run on *Google Colaboratory* cloud service using their provided GPUs and computing resources. Their run for 1,200,000 epochs each, divided in 400,000 exploration and 800,000 of exploiting epochs. Two exploration schemes were used, for some models a linear annealing of ϵ from 1.0 to 0.10 and for others a simple heuristic that moves the helicopter to any fire cell on the neighborhood, choosing randomly when more than one fire cell was present.

The employed optimization algorithm was Adam (Kingma and Ba 2014) with

Table 4.1: Constant DQN hyperparameters values across the 9 experiments.

Hyperparameter	Value
Total Training Steps	1,200,000
Exploration Steps	400,000
Exploitation Steps	800,000
Policy and Target Networks Synchronization	10,000
Experience Replay Size	200,000

default *Pytorch* values, with the exception of *learning rate* that was set manually per experiment (see Table 4.2). Each transition tuple in the *experience replay* only estimates a single $Q(s, a)$ value, however the output of the $Q_{network}$ is a vector of Q values for all actions, so during optimization steps, for each example in the minibatch, only the weights involved in the calculation of a single $Q(s, a)$ must be taken into account for the weights updating.

4.1.7 Evaluation Procedure

The trained agents were evaluated by playing 100,000 steps in the environment following ϵ-greedy policies (ϵ of 0.00, 0.02, 0.05, 0.10) over the learned Q values. The *return*, *sample mean* and *sample standard deviation* of all evaluation *rewards* per run were computed. To provide a comparison baseline the same statistics were calculated for two other agents, a heuristic and a random ones. The heuristic was the previously described of following fire cells in a 1-step Moore neighborhood. Due to the continuing nature of our RL task the return is essentially the summation of all the *rewards* obtained during the evaluation steps, no discount is used and no resets to starting states are happening between evaluation steps per experiment.

4.2 Results

4.2.1 Hyperparameter Comparison

The obtained *return*, per experiment, from interacting with the environment 100,000 steps is shown in Table 4.2. Five experiments were able to perform better than the base heuristic (c, a, d, i, g runs). Three were worse than random (h, f, e runs) and one (b run) was in between. The maximum obtained *return* was 640,643 a 1.27 fold increase in performance over the base heuristic (return of 503,521).

The comparison of the obtained *mean reward per step* and *reward standard*

Table 4.2: Comparision of obtained returns from the nine runs. The return was computed from playing 100,000 steps per run following the learned policy. The runs are ordered from best to worst and are named from a to i, the "heuristic" and "random" baselines are marked as "H" and "R" respectively.

Run	Return	Exploration	Unrolling	Architecture	LR	Batch Size	Gamma
c	640,658	heuristic	2	A3	0.0001	16	0.99
a	638,094	linear	2	A1	0.0003	32	0.99
d	615,091	heuristic	3	A3	0.0003	32	0.99
i	591,021	linear	2	A3	0.0003	32	0.99
g	507,313	linear	10	A3	0.0003	32	0.99
H	503,521						
b	327,597	linear	1	A2	0.0001	256	0.90
R	319,833						
h	294,965	linear	1	A3	0.0001	16	0.99
f	293,722	heuristic	1	A3	0.0003	32	0.99
e	293,600	heuristic	1	A2	0.0001	16	0.90

deviation is shown in Figure 4.5. It can be observed that the best runs (c, a, d, i, g) also have the an smaller *standard deviation*.

Furthermore a comparison of agents following ϵ-*greedy* policies is shown in Figure 4.6. It could be seen that the performance is close to their respective *greedy* policies.

4.2.2 Training Dynamics

During training of each DQN run the change in *rewards* was monitored, logging the *reward* each 10 iterations. From this loggings the *mean reward per step* was calculated by a sliding window of size 1,000. The *mena rewards per step* during learning are shown in Figure 4.7.

Similarly the values of the error were logged each 10 step. The error behavior during training can be seen in Figure 4.8.

Visual demonstrations are available for the best two runs:

- Run C: *https://www.youtube.com/watch?v=9qFw78___sSM*
- Run A: *https://www.youtube.com/watch?v=DHdRd96KEZA*

4.3 Disscusion

The main objective of this thesis is to design and characterize a novel environment for *Reinforcement Learning*. It must be based on Cellular Automata (CA), with

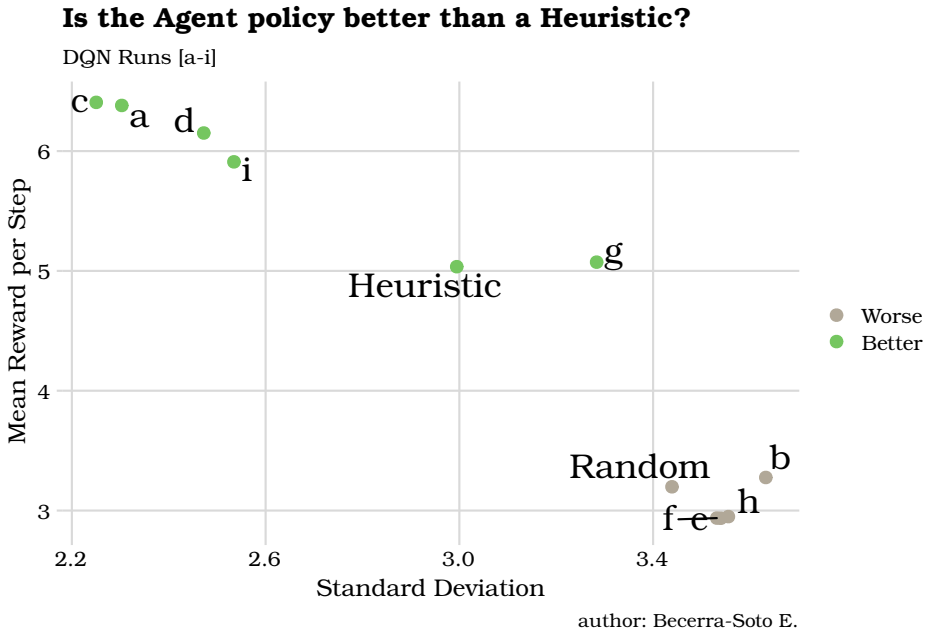

Figure 4.5: The mean and standard deviation for each model was computed from playing the environment 100,000 steps, following greedy policies ($\epsilon = 0$).

the rationale that a handful of well constructed CA based environments would provide an "interesting" set of benchmark tasks for Reinforcement Learning algorithms.

The appeal of such tasks arises from the properties of CA. Some theoretical and practical properties that could make CA a good RL benchmark are:

1. They are easy to implement.
2. They can model complex real world phenomena.
3. Some CA have *universal computation* capabilities.

Under the previous justifications, we have shown that is possible to design a CA based Reinforcement Learning task. Additionally the task has real world semantics, namely a Helicopter extinguishing a wild fire. Our particular proposed environment is an agent affecting cells on top of a Drossel and Schwabl forest fire model (DSM) (Drossel and Schwabl 1992). To further characterize the proposed environment we have applied DQN to try to solve it.

4.3.1 Effect of Hyperameters

Training DRL algorithms is difficult. One of the reasons is that they are quite sensitive to hyperparameters (Irpan 2018). The improvement of training

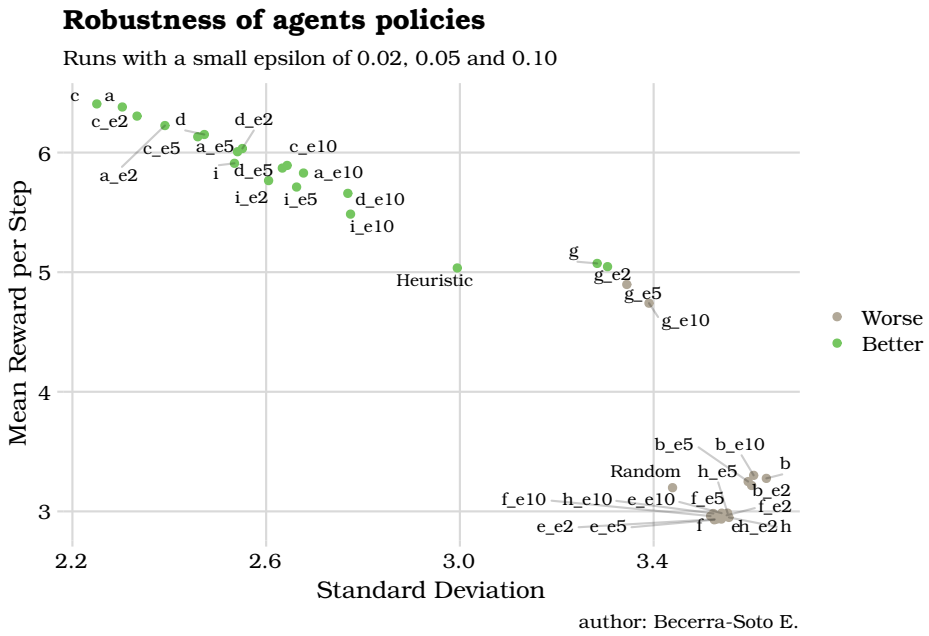

Figure 4.6: Performance of ϵ-*greedy* policies with *epsilonvalues* of $0.02, 0.05$ and 0.10.

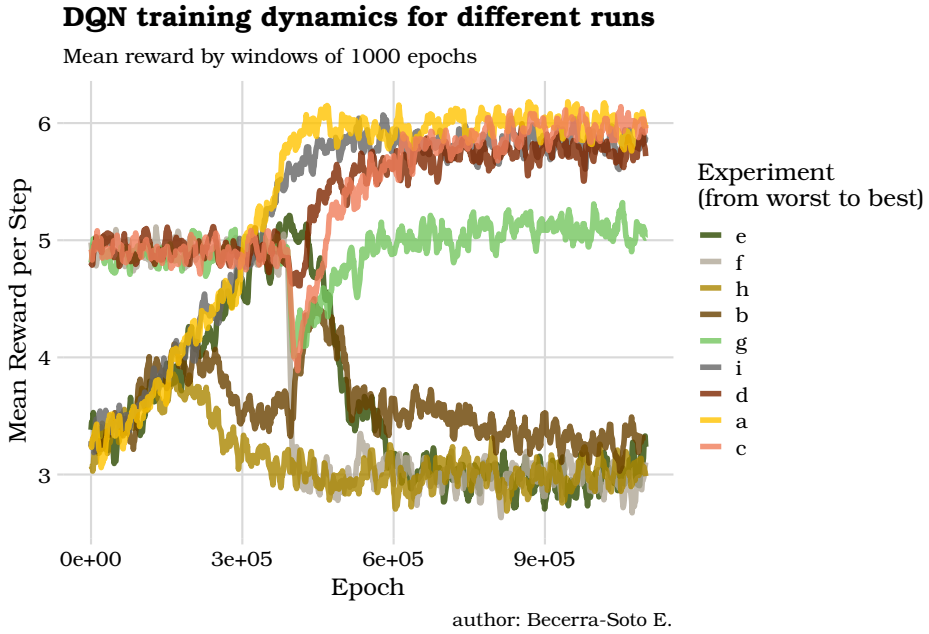

Figure 4.7: Mean Rewards per step during training (1,000 steps sliding window).

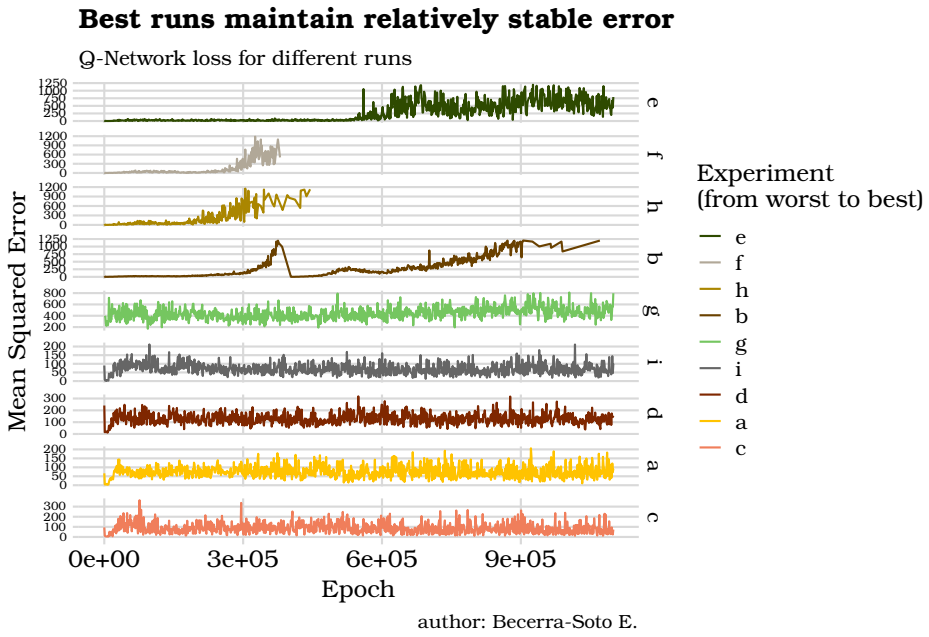

Figure 4.8: Mean Squared Error (MSE) during training. The scale of the y-axis varies between experiments as different values of γ and *unrolling* were used. Also the error was capped at 1,200 due to the increasing error of the divergent models.

stability is a substantial part of DRL research. An advice to train RL algorithms (see *http://rll.berkeley.edu/deeprlcourse/docs/nuts-and-bolts.pdf*) is to start with small instances of the problem and then proceed with a hyperparameter search to try to detect *"life signals"* (a successful DRL run), if despite this the DRL algorithm is not learning anything, the difficulty of the task should be decreased until *"life signals"* are detected. This training strategy was followed. Consequently profiling runs were made with grid configurations of 1×9, 3×3, 5×5 and 8×8 (data not shown). We decided to settle on the 3×3 grid, to further extend hyperparameter search, as bigger grid sizes were not showing *"life signals"*.

The obtained results from the non-exhaustive hyperparameter search can be seen in Table 4.2. Answering the question of to what degree the hyperparameters affect the end results of each run is not trivial and mostly experimental arguments can be provided. This is because attribution becomes difficult when multiple hyperparameters are changed at the same time since their interactions can be non-linear, yet carefully tuning a single hyperparameter can be inefficient as we would only explore a single dimension of hyperparameter space.

The experiments c, a, d, i and g have a better performance than the baseline heuristic. The hyperparameter that seems to have the greater effect is *Unrolling*. When it is equal to 1 the experiments failed to learn (beating the heuristic). *Unrolling* helps to "accelerate" learning as future steps rewards information is used to estimate the Q values. However in our case this "acceleration" meant the difference between learning something or nothing at all. A comparison between experiments c, h, and f is interesting as they have similar hyperparameters values but h and f failed to learn and arguably the difference can be attributed to the *Unrolling* hyperparameter.

The tried *learning rate* values were 0.0001, 0.0003, both of them worked. A small *minibatch size* (16 and 32) and a γ of 0.99 also worked.

One of the approaches to make the task easier was to allow some agents to train following a heuristic policy during exploration. Even though the best performing run used the heuristic exploration, other agents were capable of learning by a simple ϵ linear decrease from 1.0 to 0.10 during the fisrt 400,000 iterations. Thus learning is achievable without the heuristic and in future experiments within the same environment it could be discarded as it can bias learning. The impact of exploring with the heuristic can be seen in the training dynamics plot (Figure 4.7) where the agents using the heuristic start the training with a much better performance, but after exploration is finished some manage to learn from the heuristic (c, d) and others not and its performance plummets (f, e). A similar behaviour can be observed in the epochs vs loss plot (Figure 4.8), where, for the heuristic runs, the error started low and it was kept in that way during the exploration phase, since the linear exploration policy is steadely changing as opossed to the heuristic. Then when exploration finished and the 0.10-greedy policy took effect, some runs had learned and its error was maintained and others had not and its errors blew up.

The choosen ANN architectures are in increasing order of complexity: A2, A1 and A3 (see Figures 4.3, 4.2 and 4.4). The best run employed the more complex network (A3). The number A3 has a number of parameters in the same order of magnitude as the environment states (2.87881×10^5 parameters vs. 3.54294×10^5 states), so it is expected to be considerably overfitting. The number of parameters of the next performing network (4.9673×10^4 parameters) is of an order below, but still it is expected to be overfitting.

Training for more iterations presumably would not increase significantly the performance of any experiment as it could be seen from the training dynamics plot (see Figure 4.7).

From the evaluation data it can be observed that the best performing runs have a lower *standard deviation* to those that failed to learn (Figure 4.5), thus the better runs not only played better but they consistentily did it. Likewise, from the same data, three overall regions, in terms of *mean reward* and *standard deviation*, can be seen. The first one groups the best runs together. The second groups the worst runs. The third one contains the Heuristic and the g run. From the third region is interesting to notice that even so agent g plays better than the heuristic it does it with a higher *standard deviation*, presumably because it performs really well in some enviroment states but badly in others, besides g used an *Unrolling* of 10 which could explain that some estimations of Q values were biased with such large *Unrolling*.

To test for the robustness and maximization bias of the learned policies, the agents were also evaluated with ϵ-greedy policies of 0.02, 0.05 and 0.10. We did not find anything unexpedted, since the performance of better than the heuristic agents deteriorated (Figure 4.6).

4.3.2 Benchmark Capabilities

Chapter 5

Conclusion

5.1 Summary

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam non enim id arcu hendrerit pulvinar. Suspendisse vehicula nibh sed leo porttitor cursus. Cras vel quam interdum, suscipit tortor in, tempus ipsum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec dictum velit non sapien vestibulum efficitur finibus ac ipsum. Proin eget diam condimentum, aliquam orci vel, auctor sapien. Nulla ac magna suscipit, tincidunt mi a, gravida quam. Proin eros leo, dapibus id luctus at, interdum et nunc. Morbi dapibus dictum ex ut volutpat. Donec consectetur eget urna facilisis hendrerit. Ut sodales cursus magna. Vivamus ipsum ipsum, congue vitae maximus eu, pellentesque vel erat.

5.2 Conclusions

Lack of variaty of benchmark bias algorithms.

5.3 Future Work

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam non enim id arcu hendrerit pulvinar. Suspendisse vehicula nibh sed leo porttitor cursus. Cras vel quam interdum, suscipit tortor in, tempus ipsum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec dictum velit non sapien vestibulum efficitur finibus ac ipsum. Proin eget diam condimentum, aliquam orci vel, auctor sapien. Nulla ac magna suscipit, tincidunt

mi a, gravida quam. Proin eros leo, dapibus id luctus at, interdum et nunc. Morbi dapibus dictum ex ut volutpat. Donec consectetur eget urna facilisis hendrerit. Ut sodales cursus magna. Vivamus ipsum ipsum, congue vitae maximus eu, pellentesque vel erat.

Appendices

5.4 Apendix 1 Algorithms

5.4.1 Agent-Environment interactions

Algorithm 1: Agent-Environment interactions.

Input : Steps to play T, Synchronization parameter m.
Output: Return G (accumulated reward).

Initialize: Automaton Grid $C_{i \times j}$
Initialize: Helicopter H: $H = (row, col) : row \in \{1, ..., i\}, col \in \{1, ..., j\}$
$G \longleftarrow 0$
$k \longleftarrow m$
for *1, ..., T* **do**
    if k *has value* 0 **then**
        $C_{i \times j} \longleftarrow$ `computeForestFire`$(C_{i \times j})$
        $k \longleftarrow m$
    else
        $k \longleftarrow k - 1$
    end
    $a \longleftarrow$ `policy`$(C_{i \times j}, H, m)$
    $H \longleftarrow$ `updatePosition`(H, a)
    $C_{i \times j} \longleftarrow$ `applyEffect`$(C_{i \times j}, H)$
    $r \longleftarrow$ `getReward`$(C_{i \times j})$
    $G \longleftarrow G + r$;
end

5.4.2 Agent-Environment interactions following Open AI Gym API

Algorithm 2: Agent-Environment interactions following Open AI Gym API.

Input : Steps to play T, *Environment* and *Agent* objects.
Output: Return G (accumulated reward).

```
/* Commonly used names: s and s' for current and new states, a for
   action and r for reward.                                        */
/* Environment object is reset to starting state, it returns the
   first observation                                               */
1  s ⟵ Environment.reset()
2  G ⟵ 0
3  FALSE ⟵ terminationSignal
4  for 1, ..., T do
5      if terminationSignal then
           /* The episode is over, reset Environment again.       */
6          s' ⟵ Environment.reset()
7          r ⟵ 0
8      else
9          a ⟵ Agent.policy(s)
10         stepData ⟵ Environment.step(a)
11         s' ⟵ stepData[0]
12         r ⟵ stepData[1]
13         terminationSignal ⟵ stepData[2]
14     end
15     G ⟵ G + r
16     s ⟵ s'
17 end
```

5.4.3 Deep Q-networks (DQN) using Open AI Gym API

Algorithm 3: Deep Q-networks (DQN) using Open AI Gym API.

Input : Total iterations T to train Q network.
Output: Trained Q network with weights θ.

Initialize replay memory D to capacity N
Initialize policy network Q with random weights θ
Initialize target network $\hat{Q}$ with weights θ^- such that $\theta^- = \theta$
Get initial observation: $s \longleftarrow$ `Environment`.*reset*()
for *t=1, ..., T* **do**
    With probability ϵ: select random action a
    Otherwise select: $a \longleftarrow \arg\max_{a'} Q(s, a'; \theta)$
    Execute `Environment`.*step*(a) to obtain reward r and next observation s'
    Store transition (s, a, r, s') in D
    Sample a minibatch of transitions (s_i, a_i, r_i, s'_i) from D
    $y_i \longleftarrow \begin{cases} r_i & \text{If the episode was over at iteration } i \\ r_i + \gamma \max_{a'} \hat{Q}\ (s'_i, a'; \theta^-) & \text{Otherwise} \end{cases}$
    Perform optimization step on $\Big(y_i - Q(s_i, a_i; \theta)\Big)^2$ with respect to θ
    Every C steps synchronize networks weights: $\theta^- = \theta$
    if *Episode was over at iteration t* **then**
        $s \longleftarrow$ `Environment`.*reset*()
    else
        $s \longleftarrow s'$
    end
end

References

Alonso-Sanz, Ramon. 2011. *Discrete Systems with Memory*. Vol. 75. World Scientific.

Bak, Per. 2013. *How Nature Works: The Science of Self-Organized Criticality*. Springer Science & Business Media.

Bak, Per, Kan Chen, and Chao Tang. 1990. "A Forest-Fire Model and Some Thoughts on Turbulence." *Physics Letters A* 147 (5-6): 297–300.

Bellemare, Marc G, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. "The Arcade Learning Environment: An Evaluation Platform for General Agents." *Journal of Artificial Intelligence Research* 47: 253–79.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. "OpenAI Gym."

Codd, EF. 1968. "Cellular Automata, Academic Press." *New York*.

Cook, Matthew. 2004. "Universality in Elementary Cellular Automata." *Complex Systems* 15 (1): 1–40.

Drossel, Barbara, and Franz Schwabl. 1992. "Self-Organized Critical Forest-Fire Model." *Physical Review Letters* 69 (11): 1629.

Elwin, R Berkelamp, John H Conway, and Richard K Guy. 1982. "Winning Ways for Your Mathematical Plays." Academic Press.

Ermentrout, G Bard, and Leah Edelstein-Keshet. 1993. "Cellular Automata Approaches to Biological Modeling." *Journal of Theoretical Biology* 160 (1): 97–133.

Ganguly, Niloy, Biplab K Sikdar, Andreas Deutsch, Geoffrey Canright, and P Pal Chaudhuri. 2003. "A Survey on Cellular Automata."

Gardner, Martin. 1970. "Mathematical Games." *Scientific American* 222 (6): 132–40.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on Imagenet Clas-

sification." In *Proceedings of the Ieee International Conference on Computer Vision*, 1026–34.

Hedlund, Gustav A. 1969. "Endomorphisms and Automorphisms of the Shift Dynamical System." *Mathematical Systems Theory* 3 (4): 320–75.

Hoekstra, Alfons G, Jiri Kroc, and Peter MA Sloot. 2010. *Simulating Complex Systems by Cellular Automata*. Springer.

Hölldobler, Bert, Edward O Wilson, and others. 1994. *Journey to the Ants: A Story of Scientific Exploration.* Harvard University Press.

Ilachinski, Andrew. 2001. *Cellular Automata: A Discrete Universe*. World Scientific Publishing Company.

Irpan, Alex. 2018. "Deep Reinforcement Learning Doesn't Work yet." https://www.alexirpan.com/2018/02/14/rl-hard.html.

Kari, Jarkko. 1994. "Reversibility and Surjectivity Problems of Cellular Automata." *Journal of Computer and System Sciences* 48 (1): 149–82.

Kauffman, Stuart A. 1984. "Emergent Properties in Random Complex Automata." *Physica D: Nonlinear Phenomena* 10 (1-2): 145–56.

Kingma, Diederik P, and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization." *arXiv Preprint arXiv:1412.6980*.

Kůrka, Petr. 1997. "Languages, Equicontinuity and Attractors in Cellular Automata." *Ergodic Theory and Dynamical Systems* 17 (2): 417–33.

Louis, Pierre-Yves, and Francesca R Nardi. 2018. *Probabilistic Cellular Automata*. Springer.

Margolus, Norman. 1984. "Physics-Like Models of Computation." *Physica D: Nonlinear Phenomena* 10 (1-2): 81–95.

———. 1995. "CAM-8: A Computer Architecture Based on Cellular Automata." *arXiv Preprint Comp-Gas/9509001*.

Millington, James DA, George LW Perry, and Bruce D Malamud. 2006. "Models, Data and Mechanisms: Quantifying Wildfire Regimes." *Geological Society, London, Special Publications* 261 (1): 155–67.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning." *arXiv Preprint arXiv:1312.5602*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (7540): 529–33.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In *Advances in Neural*

Information Processing Systems 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, 8024–35. Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Popovici, Adriana, and Dan Popovici. 2002. "Cellular Automata in Image Processing." In *Fifteenth International Symposium on Mathematical Theory of Networks and Systems*, 1:1–6. Citeseer.

Poundstone, William. 2013. *The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge*. Courier Corporation.

Smith III, Alvy Ray. 1971. "Simple Computation-Universal Cellular Spaces." *Journal of the ACM (JACM)* 18 (3): 339–53.

Sutton, Richard S. 1988. "Learning to Predict by the Methods of Temporal Differences." *Machine Learning* 3 (1): 9–44.

Sutton, Richard S, and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction*. MIT press.

Toffoli, Tommaso. 1977. "Cellular Automata Machines."

———. 1984. "CAM: A High-Performance Cellular-Automaton Machine." *Physica D: Nonlinear Phenomena* 10 (1-2): 195–204.

Tsitsiklis, John N, and Benjamin Van Roy. 1997. "Analysis of Temporal-Diffference Learning with Function Approximation." In *Advances in Neural Information Processing Systems*, 1075–81.

Turing, Alan Mathison. 1936. "On Computable Numbers, with an Application to the Entscheidungsproblem." *J. Of Math* 58 (345-363): 5.

Vichniac, Gérard Y. 1984. "Simulating Physics with Cellular Automata." *Physica D: Nonlinear Phenomena* 10 (1-2): 96–116.

Von Neumann, John, Arthur W Burks, and others. 1966. "Theory of Self-Reproducing Automata." *IEEE Transactions on Neural Networks* 5 (1): 3–14.

Von Neumann, John, and others. 1951. "The General and Logical Theory of Automata." *1951*, 1–41.

Walt, Stéfan van der, S Chris Colbert, and Gael Varoquaux. 2011. "The Numpy Array: A Structure for Efficient Numerical Computation." *Computing in Science & Engineering* 13 (2): 22–30.

Weiner, N, and A Rosenblunth. 1946. "The Mathematical Formulation of the Problem of Conduction of Impulses in a Network of Connected Excitable Elements Specifically in Cardiac Muscle."

Wolfram, Stephen. 1983. "Statistical Mechanics of Cellular Automata." *Reviews of Modern Physics* 55 (3): 601.

———. 2002. *A New Kind of Science*. Vol. 5. Wolfram media Champaign, IL.

Zinck, RD, and V Grimm. 2008. "More Realistic Than Anticipated: A Classical Forest-Fire Model from Statistical Physics Captures Real Fire Shapes." *The Open Ecology Journal* 1 (1).

Zinck, Richard D, Karin Johst, and Volker Grimm. 2010. "Wildfire, Landscape Diversity and the Drossel–Schwabl Model." *Ecological Modelling* 221 (1): 98–105.

Zuse, Konrad. 1970. *Calculating Space*. Massachusetts Institute of Technology, Project MAC Cambridge, MA.