CIMAT

Centro de Investigación en Matemáticas, A.C.

GAME BASED DEEP REINFORCEMENT LEARNING FOR TARGET TRACKING

T E S I S

Que para obtener el grado de

Maestro en Ciencias

con Especialidad en

Computación y Matemáticas Industriales

Presenta

Marco Antonio Esquivel Basaldua

Director de Tesis:

Dr. Israel Becerra Durán

Autorización de la versión final

Guanajuato, Gto., 26 de enero de 2022

To Brenda, Remi and Suki... my little family

Acknowledgements

First of all I want to thank Brenda for her unconditional love and company, for all of these years spent together and the more to come.

I would like to express my gratitude to my advisor, Dr. Israel Becerra Durán for his guidance and support along this project. I thank also Dr. Rafael Eric Murrieta Cid and Dr. Eduardo Morales Manzanares for their time and valuable feedback.

I want to thank CIMAT to have received me in this Master program and for the financial support at the end of this project. Also to give me access to the super-computing laboratory where I could run some of my implementations. Especially, I wan to to thank my professors whose wisdom has inspire in me a higher desire on learning and applying new knowledge.

I am thankful to the National Council of Science and Technology (CONACYT) for the financial support that allowed me to achieve this degree.

Finally, I thank my family. I know I can always count on them.

Abstract

This work proposes a methodology for solving the tracking problem under classic visibility in the 2D Euclidean space for a pair of omnidirectional antagonistic players, a pursuer and an evader. The methodology starts proposing motion policies for the players in a discrete state-space applying optimal motion planning in a pursuit-evasion game. The first approach in the continuous state-space consists of two neural networks, one per each player, acting as motion policies whose entries are states in the environment and outputs are the actions to perform. The policies are trained from the behaviour in the discrete state-space. Finally, we implement an improvement for the pursuer motion policy using deep reinforcement learning (DRL) considering a fixed trajectory for the evader. In all these cases, the action-space is discrete. A DRL approach from scratch is compared to a initialized DRL approach, using the weights in the neural network trained from the optimal motion planning, and a DRL approach using a master policy (the same neural network trained from the optimal motion planning) which generates transitions in training for a pursuer in two proposed environments. Results show that a simple initialization is enough to achieve favorable outcomes in a simple environment while the use of a master policy is preferred in a more complex one.

Keywords: Tracking problem, pursuit-evasion game, deep learning, deep reinforcement learning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the tracking problem under classic visibility, we seek to keep one or more agents, called evaders, inside the visibility region of one or more pursuers. In an antagonistic case, we can consider this problem as a *pursuit-evasion* game, where the evader (or evaders) tries to step out from the pursuer's (or pursuers') visibility region as quick as possible, while the pursuer tries to keep the evader inside it most of the time. This kind of problem is important in surveillance cases where we want to visualize an object for all time, or in accompaniment cases to have an autonomous companion.

The work done in this thesis is separated into two main objectives: formulate motion policies in the discrete space, and formulate motion policies in the continuous space. As particular objectives in the discrete part, we have to discretize the environment and then establish the behaviour for every player given their map locations. In the continuous part, we first need to collect data so we can train artificial neural networks to be used as motion policies for both players and lately we use reinforcement learning to improve the pursuer motion policy.

1.1 Previous work

The target tracking problem has been previously tackled applying a combination of vision and control techniques [19, 41], both in environments with obstacles [5] and without obstacles [45]. However, the use of deep learning and reinforcement learning techniques is recent in general [35, 49, 51] and in pursuit-evasion games [11, 20].

In our work, we consider the motion plan for both the evader and the pursuer, like in [6], in an environment with obstacles. In [6], the authors present a pursuit-evasion game-theoretical analysis based on visibility in a planar continuous environment with obstacles. Both players, pursuer and evader, are holonomic with bounded speed and know each other's current location. The strategies proposed are functions of the value of the game being in Nash equilibrium, they are constructed near the termination situation which is a corner in the environment. Even thought the work claims to be formulated in environments with obstacles, the strategies only consider a corner remaining open the case for more complex environments, as we propose with simulations in more obstacle populated environments.

Optimal control has been used in [30] to formulate the pursuer's strategy against an intelligent evader in an obstacle-free continuous environment. Capture is done in a time-efficient and robust fashion although the pursuer's actions are not instantaneous optimal. Numerical examples are presented and compared to conventional motion tracking algorithms showing lower capture time. Contrary to this work we are not proposing the use of optimal control in our approach.

Other work using optimal planning, as we are using, is [55], also presented in [56]. In this work, discrete optimal planning is applied to formulate the motion strategy for a group of robotic routers (pursuers) to provide net connection to a mobile user (evader) as long as possible. Two cases are covered: (1) known user trajectory and (2) adversarial user trajectory. In the first case, dynamic programming is used to formulate the strategy based on the available user trajectory by constructing a table storing maximum connection time and then applying backtracking. In the second case, an algorithm is presented which computes the shortest escape trajectory from a given initial configuration. Later on in chapter 3, we use the ideas in this work to formulate the motion policies in the discrete space.

The work in [5] determines whether or not a pursuer is capable to maintain strong mutual visibility of an evader in a known polygonal environment. The evader is only able to move in the reduced visibility graph or RVG (contrary to our approach where the evader is not constraint to this path); nevertheless, it is unpredictable and antagonist. On the other hand, the pursuer is free to move within the entire free space, as in our case. The method is based on an algorithm that computes areas all of the time solving the decision problem of determine strong visibility between the players. Other works [27, 38] deal with the tracking problem under visibility only formulating strategies for the pursuer, considering an unpredictable or partially-predictable evader.

While in the preceding works, models of the environment and the motion of the players are needed, an implementation using reinforcement learning (RL) [53] techniques allows to get rid of these models since the learning process consists in getting information applying actions by the players and interacting with the environment. RL has successfully been applied to complex real-world problems like Atari games [35] and robotic locomotion [29]. The two main approaches in RL come in the form of Q-learning [60] and policy gradient [54]. In the first, a state-value function is approximated, which is then used in a policy to get the best action, while in the other, a direct map is learned between a state and an action.

The work done in [59] presents a reinforcement learning approach in the pursuit-evasion problem for one evader and one pursuer, which are car-like robots, in an obstacle-free continuous environment. In it, Deep Deterministic Policy Gradient (DDPG) [50], which is an Actor-Critic algorithm [26], is used first to train a pursuer to catch an evader that follows an optimal strategy. Then, both players are trained using DDPG with no prior information about the environment showing that it is possible to apply RL to come up with motion strategies for antagonistic players in a pursuit-evasion game. This work is different from our research in that we are tackling the tracking problem under visibility. Even though both approaches are formulated as pursuit-evasion games (ours and the one cited in [59]), there is no

previous initialization on the players strategies as we propose. Note also that we are proposing environments with obstacles, opposite to this work.

Our intention is to come up with motion strategies for an evader and a pursuer in the tracking problem under visibility in a planar space by first formulating policies approximations with optimal planning and then improve these strategies using policy gradient in RL for the pursuer tracking an evader with a fixed trajectory, since a purely RL implementation risks to need a great amount of episodes, and time, to converge with no guarantees on the achieved performance. Other works have already present the idea of employing some kind of initialization to a RL technique. However, works like [65], where an evasion strategy is learned for an evader in a pursuit-evasion game against several pursuers, propose end-to-end learning using Deep Q-Network (DQN) by incorporating the artificial potential field method to the reward function resulting in a combination of artificial potential field and RL.

In [48], a capture version of the pursuit-evasion games is addressed where the problem to be solved is driving an evader to a goal destination and at the same time avoiding being captured by a group of pursuers. The problem is formulated in both continuous state-space and action-space. However, the multi-agent pursuit-evasion game is first formulated as a sequence of discrete matrix games since the task is a high-dimensional and intractable problem. The low-dimensional manifold is obtained by implementing a nonlinear transformation on the continuous state-space and a discrete abstraction on the continuous action-space. Min-max Q-learning is used to get a Q-function approximation to the evader's reward for actions taken from any state by different players. With this Q-function a matrix describing a two-player game per stage is constructed since only the closest pursuer attempts to capture the evader. The solution represents the evader strategy at the current stage. Numerical simulations evaluating the evader's performance are presented.

In [64], a control policy is used as a supervisor policy to train a new learned control policy for a robot by using a weighted average of the supervisor and the learned policy during trials. A heavier weight to the supervisor policy is set initially and, as the process goes on, the weights are adapted to favor the learned policy. This idea is applied to perform a safer and quicker training. Actor-Critic, which is a RL algorithm that implements a combination of Q-learning and policy gradient, is used in the training phase. At the end, the learned policy performance out-performs or at least matches the supervisor policy performance. This method is applied in the OpenAI Gym to show its effectiveness. We propose a simpler algorithm to tackle the RL part in our approach rather than Actor-Cited as cited here.

RL has also been applied in the UAV's (Unmanned Aerial Vehicle) field [1, 9, 23]. Among these, one using an initialization to RL is the one presented in [39], where a planning method is implemented in an autonomous drone racing, combining optimal control and deep reinforcement learning on a single quadrotor that has to cross a series of marked gates in minimal time. The optimal control stage is made by solving the discretized Hamiltonian-Jacobi-Bellman (HJB) equation in a simplified, reduced model producing a closed-loop policy. Before going through reinforcement learning, a policy consisting in a neural network is trained in a supervised learning manner to mimic the HJB performance. This trained policy is improved applying a policy

gradient approach allowing the quadrotor, in simulation, to pass over a gate from a given initial state. At the beginning of the policy gradient stage, initial states near the center of the gate are preferred. As training goes forward, further initial states are chosen resulting in the so-called HJB-RL policy. The race is achieved by applying the trained policy sequentially one gate at a time. Results on simulation are presented comparing a model-based HJB policy, a supervised learning policy and a trajectory planning policy performance to the one achieved by the HJB-RL policy. It is showed that the HJB-RL policy outperforms all the other policies, and a purely RL policy was unable to complete the race. As aforementioned, [39] presents an initialization to RL similar to the one in the present work. However, in addition to utilizing a policy computed from a discrete state-space as a warm start for RL, we also propose another approach that utilizes the pretrained (master) policy to generate useful examples for RL, initially guiding the search for the RL policy.

Real-world implementations have already been applied using computer vision to track a mobile target [32, 15, 24]. These approaches are only focused on the tracker (pursuer), as we propose for the RL part.

1.2 Contributions

The aim in the this thesis document, is to present a methodology allowing to migrate from the discrete to the continuous state-space in the target tracking problem to formulate motion policies for both the evader and the pursuer. Nevertheless, in the last step of this work we come up with motion policies for the pursuer only.

As first stage, given the work done by Onur Tekdas and Vulkan Isler in [55, 56] about robotic routers (where the aim is to provide net connection to a mobile user most of the time given a *mobile router network* under a discretized environment) their algorithm and solutions are adapted to the tracking problem in this work with only one evader and one pursuer. These solutions formulate motion policies for every agent in a given environment.

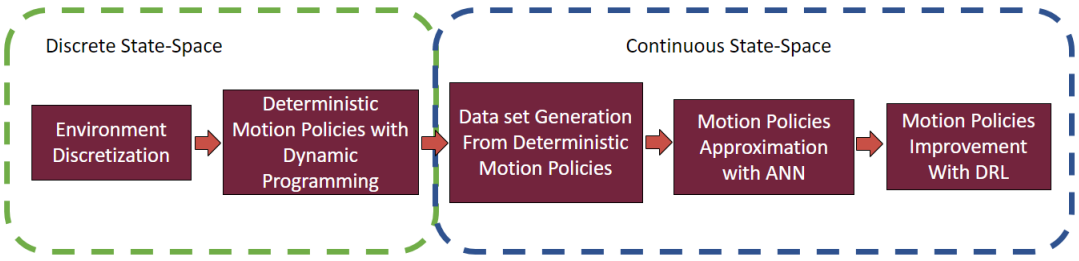
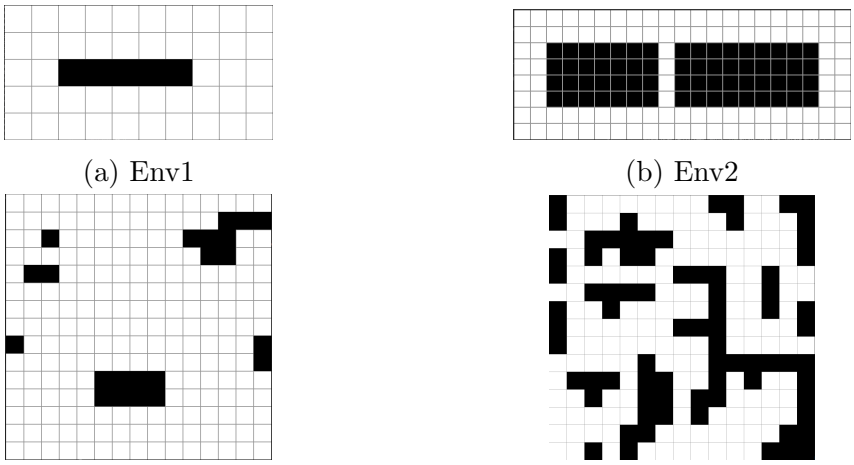


Figure 1.1: Stages in the game based learning methodology.

Later on, the motion policies derived for the discrete state-space are used to generate data sets to feed two dense[1] neural networks, one for each agent, whose

[1] Also called fully-connected layer. Every neuron in a layer is connected to all of the neurons in the next layer.



(a) Env1 (b) Env2

(c) Env3 (d) Env4

Figure 1.2: Environments used in experimental simulations.

models are next used to produce motion policies for each agent now on the continuous state-space. To get a better approximation in the continuous space, the grid which separates every cell in any proposed discretized environment is augmented in resolution. Like this, more data is available to train the models. Thereafter, reinforcement learning [53] techniques, namely policy gradient, are applied using the previous trained models as initializers, which allows faster learning. Alternatively, the previous trained models can be employed to perform an informed search to the reinforcement learning motion policy, by means of generating suitable examples to enhance learning. The described stages above are shown sequentially in the Figure 1.1.

The main contributions of the present work are as follows:

- An optimal motion strategy for both the pursuer and evader that solves the tracking problem in a discrete state-space with a finite control set.
- An extension of the discrete motion policy to a continuous state space utilizing artificial neural networks.
- Making use of reinforcement learning, a refinement of the continuous state space policy for the pursuer that enhances its tracking capabilities is presented. Such refinement can also be utilized as a warm initialization of an RL policy, or as an approach to implement an informed search for the sought policy, which speeds up and improves learning.

1.3 Simulation experiments

All along the text, descriptions of every stage in the methodology are presented, as well as experiments over four proposed environments which are named *Env1*, *Env2*, *Env3* and *Env4*. They can be seen in Figure 1.2. The grid in the images are drawn

to show the environment's dimensions and the obstacles positions. Each cell is one unit long per side. The obstacles in the environments are all polygonal and with right corners.

Env1 is the simplest environment of them all since it only presents an obstacle in the center of the scene. *Env2* presents a corridor-like environment. *Env3* is the environment with the largest free-space[2] among the four proposed. The most obstacle-populated environment is *Env4*.

All experiments and simulations are carried out in a laptop whose specifications are shown in table 1.1. The simulated environments are executed using the *Pygame* module in *Python*.

CPU	Intel Core i7-9750H
Clock frequency	2.60FHz
GPU	RTX 2060
RAM	16 GB

Table 1.1: Specifications of the lap top used in experiments and simulations.

The rest of the content of this document is organized in five more chapters as follows. In chapter 2, the theoretical framework used for the comprehension of the document is presented. On it, pursuit-evasion games, discrete optimal planning, deep learning and reinforcement learning are explained. In chapter 3, the deterministic case is developed using dynamic programming to come up with motion policies to be applied by both the evader and the pursuer in discrete environments, the interaction with the environment generates data to be used later on. Using the information gathered in chapter 3, artificial neural networks are trained in chapter 4, with the aim of learning the motion policies given each agent positions (state) migrating from discrete to continuous domain solving a classification problem. Once the neural networks are trained, their models will be used, in chapter 5, in a reinforcement learning framework as a warm initialization. The goal is to create new policies for the pursuer that outperform the previous ones and provide a method that hastens the learning process. In the last chapter, conclusions and future work are provided.

[2]Space without obstacles.

Chapter 2

Theoretical Framework

In this chapter, a theoretical framework is given in order to understand the basis used along the next chapters. We start presenting a definition of the pursuit-evasion games, where the tracking problem takes place. Next, a brief explanation on discrete optimal planning is given which will be used in the discrete state-space formulation of the problem. Neural networks are used in a supervised fashion before leading to the reinforcement learning (RL) part of the method, thus a section in this chapter is dedicated to deep learning defining the artificial neural networks. We then move to RL and deep reinforcement learning (DRL) where a general framework is set emphasizing on the policy gradient (PG) theorem which is used as the RL technique in the methodology proposed in this work.

2.1 Pursuit-evasion games

In the family of problems known as pursuit-evasion, one group of agents or players (both terms are used indistinctly throughout the document) attempts to track members of another group in a given environment. The main goal is to come up with strategies that enable an autonomous player to accomplish a series of actions against an opponent. This kind of games have relevant applications in aerospace [52, 58] and robotics [10].

Being the environment a key piece in the formulation of this kind of problems, it was firstly modeled geometrically [19], which is sometimes referred to as *continuous pursuit-evasion*, to subsequently use a formulation where movement is constrained by a graph [42] called *discrete pursuit-evasion* or even *graph search*.

In the continuous model, the Euclidean plane, or another manifold, is typically chosen. On its variants, maneuverability constraints are applied to the players (range of speed or acceleration), and the presence of obstacles is included in the geometry of the environment. In the context of a dynamical system and differential game theory, these formulations are closely related to optimal control problems where there exists a single control $u(t)$ and a single criterion to be optimized. In differential games, two controls $u_1(t)$, $u_2(t)$ and two criteria are applied (one for each player). Each player tries to control the state of the system, according to its goal, which responds

to the inputs of the player.

In the discrete scenario, the environment is modeled as a graph. Pursuers and evaders assume places at nodes of this graph and, alternatively, make movement decisions on where to go along the edges of the graph. In the simplest case where velocity is considered the same for all the agents, at every turn every member in both sides must decide whether staying on its node or move to an adjacent one along the edges of the graph. Different velocities for each agent could be considered as the maximum number of edges that they can move along in a single turn, being the minimum one edge (or zero edges in the case where no motion per turn is allowed), and the maximum the idea of infinite velocity which allows an agent to move to any node in the graph as long as there is a path to travel to.

According to the task assigned to the players, pursuit-evasion problems may have different interpretations, being the most common:

- The pursuer (pursuers) has to *capture*, where capture means "get closer to a certain distance", the evader (evaders) [4, 19, 33, 46].
- The pursuer (pursuers) has to stay at a certain distance from the evader (evaders) [36, 37].
- The pursuer (pursuers) has to find the evader (evaders), i.e., pursuer must bring the evader into its visibility region [14, 21, 57].
- The pursuer (pursuers) has to maintain the evader (evaders) inside its *visibility region*. This task is also known as *target tracking* [5, 7, 8, 36].

In antagonistic games, the evader (evaders) attempts the opposite to prevent, or at least hinder, the pursuer's task; namely, get further a certain distance, break the constant distance between players, avoid being seen and escape from the pursuer's visibility region.

In this thesis work, we address the last of these interpretations first considering a discrete pursuit-evasion formulation and next move to a continuous configuration. We consider only one pursuer and one evader. The former tries to keep the latest inside its visibility region all the time, meanwhile this last one tries to escape as soon as possible.

2.2 Discrete Optimal Planning

In discrete planning, a strategy is formulated as a sequence of actions that leads to a goal state or set of states. In optimal planning, the problem permits not only to find this sequence, or a set of sequences, but to prefer the one that optimizes a certain criterion, such like time, distance or consumed energy. For this, a stage index is used to indicate the current plan step, a cost function is formulated as a manner to tell the cost gathered during the plan execution, and a termination action is introduced to indicate when to stop the plan. Two main approaches exist: fixed-length optimal

plans, and variable-length optimal plans. Before briefly explaining each one of them lets introduce some notations according to [28].

Let π_K be a plan of length K, this plan is a sequence $(u_1, u_2, \cdots, u_K)$ of K actions where $u_i \in U$. A sequence of states $(x_1, x_2, \cdots, x_{K+1})$, where $x_i \in X$ can be derived if π_K and $x_I = x_1$ are given by applying the state transition function, f, on which $x_{k+1} = f(x_k, u_k)$. There exists a goal set of states, $X_G \subset X$, that give termination to a plan. A cost function or loss function, L, is formulated as:

$$L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F). \tag{2.1}$$

The cost term, $l(x_k, u_k)$, generates a real value for each $x_k \in X$ and $u_k \in U$. F denotes the final stage, $F = K + 1$. The final term, $l(x_F)$, remains outside the sum operator and it is defined as:

$$l_F(x_F) = \begin{cases} 0 & if \quad x_F \in X_G, \\ \infty & i.o.c. \end{cases}$$

The objective now is to find a plan that minimizes the cost function, L. There is the possibility that the term l_f may vary for different states, $x \in X_G$, to give preference to certain final states over others.

In fixed-length plans one could naively compute all the feasible plans of length K from x_I to x_G and then choose the one with best cost. However this would require a $O(|U|^K)$ running time. *Dynamic programming* allows to avoid this exponential complexity, under the idea that portions of an optimal plan are optimal by themselves. The optimality principle leads to an iterative algorithm called *value-iteration* [28]. This algorithm iteratively computes optimal cost functions over the state-space. There are two equivalent but still different versions on the value-iteration algorithm: *Backward value iteration* and *Forward value iteration*. The cost functions computed by every one of them are cost-to-go and cost-to-come values, respectively. Using the backward iteration version we are finding optimal plans with fixed length **from** every state as an initial state to a immovable final state, x_F, determined at the beginning of the algorithm. On the other hand, the forward iteration version finds optimal plans from a fixed initial state, x_I, **to** every other state as a final state.

A generalization can be achieved considering plans of unspecified length, where there is not a bound on the maximal length K. However, a special action, u_T, is introduced. The elements explained in the previous formulation are here preserved (X, $U(x)$, f, x_I and X_G) with the only difference that in the cost function, L, in equation (2.1) K is not a constant value, varying depending on the length of the plan. The special action u_T is the key component when computing optimal planning with unfixed length, once this action is applied at a state, x_k, it will be applied forever unchanging the state and without accumulating any cost in the equation (2.1). Then, $\forall i \geq k,\ u_i = u_T,\ x_i = x_k$ and $l(x_i, u_T) = 0$. This generalization has its own versions *backward* and *forward* for any value K on the optimization for plans of length K or lower. Detailed information and examples can be found in [28].

In the discrete pursuit-evasion game formulation on this work we apply discrete optimal planning to determine the plan for every player to be executed according to the time steps needed for the evader to step out from the pursuer's visibility region as a cost function.

2.3 Deep Learning

In order to understand deep learning (DL) lets first define what machine learning (ML) is. In [63], authors define machine learning as "the study of powerful techniques that can learn from experience" while IBM in [18] defines it as "a form of AI that enables a system to learn from data rather than through explicit programming". From both definitions we can pick the words "experience" and "data" as the keys to learn complex functions allowing, for example, to predict weather in some region given geographic information (data) and a record of past weather (experience) or to, given a question, answering it correctly in text or voice form.

DL exists as a branch in the ML techniques, just like this one is part of the larger domain of artificial intelligence (AI). The relation between AI, ML and DL is shown in Figure 2.1.

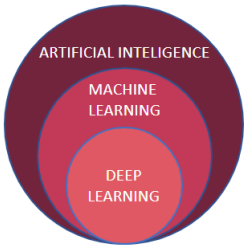


Figure 2.1: Artificial Intelligence, Machine Learning and Deep Learning.

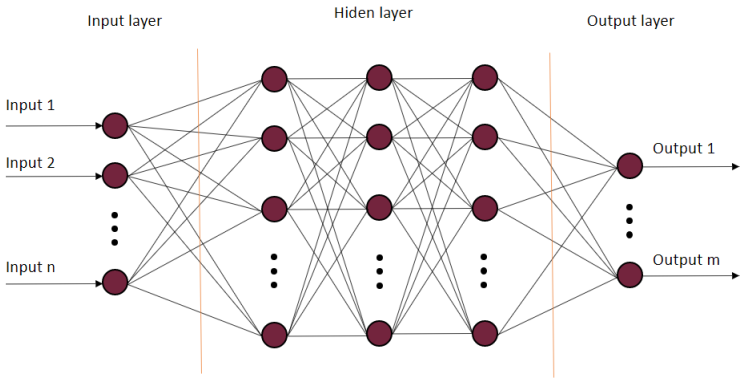


Figure 2.2: Artificial neural network architecture.

DL makes use of *artificial neural networks*, abbreviated as ANN or simply NN (for neural networks), in a set of techniques, named *representation learning*, that allow a system to automatically discover the structures and representations from

raw data to solve problems like feature detection, classification or regression. ANN replace manual feature engineering and permits a system to use features to perform a task. The adjective "deep" in deep learning comes from the structure of an ANN which uses multiple layers of perceptrons, or artificial neurons as shown in Figure 2.2. Every neuron has multiple numeric values that can be tuned during the training phase according to entering data in order to best relate features with the network outputs and be used with data not seen before getting reasonable results.

Key components on any deep learning problem are:

- The *data*, used to learn from.
- A *model* (corresponding to the neural network architecture), which will tell us how to transform the data.
- An *objective function*, used to quantify how well the model is performing.
- An optimization *algorithm*, to tune the model's parameters in order to optimize the objective function.

There exists three main kinds of learning problems where DL is widely applied:

- **Supervised learning**. Input data used to feed the system consists of feature-label pairs. The goal is to come up with a model that maps any input to a label prediction.
- **Unsupervised learning**. Opposite to supervised learning, input data only consists in features, no labels are provided. The goal is to find patterns in the entering data. A very common use to this is clustering.
- **Reinforcement learning**. Being the main topic in this document, it will be deeply explained in section 2.4. By now, lets define it as the kind of learning problem where no data is provided, instead an agent is allowed to interact with an environment on which every action will correspond to a reward for the agent. At the end, the agent will come up with a function, named *policy*, that maps *states* or *observations* (for instance, locations of the agent in the environment) to actions willing to maximize the received rewards. Reinforcement learning has largely been applied in robotics in recent years [25].

Artificial Neural Networks

The core of deep learning are the ANN. To understand how they work, lets first consider the behaviour of a single perceptron, which is also known as an artificial neuron due to the similarity with respect to a biological neuron. A biological neuron is composed of *dendrites*, a *nucleus*, the *axon* and the *axon terminals*, as shown in Figure 2.3.

We can determine equivalencies between the biological neuron and the artificial one (the perceptron) as:

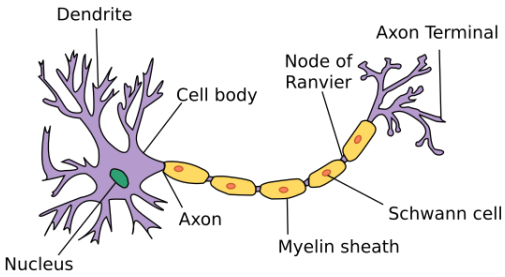


Figure 2.3: A biological neuron representation.

Dendrites → Input terminals
Nucleus → CPU
Axon → Output wire
Axon terminals → Output terminals

As a simplified picture, their behaviour is very similar: input signals, x_i, coming from the environment via sensors or from other neurons are captured by the input terminals (or dendrites). This information is weighted by *synaptic weights*, w_i, which determine the effect of the inputs. These weighted inputs are added together in the CPU (or nucleus) where a bias value, b, is aggregated, $y = \sum_i x_i w_i + b$. Typically, after this summation, a nonlinear function is applied[1], $f(y)$. This processed information is sent, via the output terminals (or axon terminals), to the output system or it can serve as input signal to another neuron.

As a manner of explanation on how neural networks work, consider the regression problems. They appear when we want to predict numerical values, which model the relationship between one or more independent variables, $\mathbf{x}$, (features) and a dependent variable, y, (label or target). The prediction is denoted by $\hat{y}$. In *linear regression* when the input consists in d features, $\hat{y}$ is expressed as in equation (2.2).

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b \tag{2.2}$$

Given a data set, the goal is to chose $\mathbf{w} = (w_1, w_2, \cdots, w_d)^\top$ and b in equation (2.2) such that the predictions made by the model best fit the values observed in data.

By collecting the features in the vector $\mathbf{x} \in \mathbb{R}^d$ and the weights in the vector $\mathbf{w} \in \mathbb{R}^d$, we can express the equation (2.2) as (2.3).

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b \tag{2.3}$$

Linear regression can be considered as a *single layer neural network*, Figure 2.4, where every input, x_i, is connected to the output in a *dense layer*.

A sole set of features put together in $\mathbf{x}$ is called a *sample*, *example* or *observation*. In practice we will have to handle a whole data set which consists of n examples, arranging this data set in a matrix, $\mathbf{X} \in \mathbb{R}^{n \times d}$, on which every row consists of an

[1]The idea is to permit a neural network to approximate universal functions. Only with additions and multiplications this is not possible

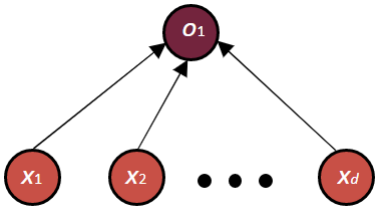


Figure 2.4: Linear regression as a single layer neural network.

example and the columns represent the features of every one of them. In this way, all of the predictions can be gathered in $\hat{\mathbf{y}} \in \mathbb{R}^n$ and then equations (2.2) and (2.3) become (2.4).

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + \mathbf{b} \tag{2.4}$$

Given the features of a training data set, $\mathbf{X}$, and the known labels, $\mathbf{y}$, the goal of linear regression is to find the weight vector, $\mathbf{w}$, and the bias term, b, such that given features of a new data example sampled from the same distribution as $\mathbf{X}$, the new examples label will (in expectation) be predicted with the lowest error. The lowest error term leads us to the definition of a loss function.

The loss function quantifies the *distance* between the *real* and the *predicted* value of the target. In general, the loss function will tell us how well or how bad our model is doing. The smaller the loss, the better the prediction. Let $\hat{y}^{(i)}$ be the prediction for an example i and the corresponding true value be $y^{(i)}$. The square error is given in equation (2.5).

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{2.5}$$

To measure the quality of a model on the entire data set of n examples, we simply average the losses on the training set, equation (2.6).

$$\mathbf{L}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) \tag{2.6}$$

In practice we may add a regularization term to keep the model simple and avoid *overfitting* [62] as it can be seen in equation (2.7). A regularization term measures the model complexity, the more complex is the model the more penalization is given to the model. On the other hand, overfitting occurs when parameters tuned while training are well adjusted to training data but not to examples not seen before. In Figure 2.5 overfitting is shown, where training data is represented by the red dots and test data by the green squares, they are assumed to come from the same distribution. The curved line represents the result of overfitting on training, whereas the straight dashed line could be a desirable result.

$$\mathbf{L}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) + \lambda R(\mathbf{w}, b) \tag{2.7}$$

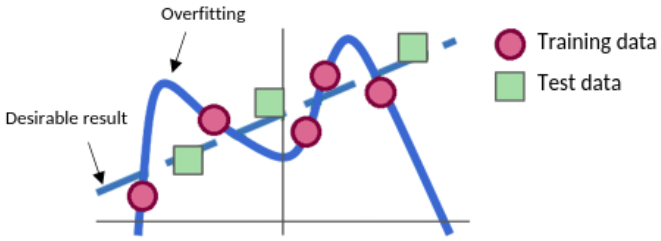


Figure 2.5: The effect of overfitting.

While training the model, the aim is to find the parameters $(\mathbf{w}^*, b^*)$ that minimize the total loss across all training examples, equation (2.8).

$$\mathbf{w}^*, b^* = \underset{\mathbf{w},b}{\operatorname{argmin}} \mathbf{L}(\mathbf{w}, b) \tag{2.8}$$

In linear regression this does not suppose any inconvenient since there exists a close solution to find the desired values in equation (2.4), appending the bias b as another value in $\mathbf{w}$, equation (2.9).

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y} \tag{2.9}$$

Finding the solution to a loss function converts the process into an optimization problem where the most common method applied is *gradient descent* or any of its variants like: stochastic gradient descent [3], Nesterov, accelerated Nesterov, Adagrad, Adadelta, Adam [44].

Depending on the problem being solved by an ANN an appropriate loss function must be chosen to expect good results. We have already seen that mean square error (also known as *quadratic loss* or even *L2 loss*) is a common choice in a regression problem and so it can be the mean absolute error (or *L1 loss*). When dealing with classification problems a better choice could be the *Hinge loss*, also called *Multi class SVM loss*, used for minimum-margin classification and support vector machines. The most common choice in classification problems is the *Cross entropy loss*, or *Negative log likelihood*, which increases as the predicted probability diverges from the actual label. As a last example in the widely existent loss functions family, the *Multinomial logistic regression*, also known as *Softmax loss*, normalizes an input value into a vector of values that follows a probability distribution making it suitable for a probabilistic interpretation in classification tasks.

Neural networks are considered to be **universal function approximators** so they can be applied other than in linear regression problems. To do this, an activation function must be applied at the end of every neuron in the network which is a nonlinear function. If there were not activation functions the whole neural network would be a single linear transformation and in that case any neural network could be treated as a single layer neural network. Common activation functions [40] are:

- Sigmoid
- tanh
- ReLu
- Leaky ReLu
- Maxout
- ELU

In the first approximation of motion policies in the continuous state-space, we use artificial neural networks to map actions from states for every player. These neural networks are trained in a supervised fashion from data collected by applying the discrete motion policies from the discrete optimal planning part.

2.4 Reinforcement Learning

Reinforcement learning (**RL**) is an area of machine learning where an agent learns to take actions by interacting with an environment. After an action is applied, a reward, r, is given to the agent as only feedback information and its state, s, is modified into a new one, s'. As training goes on, the agent attempts to apply the actions that maximise an accumulated reward, also called the expected return. The RL process can be summarized in Figure 2.6.

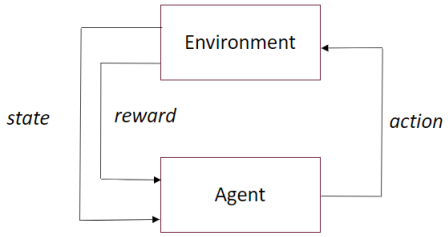


Figure 2.6: The Reinforcement Learning process.

When the agent is at a state, s, it attempts to apply an action that gives it the highest reward possible, but by doing so the agent is only aware of the present reward and will not take into account the impact of future rewards along the current episode, where an episode can be seen as the sequence of all states between an initial state and a terminal state. To overcome this problem the concept of *discounted expected return* or *discounted expected reward* G_t, equation (2.10), is introduced, which the agent has to maximize in order to consider the rewards in the long term. In equation (2.10), $\gamma \in [0, 1)$ is the discount factor. The greater is γ, the more important the long term reward becomes.

$$G_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.10}$$

Typically, the environment is modeled in the form of a Markov decision process (MDP) [43] since many reinforcement learning techniques make use of dynamic programming [28]. MDP is a discrete-time stochastic control process, which provides a framework in decision making situations where actions are determined partially

at random and partially by an optimal controller. In it, an agent in a state s, takes an action, a, which generates a new state, s', receiving a reward, r. In the new state reached, a new decision has to be made to chose an action and receive a new reward. This process continues until a termination criterion is met.

An MDP is represented by a tuple (S, A, Φ, R) where:

- S is a finite set of states called the *state-space*.
- A is a set of actions called the *action-space*. This set can be dependant over each state $a_j(s_i) \in A, j = \{1, \cdots, m\}$.
- $\Phi(s'|s, a)$ is the state transition function describing the probability of reaching the state $s' \in S$ from state $s \in S$ after applying the action $a \in A$. $\Phi : S \times A \to S$.
- $R(s, a)$ is a reward function which gives a scalar value according to the state-actions pairs. This indicates how desirable it is of being at the state $s \in S$ and apply the action $a \in A$. $R : S \times A \to \mathcal{R}$.

MDP's satisfy the *markovian property* on which the action to be applied, a, and thus the reward, r, only depend on the current or present state, s, and the environment "rules" remain stationary all along the execution, this is that the transition function, Φ, never changes.

An MDP defines a problem, and the solution to this problem comes in the form of a policy, π, (which determines the agent's behaviour) or in the form of a value function, V^π or Q^π. The policy, π, is a function that indicates which action to take at state s, $\pi(s) \to a$. There could be a preferable policy, π^*, that optimizes the amount of expected reward all over an execution. The value function is the expected accumulated rewards that an agent anticipates to receive in a state s, $V^\pi(s)$:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\{G_t \mid s_t = s\} \\ &= \mathbb{E}_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\}, \end{aligned}$$

or in a state s taking an action a, $Q^\pi(s, a)$:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi\{G_t \mid s_t = s, a_t = a\} \\ &= \mathbb{E}_\pi\left\{\sum_{k=o}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\}, \end{aligned}$$

At the end of the day, what we want to learn in any RL framework is the optimal policy π^*. Like in a MDP, there are two approaches to train an agent to find π^* distinguishing two family of methods:

- **Value-based** methods where the optimal policy is learned indirectly by teaching the agent which state is more valuable and then taking the action that leads to the more valuable states.
- **Policy-based** methods on which the optimal policy is learned directly by teaching the agent which action to take given an state.

There exist other criterion used to classify the different algorithms in reinforcement learning based on the strategies of learning (Monte-Carlo and Temporal-Difference methods) and the kind of policies (Off-Policy and On-Policy).

Monte-Carlo and Temporal-Difference strategies both make use of experience, by interacting with the environment, to update the policy or the value function. The distinction between them remains in the amount of steps needed before performing an update:

- **Monte-Carlo**: Knowledge is attained incrementally episode-by-episode. The return is calculated only after an episode terminates. A new episode begins with updated knowledge.
- **Temporal-Difference**: This strategy differs to the previous one in that it only requires a step to learn using r_{t+1} and an estimate of G_t (since we can not compute it because a whole episode is not performed before).

The two kind of policies are briefly explained below:

- **On-Policy**: These algorithms estimate the value of a policy while using this same policy for control.
- **Off-Policy**: There are two types of policies. A *behavior* policy, used to generate the agent's actions, and a *target* policy, which is the one improved throughout training.

Deep reinforcement learning (DRL) uses deep neural networks to solve RL problems allowing to take large input data (such like every pixel rendered to the screen in a video game) that, in the traditional RL techniques, could not be possible to compute due to its high dimensionality.

Nowadays, there is so much hype in reinforcement learning, being an important framework with a lot of potential. However, there are limitants to have in mind before deciding to apply RL to solve a problem. RL, and especially DRL, can be sample inefficient requiring a long time for an agent to learn which actions are good and which are bad, according to the received rewards, making several mistakes on the way. This is why in some tasks, like self-driving cars, RL is not the appropriate solving approach (at least not yet), unless knowledge could be transferred from simulations to real-world [22]. Different approaches exist to reduce errors in RL by maximizing sample efficiency [13], learning from demonstrations [16] or using external knowledge [2].

Other aspects to take into account when deciding whether or not to apply RL are if states can properly be defined and if an appropriate reward function can be built in order to characterize the actions in the environment. On the other hand, if having a convenient reward function, RL can be used in cases where the dynamics of the environment (state transition functions) are not available or are inaccurate.

In our work, reinforcement learning is used to train an autonomous pursuer while considering an evader with a fixed path on a given environment. We try training the pursuer both from scratch and from the prior information gathered from the

discrete formulation and from the artificial neural networks trained in a supervised manner. The RL technique used in this document is explained below.

2.4.1 Policy Gradient

As we have settled previously, the objective of a reinforcement learning agent is to maximize an expected reward following a policy. Lets define a policy, which maps a state s to an action a, as π_θ or simply as π where θ represents the set of parameters that parameterize this policy. In our case, using an ANN, θ represents the set of weights and biases units in the neural network representing the policy π. In the Policy Gradient algorithms, the policy π is a probability distribution over actions given states. It is more properly defined as $\pi_\theta(a|s)$. Lets also define a trajectory, τ, as the sequence of states, actions rewards gathered by the agent during an episode of length T, then τ can be written as:

$$\tau = (s_0, a_0, r_1 \cdots, s_{T-1}, a_{T-1}, r_T)$$

If we represent the total discounted reward for a given trajectory τ as $G(\tau)$ we can formulate the objective function, $J(\theta)$, as

$$J(\theta) = \mathbb{E}_\pi[G(\tau)] = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t r_{t+1} \right] \tag{2.11}$$

Note that r_{t+1} is the reward received from the state s_t if applied the action a_t. The total reward $R(\tau)$ could also be applied instead of the total discounted reward $G(\tau)$, where $R(\tau) = \sum_t r_{t+1}$.

If we are capable to come up with the parameters θ^* that maximize J, then we would have solved the task. Like in many other machine learning approaches, gradient computation is used to iteratively approximate the solution. In this case we use *gradient ascent* to update the parameters θ applying the rule in equation (2.12) where α denotes a learning rate value.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.12}$$

Where, according to the policy gradient theorem [61], the gradient of the objective function is expressed as:

$$\nabla J(\theta) = \mathbb{E} \left[\sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t, s_t) G(\tau) \right]. \tag{2.13}$$

The derivation of the equation (2.13) can be seen in the appendix A.

These ideas and the derivation lead to the REINFORCE [61] algorithm (see algorithm 1), which follows a Monte Carlo approach, i.e., it updates the parameters after completing a whole episode.

Algorithm 1 The REINFORCE algorithm.

Initialize θ arbitrarily
for each episode $\{s_1, a_1, r_2, \cdots, s_{T-1}, a_{T-1}, r_T\}$ **do**
    for $t = 1$ to $T - 1$ **do**
        $\theta \leftarrow \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) G(\tau)$
    end for
end for
Return θ

In this chapter a theoretical framework was given covering the knowledge needed in the comprehension of the concepts used throughout this document. In the next chapters the solution to the target tracking problem in the discrete state-space is formulated, a policy initialisation in the continuous state-space is proposed using artificial neural networks and their improvement with reinforcement learning is performed. Conclusions and future work are also discussed.

Chapter 3

Environment Discretization and Deterministic Motion Policies with Discrete Optimal Planning

In discrete environments, where agents interact according to a certain profit and where time is also discretized, it is often possible to establish a deterministic behaviour indicating which cell to visit at every time step t. This is the case in the work by Tekdas and Isler [55] where the goal is to formulate motion policies for a group of *robotic routers* in a discretized environment composed by a series of corridors. This motion policies determine, for every robotic router, where to move as the time (discretized) elapses, being the objective to provide net connection to a mobile *user*. Based on the *adversarial* approach between the agents, we adapt the proposed algorithm (which uses discrete optimal planning) to determine the shortest escape trajectory length for the evader to step out of the pursuer's visibility region.

3.1 Adaptation from Robotic Routers[55] solution to the target tracking problem under classic visibility

Being $\mathcal{W}$ the discretized space composed by n cells where each one of them is represented as a tuple of numbers, in a coordinates fashion way, indicating the free cells in the environment, let $e_0 \in \mathcal{W}$ and $p_0 \in \mathcal{W}$ be the evader and pursuer initial location respectively. The tuple formed by (e_t, p_t) is named a **state**. $N_e(e_t)$ and $N_p(p_t)$ are the reachability or neighborhood functions (both terms are used indistinctly throughout the text) for the evader and pursuer at time t which contains the reachable locations of the agents in a time step from e_t and p_t, respectively. We want to determine if there exists an escape trajectory for the evader in a pursuit-evasion game under classic visibility. If this trajectory exists we determine the steps (locations in $\mathcal{W}$) to be followed by the evader in a discretized time to get out of the pursuer's visibility region. In the case where this trajectory does not exist, we

determine the pursuer motion policy to always keep the evader inside its visibility region.

From the solutions in [55] and [56] in the case where the user trajectory is unknown, the algorithm is adapted considering the case with only one pursuer and one evader. Pursuer visibility from any cell location to the rest of the environment is assumed to be given accurately[1]. We pre-compute the visibility between every pair of cells in $\mathcal{W}$ and save it into a table by drawing a straight line segment between the centers of the cells and checking overlapping on every obstacle in the environment. If overlapping exists, visibility is said to be non-existent between those cells. The table built is used in the algorithm 2 to consult visibility (lines 3-5).

Shortest escape length trajectories from any initial state are computed in algorithm 2.

Algorithm 2 Escape Length Trajectory

1: $\forall e \forall p \; E[e,p] \leftarrow \infty$
2: $\forall e \forall p$
3: **if** e is out of sight from p **then**
4: $\quad E[e,p] \leftarrow 0$
5: **end if**
6: **for** $k = 1$ to n^2 **do**
7: $\quad \forall e \forall p$
8: $\quad$ **if** $\min_{e' \in N_e(e)} \max_{p' \in N_p(p)} E[e',p'] = k-1$ **then**
9: $\quad\quad E[e,p] \leftarrow k$
10: $\quad$ **end if**
11: **end for**

At the beginning of algorithm 2, a table E of dimension 2 and size n^2 is created whose entries are all initialized as infinite. The entry $E[e,p]$ denotes the length of the shortest escape trajectory for the evader from the initial agents' locations, $e \in \mathcal{W}$ and $p \in \mathcal{W}$. This being said, the aim for the pursuer will be to maximize this value meanwhile for the evader it will try to minimize it. If at the end of algorithm 2 one entry $E[e,p]$ remains as infinite, we conclude that there is not a possible escape trajectory for the evader from e and p, in other words, the pursuer is able to keep the evader inside its visibility region no matter the actions applied by the evader. In lines $2-5$ we set $E[e,p] \leftarrow 0$ in the initial states where the evader is already out of sight from the pursuer location, by consulting the visibility table built previously. This is equivalent to say that the evader needs 0 steps to escape from the pursuer's visibility region (it has already escaped). After these initialization steps, the process in lines $7-10$ repeats n^2-1 times. In these lines, the $min-max$ relationship is applied to the entire table, at each iteration the entry $E[e,p]$ is updated only if the shortest escape trajectory length is equal to k.

By setting n^2 as the limit of iterations for lines $7-10$ we assume that every combination of locations has been visited (n positions for the evader from any n

[1] An elternative is to use a visibility function which tells if the evader is inside of the pursuer's visibility region given both the evader and the pursuer locations.

positions for the pursuer) and visibility has been proved to exist for all of them, thus considering any other iteration would be worthless and the escape length is kept as infinite.

The time complexity of the algorithm is $O(n^6)$: there exist n^2 entries in the table E and on each iteration the entire table is visited which has complexity $O(n^2)$, this process iterates for n^2 times, to consider any escape length from one step to n^2 steps, which sum up to the claimed complexity. In Theorem 1, the correctness and optimality of the algorithm is showed. This theorem and its proof are taken from [55].

Theorem 1. *Suppose there exists a shortest escape trajectory such that the evader is initially at location e and the pursuer at location p. Let $\tau(e,p)$ be the length of this trajectory.*

1. *$E[e,p] = k$ if and only if the length of the shortest escape trajectory $\tau(e,p)$ is k.*

2. *$E[e,p] = \infty$ if and only if the pursuer can maintain the evader inside the visibility region for all time.*

Proof. Proof of (1): induction on k is used to show that $E[e,p] = k \Leftrightarrow \tau(e,p) = k$.

The basis case $E[e,p] = 0 \Leftrightarrow \tau(e,p) = 0$ holds due to the initialization step in lines 2-5, which means that the evader is not inside the visibility region in the initial state and $E[e,p] \leftarrow 0$ is set.

In the inductive step we assume that $\forall k,\ E[e,p] = k \Leftrightarrow \tau(e,p) = k$ holds. We prove that $E[e,p] = k+1 \Leftrightarrow \tau(e,p) = k+1$ by showing that both directions hold in the conditional statement.

To prove $E[e,p] = k+1 \Leftrightarrow \tau(e,p) = k+1$, for contradiction, we suppose that $E[e,p] = k+1$ but $\tau(e,p) \neq k+1$. Due to the inductive step we have $\tau(e,p) \geq k+1$ (Condition 1). Due to the inductive hypothesis $\tau(e,p) < k+1$ would imply $E[e,p] < k+1$, which is a contradiction. When $E[e,p]$ is set to $k+1$ the following holds due to the min-max relation (lines 8-9): $\exists e' \in N_e(e),\ \exists p' \in N_p(p)$ such that $E[e',p'] = k$ and $\forall p'' \in N_p(p),\ E[e',p''] \neq k$. From the inductive hypothesis and the inequality $\forall p'' \in N_p(p),\ E[e',p''] \neq k$, we have $\forall p'' \in N_p(p),\ \tau(e',p'') \neq k$. This gives us $\tau(e',p'') \neq k$ (Condition 2). This is because the evader can choose to go to e' and follow an escape trajectory of length k afterwards. From conditions (1) and (2), we have $\tau(e,p) = k+1$ which contradicts the original claim. Thus, $E[e,p] = k+1 \Rightarrow \tau(e,p) = k+1$ holds (Condition 3).

To prove the backwards statement $\tau(e,p) = k+1 \Rightarrow E[e,p] = k+1$ by contradiction, lets assume that $\tau(e,p) = k+1$ but $E[e,p] \neq k+1$. From the inductive hypothesis $E[e,p] \geq k+1$ holds (Condition 4). Let ψ be an escape trajectory of length $\tau(e,p) = k+1$ with initial positions of the agents in e and p. Let $e' \in N_e(e)$ be the evader location in the second step of ψ. Since the escape trajectory length is exactly $k+1$, $\forall p'' \in N_p(p),\ \tau(e',p'') \neq k$. Otherwise the pursuer can increase the surveillance time by going to p' where $\tau(e',p') > k$. Furthermore, $\exists p' \in N_p(p)$, such that $\tau(e',p')$ is exactly k (otherwise by going p' the pursuer reaches an escape trajectory of length less than $k+1$ which is a contradiction). By the induction hypothesis:

$\forall p'' \in N_p(p),\ \tau(e', p'') \neq k$ applying the min-max relation yields $E[e, p] \neq k + 1$ (Condition 5). From conditions (4) and (5) we have $E[e, p] = k + 1$, which is a contradiction of the original statement. Thus $\tau(e, p) = k+1 \Rightarrow E[e, p] = k+1$ holds (Condition 6).

From conditions (5) and (6) the inductive step is proved and $\forall k\ E[e, p] = k \Leftrightarrow \tau(e, p) = k$ is showed.

Proof of (2):

For the second statement, the proof is very straightforward. The only values for $E[e, p]$ are $k \leq n^2$ or ∞ and the evader can escape according to its shortest escape trajectory $\tau(e, p)$ or stay inside the visibility region of the pursuer for any time t, since the number of iterations in lines 6-10 are never bigger than n^2. Assume that there is an escape trajectory whose escape length is $\tau(e, p) > n^2$ a cycle in the sequence of tuples since the number of permutations of the tuples (e, p) is n^2. But then a shorter escape trajectory, which avoids that cycle, can be found which is a contradiction. ■

The time needed to compute the E table for every of the proposed environments is shown in Table 3.1.

Environment	*Env1*	*Env2*	*Env3*	*Env4*
Time	0.17	1.49	54.26	6.28

Table 3.1: Time, is seconds, needed to compute E table.

3.2 Deterministic Motion Policies (DMPs)

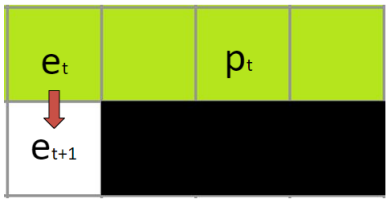
By now, we have determined if given an initial state, (e_0, p_0), there exists an escape trajectory for the evader and the time steps needed to complete this path by simply consulting the value $E[e_0, p_0]$. Nevertheless the behaviour for each agent has not been yet discussed.

In the discretized environment, we will call a *Deterministic Motion Policy* (DMP for short) to be the equation, or the set of equations, that rules the way an agent has to react given the current state at a certain time step t. A time step t consists in the motion of both agents. There exists the DMP_{evader} and the $DMP_{pursuer}$. Evader and pursuer make decisions on where to move at every time step within the execution of a game, assuming that they have access to information that helps to determine the accurate locations of each other. For the evader it is enough to know its current position and to stare at the pursuer's location whereas for the pursuer, at time t, it requires to know the evader's position e_{t+1} in order to apply its DMP to move from p_t to p_{t+1}. This is illustrated in the following example.

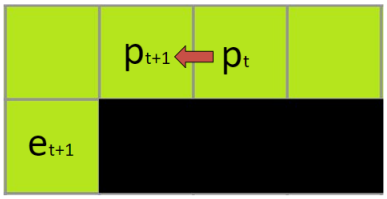
Consider the case of a corner as in Figure 3.1 where both agents move in a 4-connectivity neighborhood[2] meaning no escape for the evader due to the closeness between the players. Black cells represent obstacles where the agents cannot

[2]If an agent is in a cell with coordinates (i, j) at time step t it can move to any valid cell $(i \pm 1, j)$ or $(i, j \pm 1)$ in time $t + 1$.

step into, green cells are free space inside the visibility region of the pursuer which increases from p_t (Figure 3.1a) to p_{t+1} (Figure 3.1b) due to the pursuer motion (visibility between cells is considered existing if a straight segment of line can be drawn from the center of each cell touching at most the corner of an obstacle), white cells are free space outside the visibility region. Imagine that at time t the evader moves first from e_t to e_{t+1} stepping out from the visibility region of the pursuer at p_t (Figure 3.1a), there is a strategy for the pursuer to maintain visibility on the evader at time $t+1$, that is to move to the location p_{t+1} (Figure 3.1b). Based only on its visibility, the pursuer would not be able to guarantee surveillance the time computed in the table E. This is why it is needed for it to know the motion for the evader at time t. This scene appears in the limits of the visibility region, that is, where the evader can step out of the visibility region in one step but the pursuer is able to prevent it also moving one step.



(a) In time step $t+1$ the evader moves from e_t to e_{t+1} and steps out from the visibility region (in green).



(b) Pursuer is able to bring back the evader inside the visibility region in time step $t+1$.

Figure 3.1: Pursuit-evasion in the limit of the visibility region.

In our approaches, we assume that the pursuer knows the exact evader's location at time step $t+1$.

Given the positions e_t and p_t, motion policies are split into three different cases according to the $E[e_t, p_t]$ value:

- Case $E[e_t, p_t] = \infty$
- Case $0 < E[e_t, p_t] < \infty$
- Case $E[e_t, p_t] = 0$

For each one of these cases the evader and pursuer DMP are explained below. Rather than using a DMP to directly map an action from a given state, $s_t = (e_t, p_t)$, we compute the players locations e_{t+1} and p_{t+1}. Then, an action is selected as a function of e_t and e_{t+1} for the DMP_{evader}, and p_t and p_{t+1} for the $DMP_{pursuer}$. Assuming a matrix-like localisation[3] in the environments, the DMP_{evader} determines the action to be applied at time step t according to equation 3.1. In the same way, the $DMP_{pursuer}$ is stated in equation 3.2.

[3]Coordinates are set with the tuple (y, x) where $y \in \mathbb{Z}^+$ and $x \in \mathbb{Z}^+$. y values go from top to bottom, x values go from left to right.

$$DMP_{evader}(e_t, e_{t+1}) = \begin{cases} right & if \quad e_{t+1} - e_t = (0,1) \\ up & if \quad e_{t+1} - e_t = (-1,0) \\ left & if \quad e_{t+1} - e_t = (0,-1) \\ down & if \quad e_{t+1} - e_t = (1,0) \end{cases} \tag{3.1}$$

$$DMP_{pursuer}(p_t, p_{t+1}) = \begin{cases} stay & if \quad p_{t+1} - p_t = (0,0) \\ right & if \quad p_{t+1} - p_t = (0,1) \\ up & if \quad p_{t+1} - p_t = (-1,0) \\ left & if \quad p_{t+1} - p_t = (0,-1) \\ down & if \quad p_{t+1} - p_t = (1,0) \end{cases} \tag{3.2}$$

3.2.1 Case $E[e_t, p_t] = \infty$

In this case, the evader can adopt any strategy since the pursuer guarantees to keep track of it for all time. For instance, the evader can hurry to the closest corner from its current location or an external user could have remote access to control its motion.

Let p_t be the pursuer current location and e_{t+1} the known evader's position at time step $t+1$, the pursuer next location is determined as follows:

$$p_{t+1} = \begin{cases} p_t & if \quad E[e_{t+1}, p_t] = \infty \\ \text{argmax}_{p' \in N_p(p_t)} E[e_{t+1}, p'] & i.o.c \end{cases} \tag{3.3}$$

As it can be seen in equation (3.3), the pursuer location can remain unchanged from time step t to time step $t+1$ if it is guaranteed that the value in the E table is still infinite even if the evader is now at position e_{t+1}.

3.2.2 Case $0 < E[e_t, p_t] < \infty$

In this case the evader is able to escape from the pursuer's visibility region. The time steps needed for the escape are given by the entry $E[e_t, p_t]$. Escape is achieved if the evader goes from e_t to e_{t+1} given by equation (3.4) every time.

$$e_{t+1} = \underset{e' \in N_e(e_t)}{\text{argmin}} \max_{p' \in \{p_t, N_p(p_t)\}} E[e', p'] \tag{3.4}$$

As in case $E[e_t, p_t] = \infty$, the pursuer needs to consider the evader's location at time step $t+1$. Pursuer next location is given by the next equations:

$$\begin{aligned} S &= \left\{ p \mid \underset{p' \in \{p_t, N_p(p_t)\}}{\text{argmax}} E[e_{t+1}, p'] \right\} \\ p_{t+1} &= \underset{S}{\text{argmin}} \, ||e_{t+1} - S||_2 \end{aligned} \tag{3.5}$$

When computing S in equations (3.5) we are looking for a position in the neighborhood, $N_p(p_t)$, of the pursuer, including its current location, p_t, that maximizes

the value in the table E according to the evader new location, e_{t+1}. This search could end up with more that one solution, we pick then the closest one to the evader in e_{t+1}. Like this, even when the evader will escape, there will be a pretense of surveillance with the pursuer following the evader until it is out of sight. Besides, this particular behavior might be useful when the evader does not follow its optimal strategy, positioning the pursuer in an advantageous location for keeping surveillance in the future.

Lets consider an example in a simple environment with only one obstacle surrendered by free spaces to show this case. In Figure 3.2 both agents move in a 4-connectivity neighborhood (just like in Figure 3.1), and the range of the visibility of the pursuer is unbounded, meaning that it can see until the border of the environment or until an obstacle blocks the view. Green cells represent free spaces inside the visibility region of the pursuer at time step t, white cells are free spaces outside the visibility region and the black ones represent obstacles. In Figure 3.2a the initial state is set, on which the length of the escape trajectory is $E[e_0, p_0] = 3$. This number can be seen in the upper right corner of the pursuer location, p_0. In Figure 3.2b the evader moves from e_0 to e_1 at time step $t = 1$, the numbers in the cells in the neighborhood of p_0 represent the length of the escape trajectory if pursuer moves to that location; as it can be seen, there are four locations with the same value, 2, including the current location. Since the pursuer already knows that the evader will escape, it can even stay at the location p_0 without changing the outcome, however there will not be a sense of tracking by the pursuer. In Figure 3.2c the pursuer moves from p_0 to p_1 also at time step $t = 1$ to give a sense of "following" the evader by choosing the closest cell to the evader among the options with highest value. In Figure 3.2d evader moves from e_1 to e_2 at time step $t = 2$, once more the length of the escape trajectories given the options in the pursuer neighborhood appears in the upper right corner of every location with the same value for all of them in this case. In Figure 3.2e pursuer moves from p_1 to p_2 at time step $t = 2$ to continue the sense of "following". Finally in Figure 3.2f evader moves from e_2 to e_3 at time step $t = 3$ stepping out the visibility region. Since the pursuer is unable to re-catch the evader (this can be seen because of the zero value in the neighborhood of p_2) the evader has escaped.

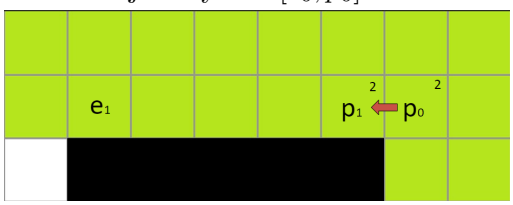
As it could be seen, the purpose of equations (3.5) is to maximize the values in the E table as the evader moves, and at the same time to move as close as possible to it. Otherwise, in cases where the evader will escape in a finite amount of steps, the pursuer could just remain at its initial state without changing the final result (evader escaping).

3.2.3 Case $E[e_t, p_t] = 0$

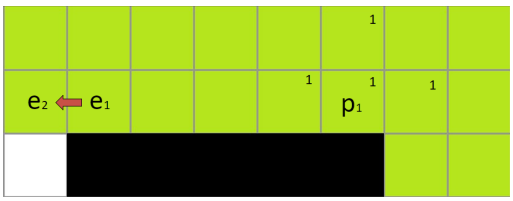
In this case the evader is already outside the visibility region of the pursuer's location at the beginning of the stage, therefore there is nothing to do: the evader has won.

(a) Initial state. The length of the escape trajectory is $E[e_0, p_0] = 3$.

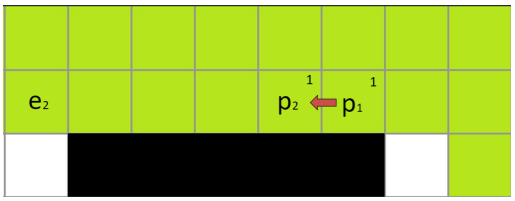
(b) Evader moves from e_0 to e_1 at time step $t = 1$.



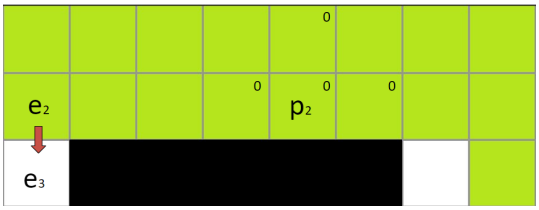
(c) Pursuer moves to p_1 location which is the closest to evader position e_1.



(d) Evader moves from e_1 to e_2 at time step $t = 2$.



(e) Pursuer moves to p_2 location which is the closest to evader position e_2.



(f) Evader moves from e_2 to e_3 and outside the visibility region at time step $t = 3$.

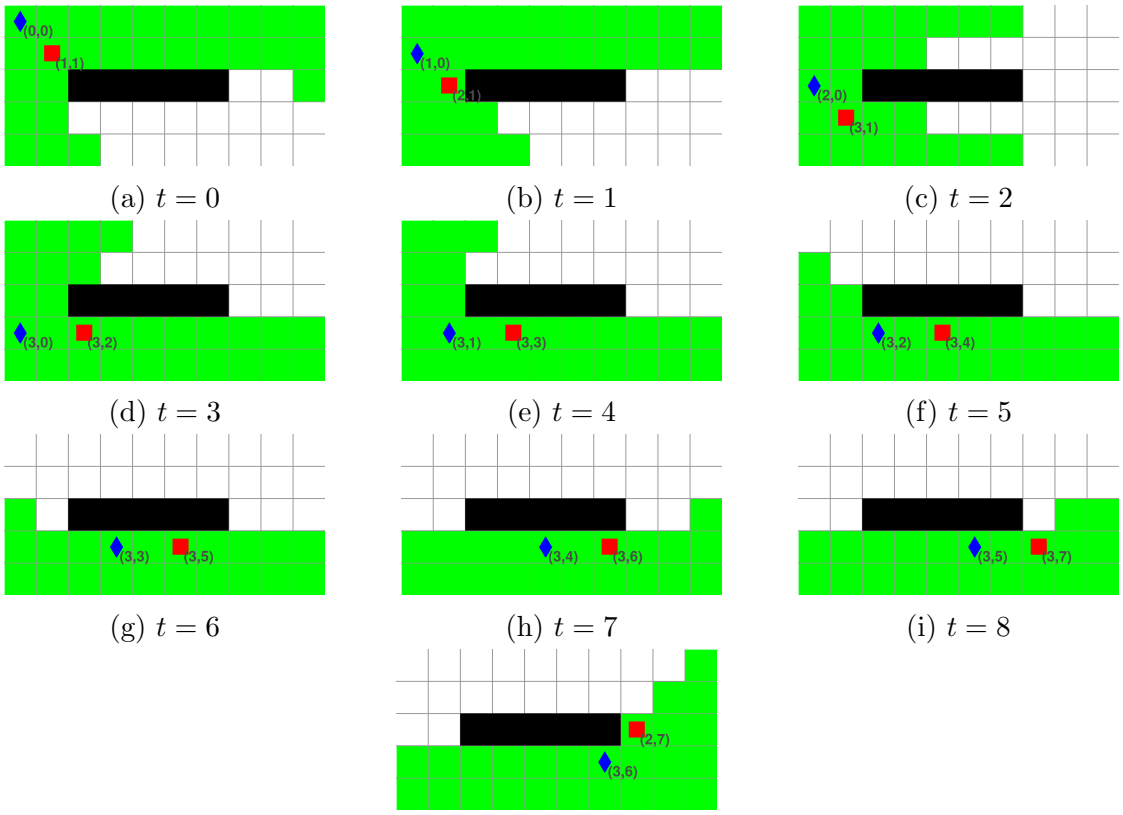
Figure 3.2: Pursuer follows the evader in cases where the escape length trajectory is finite.

3.3 Deterministic Motion Policies (DMPs), Simulations

Simulations have been run testing several initial states on the environments proposed in chapter 1 applying the equations (3.1), (3.2), (3.3), (3.4) and (3.5). Some of the simulations are shown below. All along the simulations in the remaining of the document, the evader is represented by a **red square** and the pursuer by a **blue diamond**, their coordinates are considered to be in a matrix-like fashion.

In Figure 3.3 a simulation over the environment *Env1* is presented with the initial condition $E[e_0, p_0] = \infty$. The evader surrounds permanently the obstacle in the center of the environment. A link to a video showing the simulation is here[4]. Figure 3.4 shows a simulation in *Env3* where $E[e_0, p_0] = 7$. The evader's path sequence is followed until it steps out to the pursuer's visibility region and the pursuer is not able to re-catch it. A link to a video showing the simulation is here[5].

[4]https://youtu.be/XvBjmohmW_E
[5]https://youtu.be/jWkCwXGXThs



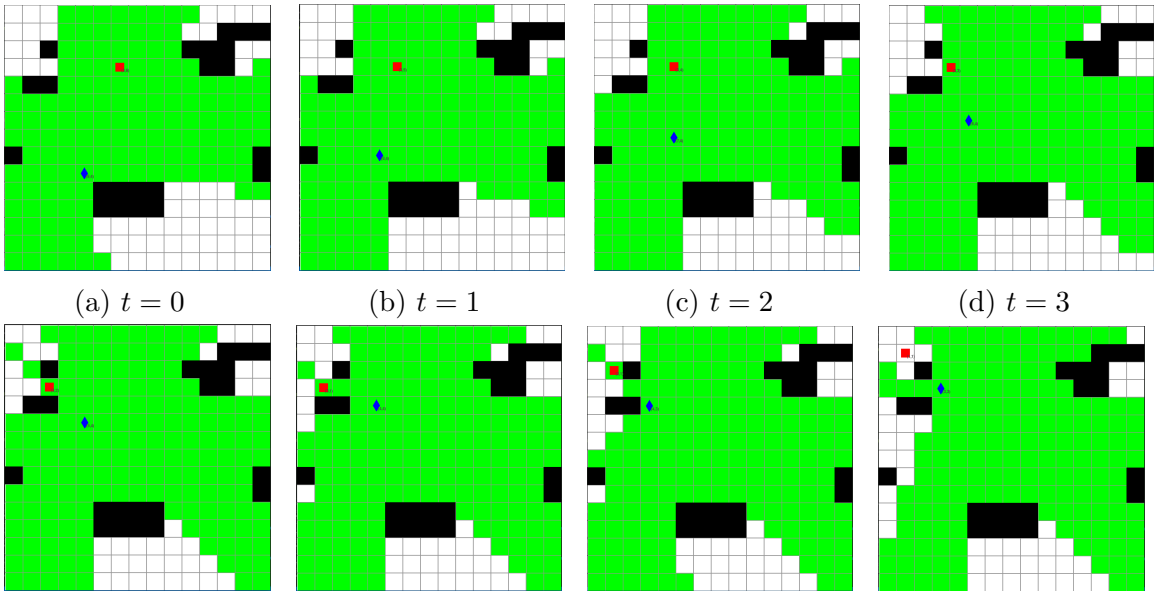
(a) $t = 0$ (b) $t = 1$ (c) $t = 2$

(d) $t = 3$ (e) $t = 4$ (f) $t = 5$

(g) $t = 6$ (h) $t = 7$ (i) $t = 8$

(j) $t = 9$

Figure 3.3: Simulation in *Env1* with $E[e_0, p_0] = \infty$.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

(e) $t = 4$ (f) $t = 5$ (g) $t = 6$ (h) $t = 7$

Figure 3.4: Simulation in *Env3* with $E[e_0, p_0] = 7$.

In this chapter, we have covered the discrete state-space case, where the locations for both the pursuer and the evader are set as cell representation in a discretized environment with obstacles. We established an algorithm, using discrete optimal planning, to determine the shortest escape length for the evader given the initial players' positions. This algorithm also allows to tell if the pursuer is able to track the evader indefinitely. Motion policies, named DMPs, for Deterministic Motion Policies, were formulated giving optimal strategies for both agents based on the escape length trajectory in the initial state, and simulations examples over two environments were shown. In the aim of applying Reinforcement Learning in our problem, in the next chapter we first train artificial neural networks to mimic the actions formulated by the DMPs of this chapter. Reinforcement learning is covered in chapter 5.

Chapter 4

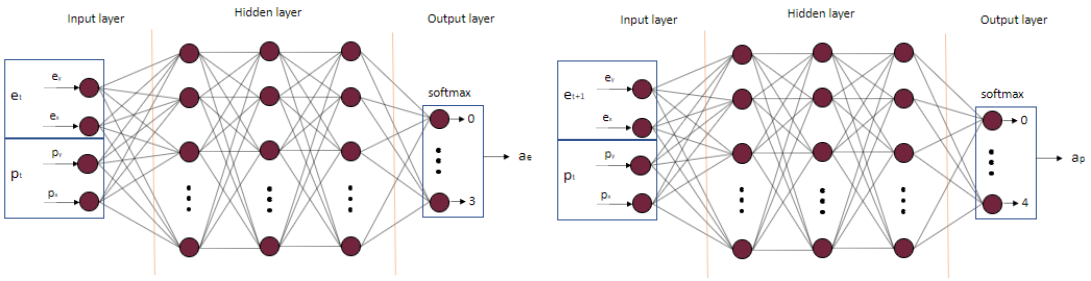
Data set Generation and Motion Policies in the Continuous State-Space with Artificial Neural Networks

Once the tracking problem is solved in the discrete state-space, the next step is to create a first motion policy approximation in the continuous state-space by imitating the actions applied by the agents while discrete optimal planning was used. In this chapter, we present the use of artificial neural networks (ANNs) to bring up motion policies in the continuous state-space domain in a supervised fashion, which will be called SMPs for *Supervised Motion Policies*. One SMP is defined for the evader and other for the pursuer. In the next sections we describe the ANNs architecture used into the SMPs, the way data is gathered from the discrete formulation of the problem and a way of collecting more information from a given environment. Finally, we show the obtained results after training and testing the SMPs in *Env1*[1].

4.1 Supervised Motion Policies (SMPs)

The core in every of the SMPs is the ANN. An ANN must be first defined by their input and output signals. Once an ANN is trained, it is implemented in the application it was designed for in a feed-forward mechanism by providing a valid input signal and getting an output value. In our case, every SMP (SMP_{evader} and $SMP_{pursuer}$) is constituted by an ANN. Recall that we are proposing an approach in the 2D space where the pursuer knows the evader location in the next time step, e_{t+1}, thus, the input for every SMP is defined separately as follows: (1) for the SMP_{evader}, it is the evader and pursuer positions at time step t; (2) for the $SMP_{pursuer}$, it is the known evader position at time step $t+1$ and the pursuer location at time t. Each agent position is defined by two values, namely, their y and x coordinates. This order in the coordinates values is considered since it was the one used in the discrete

[1]For the results in the rest of the environments please consult appendix B.



(a) $SMPevader$ input/output architecture.

(b) $SMPpursuer$ input/output architecture.

Figure 4.1: Evader and pursuer SMP architectures. Their corresponding inputs and outputs are shown.

formulation (a matrix-like location); nevertheless, any another convention could be used. The inputs are illustrated in Figure 4.1 where the location of the evader is given by the tuple (e_y, e_x) and of the pursuer by (p_y, p_x).

The outputs of the SMPs correspond to the agents' action to apply at time step t, a_e for the SMP_{evader} and a_p for the $SMP_{pursuer}$. Opposite to the DMPs where we needed first to calculate the agents next positions, a SMP maps directly a state to an action.

Even though from this stage the games are executed in a continuous state-space, **the actions remain discrete**. An action corresponds to moving an agent in discrete direction right, left, up or down according to the encoding for every player explained below. Once the direction chosen, the player moves one unit in distance with respect to the environment. We are seeking to have a continuously moving evader, meanwhile, the pursuer is allowed to stay still from time step t to $t+1$ as it was formulated in the discrete state-space scenario. Thus, the codification is different for every player. This codification is shown below and in Figure 4.2, where every direction represented by the arrows is coded into the integer value on it. It can also be appreciated in Figure 4.1 that an action is chosen, among the possible ones, after applying a *Softmax* function[2] in the output layer. The best action to chose, a_e and a_p respectively, is the one with the highest value in this Softmax layer.

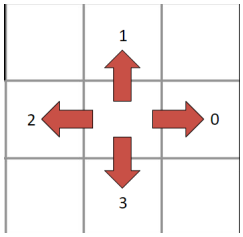
Evader actions codification:

- 0: move right
- 1: move up
- 2: move left
- 3: move down

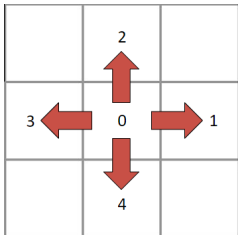
Pursuer actions codification:

- 0: stay still
- 1: move right
- 2: move up
- 3: move left
- 4: move down

[2]The *Softmax* function is widely used in multinomial classification. It turns a vector of k real values into a normalized vector of k values that sum 1, so that they can be interpreted as probabilities.



(a) Evader actions codification.



(b) Pursuer actions codification.

Figure 4.2: Players actions codifications.

For our purpose, each SMP is constituted by an ANN which in turn is comprehended of 4 hidden layers with 256 neurons per layer, each one of this layers uses the *ReLU* function as activation except for the last one where a *Softmax* function is required. Connections are made in a dense fashion. This architecture was proposed and preferred among other configurations after trying them all and measuring their performance. These architectures can be seen in appendix B.

4.2 Data collection for training

Once a neural network architecture is defined, the next step is collecting data. Data is collected by applying the DMPs constructed in chapter 3 in a given environment. Since there are two neural networks (one per SMP), two data sets are built. The first one of them is made up for the evader in which the features of each sample represents every state in the environment where there is visibility between the players and the corresponding label is the action applied by the evader in that state, the features and label are shown in Table 4.1. For the case of the pursuer, remember that it knows the evader location at time step $t + 1$, thus its data set is constructed by considering the states where there is visibility, as in the evader case, and all those cases where the pursuer can establish visibility by taking one step. The features and label in one sample of this data set can be seen in Table 4.2.

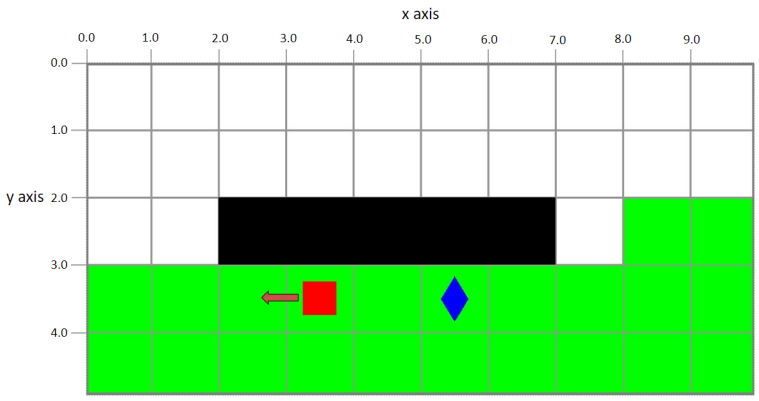
e_y	e_x	p_y	p_x	a_e

Table 4.1: Features and label (action) in a sample in the evader data set.

e_y	e_x	p_y	p_x	a_p

Table 4.2: Features and label (action) in a sample in the pursuer data set.

Every state is considered to be in the center of the cell which give us a localisation in the continuous space. For instance, consider the case in Figure 4.3a, where the evader's action is represented by the respective arrow. This example is added into the data set as in Figure 4.3b according to the actions codification explained in the previous section.



(a) Example of states and actions added to the data set.

3.5	3.5	3.5	5.5	2

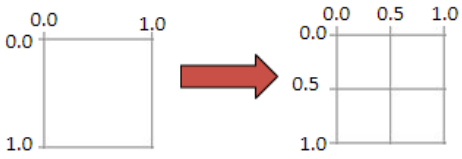
(b) The evader location is saved as the first two features in the example, the pursuer locations goes next and finally the evader's action as the label.

Figure 4.3: Evader data set sample construction.

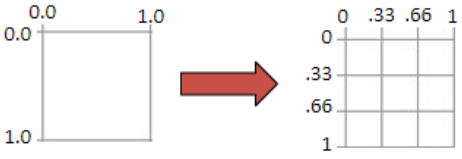
4.2.1 Increasing environment resolutions to grow information

It is well known that neural networks require a huge amount of data. The more data available for training the better a neural network will perform in generalization. For this reason, as a manner to expand the volume of data available per environment, increasing the grid resolution is proposed. This is made by dividing every cell in the original environment grid so that smaller cells will be formed inside. Once this division is done, the centers of the new cells are considered to be possible locations for the agents to step into and the equations formulated in chapter 3 for the DMPs are employed to come up with a larger data set.

We say to be working with an environment with resolution 2 if a cell is divided as in Figure 4.4a or to be working with resolution 3 if a division like in Figure 4.4b is preferred.



(a) Cell increased to resolution 2.



(b) Cell increased to resolution 3.

Figure 4.4: Examples of increased resolution over a cell.

When dealing with an augmented resolution in simulation, one step performed by the agents (moving from one location to the next) will be affected since a smaller distance will be covered, thus a time step duration has to be indirectly proportional

Environment	Time
Env1	0.17sec
Env1 res2	37.34sec
Env1 res3	39min 31.34sec
Env2	1.49sec
Env2 res2	11min 33.99sec
Env2 res3	9h 28min 56.2sec
Env3	54.26sec
Env3 res2	9h 56min 35.43sec
Env4	6.38sec
Env4 res2	1h 15min 10.03sec

Table 4.3: Time needed to compute E table.

to the resolution used in a given environment. For instance, if dealing with resolution 2, the time between time steps has to be half the time required in that same environment with no increased resolution, and so on.

We propose expanding *Env1* and *Env2* into resolutions 2 and 3, meanwhile *Env3* and *Env4* into resolution 2 only.

In these resolution augmented environments, we have also computed the E table explained in chapter 3. The complete time results for computing the E table are shown in Table 4.3. It can be seen how time grows by increasing the resolution due to the use of dynamic programming in the optimal motion planning process.

4.3 SMPs Training on the Simulated Environments

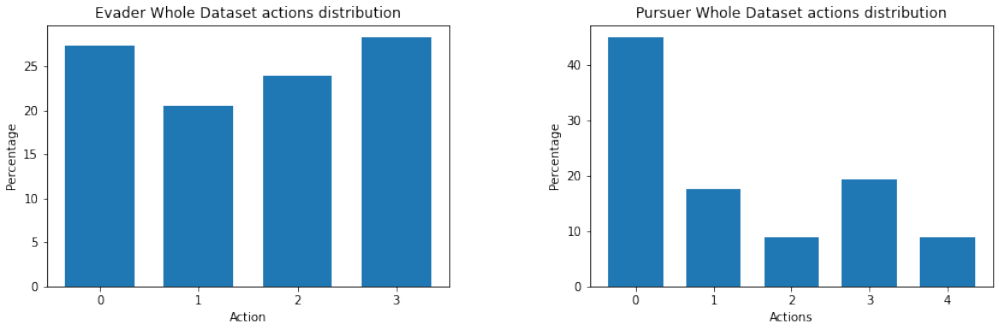
In this section, we present the results over training the ANNs in the SMPs for both players in the *Env1* environment. The results for the rest of the environments, and for the ones with augmented resolutions, can be found in appendix B.

Typically, when training an ANN, samples in the data set are split into a *training set* and a *validation set*. The observations in the first are used for training, as its name suggests, and the ones in the latter are used to compare the ANN performance to check how well is the network doing on generalizing and avoiding overfitting. We propose a random data set partition of 80% for the training set and 20% for the validation set on every environment data. This partition can be seen in Table 4.4 where the amount of samples for every player is shown. In Figure 4.5, the total of observations on each data set are split according to the action (label) assigned to it and shown in form of percentage of observations per action. We can see that, at least for the pursuer's case, this distribution is unbalanced since more than 40% of the observations are assigned with action 0 (*stay*), while observations with actions 2 (*move up*) and 4 (*move down*) correspond to less than 10% each. While training, the *Cross entropy loss* function is used since it is preferred when dealing with unbalanced

train sets, as it is the case for most of the rest of the data sets (see appendix C).

Samples, train and validation partition		
	Evader	**Pursuer**
Total samples	1209	1425
Training samples	967	1140
validation samples	242	285

Table 4.4: Number of total observations, observations for training and validation.



(a) Evader's actions distribution.

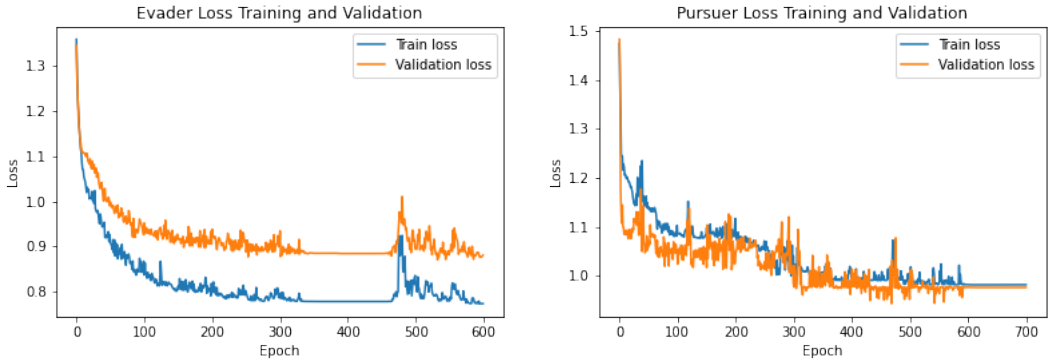
(b) Pursuer's actions distribution.

Figure 4.5: Data set distribution of action labels for both players in *Env1*.

In every training case, a batch size of 256 is fixed and a *learning rate* of value $1e-3$ is chosen. Other aspects like the number of epochs required for training, the optimizer and the results over accuracy and loss values are listed in Table 4.5. In Figure 4.6, the loss function values, for the training and validation sets, along the epochs in training are displayed for both players. It can be seen how this value decreases, meaning that the neural networks weights are well adjusting. Finally, in Figure 4.7, the accuracy values, along the epochs, are shown representing how well are the neural networks doing on learning the right action given a state in both the training and validation sets. This value increases until it settles; at this moment, training is stopped.

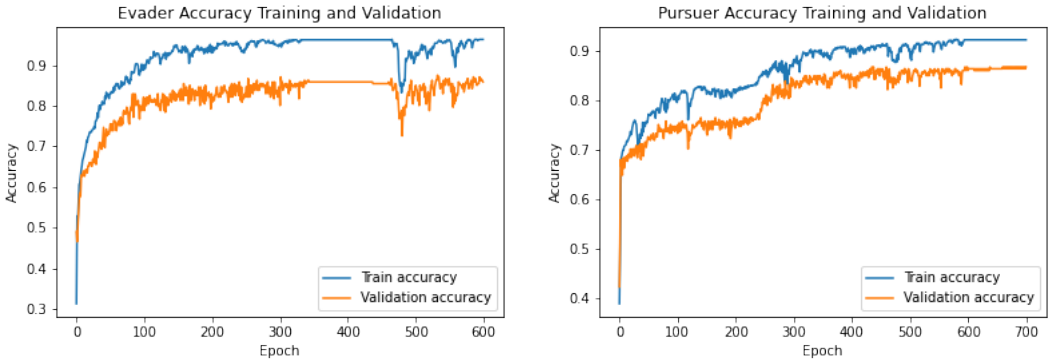
Results at the end of training		
	Evader	**Pursuer**
Epochs	600	700
Optimizer	Adam	Adam
Training time	1min 38.7sec	2min 16.2sec
Latest train accuracy	96.38%	92.11%
Latest validation accuracy	85.89%	86.62%

Table 4.5: Training results for Evader and Pursuer in *Env1* environment.



(a) Evader loss function values. (b) Pursuer loss function values.

Figure 4.6: Evolution of loss function over epochs for the training and validation sets for both players in *Env1*.



(a) Evader accuracy. (b) Pursuer accuracy.

Figure 4.7: Accuracy over epochs for the training and validation sets for both players in *Env1*.

The accuracy metrics in the previous results measure how well a neural network does on learning the right action computed with the DMP given a state. Nevertheless, this measurement is meaningless in the execution of the games, considering a game as a sequence of actions applied by both players, since a player could start in a non-well learned state applying a bad action spoiling the rest of the game, and considering that now the players can start a game in any place in the continuous space inside the workspace $\mathcal{W}$.

As a manner of quantifying the neural networks performance in the simulation of games, we propose comparing the evader's escape length in time steps for several simulations with random initial states in four game scenarios:

- *Evader evaluation*: $SMPevader$ vs $DMPpursuer$.
- *Pursuer evaluation*: $SMPpursuer$ vs $DMPevader$.

- *Evader and Pursuer simultaneous evaluation*: $SMPevader$ vs $SMPpursuer$.
- *Evader and Pursuer simultaneous evaluation with generalization*: $SMPevader$ vs $SMPpursuer$.

In the first three game scenarios, all initial states are coincident to the center of cells in the discrete state-space formulation. In the fourth game scenario, the initial states are perturbed from the center of the cells by adding a 2D Gaussian noise with mean, μ, equals zero and standard deviation, σ, equals $1/(4r)$ to the cell center, where r is the resolution of the environment which could be an integer value in the set $r \in \{1, 2, 3\}$, so 95% of the perturbed initial states will remain inside the cell they were extracted from[3]. We consider cell locations in order to have access to ground truth information, in this case the evader's escape length provided by the table E, and compare the SMPs performance to it.

Since we have access to the total number of states with visibility (from the computed data sets), we can use the formula in equation (4.1), taken from [34], to estimate a sample size and run experiments to check the neural networks performance. This formula gives us an idea about the sample size needed given the total amount of data in a population. Other parameters must be provided, such as the estimation error and the trust level.

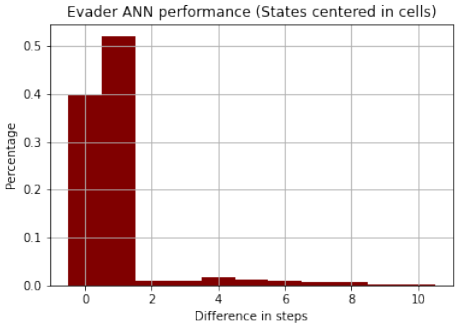
$$m = \frac{NZ^2}{4e^2(N-1) + Z^2} \tag{4.1}$$

In (4.1), m is the total simulations considered in the statistical analysis, N is the total of states in the discrete environment where there exists initial visibility, e is set to 0.05 accepting 5% of estimation error, Z is equal to 1.96 indicating a 95% trust level. In the case of the environment *Env1*, 292 simulations are considered for the evaluations.

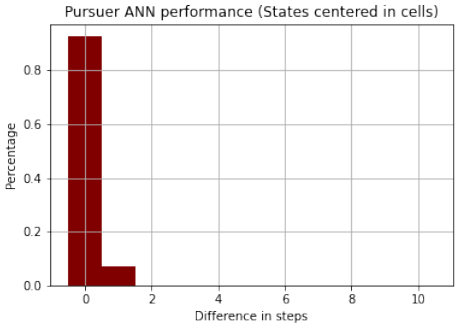
In each of the four game scenarios, the evaluations are presented in two ways: a plot representation and a confusion matrix representation. This is done taking into consideration the evader's escape length achieved with the DMPs as ground truth.

In the plot representation, the horizontal axis shows the difference, in time steps, between the achieved evader's escape length and the one resulting after applying the DMPs from a given initial state. The vertical axis shows the percentage of simulations for each difference in the horizontal axis. The percentage in the first bin (zero difference) means that the escape length using the SMP matches the expected result with the DMP. This comparison analysis is used since the E table only provides information about the evader's escape length given an initial state. In Figure 4.8, we present the results for the environment *Env1*. In the *Evader evaluation* (Figure 4.8a) 40% of the simulations presents the same evader's escape length as if using the DMP_{evader}, while in the *Pursuer evaluation* (Figure 4.8b) nearly 90% of the simulations achieved this length. We can conclude that learning is more favorable for the pursuer $SMP_{pursuer}$. It is harder to tell which of the SMP is better learning

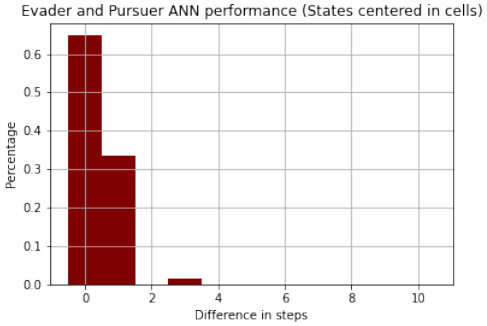
[3]Recall that for the normal distribution, the values less than two standard deviation from the mean account for 95.45% of the distribution.



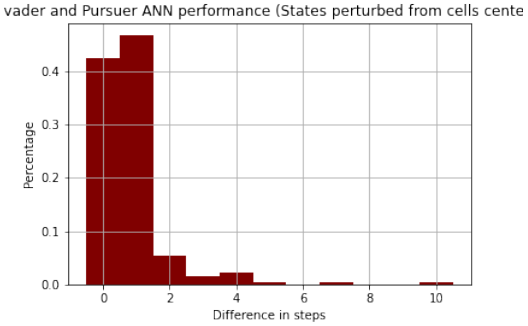
(a) *Evader evaluation.*



(b) *Pursuer evaluation.*



(c) *Evader and Pursuer simultaneous evaluation.*



(d) *Evader and Pursuer simultaneous evaluation with generalization.*

Figure 4.8: SMP performance as a comparison of the evader's escape lengths in different cases.

in the *Evader and Pursuer simultaneous evaluation* cases (Figures 4.8c and 4.8d) since both policies have similar performance. Nevertheless, we can see that mostly the evader's escape length presents a difference of 0 or 1 with respect to the DMPs which can be interpreted as a good performance.

The employed confusion matrices compare whether the evader could or not escape. To this end, the format shown in Table 4.6 is used. Here **X** represents the total simulations where the evader escapes both using the DMPs for both of the players and the combination of policies in the evaluation analysis (*Evader evaluation*, *Pursuer evaluation*, *Evader and Pursuer simultaneous evaluation* and *Evader and Pursuer simultaneous evaluation with generalization*). **Y** is the total simulations where the evader escapes using the DMPs but it does not using the combination of policies in that evaluation. **Z** is the total simulations where the evader escapes applying the combination of policies in the evaluation but it does not while using the DMPs. Finally, **W** represents the number of simulations where the evader does not escape with the DMP neither with the combination of policies in evaluation. It is important to note that in the *Evader evaluation* case, **Z** is always equal to zero since the SMP_{evader} cannot do better than the DMP_{evader} (since it was formulated using optimal motion planning). Similarly, in the *Pursuer evaluation* case, **Y** is equal to zero for the same reason considering the pursuer's SMP and DMP. In an

ideal result, both values **Y** and **Z** would be equal to zero which would mean that there is not difference applying the SMPs or DMPs, however this result is never achieved.

In Tables 4.7, 4.8, 4.9 and 4.10 we present the obtained results for the four evaluation cases considered in the environment *Env1*. The nature of the game is here presented, in the counter diagonal values, where if the pursuer mistakes a single step it can lose visibility affecting substantially its statistics in the simulations where there should be no escape. On the other hand, if the evader makes a mistake the game can continue ending in a later escape. The game is less forgiving for the evader than for the pursuer.

		Evaluation policies	
		Evader escapes	Evader does not escape
DMPs	Evader escapes	**X**	**Y**
	Evader does not escape	**Z**	**W**

Table 4.6: Confusion matrix format comparing the evader's escape.

		SMP_{evader} $DMP_{pursuer}$	
		Evader escapes	Evader does not escape
DMPs	Evader escapes	118	22
	Evader does not escape	0	152

Table 4.7: Results for the case *Evader evaluation*.

		DMP_{evader} $SMP_{pursuer}$	
		Evader escapes	Evader does not escape
DMPs	Evader escapes	140	0
	Evader does not escape	136	16

Table 4.8: Results for the case *Pursuer evaluation*.

		SMP_{evader} $SMP_{pursuer}$	
		Evader escapes	Evader does not escape
DMPs	Evader escapes	138	2
	Evader does not escape	118	34

Table 4.9: Results for the case *Evader and Pursuer simultaneous evaluation*.

		SMP_{evader} $SMP_{pursuer}$	
		Evader escapes	Evader does not escape
DMPs	Evader escapes	132	8
	Evader does not escape	78	74

Table 4.10: Results for the case *Evader and Pursuer simultaneous evaluation with generalization*.

In this chapter, we have presented the architecture of the ANNs inside the SMPs used in the experiments as our first motion policies in the continuous state-space, as well as the manner in which data sets are constructed and a form of disposing a larger amount of data to train these neural networks. Results and evaluations were also presented achieving a considerable learning for a first approximation in the continuous state-space. In the chapters to come, the application of DRL is explained and results over different proposed environments are shown.

Chapter 5

Motion Policies Improvement with Deep Reinforcement Learning

We have covered the tracking problem in a discrete state-space using optimal planning and formulating the DMPs (Deterministic Motion Policies). Then, we have moved to the continuous state-space applying supervised learning formulating the SMPs (Supervised Motion Policies) to attempt to mimic the results in the discrete state-space. In this chapter, we move forward in the continuous state-space and use DRL techniques, both to learn from scratch and to learn from the SMPs in order to enhance the already achieved performance. In particular, we adapt the REINFORCE algorithm[1] to our purposes. We explain these changes in the sections to come. However, we only employ these algorithms to make the pursuer learn while setting a fixed trajectory for the evader as a first approach of employing DRL in our problem and due to the training time. We focus on two different environments, due to the time required for training and evaluation. At the end, we compare the achieved performance using supervised learning and DRL in a given set of initial states on every environment. It is worth to say that we continue considering the actions for every player to be in a discrete action-space: the same formulation proposed in the previous chapter.

5.1 The REINFORCE Algorithm Adaptation

In this section, we explain the adaptations implemented in the REINFORCE algorithm to bring us to our three proposed algorithms: *Policy Gradient with ϵ-greedy exploration* (algorithm 3), *Policy Gradient with ϵ-greedy exploration and initialization* (algorithm 4) and *Policy Gradient with ϵ-Master assistance* (algorithm 5). At the end of each algorithm, a policy is returned conformed by an artificial neural network whose input is a continuous state and output is the action to be applied by the pursuer at that given state. In the first algorithm, training from scratch is implemented getting as result the RMP, for *Reinforced Motion Policy*. In the second algorithm, the weights in the $SMP_{pursuer}$ are used to initialize one of the

[1] Algorithm 1 in chapter 2.

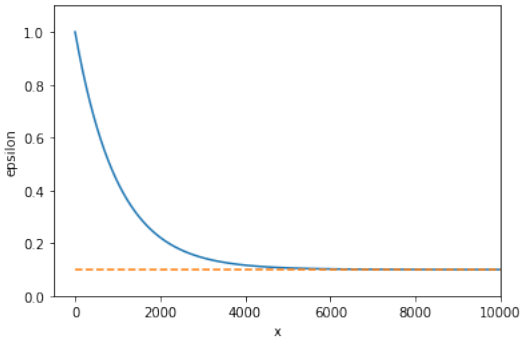


Figure 5.1: Exponential decay graph with $\epsilon_{max} = 1$, $\epsilon_{min} = 0.1$ and $\lambda = 0.001$.

neural networks in the algorithm. After that, the DRL process is executed as in the first approach. The resulting policy is called the IRMP, for *Initialized Reinforced Motion Policy*. In these two algorithms, an exploration-exploitation trade-off is employed, as opposite to the REINFORCED algorithm, to generate the transitions[2] on each trajectory, favoring an uniform exploration in the action-space at the beginning of training and a policy exploitation at the end. In the third algorithm, the $SMP_{pursuer}$ is now used as a master policy to learn by generating transitions with it in the training process. As training goes by, more transitions are generated using the policy still in training and less by the $SMP_{pursuer}$. Uniform exploration is no longer used in this last algorithm. The produced policy is called now MRMP, for *Master Reinforced Motion Policy*.

The way exploration-exploitation in algorithms 3 and 4 is done, is by implementing an epsilon-greedy strategy as in the Q-learning [60] approach. In a similar way, in algorithm 5, we use this epsilon value to determine a transition either from the master policy or from the still learning neural network. In both cases, the epsilon value decreases exponentially from an ϵ_{max} value approaching asymptotically an ϵ_{min} level according to a decay rate value, λ, and the current number of episodes while training. Figure 5.1 shows an example of exponential decay with values $\epsilon_{max} = 1$, $\epsilon_{min} = 0.1$ and $\lambda = 0.001$.

Another modification in the three proposed algorithms with respect to REINFORCE, is the incorporation of an evaluation part during training and a *training policy* network. After the weights in the *training policy* are updated, the policy is evaluated using the best action (the one with highest value in the softmax neural network layer) in every time step within a series of evaluation initial states. Every time a better result occurs as the sum of the obtained rewards in the evaluation, the weights in the *training policy* network are copied into the RMP, IRMP or MRMP network, respectively, returned at the end of every algorithm. This is done to keep the best version of the *training policy* at the end of training. In practice, we use batches in training so the evaluation part comes after a batch is completed and used to update the *training policy* weights.

Algorithm 3 begins with a random initialization of the policy weights θ_{RMP} and

[2]Moving from one state to another.

Algorithm 3 Policy Gradient with ϵ-greedy exploration

1: **Inputs:** Environment, T, α, γ, ϵ_{min}, ϵ_{max}, λ
2: Initialize pursuer policy weights θ_{RMP} randomly
3: Initialize training policy weights θ_{train} randomly
4: $R_{best} = 0$
5: **for** $t = 0$ to T **do**
6: Set players on their initial state, s_0
7: **for** $h = 0$ to $H - 1$ **do**
8: $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-\lambda t}$
9: Generate a random value x from distribution $X \sim \text{Unif}(0, 1)$
10: **if** $x < \epsilon$ **then**
11: Choose an action $a_h(s_h)$ uniformly from $\{a_0, a_1, a_2, a_3, a_4\}$
12: **else**
13: Choose an action $a_h(s_h)$ using $\pi_{\theta_{train}}$
14: **end if**
15: **end for**
16: Collect trajectory $\tau = \{s_0, a_0, r_1, \cdots, s_{H-1}, a_{H-1}, r_H\}$
17: Estimate return: $\mathcal{R}(\tau) = (G_0, G_1, \cdots, G_{H-1})$
18: where $G_k = \sum_{i=k+1}^{H} \gamma^{i-k-1} r_i$
19: Compute loss value: $L(\theta_{train}) \leftarrow \sum_{i=0}^{H-1} \log \pi_{\theta_{train}}(s_i, a_i) G_i$
20: Update train policy weights: $\theta_{train} \leftarrow \theta_{train} + \alpha \nabla_{\theta_{train}} L(\theta_{train})$
21: Generate trajectory $\{s_0, a_0, r_1, \cdots, s_{H-1}, a_{H-1}, r_H\} \sim \pi_{\theta_{train}}$
22: Get trajectory accumulated reward: $R \leftarrow \sum_{i=1}^{H} r_i$
23: Update pursuer policy weights:
24: **if** $R > R_{best}$ **then**
25: $\theta_{RMP} \leftarrow \theta_{train}$
26: $R_{best} \leftarrow R$
27: **end if**
28: **end for**
29: **Return** $\pi_{\theta_{RMP}}$

θ_{train} (lines 2-3) for the RMP and the *training policy*, respectively. A value R_{best} is initialized to 0 (line 4); this value will be used later to update the weights θ_{RMP} in the RMP network. Next, we iterate for the total number of episodes, T, (lines 5-28). We set the players on their initial state s_0 (line 6) from where a trajectory will be collected. For every transition in this trajectory (lines 7-15), the ϵ value is determined and an x value is grabbed from an uniform distribution $X \sim \text{Unif}(0, 1)$ (lines 8-9). If x is lower than ϵ an action is chosen randomly, in an uniform fashion, from the available set of actions on that given state $a_h(s_h)$, otherwise, an action is taken applying the policy $\pi_{\theta_{train}}$ (lines 10-14). This will favor a random action selection at the beginning of training to provide more exploration at this stage, and more policy exploitation at the end as the ϵ value begins to fade. After a trajectory is collected, the returned discounted reward is estimated (lines 16-18), which is used to compute the loss value (line 19) and update the weights θ_{train} applying gradient

ascent (line 20). Once this update is done, a new trajectory is generated (line 21) to evaluate the *training policy* performance by computing the accumulated reward R (line 22). If R is greater than R_{best}, the θ_{train} weights are copied to the θ_{RMP} weights (lines 23-27). This helps us to keep the best current trained policy. At the end of this process, the RMP is returned.

Algorithm 4 Policy Gradient with ϵ-greedy exploration and initialization

1: **Inputs:** Environment, T, α, γ, ϵ_{min}, ϵ_{max}, λ, $\theta_{pretrained}$
2: Initialize pursuer policy weights θ_{IRMP} randomly
3: Initialize training policy weights: $\theta_{train} \leftarrow \theta_{pretrained}$
4: $R_{best} = 0$
5: **for** $t = 0$ to T **do**
6: Set players on their initial state, s_0
7: **for** $h = 0$ to $H - 1$ **do**
8: $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-\lambda t}$
9: Generate a random value x from distribution $X \sim \text{Unif}(0, 1)$
10: **if** $x < \epsilon$ **then**
11: Choose an action $a_h(s_h)$ uniformly from $\{a_0, a_1, a_2, a_3, a_4\}$
12: **else**
13: Choose an action $a_h(s_h)$ using $\pi_{\theta_{train}}$
14: **end if**
15: **end for**
16: Collect trajectory $\tau = \{s_0, a_0, r_1, \cdots, s_{H-1}, a_{H-1}, r_H\}$
17: Estimate return: $\mathcal{R}(\tau) = (G_0, G_1, \cdots, G_{H-1})$
18: where $G_k = \sum_{i=k+1}^{H} \gamma^{i-k-1} r_i$
19: Compute loss value: $L(\theta_{train}) \leftarrow \sum_{i=0}^{H-1} \log \pi_{\theta_{train}}(s_i, a_i) G_i$
20: Update train policy weights: $\theta_{train} \leftarrow \theta_{train} + \alpha \nabla_{\theta_{train}} L(\theta_{train})$
21: Generate trajectory $\{s_0, a_0, r_1, \cdots, s_{H-1}, a_{H-1}, r_H\} \sim \pi_{\theta_{train}}$
22: Get trajectory accumulated reward: $R \leftarrow \sum_{i=1}^{H} r_i$
23: Update pursuer policy weights:
24: **if** $R > R_{best}$ **then**
25: $\theta_{IRMP} \leftarrow \theta_{train}$
26: $R_{best} \leftarrow R$
27: **end if**
28: **end for**
29: **Return** $\pi_{\theta_{IRMP}}$

Algorithm 4 presents essentially the same steps and structure than algorithm 3 except that the *training policy* weights are initialized by taking the weight values from a pretrained neural network, that is another input to the algorithm. This pretrained network is assumed to behave reasonably well in the given environment. In our case, the weights are provided by means of the weights of the $SMP_{pursuer}$ network trained in the last chapter. The resulting policy is called IRMP, for *Initialized Reinforced Motion Policy.*

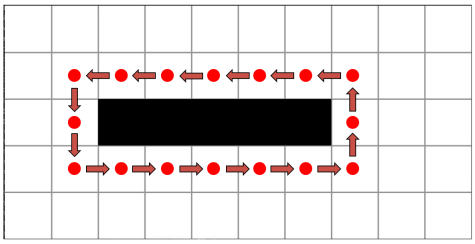
Algorithm 5 Policy Gradient with ϵ-Master assistance

1: **Inputs:** Environment, T, α, γ, ϵ_{min}, ϵ_{max}, λ, $\pi_{\theta_{Master}}$
2: Initialize pursuer policy weights θ_{MRMP} randomly
3: Initialize training policy weights: θ_{train}
4: $R_{best} = 0$
5: **for** $t = 0$ to T **do**
6: Set players on their initial state, s_0
7: **for** $h = 0$ to $H - 1$ **do**
8: $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-\lambda t}$
9: Generate a random value x from distribution $X \sim \text{Unif}(0, 1)$
10: **if** $x < \epsilon$ **then**
11: Choose an action $a_h(s_h)$ using $\pi_{\theta_{Master}}$
12: **else**
13: Choose an action $a_h(s_h)$ using $\pi_{\theta_{train}}$
14: **end if**
15: **end for**
16: Estimate return: $\mathcal{R}(\tau) = (G_0, G_1, \cdots, G_{H-1})$
17: where $G_k = \sum_{i=k+1}^{H} \gamma^{i-k-1} r_i$
18: Compute loss value: $L(\theta_{train}) \leftarrow \sum_{i=0}^{H-1} \log \pi_{\theta_{train}}(s_i, a_i) G_i$
19: Update train policy weights: $\theta_{train} \leftarrow \theta_{train} + \alpha \nabla_{\theta_{train}} L(\theta_{train})$
20: Generate trajectory $\{s_0, a_0, r_1, \cdots, s_{H-1}, a_{H-1}, r_H\} \sim \pi_{\theta_{train}}$
21: Get trajectory accumulated reward: $R \leftarrow \sum_{i=1}^{H} r_i$
22: Update pursuer policy weights:
23: **if** $R > R_{best}$ **then**
24: $\theta_{MRMP} \leftarrow \theta_{train}$
25: $R_{best} \leftarrow R$
26: **end if**
27: **end for**
28: **Return** $\pi_{\theta_{MRMP}}$

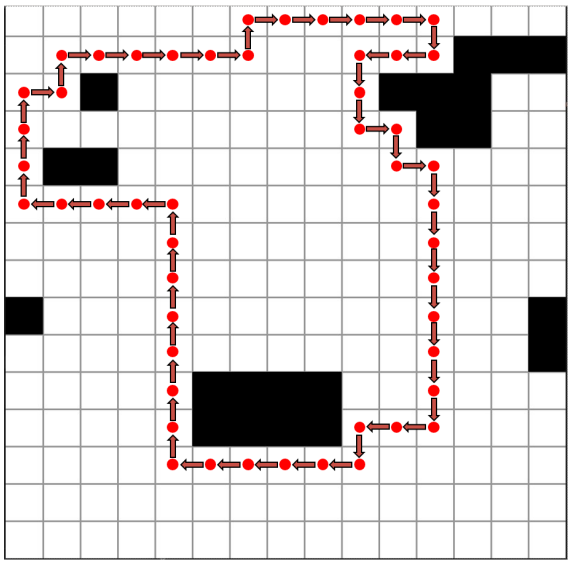
Finally, in algorithm 5, we make a slightly different use of the ϵ value and the pretrained neural network which is called now the *master policy*, π_{master}. The $SMP_{pursuer}$ policy is used to this aim. The rest of the inputs remain the same as in the previous algorithms. It is not necessary to initialize the *training policy* weights, θ_{train}, with specific values, so a random initialization is preferred. The same as for the MRMP weights θ_{MRMP} (line 2). The ϵ value is computed as before, but this time, it is used to determine whether to generate a transition by applying the *master policy*, π_{master}, or the *training policy*, π_{train} (lines 8-14). Again, the returned discounted reward is estimated, which is then used to calculate the loss value and update the θ_{train} weights (lines 16-19). At the end the MRMP is returned, which stands for *Master Reinforced Motion Policy*.

5.2 Training Results and Simulations

Before going into the results and simulations, we must first define the environments where reinforcement learning is to be applied and the evader's trajectories to be executed on each of them. Recall that we are only considering a RL approach for the pursuer, resulting in a pursuer that watches over an evader that follows a fixed trajectory. We consider two training cases based on the environments *Env1* and *Env3*. The proposed trajectories can be appreciated in Figure 5.2, in which the red spots are the locations, at the center of the cells, where the evader can step into and the rows represent the direction from one spot to the next. The evader can be initially placed on any of the red spots The grid in the figures is only for dimension reference.



(a) Evader trajectory in *Env1*.



(b) Evader trajectory in *Env3*.

Figure 5.2: Proposed trajectories for the evader in the RL process.

Even though the evader is located at the center of the cells and its actions correspond to the ones allowed in the discrete state-space formulation, the pursuer is able to be positioned in any place within the environment free-space.

5.2.1 Results in *Env1* using RL

Now that we have defined the evader's behavior in every environment, the next step is to train the pursuer and to visualize its performance over a variety of simulations.

In the case of environment *Env1*, we vary how data is gathered while training in three ways. In the first and second way, we are interested to train the pursuer to track the evader from the same start location, $s_0 = [1.5,\ 1.5,\ 0.5,\ 0.5]$, on each simulation. We first generate all training data from s_0, then we propose a set of hand picked initial states along the evader's route. Finally, in the third case, we propose a generalized version of the problem where the training initial states are randomly generated all over the environment free-space.

Since s_0 corresponds to the centers of a pair of cells, it is possible to establish that from s_0 it is guaranteed that the pursuer is able to keep the evader inside its visibility region for any given time steps duration[3]. Thus, we have a baseline for further comparisons. Considering that, the objective of using RL, in the first training case, is to come up with the pursuer strategy to be able to always maintain the evader inside its visibility region. We apply algorithms 3, 4 and 5 for this goal. In every case, the hyperparameters shown in Table 5.1 were chosen after some trial and error tests that consisted in decreasing the learning rate vale, α, and choosing between setting ϵ_{min} to 0 or to 0.1 to allow a fixed level of exploration at the end of every algorithm.

Hyperparameters in *Env1*	
Total episodes	12000
batch size	12
α	0.0005
γ	0.99
ϵ_{max}	1
ϵ_{min}	0.1
λ	0.001

Table 5.1: Hyperparameters in all training cases for environment *Env1*.

The reward function gives a **+1** value as reward to the pursuer if it is able to maintain the evader inside its visibility region after an action is performed. If visibility is lost, a **0** reward value is given and the episode is marked as complete. In practice, it is necessary to establish a total reward limit so an episode in training will not run forever. In our case, this value is set to be **33**, meaning that the pursuer kept visibility for twice the evader's length path plus 1 and so it is guaranteed that the pursuer can track the evader for any amount of time steps. This is the maximum value we seek to achieve at the test part for all of the algorithms.

Other variation to keep in mind is the way initial states in one batch are collected for training and how the initial states for testing are considered. First, we gather trajectories all starting from $s_0 = [1.5,\ 1.5,\ 0.5,\ 0.5]$ in the training batch, the test are also performed from s_0. Results are shown in Figures 5.3, 5.4 and 5.5 where the

[3]According to the results applying the pursuer *Deterministic Motion Policy* ($DMP_{pursuer}$).

horizontal axis represents the episodes in training and the vertical axis the reward value. Once a batch has been used for training, the total rewards per trajectory are averaged to come up with the blue curve. The black curve is a smooth version to the blue one that takes the average of the ten past rewards, this gives us an idea on how rewards are rising or lowering. The orange curve is the resulting reward at the test part of the algorithm, after applying the best learned actions from s_0.

The time required to complete the 12000 episodes in training for each algorithm is approximately 3 hours. The precise time is shown in Table 5.2.

Time for training	
Algorithm	**Time**
Algorithm 3	3h 9min 22.5sec
Algorithm 4	3h 2min 33.8sec
Algorithm 5	3h 9min 5.5sec

Table 5.2: Time required for training in environment *Env1*.

Figure 5.3: Training results in *Env1* applying algorithm 3.

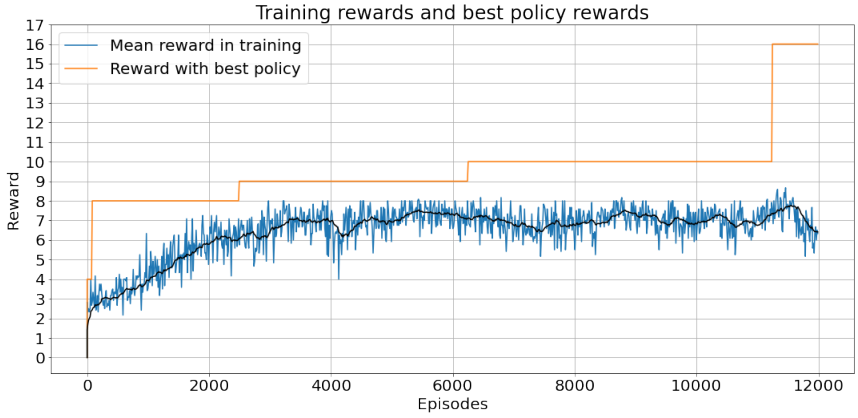


Figure 5.4: Training results in *Env1* applying algorithm 4.

Figure 5.5: Training results in *Env1* applying algorithm 5.

From Figures 5.3, 5.4 and 5.5, we can see that none of the algorithms were able to reach the reward limit (set up to 33) at the end of the training episodes. Nevertheless, it is clear that algorithm 4 achieved the best results based on the reached rewards at the end of training. To show the performance of the result policies, simulations in this link[4] are available. In these videos we can see how a mistaken step in the IRMP made it lost an entire turn around the obstacle while the RMP could only follow the evader half of a turn and the MRMP did not move after a couple of steps.

Moving forward, we propose collecting, in every training batch, trajectories starting from a set of hand-picked initial states, leaving the test trajectories unchanged (starting from s_0). The proposed initial states were chosen such that they provide suitable locations where the pursuer can maintain evader surveillance. Those states can be appreciated in Figure 5.6. Every pair of evader and pursuer locations in this set is distinguished with the same marker, coloring in red the evader's positions and in blue the pursuer's ones. Results using this approach can be seen in Figures 5.7, 5.8 and 5.9. As we can see, the reward limit is only reached using algorithms 4 and 5, namely, the algorithms that leverage on a warm initialization or guided search based on a dynamic programming solution in a discrete state space. Nevertheless, we reach this level earlier with algorithm 5, after about 2000 episodes instead of 4000 with algorithm 4. The simulation videos are available in this link[5]. In these videos we can see how the RMP almost achieved an entire turn around the obstacle while the IRMP and MRMP managed to complete two turns around it proving, in simulation, that taking several initial states in training and using a pretrained policy is useful to achieve favorable results in this environment. The time in training is displayed in Table 5.3, notice that the time needed in algorithm 5 is slightly lower than the one needed for algorithm 4.

[4]https://youtu.be/YrLybLydIuI
[5]https://youtu.be/33WTsCJT6Ks

Time for training	
Algorithm	**Time**
Algorithm 3	3h 2min 33.8sec
Algorithm 4	3h 16min 35.4sec
Algorithm 5	3h 4min 8.3sec

Table 5.3: Time required for training in environment *Env1* with middle initial states for training.

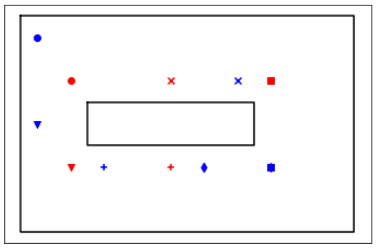


Figure 5.6: Initial states per batch training.

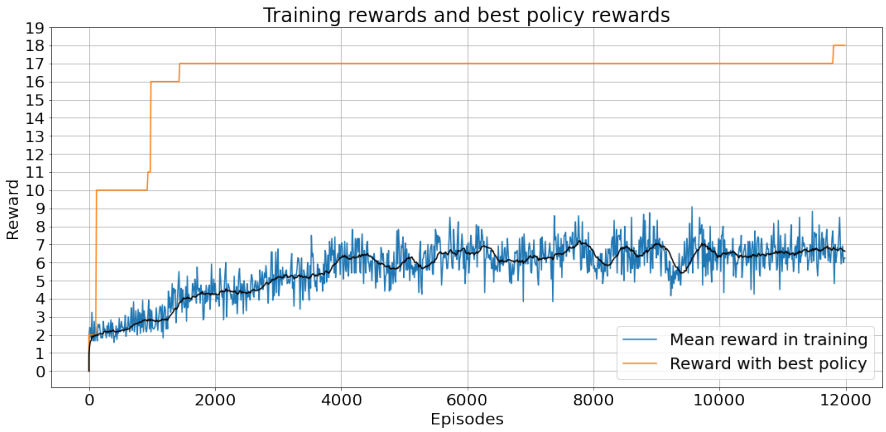


Figure 5.7: Training results in *Env1* applying algorithm 3, using Figure 5.6 initial states for training.

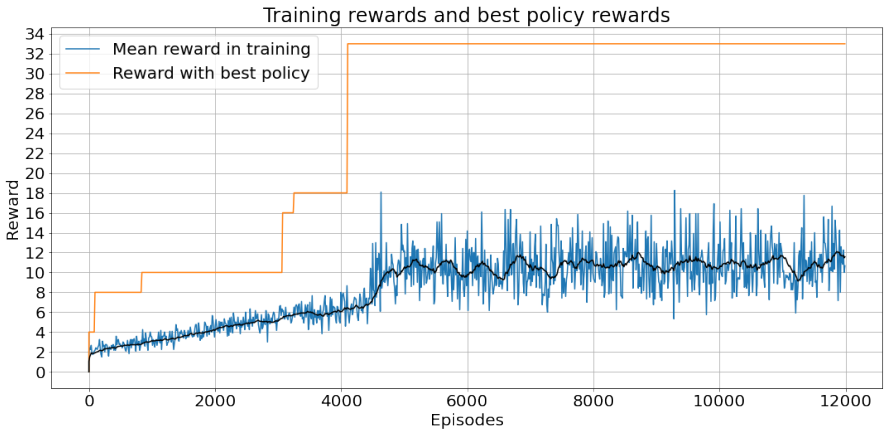
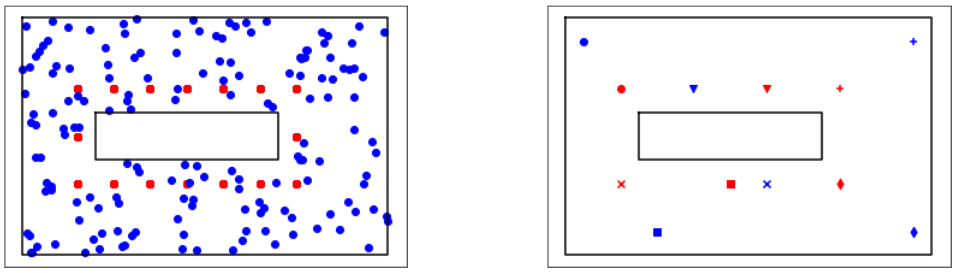


Figure 5.8: Training results in *Env1* applying algorithm 4, using Figure 5.6 initial states for training.

Figure 5.9: Training results in *Env1* applying algorithm 5, using Figure 5.6 initial states for training.

Finally, we generalize this approach by generating a random set of initial states and choosing, randomly again, a subset from it as batch for training. We also propose a set of initial states for the test part in the algorithms. The training and test sets can be seen in Figure 5.10, the red and blue spots are evader and pursuer possible locations, respectively. The total training states are 160. The test locations are coincident with cell centers, being these:

$$
\begin{aligned}
s_{test_1} &= [1.5,\ 1.5,\ 0.5,\ 0.5] \\
s_{test_2} &= [1.5,\ 5.5,\ 1.5,\ 3.5] \\
s_{test_3} &= [1.5,\ 7.5,\ 0.5,\ 9.5] \\
s_{test_4} &= [3.5,\ 7.5,\ 4.5,\ 9.5] \\
s_{test_5} &= [3.5,\ 4.5,\ 4.5,\ 2.5] \\
s_{test_6} &= [3.5,\ 1.5,\ 3.5,\ 5.5]
\end{aligned}
$$



(a) Training initial states.

(b) Test initial states.

Figure 5.10: Training and test initial states for *Env1* in a generalized approach.

By applying the $DMP_{pursuer}$ in the test set of Figure 5.10b we know that the average escape length, with an optimal pursuer policy, from every initial state in the test samples is equal[6] to 20. In the test part of algorithms 3, 4 and 5 we keep track now of the average evader's escape length from every initial state in the test set. Results can be appreciated in Figures 5.11, 5.12 and 5.13. Training time can be seen in Table 5.4. As in the previous case, the time needed for algorithm 5 is lower than the one needed for algorithm 4 suggesting that a guided policy search is preferable for a minor training time.

Time for training	
Algorithm	**Time**
Algorithm 3	9h 12min 34.5sec
Algorithm 4	10h 42.9sec
Algorithm 5	7h 34min 55.9sec

Table 5.4: Time required for training in environment *Env1* with generalization.

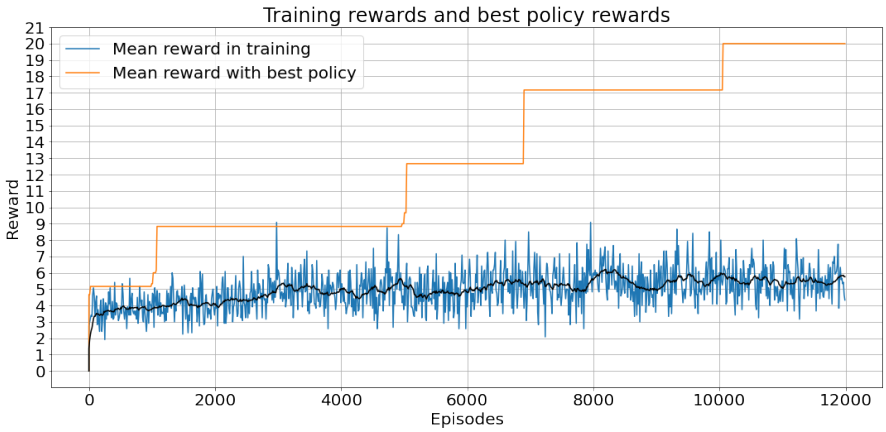


Figure 5.11: Training results in *Env1* applying algorithm 3, using Figure 5.10a initial states for training and 5.10b for testing.

[6]This was found after running simulations from every state in the test set using the fixed trajectory for the evader and the DMP for the pursuer.

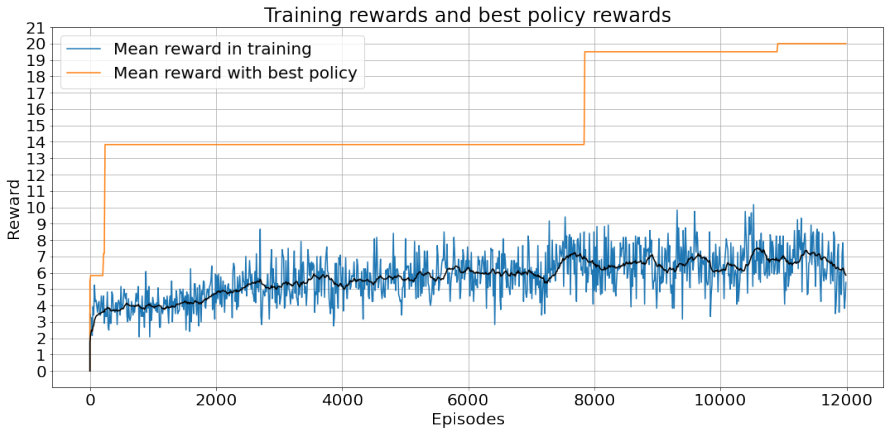


Figure 5.12: Training results in *Env1* applying algorithm 4, using Figure 5.10a initial states for training and 5.10b for testing.

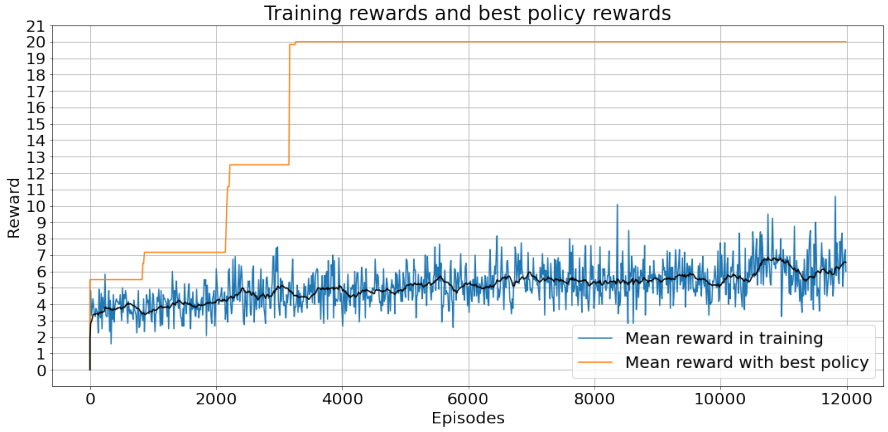


Figure 5.13: Training results in *Env1* applying algorithm 5, using Figure 5.10a initial states for training and 5.10b for testing.

Using this approach, all algorithms reach the maximum reward limit in testing in the given number of episodes. Nevertheless, algorithm 5 does so earlier, at nearly 3300 episodes, while algorithm 4 takes approximately 11000 episodes and algorithm 3 takes 10000.

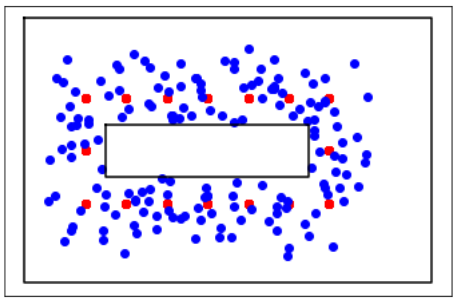


Figure 5.14: Initial states for evaluation in *Env1*.

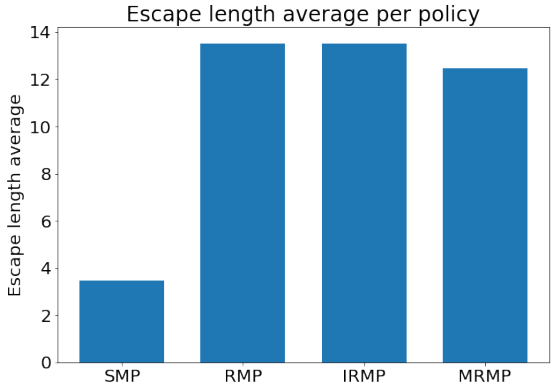


Figure 5.15: Evader's escape length average per policy in evaluation. The values in the graph are 3.46, 13.51, 13.53, 12.45 from left to right.

Additionally, we have implemented a complementary evaluation analysis to compare the SMP, RMP, IRMP and MRMP policies in the generalized training performance (where random initial states are generated to train the policies) over a series of random initial states. In this environment, 160 initial states are chosen such that the distance between the evader and the pursuer is no larger than one unit, this is done so that the pursuer could be able to maintain visibility for most of the time possible. These initial positions can be appreciated in Figure 5.14. Figure 5.15 shows the results of averaging the evader's escape length of every initial position using every one of the policies. It can be appreciated that RMP, IRMP and MRMP outperform the results using SMP and that the value achieved with IRMP is slightly higher than the levels reached with RMP and MRMP. Since the mean values as only comparison risks to be insufficient, we include the box plot[7] comparison displaying the central values and variability on the escape lengths gathered in the evaluation games in order to find out how much these values spread along the minimum and

[7] A box plot is a way to visualize data using five summary values: the minimum, the first quartile (Q1), the median, the third quartile (Q3) and the maximum.

maximum escape lengths identifying possible outliers. In figure 5.16 the SMP, RMP, IRMP and MRMP box plots are displayed in blue, orange, green and red, respectively. We can see, as in figure 5.15, how the RMP, IRMP and MRMP outperform the SMP results. The median in the cases where RL was used are very close one to each other being this value slightly higher in the IRMP results, additionally the variability with this policy is less noticeable. Other very important aspect with the IRMP is the absence of games resulting in escape lengths near the 0 value.

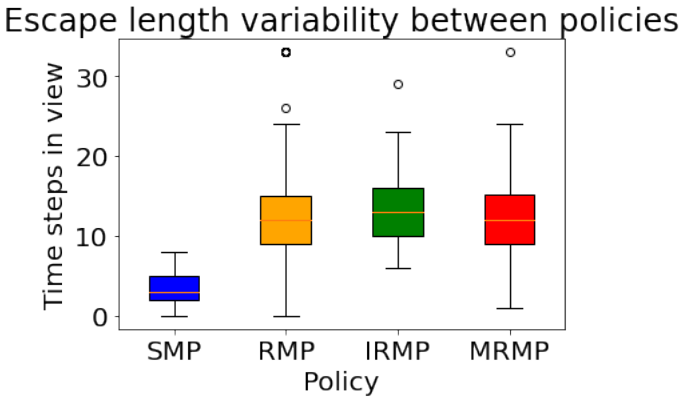


Figure 5.16: Box plot graphs comparing the evaluation results in *Env1*.

In Figure 5.17, a histogram, where the horizontal axis shows the evader's escape length and the vertical axis the total of initial states reaching that escape, compares every policy one next to each other. We can see how similar are the performance reached by the RMP, IRMP and MRMP policies and how poor is the one reached by the SMP. One more thing that can be seen is that only the RMP and the MRMP achieved escapes of value 33 which is the highest possible reward (or time in view) in a game. At this point, it is worth to mention that 19 of the initial states with RMP reached this value compared with only one with the MRMP; however, these 19 initial states were labeled as outliers in the corresponding box plot (see Figure 5.16). In this case a uniform exploration seemed to deliver better results than the use of the SMP to generate transitions. We compare separately the SMP, RMP and IRMP to the MRMP using this histogram fashion graph. These graphs are shown in Figures 5.18, 5.19 and 5.20. In this simple environment it seems that, in the evaluation, it is not necessary to start a reinforce learning with any kind of initialization or assistance since a purely DRL from scratch approach achieved the most of initial states reaching the highest value in reward. Two initial states are shown in simulation in this link[8]. Pursuer and evader in the simulations are considered to be point agents, the figure representing every one of them is used only to distinguish them. In the simulations, only two instances are shown which are not enough to bring up conclusions. That is the reason why we use average values and box plots in the escape length for multiple random initial states in figures 5.15 and 5.16.

[8]https://youtu.be/BMH8vpmWCYw

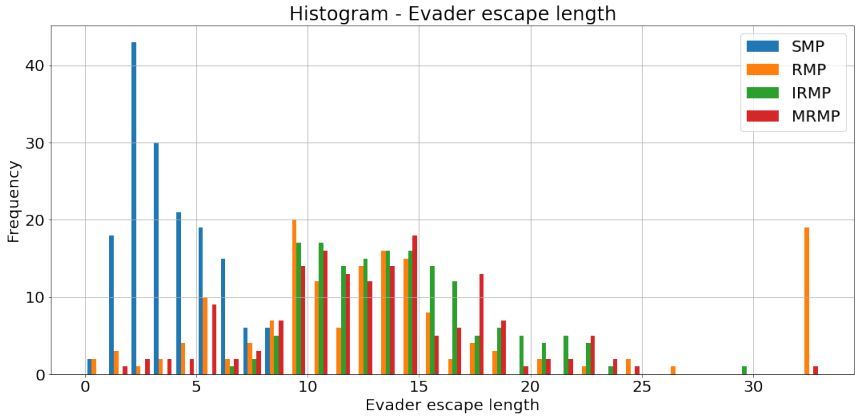


Figure 5.17: Histogram performance for SMP, RMP, IRMP and MRMP in *Env1*.

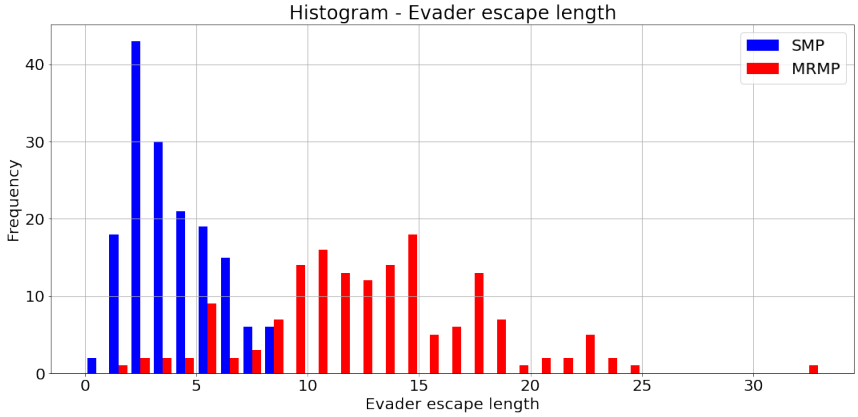


Figure 5.18: SMP and MRMP performance in *Env1*.

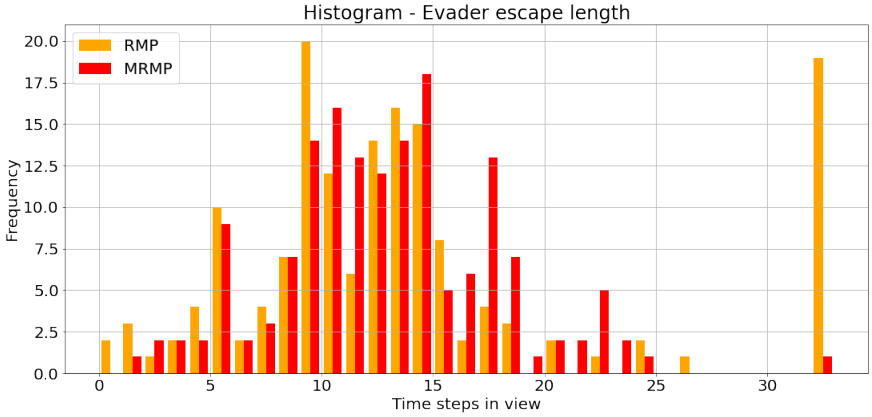


Figure 5.19: RMP and MRMP performance in *Env1*.

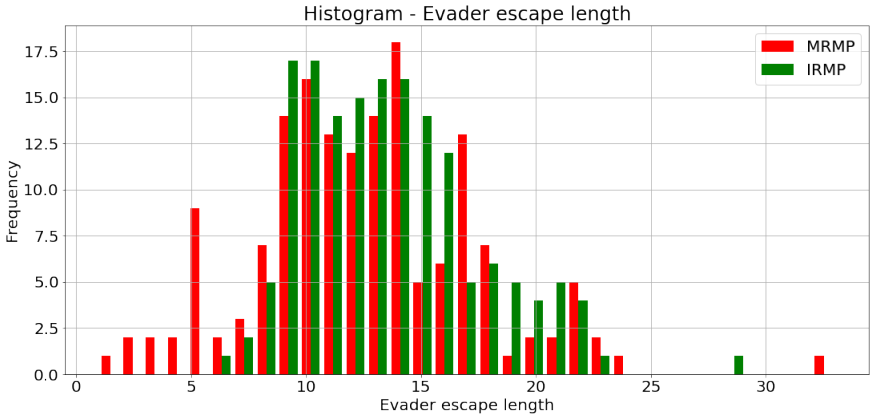
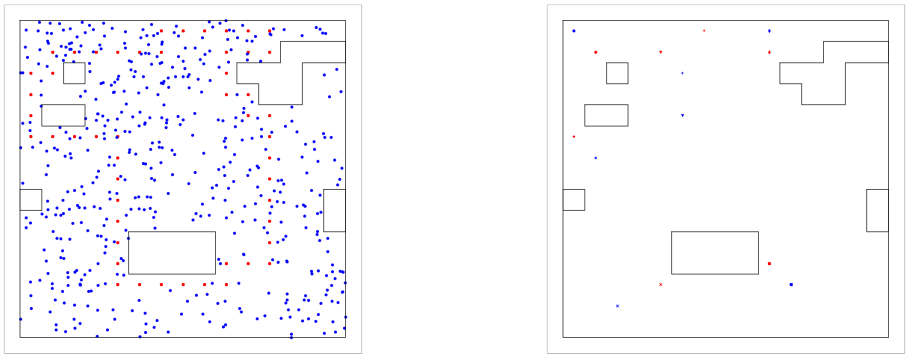


Figure 5.20: IRMP and MRMP performance in *Env1*.

5.2.2 Results in *Env3* using RL

In the *Env3* case, we go directly to a generalized training generating random initial states for training. In the test part, for every RL algorithm, the initial states are listed below. Training and test initial states can be appreciated in figure 5.21. The train initial states sum up 500 states.

$$
\begin{aligned}
s_{test_1} &= [1.5,\ 1.5,\ 0.5,\ 0.5] \\
s_{test_2} &= [1.5,\ 4.5,\ 4.5,\ 5.5] \\
s_{test_3} &= [0.5,\ 6.5,\ 2.5,\ 5.5] \\
s_{test_4} &= [1.5,\ 9.5,\ 0.5,\ 9.5] \\
s_{test_5} &= [11.5,\ 9.5,\ 12.5,\ 10.5] \\
s_{test_6} &= [12.5,\ 4.5,\ 13.5,\ 2.5] \\
s_{test_7} &= [5.5,\ 0.5,\ 6.5,\ 1.5]
\end{aligned}
$$



(a) Training initial states. (b) Test initial states.

Figure 5.21: Training and test initial states for *Env3* in a generalized approach.

Hyperparameters in *Env3*	
Total episodes	8000
batch size	64
α	0.003
γ	0.99
ϵ_{max}	1
ϵ_{min}	0.1
λ	0.001

Table 5.5: Hyperparameters in all training cases for environment *Env3*.

The chosen hyperparameters are shown in Table 5.5. By applying the $DMP_{pursuer}$ in the test set of Figure 5.21b, we know that the expected average evader escape from every initial state in the test samples is set to 101 since the pursuer could maintain surveillance for an unlimited amount of time steps and we limit the time in view in this cases to be at most twice the evader trajectory length plus 1. The training results in this environment are shown in Figures 5.22, 5.23 and 5.24. Here we can see that the only algorithm able to keep surveillance for the entire evader trajectory is algorithm 5, the one that uses the SMP as a master policy. In this more complex environment, the use of a pretrained neural network as a master policy, to perform a guided exploration, threw better results than only using its weights as initializers in the training network and than training a model from scratch keeping some exploration at the end of training. The time required for training in this environment is shown in Table 5.6. Since algorithm 5 achieved better results, it was also the one that required more time for training.

Figure 5.22: Training results in *Env3* applying algorithm 3, using Figure 5.21a initial states for training and 5.21b for testing.

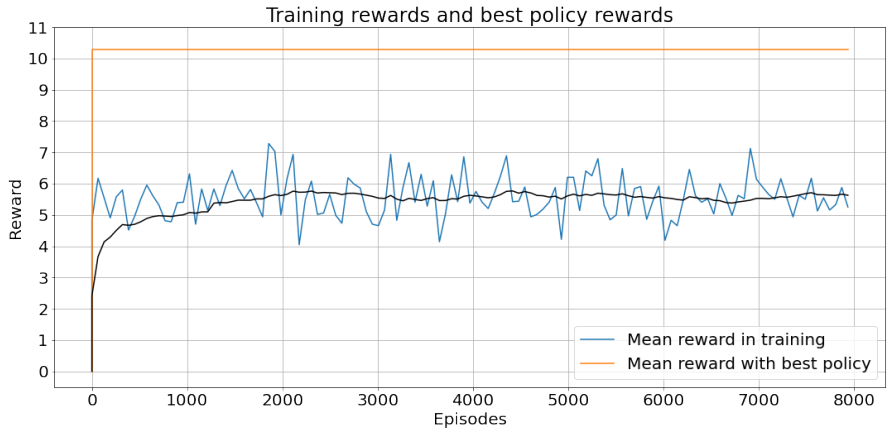


Figure 5.23: Training results in *Env3* applying algorithm 4, using Figure 5.21a initial states for training and 5.21b for testing.

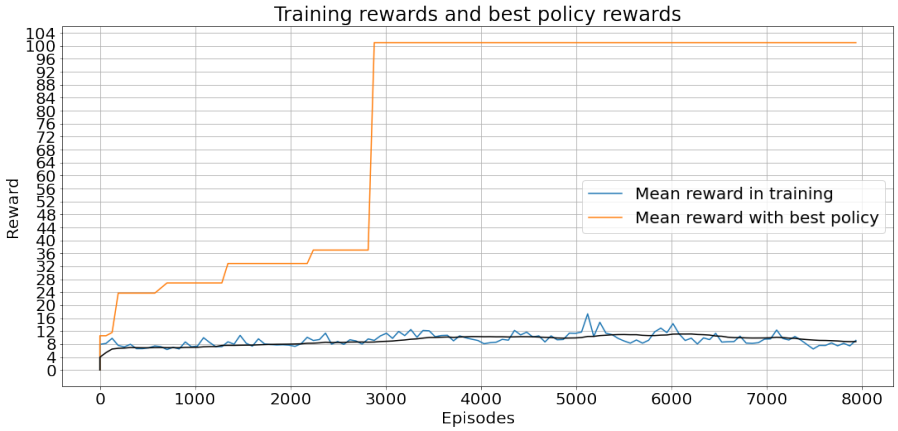


Figure 5.24: Training results in *Env3* applying algorithm 5, using Figure 5.21a initial states for training and 5.21b for testing.

Time for training	
Algorithm	**Time**
Algorithm 3	31h 14min 17.4sec
Algorithm 4	24h 22min 31.9sec
Algorithm 5	43h 33min 23.9sec

Table 5.6: Time required for training in environment *Env3* with generalization.

For *Env3* environment we also run a set of games for evaluation as in the previous environment. We choose randomly a set of 500 initial states such that the initial distance between players is not greater than 3 units, this is done so that a game would not end quickly. This distance only applies for the complementary evaluation.

The mean escape length value on these games can be appreciated in Figure per each policy. Conversely to the results for *Env1* shown in Figure 5.16, Figures 5.25 and 5.26 shows that the MRMP clearly outperforms the SMP, RMP and IRMP. It is worth noting that the MRMP has reached a point in the training in which it has already found the highest possible reward (101) for several initial states. This is noticed in the corresponding box plot (see Figure 5.26) where the third quartile (Q3) mark is on the 101 reward value, meaning that the maximum reward has already been found for at least 25% of the initial states. On the other hand, more training episodes are still needed since the interquartile range is still large; nonetheless, the first quartile (Q1) for the MRMP is above the Q1 of the other three policies.

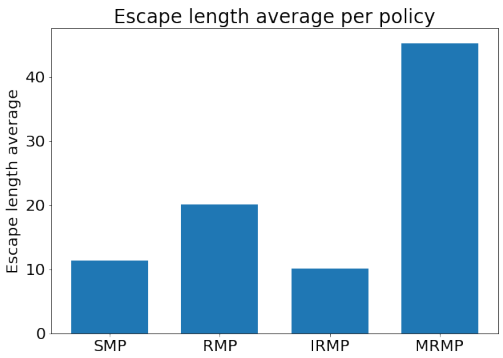


Figure 5.25: Evader's escape length average per policy in *Env3* in evaluation. The values in the graph are 11.31, 20.10, 10.1, 45.28 from left to right.

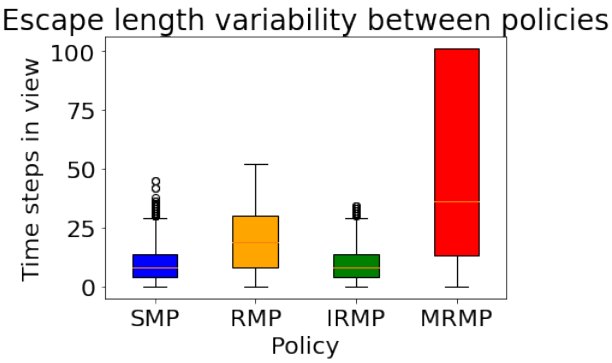


Figure 5.26: Box plot graphs comparing the evaluation results in *Env3*.

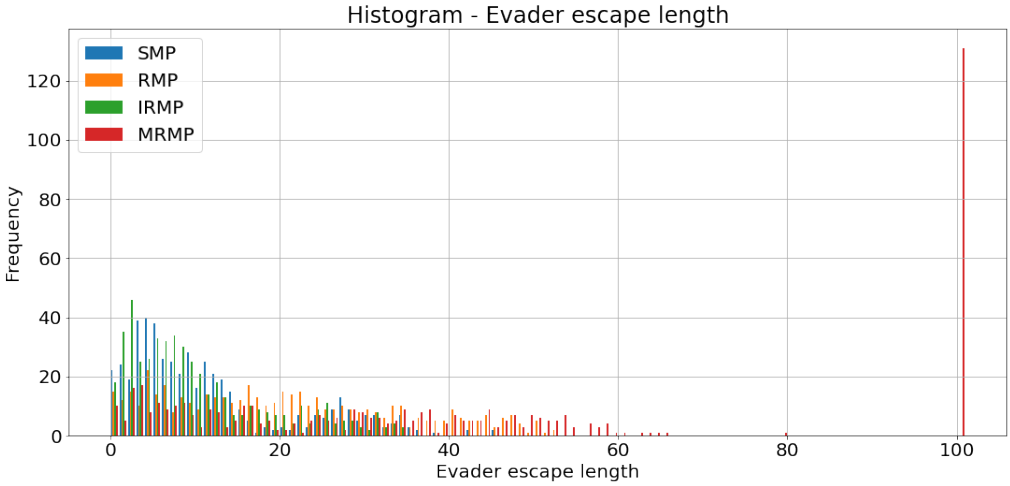


Figure 5.27: Histogram performance for SMP, RMP, IRMP and MRMP in *Env3*.

In Figure 5.27 we can see how the MRMP outperforms the results achieved by the rest of the policies, since it is the only one able to reach the maximum allowed time in view (101 steps). We can also see that the RMP policy achieved a better result with respect to a simple initialization with the IRMP. The results in form of a histogram comparing the MRMP with the rest of the policies are shown in Figures 5.28, 5.29 and 5.30. Two initial states are shown in simulation in this link[9]. As in the simulations in the previous environment, the agents are considered point-like players.

In this more complex environment, the MRMP achieves better results than any other approach while the IRMP does not get any improvements compared to a DRL approach trained from scratch.

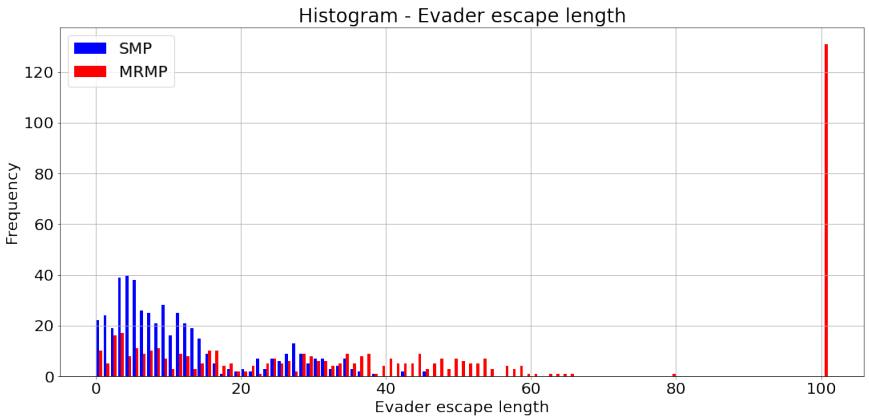


Figure 5.28: SMP and MRMP performance in *Env3*.

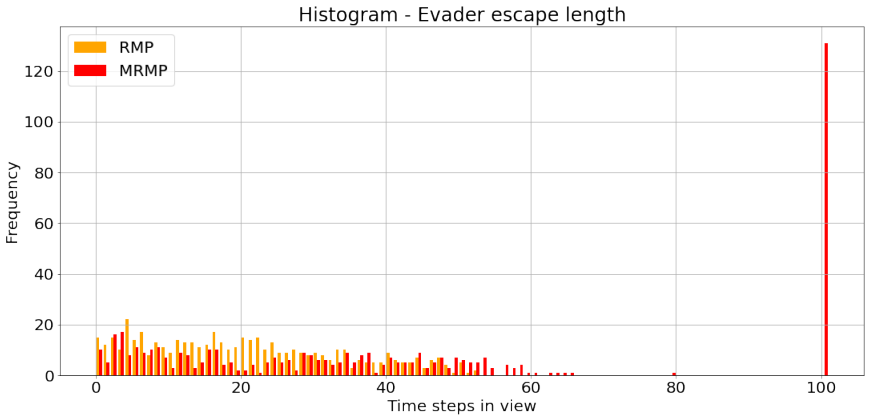


Figure 5.29: RMP and MRMP performance in *Env3*.

[9]https://youtu.be/WNluBWtaF90

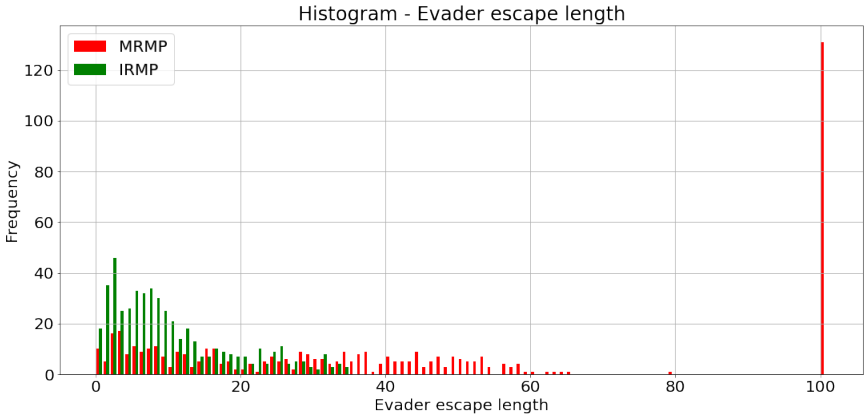


Figure 5.30: IRMP and MRMP performance in *Env3*.

From environment *Env1* we observe how important it is to generate trajectories from several initial states along the environment rather than always from the same initial state even if we want the pursuer only to be able to track the evader from this very location. Training seems to be improved since the RL approach let the pursuer learn how to act in different positions along the evader trajectory given the diversity of the training initial states. Also in *Env1*, we observe good results by applying a RL approach from scratch, which we called RMP. Nevertheless, the algorithms that make use of a pretrained policy achieved better results with a need of less episodes in training: the IRMP and the MRMP. These two policies are clear options to get faster results. Finally, we see that the MRMP in *Env1* shows competitive results in evaluation compared to both the RMP and the IRMP policies, but in *Env3* it exhibits a clear superiority making it the best option in more complex environments.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have worked with the tracking problem under classical visibility formulated as a pursuit-evasion game. Our aim was to design motion policies in the discrete state-space and to move progressively to motion policies in the continuous state-space using deep learning and deep reinforcement learning approaches. We contribute by solving the tracking problem in the discrete state-space with optimal motion strategies for both players with a finite control set each, we next extend the problem to a continuous state-space using artificial neural networks, to finally enhance the pursuer motion policy in the continuous state-space using reinforcement learning both from scratch and using prior information as a warm initialization or as a way to guide policy searching to speed up and improve learning. We present results for different given environments.

Using optimal motion planning in the discrete state-space by means of dynamic programming, we design Deterministic Motion Policies (DMP for short) for both the evader and the pursuer. At the beginning of a game, given the initial state (where a state is formed by the evader and pursuer locations), it is possible to know whether the pursuer is able to keep the evader inside its visibility region indefinitely or not. If it does not, the amount of time steps needed for the evader to escape is returned. In the case where the evader is able to escape, the $DMP_{pursuer}$ is designed such that the pursuer will move as close to the evader as possible. Even though the evader will escape if it applies its optimal policy, placing the pursuer as close to it as possible will set the pursuer in an advantageous position if the evader makes a mistake or if it does not applies its optimal policy, also this plan exhibits a chasing behaviour by the pursuer. We consider discrete actions in the 4-connectivity neighborhood of each player.

The first approach of motion policies in the continuous state-space consists of two artificial neural networks, one network per player in a given environment, whose inputs are continuous states in the environment and the outputs are the discrete action to be applied by the player (moving on its 4-connectivity neighborhood). The neural networks are trained with data gathered from games played in the discrete state-space using the DMPs considering that the player's locations lay in the center

of a cell. Once a neural network is trained it is called an SMP, for *Supervised Motion Policy* since training is done in a supervised fashion. The SMPs try to mimic the DMPs performance but extending to the continuous state-space. Increasing the resolution in the discrete scenario is also proposed to come up with a higher amount of data to train the neural networks.

Finally, we propose the use of deep reinforcement learning as a manner to enhance the motion policies in the continuous state-space only for the pursuer, considering a fixed trajectory for the evader. We present three algorithms, all based on the REINFORCE algorithm from the literature. The resulting policies at the end of each algorithm are called *Reinforced Motion Policy* (RMP), *Initialized Reinforced Motion Policy* (IRMP) and *Master Reinforced Motion Policy* (MRMP), respectively.

The DRL approaches are implemented in a simple environment (*Env1*) and in a more complex one (*Env3*). Comparisons between the generated policies in the continuous state-space are shown, for a random set of initial states. In the simple environment, the RMP and the IRMP policies appeared to have the best performance while in the complex environment, the MRMP outperformed the rest of the policies. Additionally, the IRMP and MRMP showed to achieve acceptable solutions in fewer episodes, hastening the policy training. Thus, utilizing approximate pretrained policies (obtained by applying discrete optimal planning) as a warm initialization or to generate sample trajectories seems adequate to enhance the training of RL approaches in continuous spaces.

6.2 Future Work

With the proposed approach, we have tackled the visibility problem in mobile robotics in simulations; however, there are still several research directions than can be followed.

In chapter 5, we apply DRL to train a pursuer considering an evader with a fixed trajectory. First on, the next step in future work is to train both agents with a DRL approach so that the evader could also come up with its own motion policy. To this end, a min-max RL approach could be employed. Moreover, the trained neural networks in chapter 4 can be used as initializers, in the DRL training, as they were used for the purser's case.

Additionally, all of the environments proposed in this project present a deterministic behavior in the sense that there is not stochasticity in the applied actions by the agents. Opposite to stochastic environment models [17] where, when an action is said to be played by an agent, it is only applied with a certain probability less than 1. This behavior allows to mimic, in some way, the uncertainty of actions in real-world scenarios. Training and simulations in this kind of environments are remaining to be considered by our approaches.

Continuous actions could also be considered in future work allowing both players to have a wider action field. One possibility is to express an action as a vector in polar coordinates in the form (r, φ) whose reference frame correspond to each agent current location at time step t. In this continuous action-space proposal, $r \in [0, r_{max}]$

is a step length bounded by a maximum admissible value r_{max} and $\varphi \in [0, 2\pi)$ is a direction. Furthermore, φ could be adapted to nonholonomic robot constraints such like DDR (differential drive robot) or car-like robots.

This approach can be achieved by first considering an 8-connectivity[1] neighborhood in the discrete optimal planning formulation, and then compute a data set for the artificial neural networks training part, whose inputs are the evader and pursuer locations and the output is the tuple (r, φ). According to each location in the neighborhood, the φ value is determined as in Figure 6.1. In this case, the ANN's are designed to solve a regression problem since we want r and φ to lay within a range of continuous values. Finally, DRL could be applied to improve the trained ANN's. When considering a continuous action-space, reinforcement learning algorithms other than Policy gradient can be used, such as PPO (Proximal Policy Approximation) [47], DDPG (Deep Deterministic Policy Gradient) [50] or even TD3 (Twin Delayed DDPG) [12]. Both the DDPG and TD3 algorithms compute internally Q-values via a neural network, as an initialization to it, a neural network could be trained to store the values in the E table computed in the discrete state-space part of the methodology, since both the Q-values and E table are value functions.

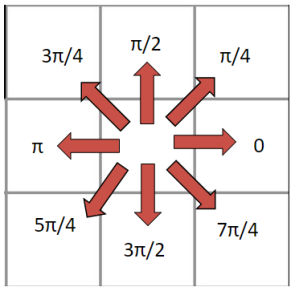


Figure 6.1: φ values according to the action direction in an 8-connectivity neighborhood.

One variation to the problem is to have a visibility region bounded by a range radius going from the pursuer location, which is closer to real world scenarios. In this case, the problem can be formulated considering relative coordinates instead of absolute coordinates as we have done in our approaches. Every player motion strategy considers its own coordinate system. Later on, Convolutional Neural Networks (CNN) can be used to learn features inside the pursuer visibility region for every possible location. This can lead to a generalization of the solution allowing to apply the motion policy learned in one environment into a completely new one.

Finally, this approaches can be tested in real-world scenarios, such like surveillance applications, by adapting the solutions according to the software and hardware availability, mainly the pursuer and evader configurations and the visibility signals perception and image processing. Once the tracking problem is solved in a planar environment, we can also move to a 3D formulation and end up applying the results to a drone as the pursuer.

[1] If an agent is in a cell with coordinates (i, j) at time step t it can move to any valid cell $(i \pm 1, j)$, $(i, j \pm 1)$ or $(i \pm 1, j \pm 1)$ in time $t + 1$.

Bibliography

[1] M. A. Akhloufi, S. Arola, and A. Bonnet. Drones chasing drones: Reinforcement learning and deep search area proposal. *Drones*, 3(3), 2019.

[2] M. F. Argerich, J. Fürst, and B. Cheng. Tutor4rl: Guiding reinforcement learning with external knowledge. In *AAAI Spring Symposium: Combining Machine Learning with Knowledge Engineering*, 2020.

[3] S. Arora, N. Cohen, and E. Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. *Proceedings of the* 35^{th} *International Conference on Machine Learning Stockholm, Sweden*, PMRL 80:244 – 253, 2018.

[4] T. Başar and G. Olsder. Dynamic non-cooperative game theory. 160, 01 1999.

[5] I. Becerra, R. Murrieta-Cid, R. Monroy, S. Hutchinson, and J.-P. Laumond. Maintaining strong mutual visibility of an evader moving over the reduced visibility graph. *Autonomous Robots*, 40(2):395 – 423, 2016.

[6] S. Bhattacharya and S. Hutchinson. On the existence of nash equilibrium for a two-player pursuit—evasion game with visibility constraints. *The International Journal of Robotics Research*, 29(7):831–839, 2010.

[7] S. Bhattacharya and S. Hutchinson. A cell decomposition approach to visibility-based pursuit evasion among obstacles. *Int. J. Rob. Res.*, 30(14):1709?1727, Dec. 2011.

[8] S. Bhattacharya, S. Hutchinson, and T. Basar. Game-theoretic analysis of a visibility based pursuit-evasion game in the presence of obstacles. In *2009 American Control Conference*, pages 373–378, 2009.

[9] D. Chen, Y. Wei, L. Wang, C. S. Hong, L.-C. Wang, and Z. Han. Deep reinforcement learning based strategy for quadrotor uav pursuer and evader problem. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2020.

[10] T. Chung, G. Hollinger, and V. Isler. Search and pursuit-evasion in mobile robotics. *Auton. Robots*, 31, 11 2011.

[11] S. F. Desouky and H. M. Schwartz. Self-learning fuzzy logic controllers for pursuit-evasion differential games. *Robotics and Autonomous Systems*, 59(1):22–33, 2011.

[12] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. 02 2018.

[13] A. Gruslys, M. G. Azar, M. G. Bellemare, and R. Munos. The reactor: A sample-efficient actor-critic architecture. *CoRR*, abs/1704.04651, 2017.

[14] L. Guibas, J. Latombe, S. Lavalle, D. Lin, and R. Motwani. A visibility-based pursuit-evasion problem. *International Journal of Computational Geometry and Applications*, 9(4-5):471–493, 1999.

[15] X. Guo, H. Liu, and H. Dai. Mobile robot target tracking system based on deep learning. In *Mobile Robot Target Tracking System Based on Deep Learning*. CSP, 2020.

[16] T. Hester, M. Vecerík, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. P. Agapiou, J. Z. Leibo, and A. Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[17] S.-M. Hung and S. N. Givigi. A q-learning approach to flocking with uavs in a stochastic environment. *IEEE Transactions on Cybernetics*, 47(1):186–197, 2017.

[18] J. Hurwitz and D. Kirsh. *Machine Learning For Dummies, IBM Limited Edition*. Jhon Wiley & Sons, Inc., Hoboken, NJ, 2018.

[19] R. Isaacs. *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*. John Wiley & Sons, New York, NY, 1965.

[20] Y. Ishiwaka, T. Sato, and Y. Kakazu. An approach to the pursuit problem on a heterogeneous multiagent system using reinforcement learning. *Robotics and Autonomous Systems*, 43:245–256, 06 2003.

[21] V. Isler, C. Belta, K. Daniilidis, and G. Pappas. Hybrid control for visibility-based pursuit-evasion games. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 2, pages 1432–1437 vol.2, 2004.

[22] M. L. Iuzzolino, M. E. Walker, and D. Szafir. Virtual-to-real-world transfer learning for robots on wilderness trails. *CoRR*, abs/1901.05599, 2019.

[23] K. Kersandt, G. Mu?oz, and C. Barrado. Self-training by reinforcement learning for full-autonomous drones of the future*. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2018.

[24] Y.-H. Kim, J.-I. Jang, and S. Yun. End-to-end deep learning for autonomous navigation of mobile robot. In *2018 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6, 2018.

[25] J. Kober, J. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013.

[26] V. Konda and J. Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 2000.

[27] S. LaValle, H. Gonzalez-Banos, C. Becker, and J.-C. Latombe. Motion strategies for maintaining visibility of a moving target. In *Proceedings of International Conference on Robotics and Automation*, volume 1, pages 731–736 vol.1, 1997.

[28] S. M. LaValle. *Discrete Planning*, pages 28 – 76. Cambridge University Press, Cambridge, U.K., 2006.

[29] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies, 2016.

[30] S. H. Lim, T. Furukawa, G. Dissanayake, and H. Durrant-Whyte. A time-optimal control strategy for pursuit-evasion games problems. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 4, pages 3962–3967 Vol.4, 2004.

[31] I. Loshchilov and F. Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[32] W. Luo, P. Sun, F. Zhong, W. Liu, T. Zhang, and Y. Wang. End-to-end active object tracking and its real-world deployment via reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(6):1317–1332, 2020.

[33] V. Macias, I. Becerra, R. Murrieta-Cid, H. M. Becerra, and S. Hutchinson. Image feedback based optimal control and the value of information in a differential game. *Automatica*, 90:271–285, 2018.

[34] C. Martínez and E. Ediciones. *Estadística y muestreo - 13ra Edición*. Ciencias exactas. Estadística. Ecoe Ediciones, 2012.

[35] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[36] R. Murrieta-Cid, T. Muppirala, A. Sarmiento, S. Bhattacharya, and S. Hutchinson. Surveillance strategies for a pursuer with finite sensor range. *The International Journal of Robotics Research*, 26(3):233–253, 2007.

[37] R. Murrieta-Cid, U. Ruiz, J. Marroquin, J.-P. Laumond, and S. Hutchinson. Tracking an omnidirectional evader with a differential drive robot. *Auton. Robots*, 31:345–366, 11 2011.

[38] R. Murrieta-Cid, B. Tovar, and S. Hutchinson. A sampling-based motion planning approach to maintain visibility of unpredictable targets. *Autonomous Robots*, 19:285–300, 2005.

[39] K. Nagami and M. Schwager. Hjb-rl: Initializing reinforcement learning with optimal control policies applied to autonomous drone racing. In *Robotics: Science and Systems*, 2021.

[40] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv e-prints*, page arXiv:1811.03378, Nov. 2018.

[41] N. Papanikolopoulos, P. Khosla, and T. Kanade. Visual tracking of a moving target by a camera mounted on a robot: a combination of control and vision. *IEEE Transactions on Robotics and Automation*, 9(1):14–35, 1993.

[42] T. D. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. R. Lick, editors, *Theory and Applications of Graphs*, pages 426–441, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[43] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[44] S. Ruder. An overview of gradient descent optimization algorithms, 2016. arXiv preprint arXiv:1609.04747.

[45] U. Ruiz. A game of surveillance between an omnidirectional agent and a differential drive robot. *International Journal of Control*, 0(0):1–13, 2021.

[46] U. Ruiz, R. Murrieta-Cid, and J. Marroquin. Time-optimal motion strategies for capturing an omnidirectional evader using a differential drive robot. *IEEE Transactions on Robotics*, 29:1180–1196, 06 2013.

[47] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017.

[48] J. Selvakumar and E. Bakolas. Min-max q-learning for multi-player pursuit-evasion games, 2020.

[49] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[50] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[51] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017.

[52] E. Staffetti, X. Li, Y. Matsuno, and M. Soler. Optimal control techniques in aircraft guidance and control. *International Journal of Aerospace Engineering*, 2019:1–2, 08 2019.

[53] R. S. Sutton and A. G. Barto. *Reinforcement Learning an Introduction, 2nd edition*. The MIT Press, Cambridge, Massachusetts, USA, 2018.

[54] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 2000.

[55] O. Tekdas and V. Isler. Robotic routers. In *2008 IEEE International Conference on Robotics and Automation*, pages 19 – 23, Pasadena, CA, USA, 2008.

[56] O. Tekdas, Y. Wei, and V. Isler. Robotic routers: Algorithms and implementation. *The International Journal of Robotics Research*, 29(1):110 – 126, 2010.

[57] B. Tovar and S. LaValle. Visibility-based pursuit - evasion with bounded speed. *I. J. Robotic Res.*, 27:1350–1360, 01 2008.

[58] V. Turetsky and J. Shinar. Missile guidance laws based on pursuit–evasion game formulations. *Automatica*, 39(4):607–618, 2003.

[59] M. Wang, L. Wang, and T. Yue. An application of continuous deep reinforcement learning approach to pursuit-evasion differential game. In *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 1150–1156, 2019.

[60] C. J. Watkins and P. Dayan. Q-learning. Kluwer Academic Publishers, 1992.

[61] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[62] S. Ying. An overview of overfitting and its solutions. *Journal of Physics Conference Series*, 1168(2):022022, 2019.

[63] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[64] Y. Zhang, D. Balkom, and H. Li. Towards physically safe reinforcement learning under supervision, 2019. arXiv preprint arXiv:1901.06576.

[65] J. Zhu, W. Zou, and Z. Zhu. Learning evasion strategy in pursuit-evasion by deep q-network. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 67–72, 2018.

Appendices

Appendix A

Derivation of the Policy Gradient Expression

We want to derive the gradient of the objective function in the Policy Gradient RL approach:

$$\nabla J(\theta) = \nabla \mathbb{E}_{\pi_\theta} [G(\tau)] \tag{A.1}$$

Lets recall the definition to the expectation of a discrete random variable X as the sum of the product of every event x times the probability of its occurrence $P(x)$.

$$\mathbb{E}(X) = \sum_{x \in X} xP(x)$$

Now we can write the gradient in equation (A.1) as:

$$\nabla J(\theta) = \nabla \mathbb{E}_{\pi_\theta} [G(\tau)] = \nabla_\theta \sum_\tau P(\tau|\theta) G(\tau) \tag{A.2}$$

where $P(\tau|\theta)$ is the probability of a trajectory given the parameter θ.

Bringing the gradient operator under the summation and multiplying and dividing by $P(\tau|\theta)$ we can rearrange:

$$\nabla \mathbb{E}_{\pi_\theta} [G(\tau)] = \sum_\tau P(\tau|\theta) \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} G(\tau) \tag{A.3}$$

Recalling the *log* derivative of a variable z

$$\nabla_\theta log(z) = \frac{\nabla_\theta z}{z}$$

we can write:

$$\nabla \mathbb{E}_{\pi_\theta} [G(\tau)] = \sum_\tau P(\tau|\theta) \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} G(\tau) = \mathbb{E}_{\pi_\theta} [\nabla_\theta log\ P(\tau|\theta) G(\tau)] \tag{A.4}$$

The probability $P(\tau|\theta)$ can be treated as a marginalization as follows:

$$P(\tau|\theta) = p(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t) \pi_\theta(a_t, s_t) \tag{A.5}$$

where $p(s_0)$ is the probability of starting at the state s_0 and $P(s_{t+1}|s_t, a_t)$ represents the transition probability of reaching the new state s_{t+1} by applying the action a_t at the state s_t.

After taking the log-probability of the trajectory in (A.5), we get:

$$log\ P(\tau|\theta) = log\ p(s_0) + \sum_{t=0}^{T-1} [log\ P(s_{t+1}|s_t, a_t) + log\ \pi_\theta(a_t, s_t)] \tag{A.6}$$

We can apply the gradient in expression (A.6) which gives us:

$$\nabla_\theta log\ P(\tau|\theta) = \nabla_\theta log\ p(s_0) + \sum_{t=0}^{T-1} [\nabla_\theta log\ P(s_{t+1}|s_t, a_t) + \nabla_\theta log\ \pi_\theta(a_t, s_t)] \tag{A.7}$$

Since $p(s_0)$ and $P(s_{t+1}|s_t, a_t)$ does not depend on the parameter θ, we can get rid of them after the gradient, thus we can write:

$$\nabla_\theta log\ P(\tau|\theta) = \sum_{t=0}^{T-1} \nabla_\theta log\ \pi_\theta(a_t, s_t) \tag{A.8}$$

Now we can replace (A.8) in (A.4):

$$\nabla \mathbb{E}_{\pi_\theta} [G(\tau)] = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta log\ \pi_\theta(a_t, s_t) G(\tau) \right] \tag{A.9}$$

Recalling that:

$$\nabla_\theta J(\theta) = \nabla \mathbb{E}_{\pi_\theta} [G(\tau)] \tag{A.10}$$

We conclude

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta log\ \pi_\theta(a_t, s_t) G(\tau) \right] \tag{A.11}$$

Appendix B

Other tested ANN architectures

Accuracy over different NN Architectures (%)											
Env	Agent	Mode	Architectures								
			4-5-5	4-10-5	4-50-5	4-5-10-5	4-10-10-5	4-10-50-5	4-5-10-5-5	4-10-50-10-5	4-20-100-20-5
Env1	Evader	Train	39.38	40.24	50.98	47.28	45.80	42.03	39.87	39.69	40.55
		Test	41.83	39.10	51.23	44.80	43.56	40.84	40.09	41.33	37.12
	Pursuer	Train	72.71	71.54	70.98	71.54	71.35	73.45	71.60	72.46	71.84
		Test	67.32	73.76	75.74	72.02	72.77	68.06	71.78	68.31	72.77
Env2	Evader	Train	69.03	76.32	75.69	65.23	66.62	76.74	60.95	62.05	76.08
		Test	66.57	77.34	75.91	65.46	66.06	76.60	61.44	64.40	75.86
	Pursuer	Train	70.90	85.33	88.48	71.89	71.60	87.26	71.81	85.67	88.92
		Test	71.70	82.10	87.93	70.68	71.84	87.42	71.01	86.13	89.55
Env3	Evader	Train	35.23	42.65	52.65	38.95	48.24	49.82	32.69	49.85	56.46
		Test	34.88	43.38	52.78	38.51	49.39	49.92	33.83	50.39	56.69
	Pursuer	Train	53.33	60.26	69.82	58.60	62.92	66.07	55.95	68.00	72.39
		Test	52.99	60.02	68.46	56.35	62.95	67.18	55.70	68.60	72.12
Env4	Evader	Train	82.21	83.23	84.04	82.40	83.24	84.03	82.08	82.51	83.85
		Test	82.40	83.82	83.31	82.97	822.95	83.24	82.92	82.38	82.27
	Pursuer	Train	90.86	90.63	90.69	90.85	90.85	90.67	90.56	90.46	90.78
		Test	90.00	90.94	90.49	90.04	90.06	90.33	91.19	91.62	90.31

Appendix C

Artificial Neural Networks Training Results - Continuation

In this appendix, we continue presenting the obtained results over training the SMPs for both players, as first presented in section 4.3. Here we present the results for the environments *Env1* with resolutions 2 and 3, *Env2* with resolutions 1, 2 and 3, *Env3* with resolutions 1 and 2, and finally *Env4* with resolutions 1 and 2[1]. As exposed in section 4.3, we first show the results in the training phase and then the performance of the trained SMPs in actual games simulations according to the same four cases as in section 4.3. These cases are recalled in form of list below:

- *Evader evaluation*
- *Pursuer evaluation*
- *Evader and Pursuer simultaneous evaluation*
- *Evader and Pursuer simultaneous evaluation with generalization*

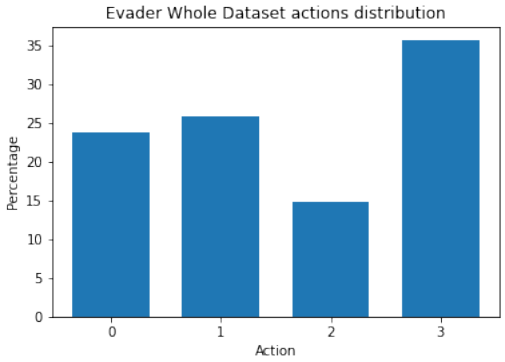
C.1 Training results

In this section, results over the neural networks training, inside the SMPs, are presented. Remember that we are proposing two SMPs per environment, one for the evader and the other for the pursuer. In the table below, the number of samples used for training and validation are displayed. In the next pages, the actions distribution in each data set are shown in a graphical manner representing the percentage of samples (in the vertical axis) in the data set that correspond to the action labeled in the horizontal axis.

[1]As resolution grows, the time needed to compute the E table increases. That is the reason why we keep only these resolutions.

Samples, train and validation partition			
	Total	Test	Validation
Env1 res 2			
Evader	19232	1585	3847
Pursuer	21052	16841	4211
Env1 res 3			
Evader	97237	77789	19448
Pursuer	103497	82797	20700
Env2			
Evader	4200	3360	840
Pursuer	4744	3795	949
Env2 res 2			
Evader	67480	53984	13496
Pursuer	72084	57667	14417
Env2 res 3			
Evader	342684	274147	68537
Pursuer	358556	286844	14417
Env3			
Evader	27195	21756	5439
Pursuer	31027	24821	6206
Env3 res 2			
Evader	429806	343844	85962
Pursuer	464801	371840	92961
Env4			
Evader	3692	2953	739
Pursuer	5028	4022	1006
Env4 res 2			
Evader	55994	44795	11199
Pursuer	69620	55696	13924

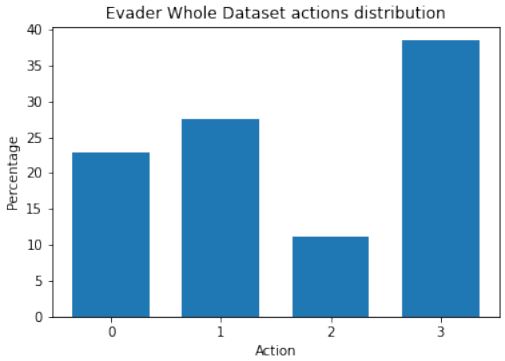
Total of observations and its partition in train and validation sets for every environment.



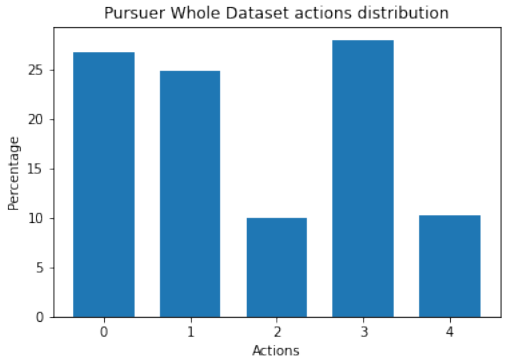
Evader's actions distribution.

Pursuer's actions distribution.

Data set distribution of action labels for both players in *Env1 res 2*.

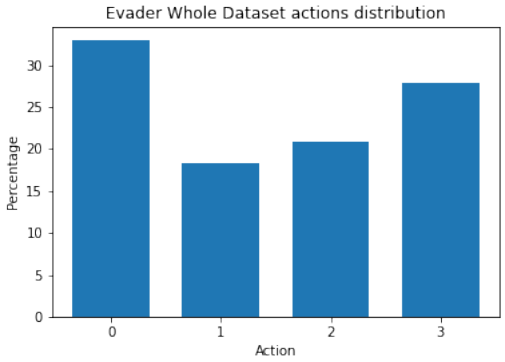


Evader's actions distribution.



Pursuer's actions distribution.

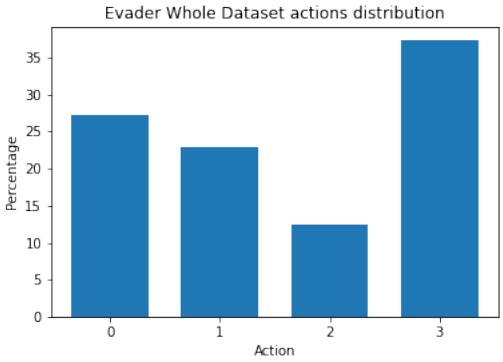
Data set distribution of action labels for both players in *Env1 res 3*.



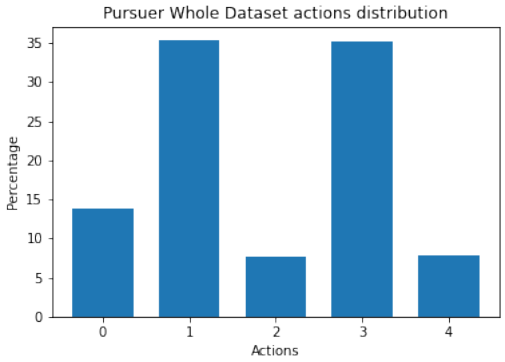
Evader's actions distribution.

Pursuer's actions distribution.

Data set distribution of action labels for both players in *Env2*.

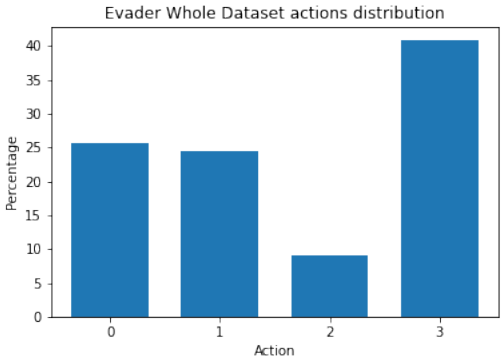


Evader's actions distribution.



Pursuer's actions distribution.

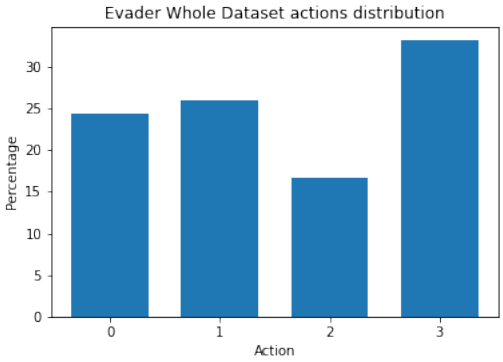
Data set distribution of action labels for both players in *Env2 res 2*.



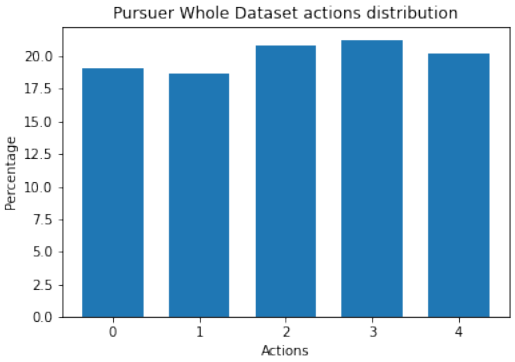
Evader's actions distribution.

Pursuer's actions distribution.

Data set distribution of action labels for both players in *Env2 res 3*.

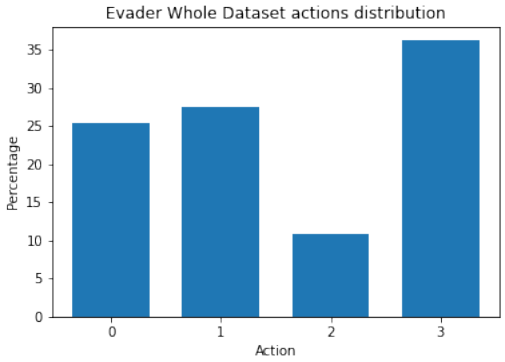


Evader's actions distribution.

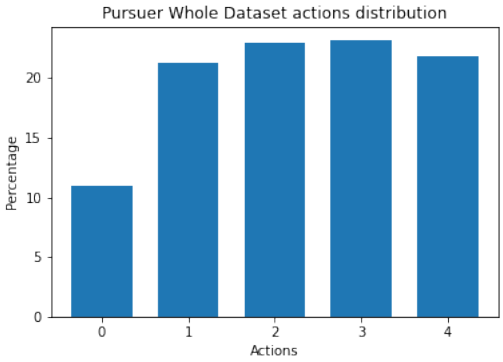


Pursuer's actions distribution.

Data set distribution of action labels for both players in *Env3*.

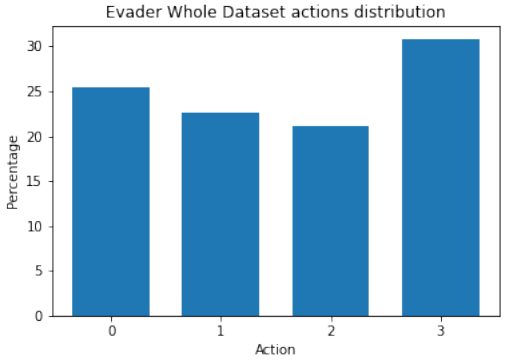


Evader's actions distribution.

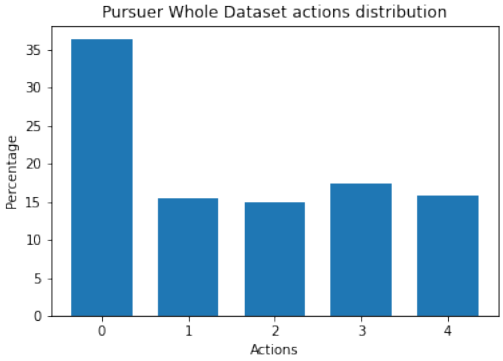


Pursuer's actions distribution.

Data set distribution of action labels for both players in *Env3 res 2*.

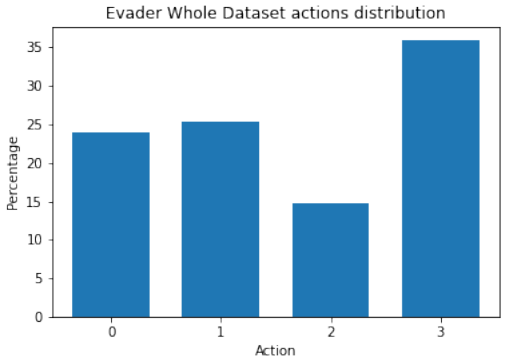


Evader's actions distribution.



Pursuer's actions distribution.

Data set distribution of action labels for both players in *Env4*.



Evader's actions distribution.

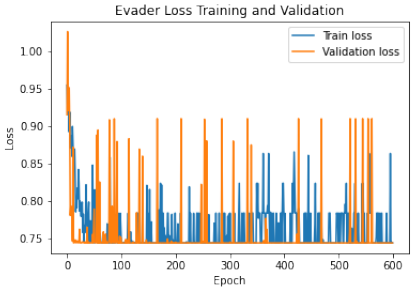
Pursuer's actions distribution.

Data set distribution of action labels for both players in *Env4 res 2*.

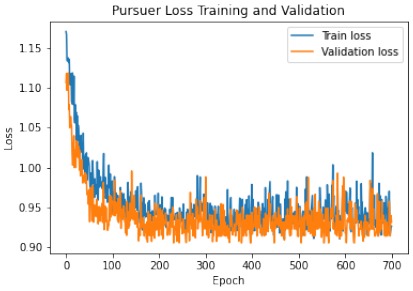
In the table below, we can appreciate the required epochs for every SMP training, the chosen optimizer, the time required for training, and the latest accuracy values for both the train and validation data sets. In the next pages, the evolution of the loss function and the accuracy values is displayed.

Samples, train and validation partition					
	Epochs	Optimizer	Training time	Train accuracy	Validation accuracy
Env1 res 2					
Evader	600	Adam	3min 4.1sec	96.88%	96.78%
Pursuer	700	Adam	4min 38.1sec	96.16%	94.7%
Env1 res 3					
Evader	100	Adam	1min 32.9sec	95.12%	94.3%
Pursuer	150	Adam	2min 33.4sec	92.37%	91.34%
Env1 res 3					
Evader	1000	Adam	3min 12.4sec	96.88%	96.78%
Pursuer	500	Adam	1min 47.2sec	96.16%	94.7%
Env2					
Evader	300	Adam	3min 18.2sec	91.33%	90.11%
Pursuer	1000	AdamW[31]	13min 6.1sec	97.72%	97.29%
Env2 res 2					
Evader	100	Adam	4min 28.5sec	92.08%	93.04%
Pursuer	200	AdamW	10min 45.1sec	96.35%	96.78%
Env2 res 3					
Evader	300	Adam	2min 6.6sec	92.55%	89.67%
Pursuer	800	AdamW	6min 7.4sec	94.66%	91.51%
Env3					
Evader	200	AdamW	15min 26.5sec	93.86%	94.41%
Pursuer	500	AdamW	39min 59.6sec	92.95%	92.86%
Env4					
Evader	2000	Adam	8min 42.9sec	90.62%	73.58%
Pursuer	1000	Adamax	4min 20.4sec	83.44%	74.03%
Env4 res 2					
Evader	100	AdamW	11min 6.3sec	93.95%	88.08%
Pursuer	1000	AdamW	13min 29.6sec	92.3%	89.5%

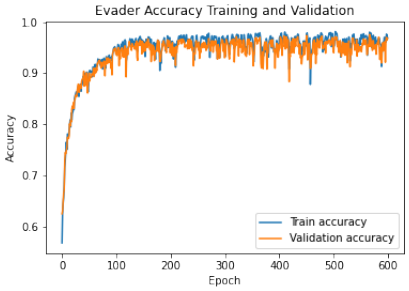
Training results for bot players in every environment.



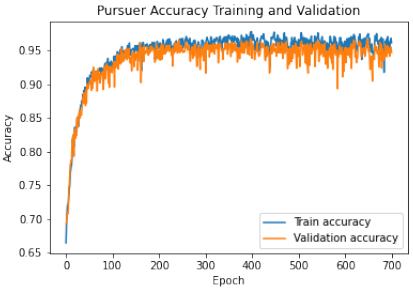
Evader loss function values.



Pursuer loss function values.

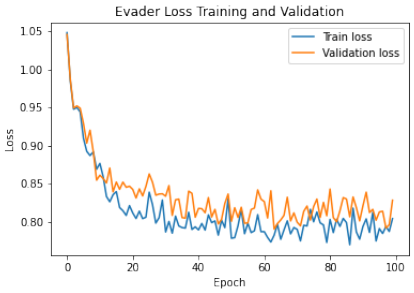


Evader accuracy.

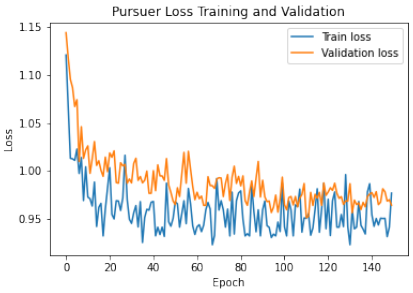


Pursuer accuracy.

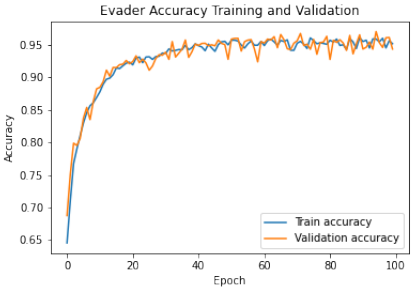
Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env1 res 2*.



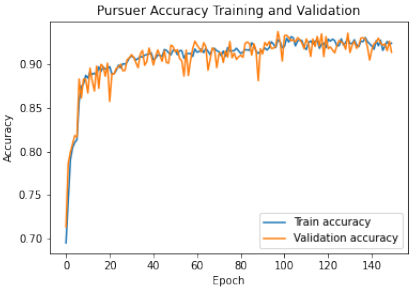
Evader loss function values.



Pursuer loss function values.

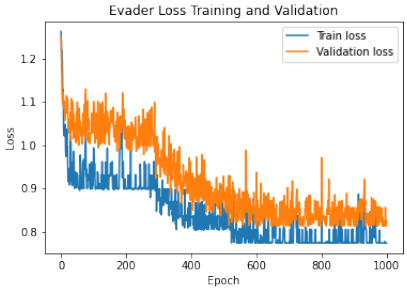


Evader accuracy.

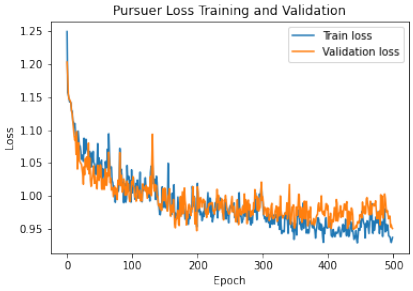


Pursuer accuracy.

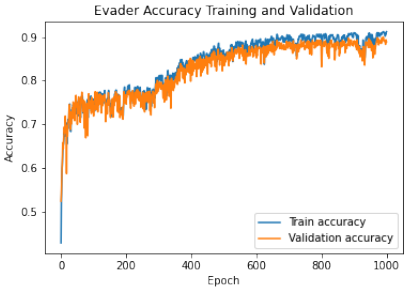
Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env1 res 3*.



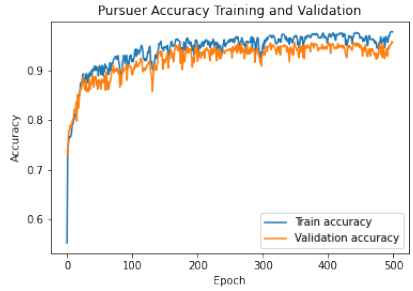
Evader loss function values.



Pursuer loss function values.

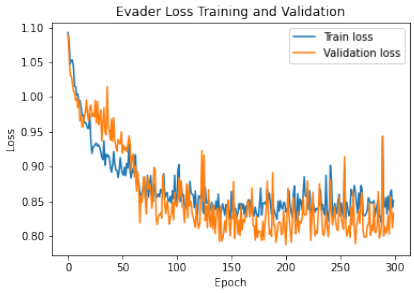


Evader accuracy.

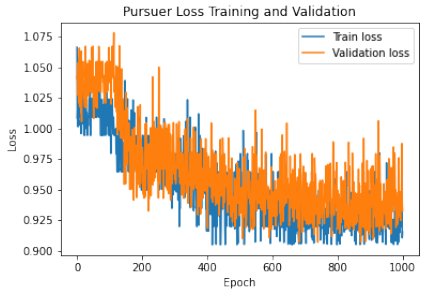


Pursuer accuracy.

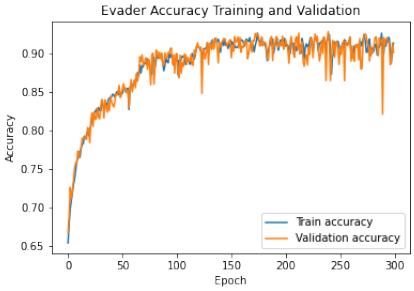
Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env2 res 1*.



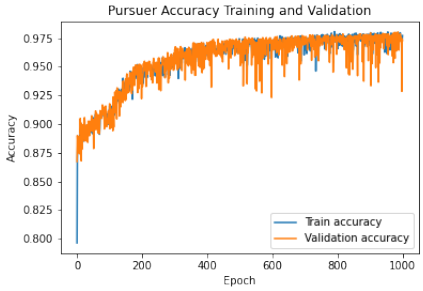
Evader loss function values.



Pursuer loss function values.

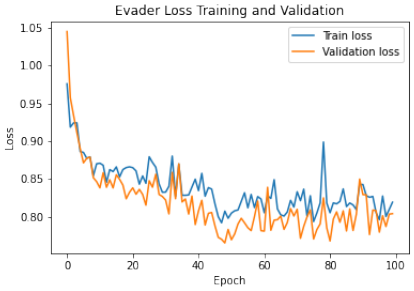


Evader accuracy.

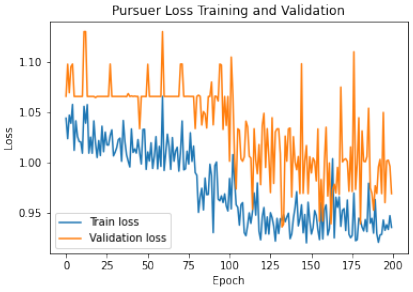


Pursuer accuracy.

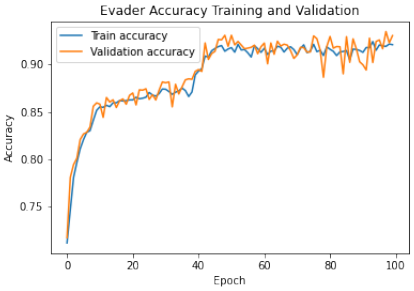
Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env2 res 2*.



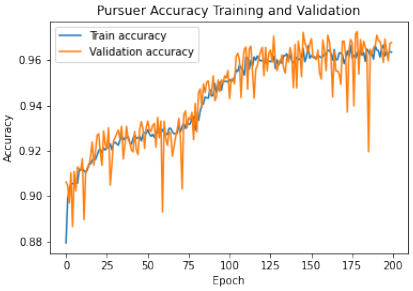
Evader loss function values.



Pursuer loss function values.

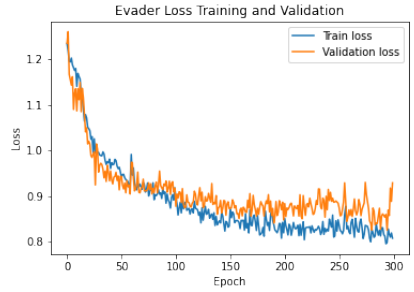


Evader accuracy.

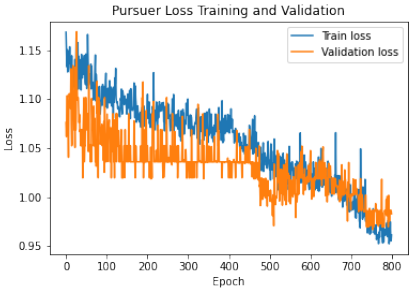


Pursuer accuracy.

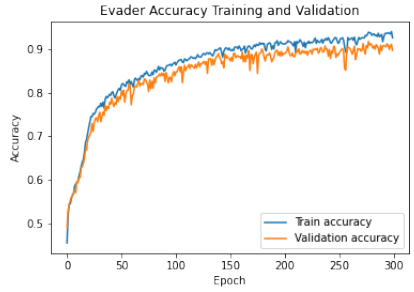
Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env2 res 3*.



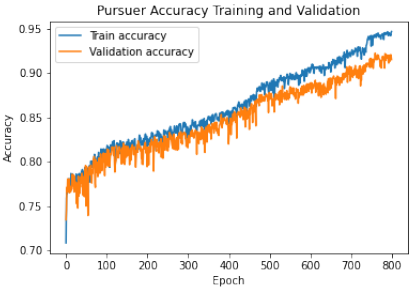
Evader loss function values.



Pursuer loss function values.

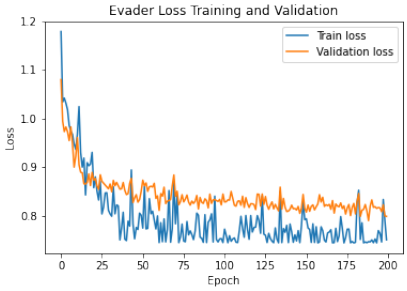


Evader accuracy.

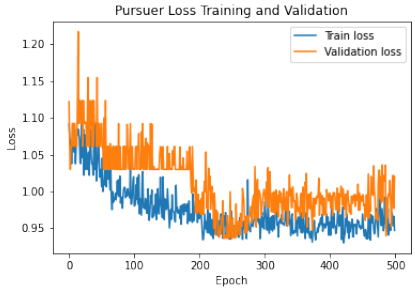


Pursuer accuracy.

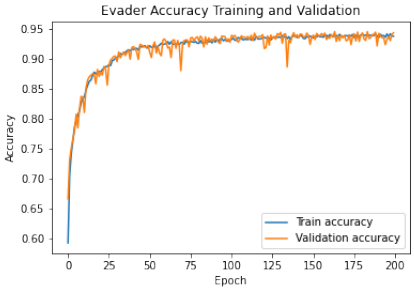
Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env3 res 1*.



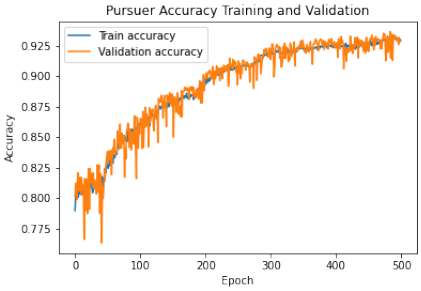
Evader loss function values.



Pursuer loss function values.

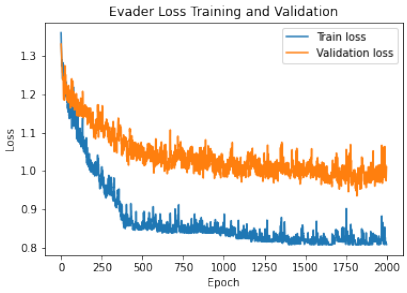


Evader accuracy.



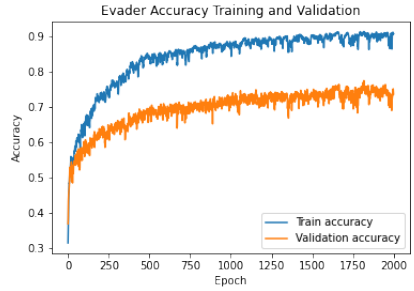
Pursuer accuracy.

Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env3 res 2*.

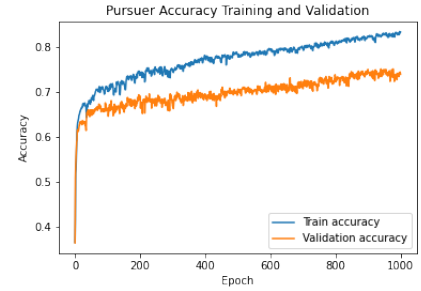


Evader loss function values.

Pursuer loss function values.

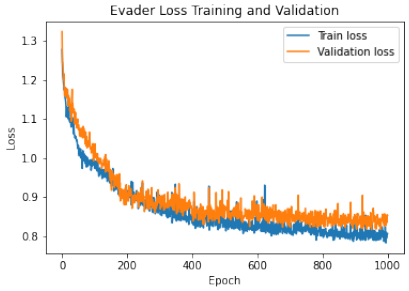


Evader accuracy.

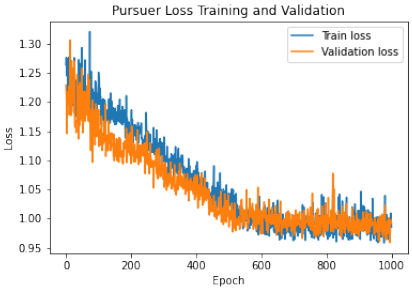


Pursuer accuracy.

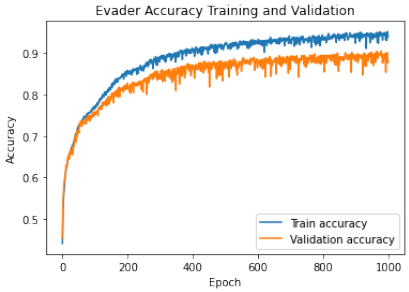
Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env4 res 1*.



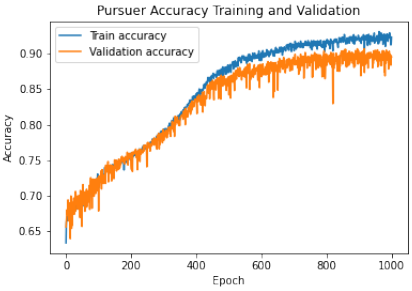
Evader loss function values.



Pursuer loss function values.



Evader accuracy.



Pursuer accuracy.

Evolution of the loss function and the accuracy over epochs for the training and validation sets for both players in *Env4 res 2*.

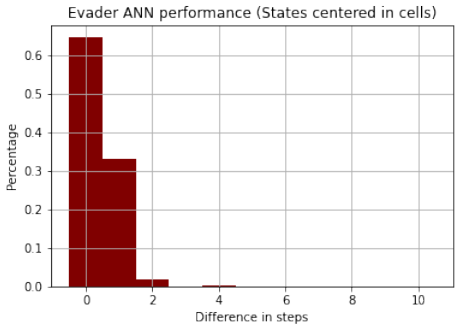
C.2 SMPs performance results

Finally the results after using the SMPs in simulated games is exposed below. The table shows the total simulations executed per environment. In the next pages, the comparison of the trained SMP taking as reference the evader's escape length, for the initial state in every simulation, obtained as $E[e_0, p_0]$ (as explained in chapter 3). Next to these comparisons we can find the confusion matrices comparing the total of cases where there is escape, or not, applying the SMPs or the DMPs.

Total of Simulations per Environment									
	Env1 res2	*Env1 res3*	*Env2*	*Env2 res2*	*Env2 res3*	*Env3*	*Env3 res2*	*Env4*	*Env4 res2*
Number of simulations	377	383	353	382	382	379	384	349	382

Total of run simulations per environment in the neural networks performance analysis.

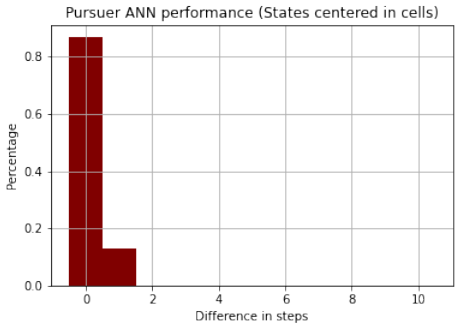
Env1 res 2



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	252	0
	Evader does not escape	0	125

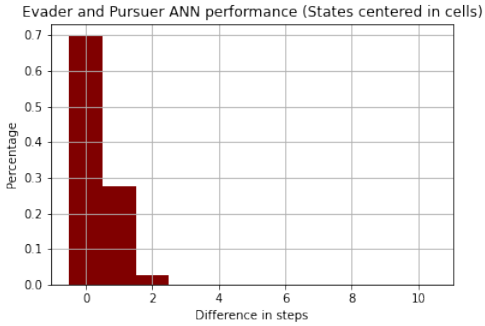
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	252	0
	Evader does not escape	97	28

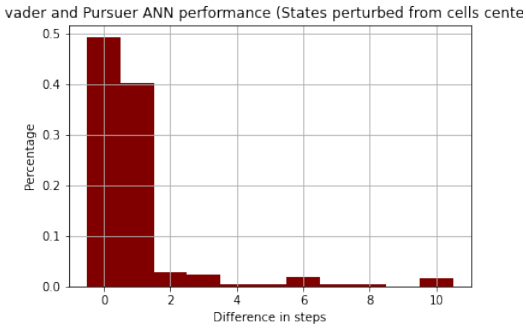
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	252	0
	Evader does not escape	61	64

Evader and Pursuer simultaneous evaluation.

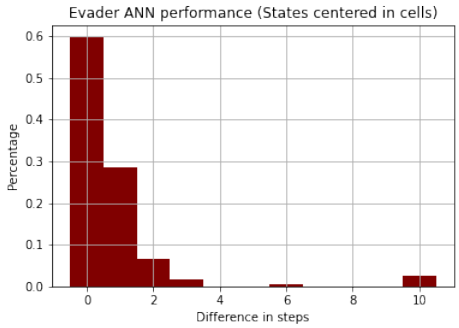


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	245	7
	Evader does not escape	43	82

Evader and Pursuer simultaneous evaluation with generalization.

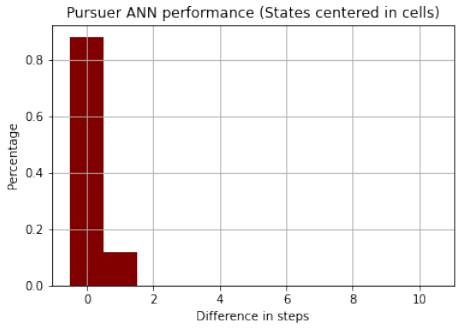
Env1 res 3



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	275	22
	Evader does not escape	0	86

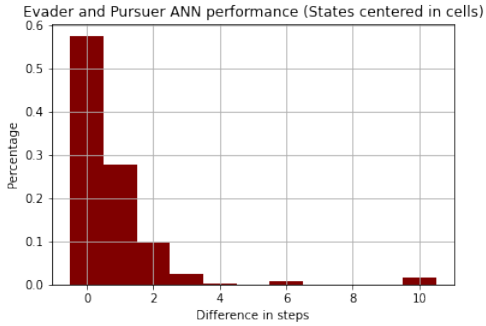
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	284	0
	Evader does not escape	71	28

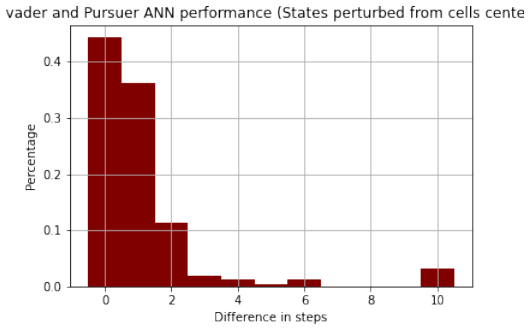
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	282	2
	Evader does not escape	41	58

Evader and Pursuer simultaneous evaluation.

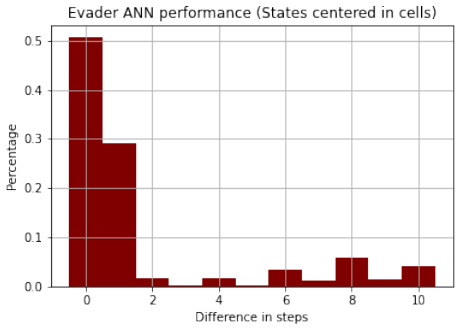


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	279	5
	Evader does not escape	36	63

Evader and Pursuer simultaneous evaluation with generalization.

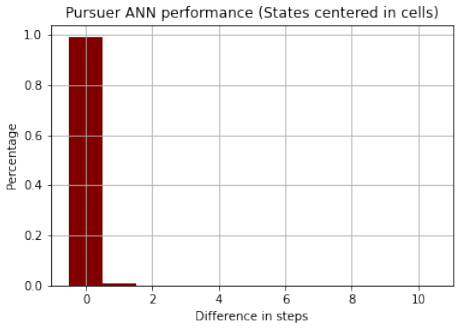
Env2



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	233	18
	Evader does not escape	0	102

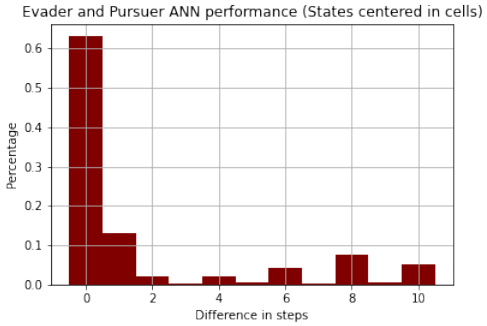
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	251	0
	Evader does not escape	99	3

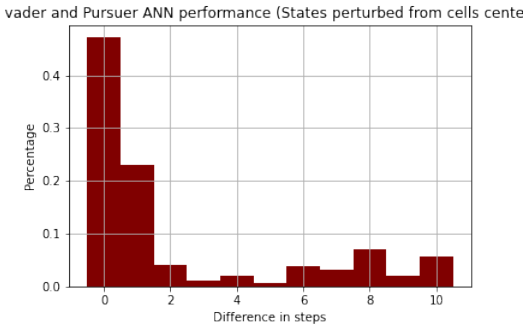
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	241	10
	Evader does not escape	63	39

Evader and Pursuer simultaneous evaluation.

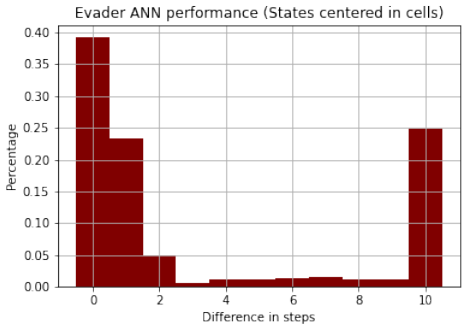


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	233	18
	Evader does not escape	54	48

Evader and Pursuer simultaneous evaluation with generalization.

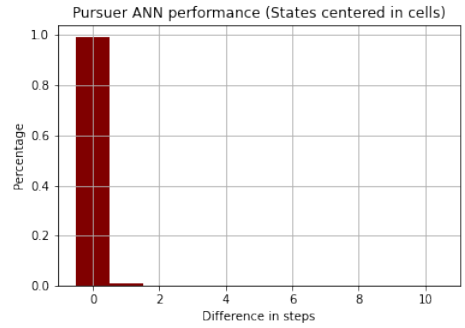
Env2 res 2



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	28	39
	Evader does not escape	0	55

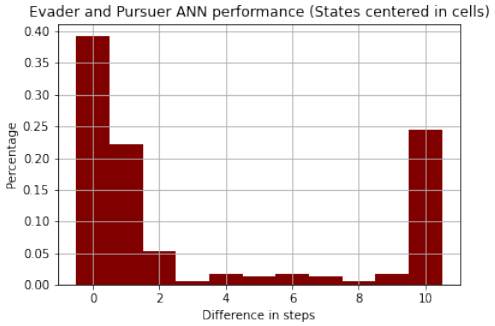
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	330	0
	Evader does not escape	50	2

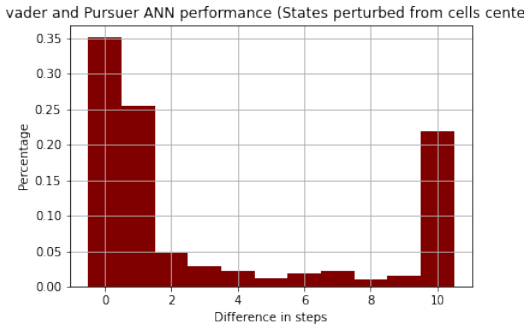
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	314	13
	Evader does not escape	31	24

Evader and Pursuer simultaneous evaluation.

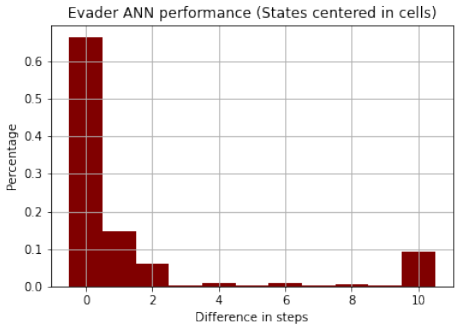


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	311	16
	Evader does not escape	20	35

Evader and Pursuer simultaneous evaluation with generalization.

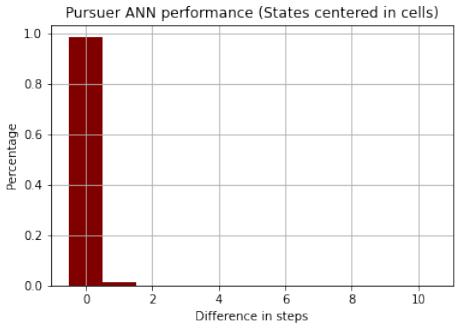
Env2 res 3



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	336	12
	Evader does not escape	0	36

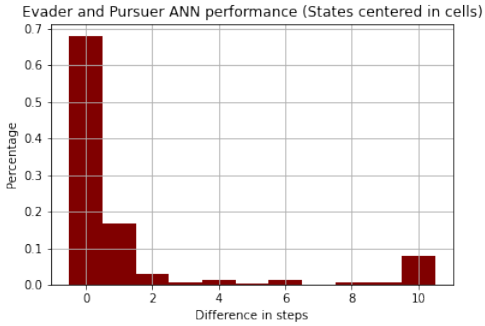
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	352	0
	Evader does not escape	28	4

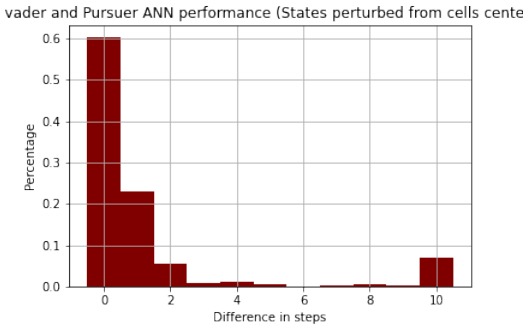
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	344	8
	Evader does not escape	7	25

Evader and Pursuer simultaneous evaluation.

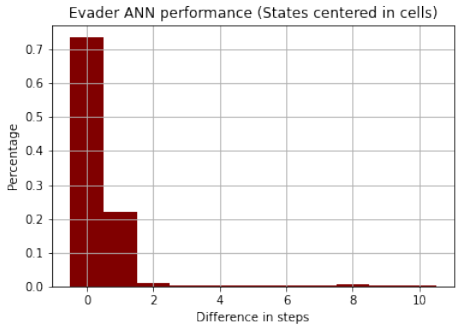


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	347	5
	Evader does not escape	9	23

Evader and Pursuer simultaneous evaluation with generalization.

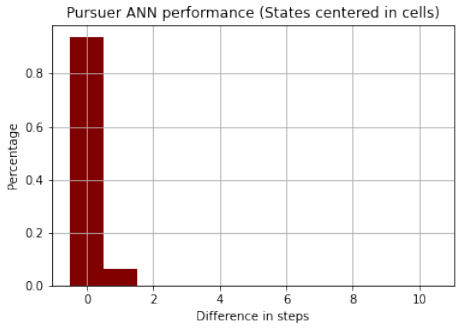
Env3



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	293	6
	Evader does not escape	0	80

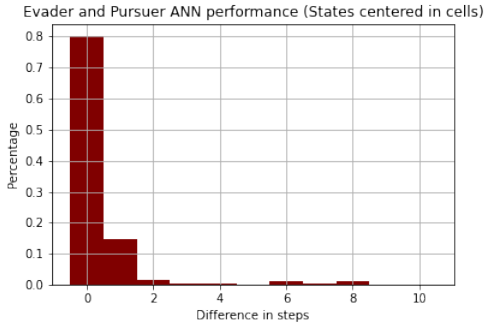
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	299	0
	Evader does not escape	69	11

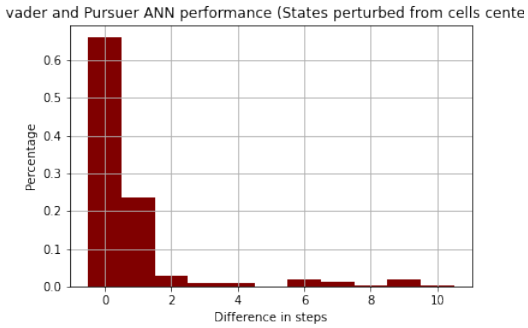
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	296	3
	Evader does not escape	53	27

Evader and Pursuer simultaneous evaluation.

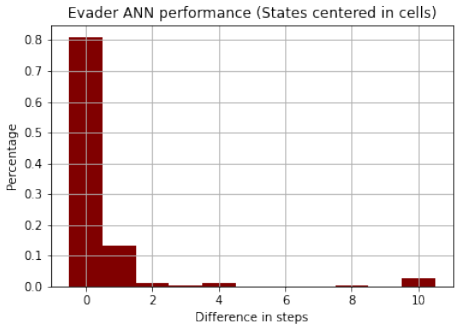


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	287	12
	Evader does not escape	43	37

Evader and Pursuer simultaneous evaluation with generalization.

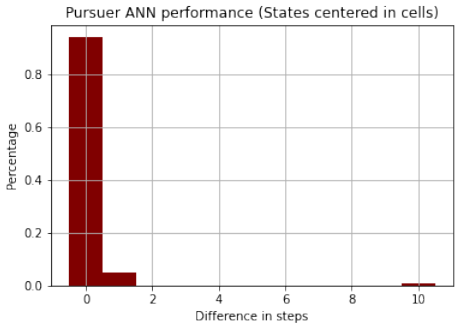
Env3 res 2



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	332	10
	Evader does not escape	0	42

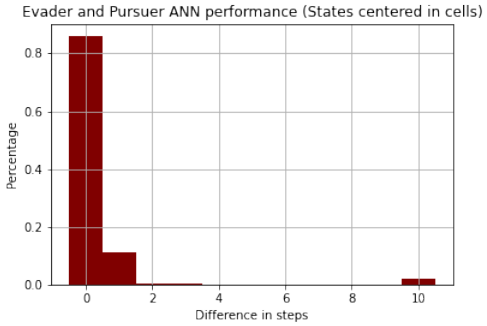
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	340	0
	Evader does not escape	34	10

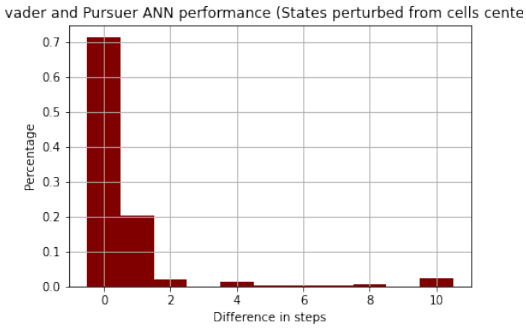
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	339	3
	Evader does not escape	26	16

Evader and Pursuer simultaneous evaluation.

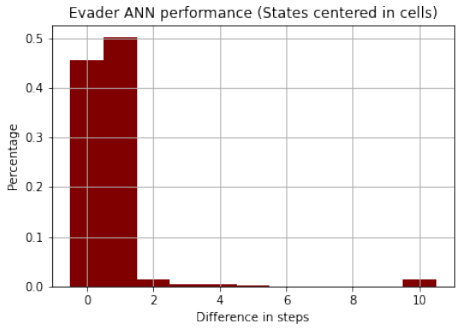


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	338	4
	Evader does not escape	22	20

Evader and Pursuer simultaneous evaluation with generalization.

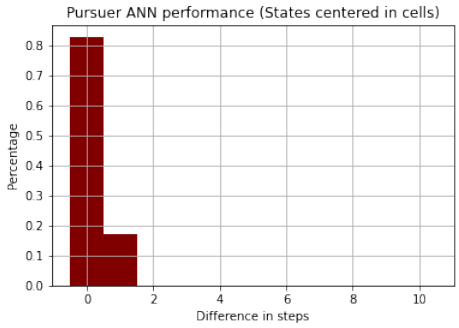
Env4



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	177	4
	Evader does not escape	0	168

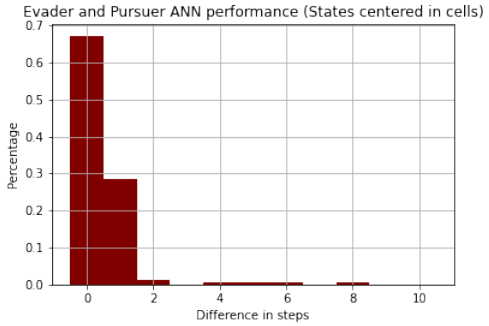
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	181	0
	Evader does not escape	148	20

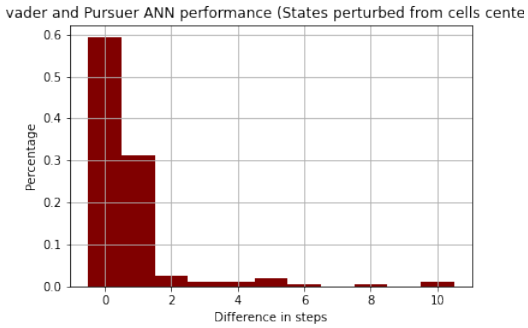
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	180	1
	Evader does not escape	130	38

Evader and Pursuer simultaneous evaluation.

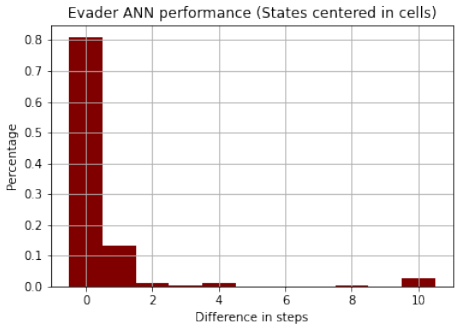


Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	179	2
	Evader does not escape	124	44

Evader and Pursuer simultaneous evaluation with generalization.

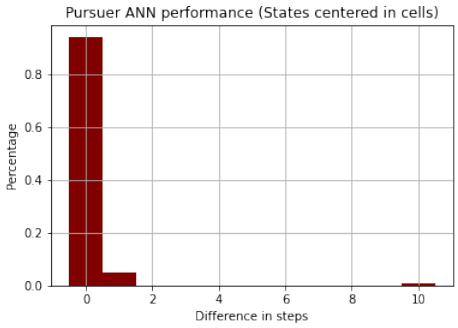
Env4 res 2



Evader evaluation.

		SMP_{evader}	$DMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	288	6
	Evader does not escape	0	88

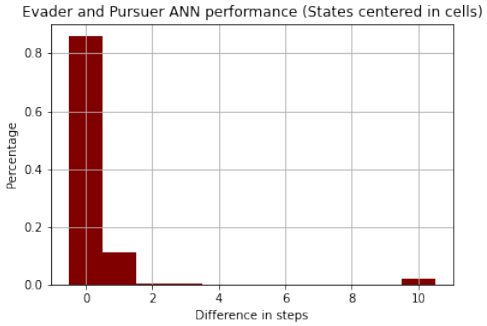
Evader evaluation.



Pursuer evaluation.

		DMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	294	0
	Evader does not escape	69	19

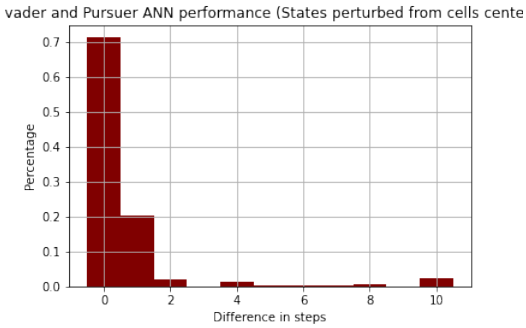
Pursuer evaluation.



Evader and Pursuer simultaneous evaluation.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	291	3
	Evader does not escape	60	28

Evader and Pursuer simultaneous evaluation.



Evader and Pursuer simultaneous evaluation with generalization.

		SMP_{evader}	$SMP_{pursuer}$
		Evader escapes	Evader does not escape
DMPs	Evader escapes	283	11
	Evader does not escape	52	36

Evader and Pursuer simultaneous evaluation with generalization.