University of Padova

Department of Information Engineering

*Master Thesis in* ICT for Internet and Multimedia

# Distributed Deep Reinforcement Learning for Drone Swarm Control

*Supervisor*
Prof. Alberto Testolin
University of Padova
Department of General Psychology

*Co-supervisor*
Prof. Andrea Zanella

*Master Candidate*
Federico Venturini

2 December 2019

To my family: my brother Marco and my parents Adriano and Daniela.

# Abstract

Reinforcement Learning is a promising Machine Learning field that has started to be widely used in many scenarios, from robotics to chemistry. Inspired by the way humans learn, it consists of an iterative approach in which an agent, by interacting with and receiving feedback from its environment, attempts to learn an optimal action selection policy.
In this project we applied Deep-Q Reinforcement Learning, a technique that combines the power of Deep Learning, in particular of Convolutional Neural Networks (CNNs), with the Q-Learning approach, to manage a swarm scenario in a bi-dimensional environment. Given a square map with some targets, the goal is to make the drones able to learn to cooperate between them, trying to track and follow the most valuable targets. We compared a distributed and a centralized approach and verified how the first can outperform the latter in a real-world scenario with limited training.

# Contents

# Listing of figures

# 1 Introduction

In the last decade, thanks to the continuing growth of computing power, Machine Learning became a must in many fields. It is very useful for the analysis of a big amount of data and the creation of models able to outperform any complex human model designed. In the context of complex scenarios, the design of engineered models is very time consuming and difficult. Instead of tuning the features of the models, in Machine Learning these are extracted from data. We can talk in this way of a model-free approach. Among the various techniques used in Machine Learning, Neural Networks are nowadays the most common. In particular Deep Neural Networks (DNNs) are state-of-the-art techniques: using the parallel computing power of Graphics Processing Units (GPUs), they are able to construct very complex models that manage and benefit from large amounts of data. They are used in a variety of tasks, including computer vision [1], speech recognition [2][3], machine translation [4], social network filtering etc.

In the last years Neural Networks have been combined with Reinforcement Learning (RL), and in particular with Q-Learning [5]. For this reason, now we talk about Deep Reinforcement Learning applications and Deep-Q-Networks (DQN).
Reinforcement Learning is a sub-field of Machine Learning where an agent that interacts with an environment learns an optimal action selection policy by receiving feedback from it. RL algorithms learn in an iterative way: at each iteration the learner observes its state, it chooses an action that leads to a subsequent environment's state. Then, the learner receives

a reward or a penalty and updates its value function, which represents the objective to maximize. Exploring the environment and collecting this informations the agents learns how to maximize the long-term reward.
Recently Artificial Intelligence reached incredible results, with the use of RL. It has been hugely successful in classic table games, like chess or Go. These games imply solving a search problem at each move, and RL amazingly boosts the efficiency of the search. AlphaGo became the first program to defeat a human world champion in the game of Go [6].
Classic video games are other good benchmarks because they are executable environments with large state spaces. With the use of a DQN, an agent was trained and tested on the challenging domain of classic Atari 2600 games [7]. It has been demonstrated that the DQN agent, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games receiving only the pixels and the game score as inputs. RL has also been applied to solve a Rubik's cube with a humanoid robot hand [8].

However, RL is also applicable in many real world scenarios: from the design of a traffic light controller to solve the congestion problem [9], to the optimization of chemical reactions [10] or personalized recommendations systems [11]. The list of other possible applications in which RL can be applied is vast. One of them is the coordination of a multi-agent system. These situations arise naturally in a variety of domains, such as: robotics, telecommunications, drones, economics, distributed control, auctions, traffic light control, etc. In such systems it is important that agents are capable of discovering good solutions to the problem at hand either by coordinating with other learners or by competing with them.

The drone scenario is the one chosen for this project. Drones can be used for search and rescue operations [12], for aerial filming, for security reasons, for maintenance, for environmental monitoring [13], delivery services [13] and many other situations. In this work we applied Deep-Q-Learning, using Conv. Neur. Networks (CNNs), to train a swarm of drones to cooperate in a grid world. The objective is to explore a map and track any number of targets, which might move around the map, have different values over time and space, or appear or disappear suddenly. In this thesis, we limited ourselves to a static case, in which the problem simplifies to the assignment of a target to each drone. We compared a distributed approach and a centralized one. A similar work on a multi-agent cooperation was developed by Egorov [14], where a pursuit-evasion game has been simulated, with two pursuers that

try to catch two evaders.

The structure of the rest of this thesis is as follows:
in Chapter II we present an introduction to RL, presenting the Markov Decision Processes (MDP) model and the Temporal Difference (TD) learning solution. In particular, the Deep-Q-Learning approach is presented , with the application of Neural Networks on the Q-Learning.
In Chapter III we describe our drone swarm scenario and how we implemented Deep-Q-Learning. Some Reinforcement Learning strategies are also described and the way we conducted Optimization Search on some hyper-parameters is presented.
In Chapter IV there are the results of a single drone scenario: how it behaves in a single target situation and in a multi targets one, how it scales with the world size and how much some parameters can affect the final performance.
In Section V we analyze the results of a swarm situation, with the comparison between a distributed and centralized approach, the study of a generalization on more complex situations and the benefits of the learning strategies and the Optimization Search described in Section III.
In Section VI we briefly analyze the results obtained, we try to propose some ways to increase the performances and we present the numerous theoretical extensions that this project could have.

# 2

# Reinforcement Learning

In this chapter, we present an introduction to the Reinforcement Learning concepts that we will use in our work [15]. Then, the Deep-Q-Learning approach is described with a general overview on neural networks, and on Convolutional Neural Networks in particular.

### What is Reinforcement Learning

Reinforcement Learning [15, Sec.I] is a sub field of Machine Learning. Machine Learning can be divided into 3 main areas:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

Supervised learning is learning from a training set of labeled examples provided by a knowledgeable external supervisor. Each example is a description of a situation together with a specification (the label) of the correct output the system should take in that situation. We can, in general, classify the main tasks of Supervised Learning in classification tasks and regression tasks.
Instead, Unsupervised learning is a kind of learning that tries to find structure hidden in

collections of unlabeled data. Clustering and dimensionality reduction are the main applications of this kind of learning.
Although one might be tempted to think of Reinforcement Learning as a kind of unsupervised learning, it is trying to maximize a reward signal instead of trying to find hidden structures in the data. We can say that RL means "how to map situations to actions". The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and all subsequent rewards. Since the learner receives a scalar value as consequent feedback of its action, we could classify RL as a supervised approach. But in the latter case the feedback received, the label, is sufficient to know the correct output of the model, in the RL case it is only a scalar that helps to find it. The two features, trial-and-error search and delayed reward, are the two most important distinguishing features of Reinforcement Learning.

Main features of RL are:

- Its goal is to design algorithms (agents) that learn to make actions in order to maximize the sum of the cumulated following reward
- The agent's initial knowledge about the environment is limited
- The agent learns by trial and error: after selecting an action, the agent observes the effects of the action on the environment, and receives a feedback signal (reward)

What makes it different from other areas in Machine Learning:

- There is no supervisor, only a reward signal
- Feedback is delayed, not immediate
- Time matters (data is received sequentially)
- The agent's actions affect the subsequent data it receives

### 2.0.1 Elements of Reinforcement

- Policy: it defines the learning agent's way of behaving at a given time. It is a mapping from perceived states of the environment to actions to be taken when in those states.
- Reward signal: it defines the goal of a RL problem. At each time step, the environment sends a scalar called reward to the RL agent. The agent's objective is to maximize the total reward it receives over the long run.
- Value function: while a reward indicates what is good in an immediate sense, a value function specifies what is good in the long run. The value of a state is the total amount of reward an agent can expect to accumulate in the future, starting from that state. A state might yield a low immediate reward but still have an high value because it is regularly followed by other states that yield high rewards. We prefer actions that lead to states of highest value, not highest reward.
- Model environment: it's something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. Given a state and action, the model might predict the resultant next state and next reward.

## 2.1 Finite Markov Decision Processes

### 2.1.1 The agent-Environment Interface

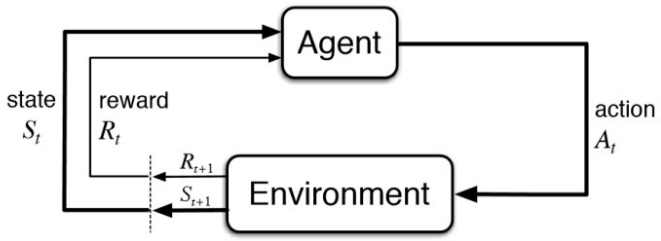


**Figure 2.1:** Reinforcement Learning framework

The agent and environment interact at each of a sequence of discrete time steps, t=0,1,2,3,... At each time step t, the agent receives some representation of the environment's state, $s_t \in S$, and on that basis selects an action, $a_t \in A(s)$. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in R$, and finds itself in a new state, $s_{t+1}$ [15, Sec. III]. The MDP and agent together thereby give rise to a sequence or trajectory that begins like this:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ...$$

In a finite MDP, the sets of states, actions and rewards (S,A,R) all have a finite number of elements. In this case, the random variables $S_t$ and $R_t$ have well defined discrete probability distributions dependent only on the preceding state and action:

$$p(s^{'}, r|s, a) = Pr[S_t = s^{'}, R_t = r|S_{t-1} = s, A_{t-1} = a] \; \forall \; s^{'}, s \in S, r \in R, a \in A(s) \tag{2.1}$$

The function p defines the dynamics of the MDP and is an ordinary deterministic function with 4 arguments:

$$\sum_{s' \in S} \sum_{r \in R} p(s^{'}, r|s, a) = 1 \; \forall \; s \in S, a \in A(s) \tag{2.2}$$

In a MDP, the state-transition and reward probability distributions completely characterize the environment's dynamics. The probability of each possible value for $S_t$ and $R_t$ depends

only on the immediately preceding state and action, $S_{t-1}$ and $A_{t-1}$, and, given them, not at all on earlier states and actions. This is a restriction on the state, not on the decision process. The state must include information about all aspects of the past agent-environment interaction that make a difference for the future. If it does, then we have the Markov property.

$$p(s^{'}|s,a) = Pr[S_t = s^{'}|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} p(s^{'}, r|s, a) \tag{2.3}$$

$$r(s,a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s^{'}, r|s, a) \tag{2.4}$$

$$r(s,a,s^{'}) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s^{'}] = \sum_{r \in R} r \frac{p(s^{'}, r|s, a)}{p(s^{'}|s, a)} \tag{2.5}$$

### 2.1.2 Goals and rewards

If we want to make a robot learn how to escape from a maze, the reward is often -1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it. Instead, for an agent that has to learn to play chess, we can use +1 for winning, -1 for loosing and 0 reward for all non-terminal positions.
It is thus critical that the rewards we set up truly indicate what we have in mind. The reward is our way of communicating to the robot what we want to achieve, not how we want it achieved (a chess player should be rewarded only for actually winning, not for achieving sub-goals such as taking it's opponent's pieces or gaining control of the center of the board. If achieving these sorts of sub-goals were rewarded, then agent might find a way to achieve them without achieving the goal).

### 2.1.3 Returns and episodes

In general we seek to maximize the expected return, where the return, denoted $G_t$, is defined as some specific function of the reward sequence. In the simplest case, the return is the sum of the expected reward:

$$G_t = r_{t+1} + r_{t+2} + ... + r_T \tag{2.6}$$

where T is a random variable which represents the final step. This approach makes sense in applications in which there is a natural notion of the final step. Each episode ends in a spe-

cial state called "terminal state", followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. The next episode begins independently of how the previous one ended. Thus the episodes can all be considered to end in the same terminal state, with different rewards for the different outcomes. These are called episodic tasks.
On the other hand, in many cases the agent-environment interaction does not break naturally into identifiable episodes, but goes on without limits. This would be a natural way to formulate an on-going process-control task and manage in this way continuing tasks. Computing $G_t$ is problematic for continuing tasks because the final step would be T=$\infty$, and the return could easily infinite. For this reason a discounting factor is used:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \tag{2.7}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad 0 \leq \gamma \leq 1 : discount\ rate \tag{2.8}$$

If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence $\{R_k\}$ is bounded.

$$G_t = R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+... = R_{t+1}+\gamma(R_{t+2}+\gamma R_{t+3}+...) = R_{t+1}+\gamma G_{t+1} \tag{2.9}$$

This works for all time steps t<T, even if termination occurs at t+1, if we define $G_T = 0$. Although the return is a sum of an infinite number of terms, it still converges to a finite value if the reward is non-zero and bounded, if $\gamma < 1$. For example if the reward is +1,

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{2.10}$$

### 2.1.4 Policies and value functions

A policy is a mapping from states to probabilities of selecting each possible action:
$\pi : s \rightarrow p(a|s)\ \forall a \in A$. If the agent is following policy $\pi$ at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. The value function of a state s under a policy $\pi$, $v_\pi(s)$, is the expected return when starting in s and following $\pi$ (the value of the terminal state, if any,

is always 0):

$$v_\pi(s) = \mathbb{E}_\pi = [G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \forall s \in S \tag{2.11}$$

Equation is defined as state-value function for policy $\pi$. We define the value of taking action a in state s under a policy $\pi$, $q_\pi(s, a)$, as the expected return starting from s, taking the action a and following $\pi$:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.12}$$

$v_\pi$ and $q_\pi$ can be estimated from experience. If an agent follows policy $\pi$ and maintains an average, for each state encountered, of the actual returns that have followed that, then the average will converge to the state's value $v_\pi(s)$ as the number of times that state is encountered approaches infinity. If separate averages are kept for each action taken in each state, then the averages will similarly converge to the action values $q_\pi(s, a)$. These methods are called Monte Carlo methods because they involve averaging over many random samples of actual returns.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] = \tag{2.13}$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \tag{2.14}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \; \forall s \in S \tag{2.15}$$

Equation 2.14 is called "Bellman equation for $v_\pi$", and it expresses a relationship between the value of a state and the values of its successor values; the value of the state must be equal to the (discounted) value of the expected next state, plus the reward expected along the way. The value function $v_\pi$ is the unique solution to its Bellman equation.

### 2.1.5 Optimal policies and optimal value functions

A policy $\pi$ is defined to be better than or equal to a policy $\pi'$ if its expected return is greater than or equal to that of $\pi'$ for all states:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \, \forall s \in S \tag{2.16}$$

There is always at least one policy that is better than or equal to all other policies: it's the optimal policy. Although there may be more than one, we denote all the optimal policies by $\pi_*$. They share all the same-value function, called optimal state-value function:

$$v_*(s) = \max_\pi v_\pi(s) \; \forall s \in S \tag{2.17}$$

Optimal policies also share the same optimal-action value function:

$$q_*(s,a) = \max_\pi \; q_\pi(s,a) \forall s \in S, a \in A(s) \tag{2.18}$$

$$\rightarrow q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma V_*(S_{t+1}) | S_t = s, A_t = a] \tag{2.19}$$

Because $v_*$ is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation, for state values. Because it's the optimal value function, however, $v_*$'s consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for $v_\pi$, or the Bellman optimality equation, that intuitively expresses the fact that the value of a state under an optimal policy must be equal to the expected return for the best action from that state:

$$v_*(s) = \max_{a \in A(s)} q_{\pi_*}(s,a) = \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] = \tag{2.20}$$

$$= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] = \tag{2.21}$$

$$= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \tag{2.22}$$

$$= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')] \tag{2.23}$$

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \tag{2.24}$$

$$= \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{2.25}$$

For finite MDPs, the Bellman optimality equation for $v_\pi$ has a unique solution independent of the policy. The Bellman optimality equation is actually a system of equations, one for each state. So if there are n states, then, there are n equations in n unknowns. If the dynamics of the environment are known, then in principle one can solve this system of equations for $v_*$ using any one of a variety of methods for solving system of equations of non-linear equations.

With $v_*$, it's easy to obtain an optimal policy. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns non-zero probability only to these actions is an optimal policy.

## 2.2 Dynamic programming

Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markon Decision Process (MDP)[15, Sec. IV]. Classical DP algorithms are of limited utility in RL both because of their assumption of a perfect model and because of their computational expense. Usually, other methods are used: methods that try to achieve the same results of DP but with less computation and without assumption of perfect environment's model. With DP, the assumptions are based on finite MDP: it's state, action and reward sets, (S,A,R) are finite. The dynamics are given by a set of probabilities

$$\text{p(s',r|s,a)} \; \forall \; \text{s} \in \text{S, a} \in \text{A(s), r} \in \text{R and s} \in \text{S}$$

Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. With continuous states and actions, state and action spaces are quantized and then applied finite-state DP methods.
The existence and uniqueness of $v_\pi$ are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy $\pi$.
If the environment's dynamics are completely known, then

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.26}$$

is a system of |S| simultaneous equations in |S| unknowns.
If we consider a sequence of approximate value functions $v_0, v_1, v_2, \ldots$ each mapping $S^+$ to R, $v_0$ is chosen arbitrarily and each successive approximation is obtained by using Bellman equation for $v_\pi$ as an update rule:

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] = \tag{2.27}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \forall s \in S \tag{2.28}$$

$v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for $v_\pi$ assures us of equality in this case. The sequence $v_k$ can be shown in general to converge to $v_\pi$ as $k \to \infty$ under the same conditions that guarantee the existence of $v_\pi$. From $v_k$ to $v_{k+1}$, iterative policy evaluation applies the same operation to each state s: it replaces the old values of the successor state s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated: expected update. Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function $v_{k+1}$. All the updates done in DP algorithms are called expected updates because they are based on the expectation over all possible next states rather than on a sample next state.

## 2.3 Monte Carlo methods

In Monte Carlo methods we don't assume complete knowledge of the environment. We need only experience: sample sequences of states, actions and rewards from actual or simulated interaction with an environment. No prior knowledge of the environment is required. Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming. Only on the completion of an episode the values estimates and policies are changed.

The value of a state is the simply average of the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. Each occurrence in a state s is called "a visit to s". There exist 2 possible methods: the *First Visit MC method* (it estimates $v_\pi(s)$ as the average of the returns following first visit to s) and the *Every-visit MC method* (it averages the returns following all visits to s) [15, Sec. V].

### 2.3.1 Monte Carlo estimation of action values

If a model is not available, then it's particularly useful to estimate action values rather than state values. With a model, state values alone are sufficient to determine a policy. Without a model, they are not. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Our goal is always to estimate $q_*$. Both *First-visit MC* and *Every-visit MC* converge quadratically to the true expected values as the number of visits to each state-action pair $\to \infty$. But many state-action pairs may never be visited. If $\pi$ is a deterministic policy, then following it one will observe returns only for one of the action

for each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience.
This is the general problem of maintaining exploration. For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying that the episodes start in a state-action pair, and that every pair has a non-zero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. This requirement is called the assumption of *exploring starts* (ES). But exploring starts cannot generalize all the case as the best option.

### 2.3.2 Monte Carlo control

Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. Let us assume that episodes are generated with exploring starts, and an infinite number of episodes. Under these assumptions, Monte Carlo methods will compute each $q_{\pi_k}$ exactly, for arbitrary $\pi_k$. Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an action-value function, and therefore no model is needed to construct the greedy policy: for any action-value function q, the corresponding greedy policy is the one that, for each $s \in S$, deterministically chooses an action with maximal action-value:

$$\pi(s) = arg\max_a q(s,a) \tag{2.29}$$

Policy improvement then can be done by constructing each $\pi_{k+1}$ as the greedy policy with respect to $q_{\pi_k}$.

$$q_{\pi_k}(s, \pi_{k+1}(s)) = q_{\pi_k}(s, arg\max_a q_{\pi_k}(s,a)) = \tag{2.30}$$

$$= \max_a q_{\pi_k}(s,a) \geq q_{\pi_k}(s, \pi_k(s) \geq v_{\pi_k}(s) \tag{2.31}$$

The theorem assures that each $\pi_{k+1}$ is uniformly better than $\pi_k$, or just good as $\pi_k$, in which case they are both optimal policies. This assures that the overall process converges to the optimal policy and optimal value function. But there are 2 unlikely assumptions:

1. episodes have exploring starts
2. Infinite number of episodes

The second assumptions is easy to remove. There are 2 ways:

1. One is to hold firm to the idea of approximating $q_{\pi_k}$ in each policy evaluation. Measurement and assumptions are made to obtain bounds on the magnitude and error probability in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small. We will have correct convergence up to some level of approximation, but it also requires far too many episodes to be useful.

2. We give up trying to complete policy evaluation before returning to policy improvement. On each evaluation step we move the value function toward $q_{\pi_k}$, but we don't expect to actually get close over many steps.

### 2.3.3 Monte Carlo control without ES

The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are 2 possible approaches:

1. On-policy methods

2. Off-policy methods

On-policy methods attempt to evaluate or improve a policy different from that used to generate the data. Monte Carlo with ES method is an example. The policy is generally soft, meaning that $\pi$(a|s)>0 $\forall$ s $\in$ S and all a $\in$ A(s), but gradually shifted closer and closer to a deterministic optimal policy $\rightarrow$ $\epsilon-$greedy. All non greedy actions are given the minimal probability of selection, $\frac{\epsilon}{|A(s)|}$, and the remaining to the greedy action: $1 - \epsilon + \frac{\epsilon}{|A(s)|}$.
$\epsilon$-greedy policies are examples of $\epsilon$-soft policies, defined as policies for which $\pi(a|s) \geq \frac{\epsilon}{|A(s)|}$ for all states and actions, for some $\epsilon$>0. Among $\epsilon$-soft policies, $\epsilon$-greedy policies are in some sense those that are closest to greedy.
As in Monte Carlo ES, we use *First-visit MC* methods to estimate the action-value function for the current policy. Without the assumption of ES, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of non-greedy actions. General policy improvement does not require that the policy be taken all the way to a greedy policy, only that it be moved toward a greedy policy. In our on-policy method we will move it only to an $\epsilon$-greedy policy. For any $\epsilon$-soft policy, $\pi$, any $\epsilon$-greedy policy with respect to $q_\pi$ is guaranteed to be better than or equal to $\pi$ by the policy improvement theorem.
The on-policy approach is a compromise: it learns action values not for the optimal policy,

but for a near-optimal policy that still explores. A possible solution is to use 2 policies: one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. They are called respectively, *target policy* and *behavior policy*. In this case the learning is from data "off" the target policy: for this reason it is called Off-policy learning. On-policy methods are generally simple. Off-policy methods require additional concepts and notation, and because the data is due to a different policy, they are often of greater variance and slower to converge. On the other hand, off-policy methods are more powerful and general: they include on-policy methods as the special case in which the target and behavior policies are the same. On policy methods estimate the value of a policy while using it for control. In off-policy methods there 2 functions are separated. The policy used to generate behavior, the *behavior policy*, may in fact be unrelated to the policy that is evaluated and improved, the *target policy*. Possible advantage: the target policy may be deterministic (e.g. greedy), while the behavior policy can continue to sample all possible actions.
Off policy Monte Carlo control methods follow the behavior policy while learning about and improving the target policy. The behavior policy must have a non-zero probability of selecting all actions that may be selected by the target policy. To explore all possibilities, we require that the behavior policy be soft (that it select all actions in all states with non-zero probabilities).
The target policy $\pi_*$ is the greedy policy with respect to Q, which is an estimate of $q_\pi$. The behavior policy b can be anything, but in order to assure convergence of $\pi$ to the optimal policy, an infinite number of returns must be obtained for each pair of state and action. This can be assured by choosing b to be $\epsilon$-soft. The policy $\pi$ converges to optimal at all encountered even though actions are selected according to a different soft policy b, which may change between or even within episodes. Possible problem: this methods learns only from the tails of episodes, when all of the remaining actions in the episode are greedy. If non-greedy actions are common, then learning will be slow, particularly for states appearing in the early portions of long episodes.

## 2.4 Temporal Difference Reinforcement Learning

Temporal Difference (TD) learning [15, Sec. VI] is a combination of Monte Carlo ideas and dynamic programming (DP) ones, and it is the method used in all this project. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on the other

learned estimates, without waiting for a final outcome.

### 2.4.1 TD prediction

A simple every-visit Monte Carlo method for non-stationary environment is:

$$V(S_t) \leftarrow +\alpha[G_t - V(S_t)] \tag{2.32}$$

Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then $G_t$ is known), TD methods need to wait only until the next time step. At time t+1 they immediately form a target and make a useful update using the observed reward $R_{t+1}$ and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.33}$$

immediately on transition to $S_{t+1}$ and receiving $R_{t+1}$. In effect, the target of the Monte Carlo update is $G_t$, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This method is called TD(0), or one-step TD. Since TD(0) bases its update in part on an existing estimate, is called bootstrapping method, like DP. We know that

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.34}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \tag{2.35}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \tag{2.36}$$

Monte Carlo methods use an estimate of (2.34) as a target, whereas DP methods use an estimate of (2.36) as a target. The Monte Carlo target is an estimate because the expected value in (2.34) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (2.36) and it uses the current estimate V instead of the true $v_\pi$.
The difference in the equation between $S_t$ and $R_{t+1} + \gamma V(S_{t+1})$ is the TD error:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.37}$$

Because the TD error depends on the next state and next reward, it's not available until one time step later: $\delta_t$ is available at time step t+1.

### 2.4.2 Advantages of TD prediction methods

TD methods update their estimate based in part on other estimates. They learn a guess from a guess: they bootstrap.

1. TD methods don't require a model of the environment, of it's reward and next-state probability distributions

2. They are naturally implemented in an "on-line, fully incremental fashion": with Monte Carlo methods one must wait until the end of the episode, because only when then the return is known, whereas with TD methods one need wait only one time step. Some applications have very long episodes, so that delaying all learning until the end of the episode is too slow; other are continuing tasks and have no episodes at all. Finally, Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken.

3. For any fixed policy $\pi$, TD(0) has been proved to converge to $v_\pi$, in the mean for a constant step-size parameter if it's sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions. TD methods have usually been found to converge faster than constant-$\alpha$ MC methods on stochastic tasks.

### 2.4.3 Optimality of TD(0)

When only a finite amount of experience is available, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function V, the increments specified by (2.32) and (2.33) are computed for every time step t at which a non-terminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges $\rightarrow$ it's called batch updating because updates are made only after processing each complete batch of training data. Under batch

updating, TD(0) converges deterministically to a single answer independent of the step-size parameter $\alpha$, as long as $\alpha$ is chosen to be sufficiently small. The constant-$\alpha$ MC method also converges deterministically under the same conditions, but to a different answer. Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions.

### 2.4.4 Sarsa: on-policy TD control

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior $\pi$ and for all states s and actions a. This can be done using essentially same TD method described above for learning $v_\pi$.
The theorems assuming the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.38}$$

where the step-size parameter $\alpha \in (0, 1]$ is constant. This update is done after every transition from a non-terminal state $S_t$. If $S_{t+1}$ is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as 0. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to the next. This quintuple gives rise to the name *Sarsa* algorithm [16].
We continually estimate $q_\pi$ for the behavior policy $\pi$, and at the same time change $\pi$ toward greediness with respect to $q_\pi$. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (for example, with $\epsilon$-greedy policies with $\epsilon = 1/t$.
Sometimes it is convenient to vary the step-size parameter from step to step. For example, using $\alpha_t(a) = \frac{1}{t}$ is guaranteed to converge to the true action values by the law of large numbers. But convergence is not guaranteed for all choices of the sequence $\alpha_t(a)$. The conditions required to assure convergence with probability 1 are [17]:

$$\sum_{t=1}^{\infty} \alpha_t(a) = \infty \tag{2.39}$$

and

$$\sum_{t=1}^{\infty} \alpha_t^2(a) < \infty \tag{2.40}$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence. Both conditions are satisfied for the case $\alpha_t(a) = \frac{1}{t}$, but not for the case of constant step-size parameter. In the latter case, this implies that the estimates never completely converge but continue to vary in response to the most recently received rewards. This is actually desirable in a non-stationary environment, and problems that are effectively non-stationary are the most common in RL. In addition, sequences of step-size parameters that meet the conditions often converge very slowly or need considerable tuning in order to obtain a satisfactory convergence rate. Although sequences of step-size parameters that meet these convergence conditions are often used in theoretical work, they are seldom used in applications and empirical research [15, Sec. II].

### 2.4.5 Q-learning: Off-policy TD control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning [5], defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.41}$$

The learned action-value function, Q, directly approximates $q_*$, the optimal action-value function, independent of the policy being followed. This simplifies the analysis of the algorithm and enables early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pair continue to be updated.
Under this and other assumptions, Q has been shown to converge with probability 1 to $q_*$.

### 2.4.6 Expected Sarsa

It is a learning algorithm that is just like Q-learning except that instead of the maximum over next state-action pairs it uses the expected value, taking into account how likely each action

is under the current policy.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma + \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)] \quad (2.42)$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma + \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (2.43)$$

Given $S_{t+1}$, this algorithm moves deterministically in the same direction as Sarsa moves in expectation: for this reason is called Expected Sarsa. Expected Sarsa is more complex computationally than Sarsa, but it eliminates the variance due to the random selection of $A_{t+1}$. Given the same amount of experience, we might expect it to perform slightly better than Sarsa, and indeed it generally does.

### 2.4.7 Maximization bias and double learning

In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias: consider a single state s where there are many actions a whose true values, q(s,a) are all zero but whose estimated values, Q(s,a) are uncertain and thus distributed some above and some below 0. The maximum of the true values is 0, but the maximum of the estimates is positive, a positive bias: this can harm the performance of TD control algorithms. We call this maximization bias.
So there will be a positive maximization bias if we use the maximum of the estimates as an estimate of the maximum of the true values. It's due to using the same samples (plays) both to determine the maximizing action and to estimate it's value. A possible solution is to divide the plays in 2 sets and use them to learn 2 independent estimates, $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value q(a), $\forall a \in A$. We could then use one estimate, $Q_1$, to determine the maximizing action $A^* = arg\max_a Q_1(a)$, and the other, $Q_2$, to provide the estimate of its value, $Q_2(A^*) = Q_2(arg\max_a Q_1(a)$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(arg\max_a Q_2(a)$.
The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time step in two, perhaps by flipping a coin on each step. If the coin comes up heads, the

update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)] \tag{2.44}$$

If the coin comes up tails, then the same update is done with $Q_1$ and $Q_2$ switched, so that $Q_2$ is updated. The two approximate value functions are treated completely symmetrically. The behavior policy can use both action-value estimates. For example, an $\epsilon$-greedy policy for Double Q-learning could be based on the average (sum) of the two action-value estimates.

### 2.4.8 Exploration vs Exploitation

One of the challenges of Reinforcement Learning is the trade-off between exploration and exploitation [15, Sec.I-II]. An agent must prefer actions that it has tried in the past and found to be effective in producing reward. On the other hand, in order to discover such actions, it has to try actions that it has not selected before. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and favor those that appear to be the best. The problem is even harder when reward is stochastic: each action must be tried many times to gain a reliable estimate of its expected reward. If we define $a_t$ as the action taken at time step t, $r_t$ the reward given at time step t, we can define the value of an arbitrary action $a$ as:

$$q_*(a) = \mathbb{E}[R_t | A_t = a] \tag{2.45}$$

The learner can only have an estimated real value: $Q_t(a)$, which should be as close as possible to $q_*(a)$. Since the true value of an action is the mean reward when that action is selected, we can define $Q_t(a)$ as:

$$Q_t(a) = \frac{sum\ of\ rewards\ when\ a taken\ prior\ to\ t}{number\ of\ times\ a\ taken\ prior\ to\ t} = \frac{\sum_{i=1}^{t-1} R_i \mathbb{1}_{A_t=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_t=a}} \tag{2.46}$$

where $\mathbb{1}_{predicate}$ denotes the random variable that is 1 if predicate is true and 0 if it is not. If the denominator is 0, then we instead define $Q_t(a)$ as some default value, such as 0. As the denominator goes to $\infty$, by the law of large numbers, $Q_t(a)$ converges to $q_*(a)$. This is called *sample-average method* for estimating action values because estimate is an average of the sample of relevant rewards. This is just one way to estimate action values, and not necessarily the best one. The simplest action selection rule is to select the action with the

highest estimated value, that is,the "greedy" action. If there are more than one greedy actions, then a selection is made among them in some arbitrary way, perhaps randomly. We write this greedy action selection method as:

$$A_t = arg\max_a Q_t(a) \tag{2.47}$$

where $arg\max_a$ denotes the action *a* for which the expression that follows is maximized. Greedy action selection always exploits current knowledge to maximize immediate reward; it spends no time at all sampling apparently inferior actions to see if they might really be better.

### $\epsilon$-greedy method

A possible alternative is to behave greedily most of the time, but every once in a while, say with probability $\epsilon$, instead select from among all the actions with equal probability, independently of the action-value estimates. These methods are called $\epsilon$-greedy methods.
In the limit, as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the $Q_t(a)$ converge to $q_*(a)$. This of course implies that the probability of selecting the optimal action converges to greater than $1 - \epsilon$, that is, to near certainty.

### Soft-max method

Another possible solution is to learn a numerical *preference* for each action *a*, called $H_t(a)$. The larger the preference, the more often that action is taken. Only the relative preference of one action over another is important. These preferences are determined according to a *soft-max* distribution as follows:

$$Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} = \pi_t(a) \tag{2.48}$$

Initially all action preferences are the same (e.g., $H_1$(a)=0, for all *a*) so that all actions have an equal probability of being-selected. On each step, after selecting action $A_t$ and receiving the reward $r_t$ the action preferences are updated by:

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)),\ and \tag{2.49}$$

$$H_{t+1}(a) = H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a), \ \forall a \neq A_t \tag{2.50}$$

where $\alpha > 0$ is the step-size parameter, and $\bar{R}_t \in \mathbb{R}$ is the average of all the rewards up through and including time t. The $\bar{R}_t$ serves as a baseline with which the reward is compared. If the reward is higher than the baseline, then the probability of taking $A_t$ in the future is increased, and if the reward is below, decreased. The non-selected actions move in the opposite direction. Some simulations show that without a baseline the performance would be significantly degraded.

## 2.5 Neural Networks

Neural networks (NNs) are a set of architectures, modeled from the human brain, that are designed to recognize patterns. A neural network can be described as a graph whose nodes correspond to neurons and edges correspond to links between them [18] [19, Sec. XX]. Such systems "learn" to estimate a function by looking at examples, without being programmed with specific rules. Nowadays NNs represent a high-performant tool widely sued in many machine learning problems; different versions of NNs exist. Neural networks can be used for classification and regression tasks: in the first case, the output layer will have as many neurons as the number of classes; in the second one, it can have an arbitrary amount of nodes. To briefly describe how they work, in what follows we present the 2 types of them: the fully-connected neural network, that can be considered as the original one, and the convolutional neural network, which is implemented in this project.

### 2.5.1 Fully Connected neural network

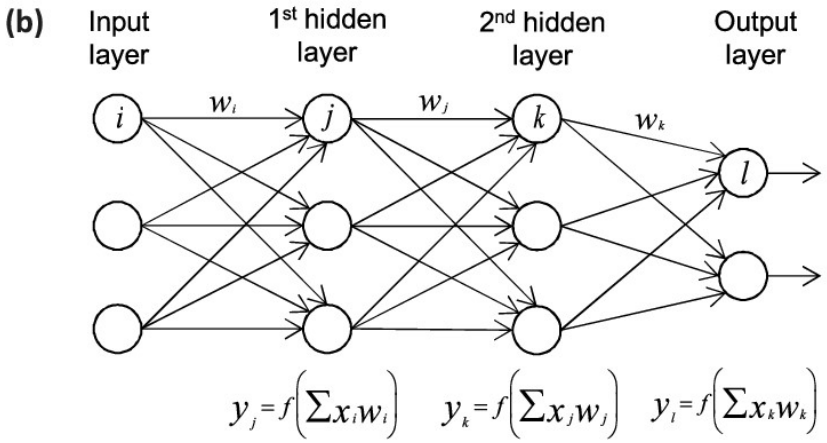


**Figure 2.2:** Fully connected neural network.

A fully connected neural network is composed by a multiple neurons that are grouped into layers. These layers are stacked subsequently from the input layer to the output layer. Each of them is composed by an arbitrary amount of nodes (neurons), and receives the output of the previous layer's nodes. Then, it propagates its output to all the neurons belonging to the next layer (Figure 2.2). In this way the data flow in a single direction, from input layer to the output layer. In Figure 2.2 you can see a very simple example of a fully connected neural network, with only 2 hidden layers and three neurons for each layer but the output

layer, which has only two neurons. More powerful neural networks are bigger and deeper, i.e. they have more layers and more neurons per layer.

Neurons do not just propagate information, but also process it. Like a neuron in the brain, a node of a neural network receives an input signal and under some conditions, it fires and propagates the signal. In a NN, each neuron receives the output signal from each of the neurons it is linked to, it multiplies each signal by a different weight and then sums them together. The result is given in input to a non-linear activation function and its output is propagated to all the following neurons that are linked. There exist different types of activation functions: the *sigmoid*, the *tanh*, *ReLU* (Figure 2.2), etc. The ReLU activation function is fast from a computationally point of view and avoids vanishing gradient problems, present with sigmoid and tanh activation functions. For this reason, ReLU is nowadays state of the art [20].

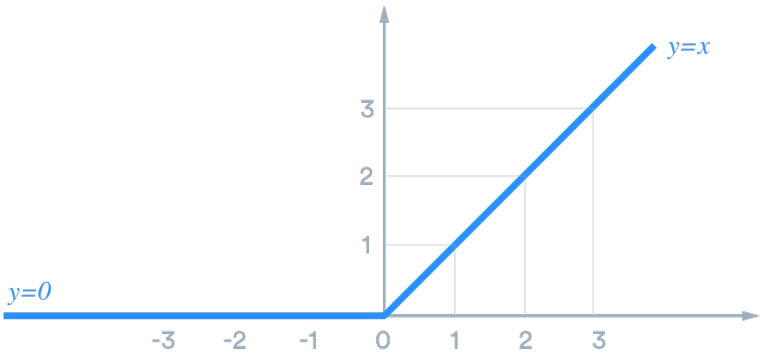


**Figure 2.3:** ReLU activation function.

To learn an input-output mapping, neural networks have to adjust the weight matrix, which contains the weights of all the links between the nodes. To do that, we have to give to the model the true output for each input. Then, it is shown we can use techniques like Stochastic Gradient Descent (SGD) to change the weights of the model towards to minimize the loss between the predicted output and the true output. If we define $J$ as the loss function, $\theta$ the set containing all the NN's weights, $\mathbf{x}$ the training sample and y the training label sampled from the their distribution $p$, the loss over the whole training set is the following:

$$J(\theta) = \mathbb{E}_{\mathbf{x},y\hat{p}_{data}} L(\mathbf{x}, y, \theta) = \frac{1}{m} \sum_{i=1}^{m} L(\mathbf{x}^{(i)}, y^{(i)}.\theta) \tag{2.51}$$

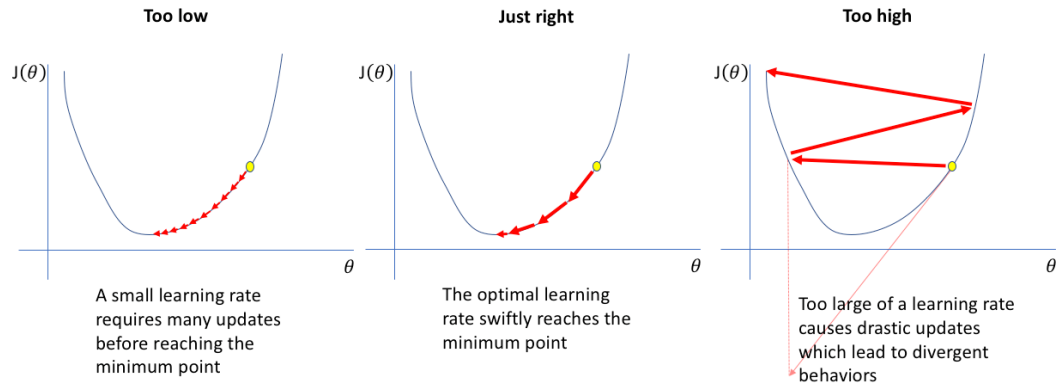


**Figure 2.4:** Effects of the learning rate on the minimization of the loss.

The loss of the gradient with respect to the weights $\theta$ can be defined as:

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(\mathbf{x}^{(i),y^{(i)}} \theta) \tag{2.52}$$

Rather than computing the gradient over the whole training set, in SGD it is approximated by using a subset containing $m$' training samples:

$$\mathbf{g} = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} \nabla_\theta L(\mathbf{x}^{(i)}, y^{(i)}, \theta) \tag{2.53}$$

The weights are then changed according to: $\theta \leftarrow \theta - \epsilon \mathbf{g}$, where $\epsilon$ is the learning rate, which progressively decreases during training. In Figure 2.4 it is possible to see the effects of different learning rates: if $J$ is defined as the loss of the model and $\theta$ as the parameters of the model, a too small learning rate requires an infeasible amount of time to learn the task, while a too big one can escape the valley of the local minima.

### 2.5.2 Convolutional Neural Networks

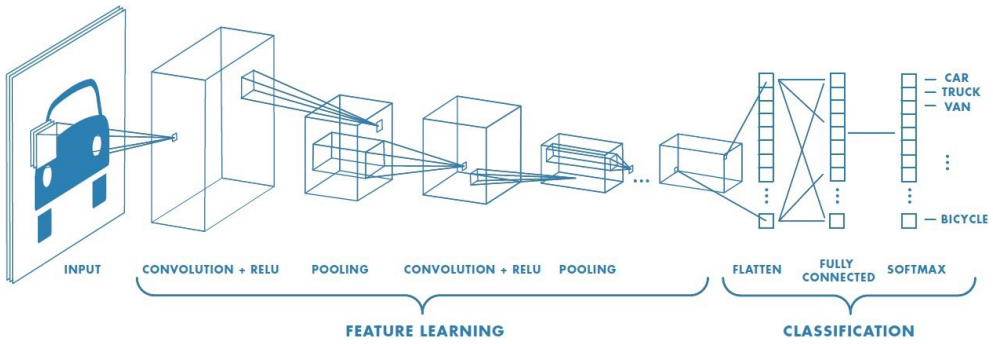


**Figure 2.5:** Example of CNN for object classification.

CNNs [21] are specialized kind of neural networks that exploits the convolution operation for processing data that have a known grid-like topology. For this reason they can be used for image data. In a fully connected neural network each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. In CNNs instead, thanks to the convolution, each member of the weight matrix, here called *kernel*, is used at every position of the input. This does not affect the run-time of the forward propagation but reduces the storage requirements for the model parameters. Most CNN are composed by:

- Convolution layer
- Non-linear activation function
- Pooling layer
- Fully-connected layer(s) (usually as output layer)

To be precise, the combination given by Convolution layer, Pooling layer and activation function can be considered as an unique layer; Layers like this are stacked subsequently many times until one or more fully-connected layers at the end of the neural network (Figure 2.5).

### Convolution layer

The convolution layer performs a 2D Convolution on the input image: the weight matrix, whose dimension is smaller than the input image and called *kernel*, is convolved on the entire input image. Each convolution layer can contain more than a single kernel: in that case we talk about the number of filters of a convolution layer. Each filter can perform a convolution on a different channel of the input image, if it has more than one. The tunable parameters of a Convolution layer are: the dimension of the kernel (on x and y directions), the strides and the padding parameter. The strides are step-size of the shift on both directions of the convolution operation, i.e. by how many pixels the kernel is shifted on the right and downward in the convolution operation. The padding consists in adding some blank pixels stacked on the borders of the input image in such a way the result of the convolution has the same size of the input image.

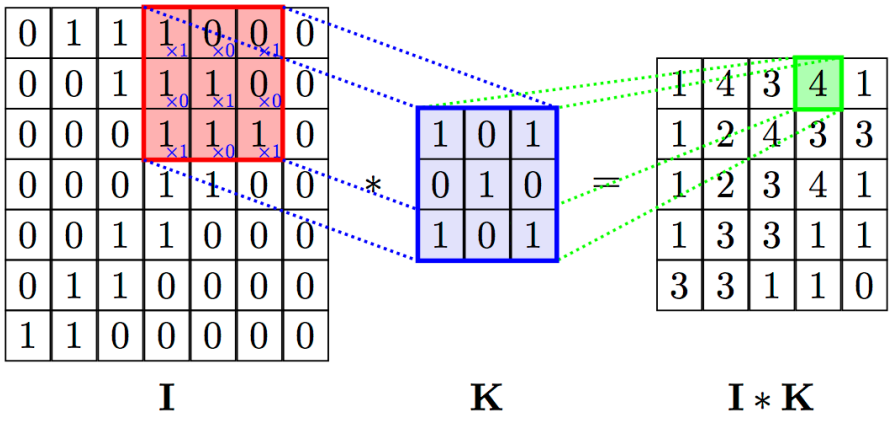


**Figure 2.6:** 2D Convolution.

### Non-linear activation function

The output of the convolution layer is one or more matrices (the matrix on the right in Figure 2.6) whose number is equivalent to the number of filters used. Each matrix is the result of the convolution of a filter of the convolution layer with the input of the convolution layer. All the values belonging to those matrices are then passed through a non -linear activation function, like in fully-connected neural networks.

### Pooling layer

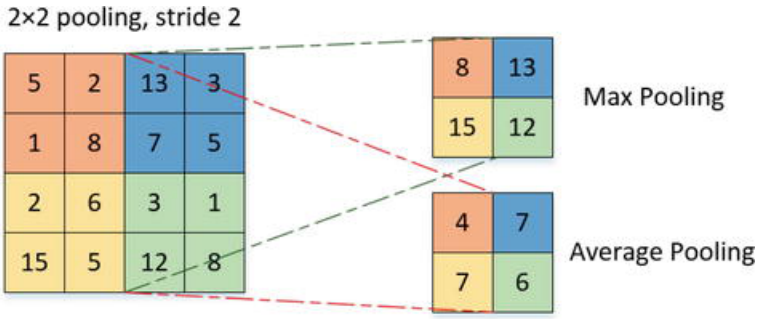


**Figure 2.7:** Mean average and max pooling.

The outputs of non-linear activation functions are combined with MaxPooling layers or with AveragePooling operations that summarize statistics of nearby outputs: the matrices that derive from the outputs of the non-linear activation functions are reduced by dimension using an average or the maximum of that region of the original matrices. Figure 2.7 shows how pooling works: a window whose size and strides are decided by the user, shifts along both dimensions of the original matrix, replacing each of the four quadrants with a single value: the maximum or the average of the original values.Pooling helps to make the representation approximately invariant to small translations of the input. Invariance to local translation can be a useful property if we care more about whether some features are present than exactly where they are. Pooling over spatial regions produce invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to [21]. Pooling is essential for handling inputs of varying size: to classify images of variable size, the input to the classification layer must have a fixed size.

Now that we have briefly explained how neural networks work, we turn back to Q-learning, and in particular to an off-policy temporal difference control. We have said that the equation that governs the action-value function updates is the following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.54}$$

Ideally, the estimation of the value function can be represented in a tabular form, for which an optimal policy can be obtained. However many real-world problems present large or continuous state spaces making training extremely slow. This can be solved by using function approximation methods like neural networks [7]. We refer to a neural network function approximator with weights $\theta$ as a Q-network. Remembering that the goal of the agent is to select actions in a way that maximizes cumulative future reward, the objective of the neural network is to approximate the optimal action-value function:

$$Q^*(s, a) = \max_\pi \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi] \tag{2.55}$$

$Q^*(s, a)$ is defined as the maximum expected return achievable by following any policy, after seeing some sequence *s* and then taking some action *a*, $Q^*(s, a) = max_\pi \mathbb{E}[R_t|s_t = s, a_t = a, \pi]$, in which $\pi$ is a policy mapping sequences to actions. A Q-network can be trained by adjusting the parameters $\theta_i$ , at iteration *i* to reduce the mean-squared error in the Bellman equation, where the optimal target values $r + \gamma \max_{a'} Q^*(s', a')$ are substituted with approximate bootstrap target values $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$, using parameters $\theta_i^-$ from some previous iteration. This leads to a sequence of loss function $L_i(\theta_i)$ that changes at each iteration *i*. The Q-learning update at iteration *i* uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \tag{2.56}$$

in which $\gamma$ is the discount factor determining the agent's horizon, $\theta_i$ are the parameters of the Q-network at iteration *i* and $\theta_i^-$ are the network parameters used to compute the target at iteration *i*. The targets depend on the network weights, differently from supervised learning where are fixed before training begins. During the optimization, the parameters from the previous iteration $\theta_i^-$ are kept fixed when optimizing the *i*th iteration loss function $L_i(\theta_i)$. Differentiating the loss function with respect to the weights, we arrive at the following gra-

dient:

$$\Delta_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))\Delta_{\theta_i} Q(s, a; \theta_i)] \quad (2.57)$$

Rather than computing the full expectations, it is common to optimize the loss function by stochastic gradient descent, updating the weights of the neural network at every time step, replacing the expectations using single samples, and setting $\theta_i^- = \theta_{i-1}$.
Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximation such as a neural network is used to represent the action-value function, due to the correlations present in the sequence of observations, the fact that small updates to Q may change the policy and therefore change the data distributions, and the correlations between the action-values (Q) and the target values.
To overcome theses instabilities, two different techniques are used:

- the use of a mechanism called *experience replay* that randomizes the selection of training samples over the data, removing correlations in the observations sequence

- the use of an iterative update that adjusts the action-values (*Q*) towards target values that are only periodically updated, reducing correlations with the target

At each time-step *t*, the agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is stored in a data set $D_t = \{e_1, ..., e_t\}$. To perform experience replay, during learning Q-learning updates are applied on samples of experience *(s,a,r,s')∼U(D)*, drawn uniformly at random from the pool stored samples D. The uniform sampling gives equal importance to all the transitions in the memory replay. More sophisticated sampling strategies that emphasize transitions from which we can learn the most are also possible: they use *prioritized memory replays*.
The target network parameters $\theta_i^-$ are only updated with the Q-network parameters ($\theta_i$) every C steps and are held fixed between individual updates. To be precise, every C updates we clone the network Q to obtain a target network $\hat{Q}$ and use $\hat{Q}$ for generating the Q-learning targets $y_j$ for the following C updates to Q. This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$, for all *a* and hence also increases the target $y_j$, possibly leading to oscillations or divergence of the q-values. Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets $y_j$, making divergence or oscillations much more unlikely.

**Algorithm 2.1** Deep Q-learning with experience replay

Initialize memory replay D
Initialize action-value function Q with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**for** *episode=1,M* **do**
    Initialize $s_1$
    **for** *t=1,T* **do**
        Select an action $a_t$ according to the behavior policy
        Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
        Set $s_{t+1} = s_t$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in D
        Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ from D
        Set $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. $\theta$
        Every C steps set $\hat{Q} = Q$
    **end**
**end**

This algorithm is model-free: it solves the RL task directly using samples from the agent experience, without explicitly estimating the reward and transition dynamics $P(r, s'|s, a)$. It is also off-policy: it learns about the greedy policy $a=arg\max_{a'} Q(s, a'; \theta)$, while following a behavior distribution that ensures adequate exploration of the state space. In practice, the behavior distribution is often selected by an $\epsilon$-greedy policy that follows the greedy policy with probability 1-$\epsilon$ and selects a random action with probability $\epsilon$.

# 3
# System model

In this chapter, we will describe the reinforcement learning model on which the project analysis is based. In particular, we will describe the environment, the neural network's architecture used and the different RL techniques. For each studied scenario, a different neural network will be trained.

## 3.1 Environment

The project analysis was achieved by leveraging an extended version of Gym. Gym is a well known python library provided by OpenAI by which it is possible to create ad-hoc reinforcement learning environments. Additionally, the Keras* library was used to build the NNs used in this project.
The environment we developed represents a generic scenario where certain *target events* are located within a certain area. These events should be identified and monitored by a swarm of drones, which represents the agents of the RL scenario. Practically, the environment is composed by a square grid where one or more agents can move. Each cell of this grid has its own value, sampled from Gaussian distribution(s). Each distribution ideally represents a target in the real world. In the case of more targets, the square grid will have regions influenced by more than a Gaussian distribution. The framework allows the possibility to choose a lot of parameters for the creation of the "world": the size of the grid, the number of drones

*https://keras.io/

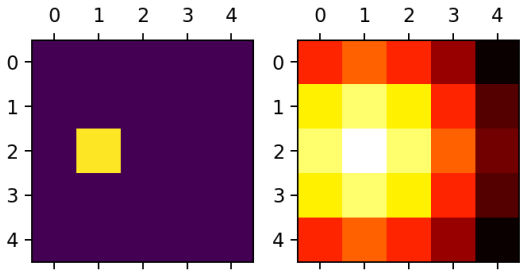


**Figure 3.1:** On the left the position of the drone, on the right the real map.

present on it, the number of Gaussian distributions present with their relative mean and variance and so on.

Figure 3.1 shows an example of the *single-drone* scenario: the environment, i.e. the grid with the Gaussian distributions, is represented by the map on the right, while the position of the agent within the grid is represented by the map on the left. The environment map is not the map that the drone sees at the beginning of each episode: the agent can only discover it completely after it has explored all the regions of the map. Hence, we can distinguish two different maps representing the scenario: the first is the *real map*, containing the true values of the targets; the second is the map discovered by the drones, the *known map*. At the beginning of the episode, all the second map cells are set to the value 1 (white color). As the drone moves within the map, it discovers the real value of each location it explores, as shown in Figure 3.2. The agent knows only the value of the cell corresponding to its position and

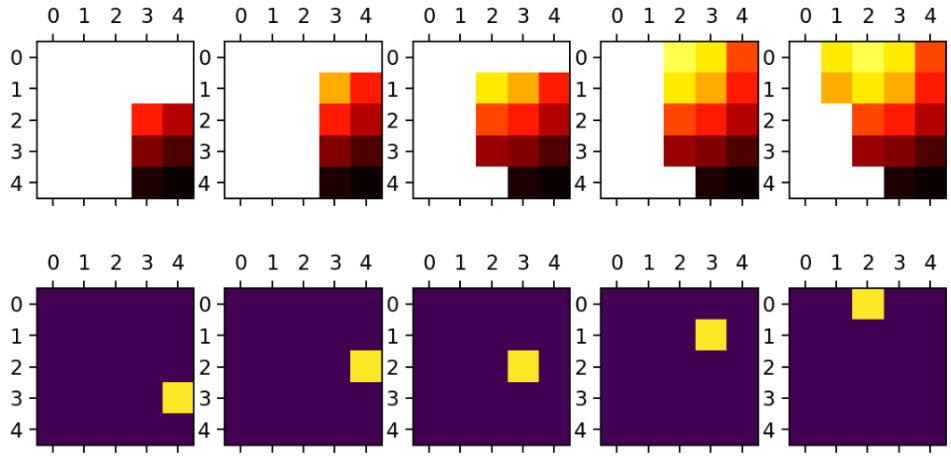


**Figure 3.2:** From left to right: at beginning the drone knows only a portion of the real map.

of the 3x3 window centered around it. The real map values are all in the range [0,1]. The size of the window is another tunable parameter before the training. This represents the fact that the drone is able to discover the value of both its position and a region in its surroundings, thanks to cameras, radars and other sensors installed on it. The idea is that the agent is encouraged to explore the map to find the center of the Gaussian distribution, i.e. the point where it can better observe the target event.

In the rest of the chapter we will analyze in detail different scenarios with different numbers of drones and targets

### 3.1.1 Single drone scenario

The first scenario studied is the one with a single drone moving within the grid. At every time step, the drone can choose one of the 5 actions *Stand, Left, Right, Down, Up* and receives a positive reward (+1) when it reaches the center of the target distribution, a negative reward when it tries to exit from the map (-1 and it will remain in the same cell), and a null reward in all the other cases (0 reward). Hence, the drone is expected to learn the following behavior: it tries to get closer to the highest point, without necessarily following the highest value cells along the path. A different behavior could be possible with a change of the rewards, that should be proportional to the values of the map, and not restricted to a subset of 3 values. For this reason the performances will be compared with a heuristic approach, the *look-ahead* algorithm, that chooses the best action at each time step that will maximize the sum of rewards of the next $t$ time steps, where $t$ is a predefined horizon. The heuristic approach requires exponentially larger computational resources as $t$ grows, and for this reason could not be applied in a real world scenario.

On this scenario we trained 3 models:

(a) One trained on a 6x6 grid with a low peak (value=0.4)

(b) One trained on 6x6 grid with an high peak (value=0.8)

(c) One trained on a 6x6 grid with 2 peaks, one lower (value=0.4) and one higher (value=0.8), receiving reward only when the highest is reached

The main studies that have been conducted on these 3 models are:

- we studied how the grid size, the learning rate, $\gamma$ and $\epsilon$ affect the learning of the task
- we compared the performances of model (a) and model (b) and we tested model (a) to situation (b) and viceversa
- we tested model (a) and model (b) in the situation (c). They should perform worse, because applied to a more difficult task than the one seen during their training.
- We tested model (c) on situation (a) and (b).
- We compared model (c) with the *look-ahead* algorithm with different number of steps

### 3.1.2 Swarm scenario

In a swarm scenario, we need to train a new model where each agent is aware of the presence of other agents. In this study, all the agents have a common goal: to reach the highest cells in the map, without overlapping each other on the same cells. To promote the partition of the targets between the agents, the system receives a reward when all the drones are located over different targets. Instead, it will receive a negative reward when two or more drones are on the same cell or a drone tries to exit from the map.
In this perspective we developed two different models: the first one is centralized, i.e., all the drones are jointly managed, the second one is distributed, which means that each drone takes decisions in an autonomous fashion. The distributed approach should train faster because the complexity of the action space is independent to the number of drones that must cooperate in the map: the same neural network is applied to each of them, so they have to choose the best action among the five ones. In this way each drone must learn which of the 5 actions to choose, taking in consideration the agent's own position, the position of the other agents and the distance from the targets in the map. On the other hand, the centralized approach needs to jointly optimize the action combinations of all the drones at once, so its complexity increases exponentially with the number of drones. We expect the centralized approach to be more efficient than the distributed version, especially in the first steps, because the first should be able to jointly optimize the action space and assign a target to each drone from the beginning, while in the latter the drones may try to reach the same target. A centralized system should be re-trained every time the features of the scenario, e.g. the number of drones, changes. Particularly, the complexity of the centralized system quickly increases as the number of agents grows. Indeed, the number of possible actions is given by $5^N$, where

N is the number of drones. In a distributed version, the model does not need a modification on the neural network architecture: we have 5 possible actions in output as always, and each agent has to choose the best one considering its position and the position of others. Hence, the only feature that changes is the composition of the system state, which also takes into account the position of the neighbor drones. In this case, we can train a model with a given number of drones, e.g. N, and extend it to a scenario with a different number of drones, e.g. $K \neq N$. It should probably perform worse in the case of more agents present in the environment with respect to the ones seen during training; however, transfer learning is possible, as the two scenarios have some similarities. The distributed approach is very attractive, since in a real scenario we must able to generalize to a number of agents not seen during training. In a centralized version instead, we have to know how many agents we have because we have to give an action to each of them. For this reason the number of output neurons of the neural network grows exponentially with respect to the number of agents, which makes this version impossible to scale to a huge number of agents, and also to a number of agents not seen during training. For this reason we want to demonstrate that, even though a centralized algorithm can have a better performance with enough training, a completely distributed approach is more flexible and adaptable.

On this scenario we trained 3 models:

(a) One trained with a distributed approach on an environment with 2 drones and 2 targets

(b) One trained with a distributed approach on an environment with 2 drones and 3 targets

(c) One trained with a centralized approach in an environment with 2 drones and 2 targets

Main studies that have been conducted are:

- We used the iterative adaptive hyper-parameter optimization search to find the best values for $\gamma$ and the value of the malus corresponding to the drone overlapping situation
- We studied how the different memory replay settings have affected the final performances

- We compared the performances of models (a) and (b) in scenarios with different number of targets
- We tested (a) and (b) on scenarios with only 1 drone and an arbitrarily number of targets
- We studied the best way to compute the reward of a centralized approach
- We compared models (a), (b) and (c) on scenarios with 2 drones and an arbitrarily number of targets

## 3.2 Techniques

### 3.2.1 Neural networks

As explained in Section 2.6, to train the model we developed, we used the double Q-learning algorithm, which was detailed explained in the previous Chapter. In particular, each system agent is associated to two neural networks, which we call *step model* and *target model*. They are identical from an architecture point of view but have different weights: the first one is the neural network trained during the learning, the second one is a copy periodically refreshed to compute the greedy-actions. Both neural networks, independently from their inner architecture, will have as many output neurons as the possible actions chosen by the agent.

In the case of a single drone scenario there are 5 possible actions, like in the distributed swarm one: (Left,Right,Stand,Up,Down). The value of the output neurons will then represent the Q-values of the 5 possible actions for the given state. The state will be the position of the drone and the map discovered by the agent until now in the case of a single-drone scenario; in a swarm scenario, there will be a second matrix with the map of all the other drones' positions. In the centralized multi-drone scenario instead, there will be more output neurons (all the possible combinations of the agent governed) and the same input as in the *Distributed* case.

The general architecture of the CNN used in the entire project is very simple:

1. Input layer with 2 or 3 square matrices (if single agent or multi agent scenario) with variable size (dependent on the size of the grid world)
2. Conv2D layer with 20 filters, kernel of size 3, padding="same", strides=(1,1) and relu as activation function

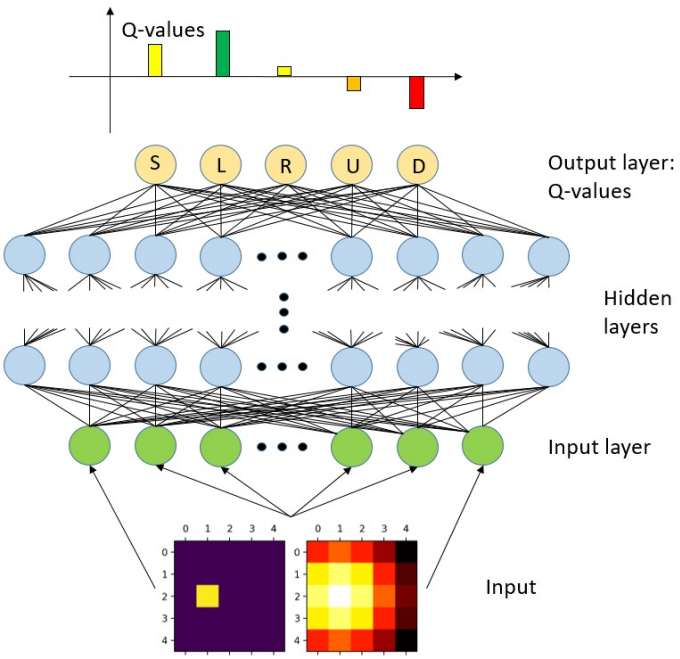


**Figure 3.3:** Single drone neural network case

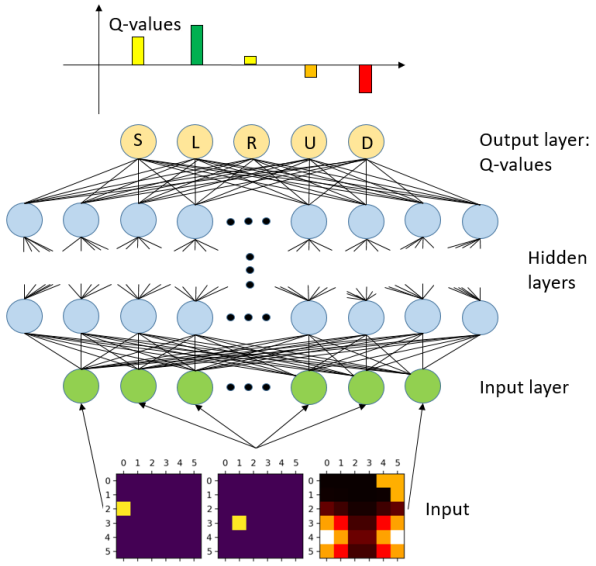


**Figure 3.4:** Distributed neural network case

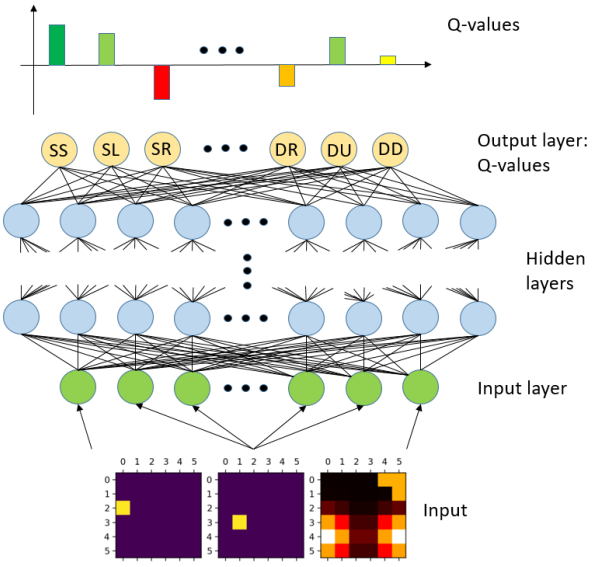


**Figure 3.5:** Centralized neural network case (2 drones)

3. Conv2D layer with 20 filters, kernel of size 2, padding="valid", strides=(1,1) and relu as activation function

4. Conv2D layer with 20 filters, kernel of size 2, padding="valid", strides=(1,1) and relu as activation function

5. Flatten layer

6. Dense layer with as many neurons as the number of possible actions (5 in the case of single agent/distributed version, 25 or more in the case of centralized), without activation function

The choice to use 3 convolutional layers is a free choice, pushed by the need to have at least some non-linear transformations from the original input. In our research, we established that three layers were sufficient to handle the complexity of the problem in the various scenarios. For the hidden layers we used *ReLU* activation function. Instead, the last layer, the output layer, does not have any activation function because we do not know the possible limits of the Q-values. The loss used is always the mean squared error.

### 3.2.2 Pre-training

Before the training, in all the considered scenarios, a pre-training phase has been done: 500 episodes with 100 steps have been computed and stored on disk to create a dataset on which the neural network can start to train just at the beginning of the training phase. This is done to make the neural network able to generalize to different situations also at the beginning of the training phase, instead of starting to train only on those cases that it has seen in the first episodes and attempting to generalize only when the exploration has just reduced a lot.
In the single-drone scenario, at every time-step the drone chooses 1 of the 5 possible actions and its position on the left map is updated, as well as the *known map*. In the distributed swarm environment instead, at every time step only one agent chooses an action while the other one remain in their current position to make the training simpler.
At every time step, a tuple [actual state, new state, action, reward] is created and stacked on top of the dataset.

### 3.2.3 Exploration policies

Exploration is a fundamental phase in RL tasks. Indeed, if the exploration done is not sufficient, the learner could not have seen which is the best action for a sufficient number of states. So, in the first phase of the training, we must ensure that the learner tries the actions at random, following the behavior policy with a probability that decreases during training. For this reason, 3 different $\epsilon$-greedy profiles have been studied: the goal is to see if they can affect the final performances.

### 3.2.4 Learning - how to train

After the creation of the pre-training dataset, the training-procedure starts: 1000 episodes are presented to the agent(s), and during these episodes the drone moves in the grid choosing either a random action or a greedy action. The probability to choose a random action inside each of the 1000 episodes is defined by one of the 3 profiles plotted in Figure 3.6. During this procedure, all the tuples [actual state,next state,action,reward] are stacked on top of the pre-training dataset, enlarging it until the end of the training phase. During each episode, a batch of some tuples is sampled randomly from the dataset and the *step model* neural network is trained on it, and less frequently the weights of the *target model* are updated with a

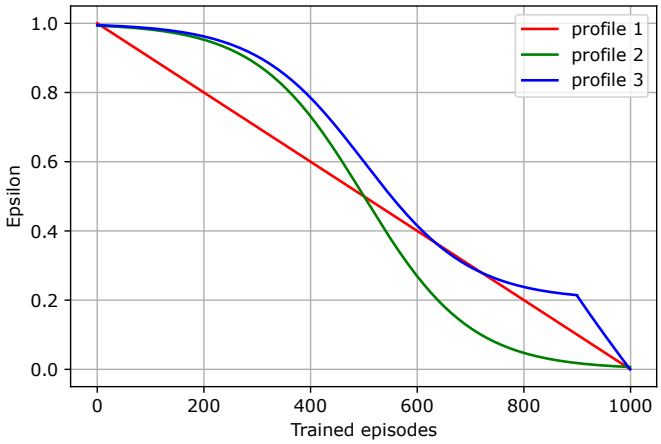


**Figure 3.6:** How much exploration is performed during the training.

copy of the actual weights of the first neural network.

As we have said in chapter 2, a common technique used in Reinforcement Learning is the memory-replay. Every time we train the neural network, a batch from there is sampled and the model is trained to predict the Q-value corresponding to the actions made in each situation inserted in the batch.
The rewards received by the agents are relatively rare for 2 reasons:

- we have non-zero rewards only when targets are reached, the agents are in the same cell or they try to exit from the grid
- with bigger maps, rewards are less probable in the state-action space.

For this reason, it is important to ensure a correct learning during the 1000 episodes, trying to show more often some situations than others. For example, there could be some situations where the neural network has trained sufficiently and that can be seen less frequently during subsequent training steps. There are also some combinations of action-states that are not so important for the learning task, because they will be never reached. For this reason, even though the neural network is not able to perform well in those situations, we can neglect them. To solve this kind of problem, 4 kinds of memory-replay have been studied:

1. Standard memory-replay: all states have equal probability to be chosen during the sampling-phase

2. Prioritized memory-replay: every 50 episodes we increase the sample probability of those transitions on which the highest Q-value has changed.

3. Rewards-based memory-replay: there are more memory-replay datasets, each containing states that bring the same rewards. For example a memory-replay with action-state combinations that bring positive rewards, null rewards, or negative rewards. During the training phase, the neural network will train more often on some of them.

4. Prioritized rewards-based memory-replay: similar to the third one but with the same sampling probability of the second one applied in each of the different memory-replay datasets.

### 3.2.5 Hyper-parameters Optimization Search

Reinforcement Learning is a difficult process in which many parameters must be tuned, as a wrong selection of them can interfere with the learning task. The RL algorithm itself has several parameters (such as the profile of $\epsilon$, the learning rate, $\gamma$, the rewards, etc), and adding neural networks further complicates the issue. Since the tasks that have to be learned in this project are relatively simple, most efforts have been focused on the RL parameters.
For this reason, a hyper-parameter search has been performed before the training of the swarm scenario, to discover which were best values for the learning rate, $\gamma$ and the malus of the overlapping drones situation. To find them, it is necessary to compute all possible combinations of the considered parameters. Such kind of search, when applied to traditional supervised learning tasks, can be boosted using only a representative fraction of the original dataset. In the RL scenario instead, is quite difficult to say which is the best way to save time during the hyper-parameters search: in this project we sampled the mini-batch less frequently during the hyper-parameter search. When the best values are found, the model is trained more. This kind of pre-training can benefit a lot from parallelism, trying a different combination of the parameters in every machine.

There exist a lot of automatic hyper-parameters optimization search. Some of them are:

- Grid Search
- Random Search
- Bayesian Search
- Gradient Search

**Grid Optimization**: when there are 3 or fewer hyper-parameters, the common practice is to perform grid search. For each hyper-parameter, the user selects a small finite set of values to explore. The grid search algorithms then trains a model for every joint specification of hyper-parameter values in the Cartesian product of the set of values for each individual hyper-parameter. The experiment that yields the best error is then chosen as having found the best hyper-parameters. Grid search usually works best when it is performed repeatedly, changing the limits of the hyper-parameters whose best values are found around there, or zooming in on the best regions of each hyper-parameters found at some point of the process. Unfortunately, if there are $m$ hyper-parameters, each taking at most $n$ values, then the number of training trials required grows as $O(n^m)$. The trials may be run in parallel, but even parallelization may not provide a satisfactory size of search, due to the exponential cost of grid search [21].

**Random Search**: Random Search replaces the exhaustive enumeration of all combinations by selecting them randomly. It can outperform Grid search, especially when only a small number of hyper-parameters affects the final performance of the machine learning algorithm. [21]

**Bayesian Optimization**: Bayesian optimization is a global optimization method for noisy black-box functions. Applied to hyper-parameter optimization, Bayesian optimization builds a probabilistic model of the function mapping from hyper-parameter values to the objective evaluated on the task. By iteratively evaluating a promising hyper-parameter configuration based on the current model, and then updating it, Bayesian optimization, aims to gather observations revealing as much information as possible about this function and, in particular, the location of the optimum [22]. It tries to balance exploration (hyper-parameters for which the outcome is most uncertain) and exploitation (hyper-parameters expected close to

the optimum). In practice, Bayesian optimization has been shown to obtain better results in fewer evaluations compared to grid search and random search, due to the ability to reason about the quality of experiments before they are run [23].

**Gradient Optimization**: For specific learning algorithms, it is possible to compute the gradient with respect to hyper-parameters and then optimize the hyper-parameters using gradient descent. The first usage of these techniques was focused on neural networks [24].

### Iterative Adaptive Hyper-parameter Optimization Search

Trying to simulate some of the features of these techniques, a custom optimization search method has been developed for this project. The algorithm requires to insert in input the limits of the hyper-parameters values, the number of attempts, the number of rounds and the search type. The search works in the following way:

1. all possible combinations of the hyper-parameters values are computed at their limits and in an arbitrarily number of additional points inside the ranges. The additional points can be sampled between the limits with a grid search (see at Round 1 of Figure 3.7) or with a random search and their numbers is dependent by the attempt value defined by the user.

2. For each hyper-parameter the algorithms focuses on the best region found in round i, and tries new values inside it at round i+1. New values are sampled with a grid search technique inside the new region (Round >1 of Figure 3.7) or with a random search.

3. The process repeats for a number of rounds defined by the user

Figure 3.7 is an example of an optimization search with only 2 hyper-parameters. At round 1 we can have 3 possible cases: the trivial case when *attempts* is set at value 0, where the optimization search is useless because it will try only the middle value of the limits of each parameter; the *attempts*=2 case, where only the borders of the hyper-parameters are tested and then 2 new values are sampled at the next round inside the best region; the *attempts*>2 case, where an additional number of equally spaced values are tested at round 1, but from the next round only 2 attempts are tested (for computational reasons).

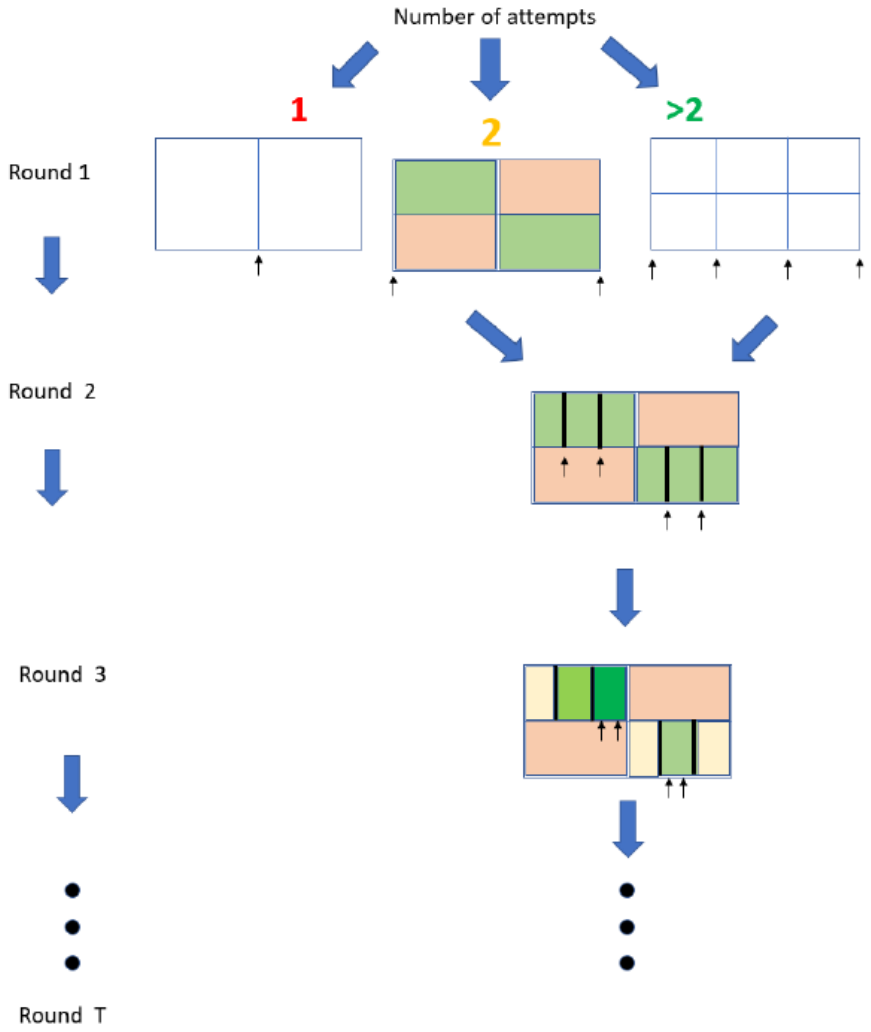


**Figure 3.7:** Optimization search procedure. Each row of the table corresponds to a different parameter that needs to be tuned; the columns represent the values studied for each parameter; the color indicates the performance of the model with that particular value for the parameter. At each round of the optimization search, the algorithm tries some new values for each parameter inside the best range found until now for each of them.

# 4

# Results - single drone scenario

## 4.1 One drone, one target

### 4.1.1 Reinforcement Learning optimization

For the pure Reinforcement Learning part, 4 aspects have been studied:

- Learning rate $\alpha$
- $\gamma$
- How performances scale with the map size
- $\epsilon$-shape

Among all the parameters of reinforcement learning and the neural network, we found that the learning rate is the most important one: with the wrong choice, the agent is never able to learn the task. For this reason, the tuning of this parameter is the first thing to do: we have studied 4 different Keras optimizers, 3 using SGD with different initial learning rates and 1 that is an evolution of Adam invariant with respect to the learning rate, called Radam[25]. The features chosen for the optimizers are the following:

| Optimizer | Learning rate | Momentum | Nesterov flag | Notes |
|---|---|---|---|---|
| SGD | 0.001 | 0.9 | True | - |
| SGD | 0.01 | 0.9 | True | - |
| SGD | 0.1 | 0.9 | True | - |
| Radam | - | - | - | Default values |

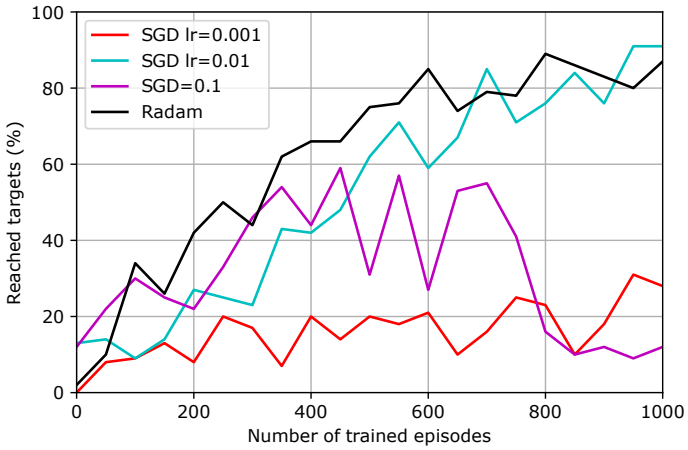


**Figure 4.1:** The learning rate of the neural network is determinant for the learning of the task.

The results that we see in Figure 4.1 are obtained with this main parameters configuration:

- 6x6 world
- 500 pre-training episodes with 100 steps each
- 1000 training episodes with 100 steps each
- Neural network trained every 2 steps
- $\gamma = 0.9$

The performance of Radam seems quite good and comparable with the right learning rate choices for SGD. Since it's invariant with respect to the learning rate and it performs well, from here we will always use it.

### The right choice of $\gamma$

Using the same parameters described above, here a study on $\gamma$ is presented: you can see that it is another an important parameter. For the next tests we will use $\gamma = 0.9$

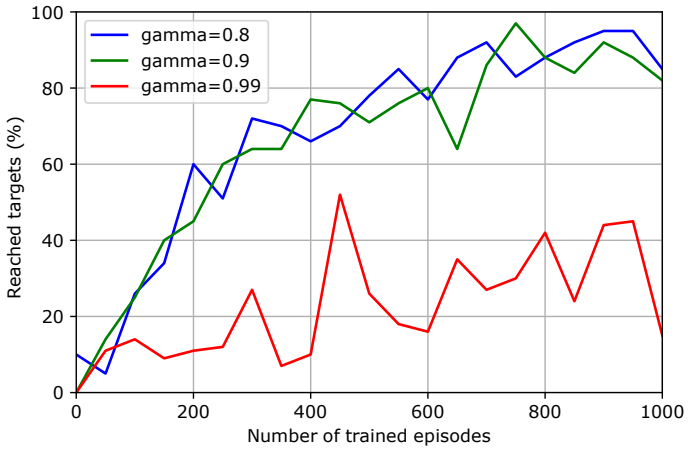


**Figure 4.2:** An high value of $\gamma$ can give more importance to future events, but it requires more training.

### Bigger map, more training

With the same parameters used until now (Radam,$\gamma = 0.9$,...) a comparison between different world size performances is illustrated in Figure 4.3 This plot shows how the performance decreases if we train the model for a bigger world with the exactly same amount of training and with the same parameters. The solution to scale to a bigger scenario is to train the model more frequently, with longer episodes and with a proper selection of the hyper-parameters for each different situation. We hope that a proper selection of the best hyper-parameters is not necessary for every possible change of the environment, such as a bigger map. Probably, an higher amount of training should be sufficient to increase the performance.


**Figure 4.3:** The performance decreases with a bigger map

### $\epsilon$-profile

For the profile of the $\epsilon$ parameter, 3 different choices have been studied (see Figure 3.6 of Chapter III). The idea is to study how the amount of exploration done during training can affect the performance on the task learning. An insufficient amount of exploration can drastically reduce performance, due to the fact that the agent has not experienced a sufficient number of transitions. For this reason, even an arbitrarily long training procedure using large computational resources, could be useless with an incorrect exploration-profile. The exploration profiles we tested did not have any significant effect on the training, as shown in Figure 4.4 (5x5 map), probably due to the relative simplicity of the task.

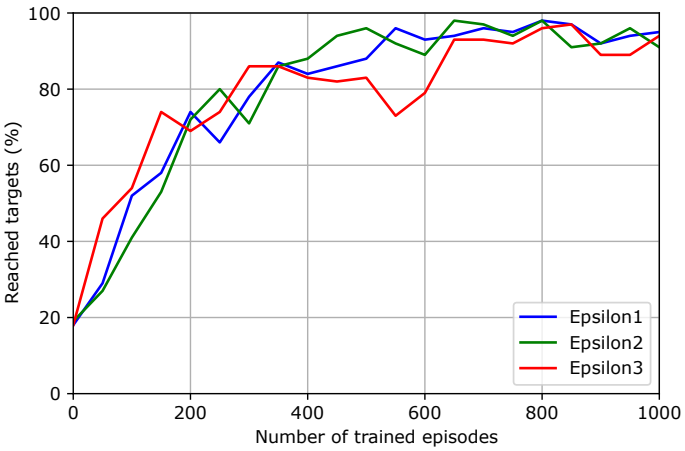


**Figure 4.4:** No particular evidences between the different epsilon-shapes.

### 4.1.2 Peak invariance

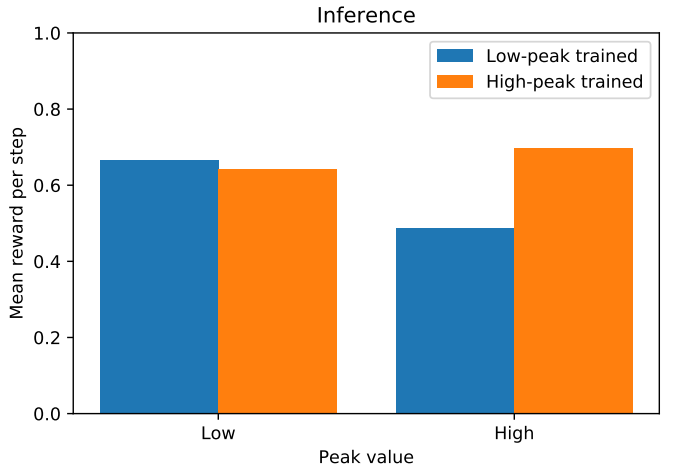


**Figure 4.5:** The model trained on an high target outperforms.

After optimizing the hyper-params, we trained 2 models where the only difference is the target's value: in the *Low* case, the target is set at 0.4, in the *High* case at 0.8. Ideally, we can think that the first one could outperform the second one, because recognizing a low target makes the agent more robust to an high one. However, from a NN point of view, recognizing a lower target could be harder, given that it continues to see the unexplored areas at an high value.

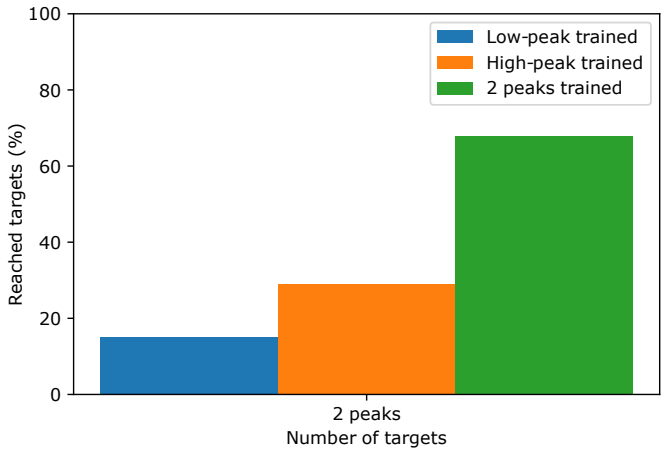


**Figure 4.6:** With a 2 targets environment, the model trained on 2 targets is the best one.

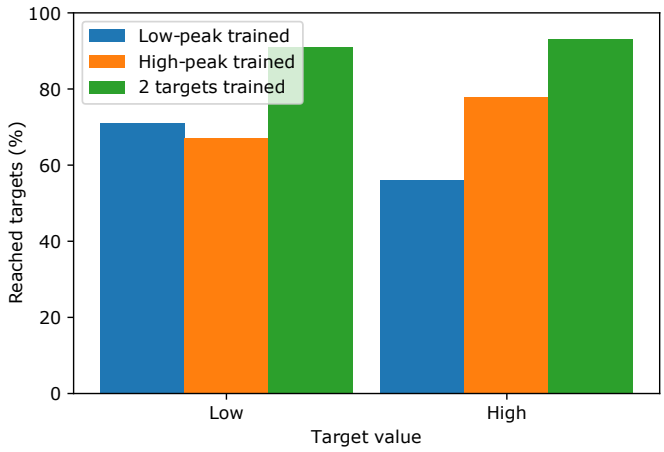


**Figure 4.7:** Model trained on 2 targets outperforms previous ones when there's only 1 target

Then we trained a model in an environment with 2 targets, one higher and one lower with the same height of the targets *High* and *Low* used in the previous section. It is interesting to see that this model is obviously the best one in this kind of situation (2 targets instead of one, Figure 4.6), but it also performs better in the 2 situations studied before (Figure 4.7).

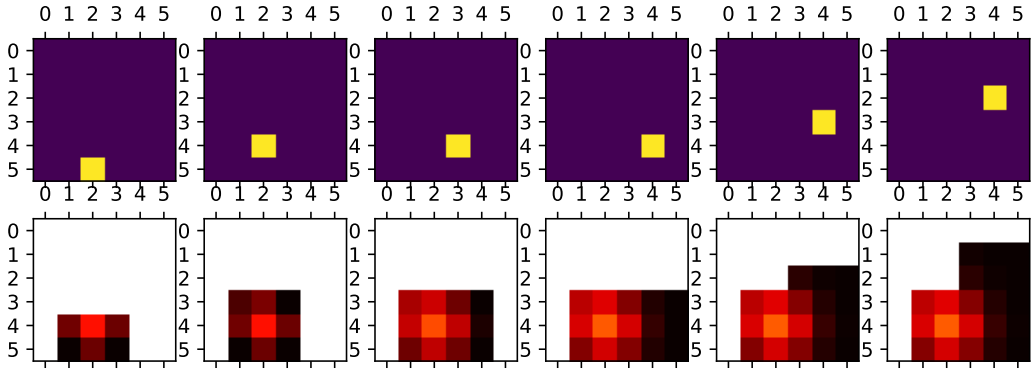

**Figure 4.8:** First 6 steps of an episode: it's possible to see that the model is able to not stop in the lower target

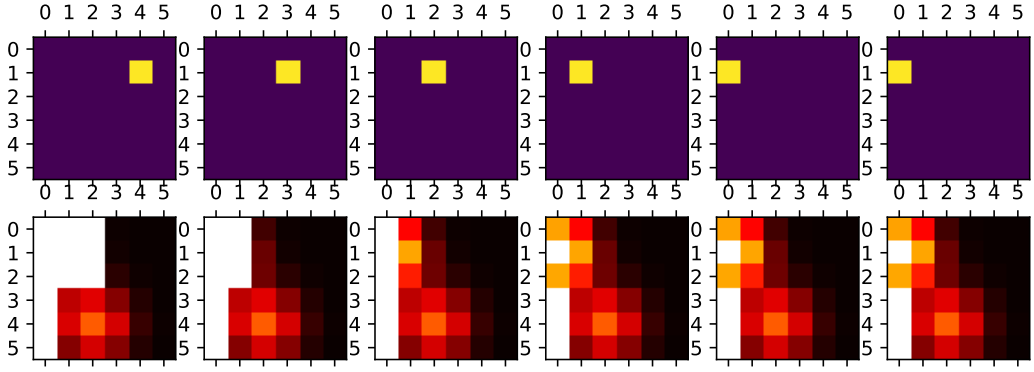

**Figure 4.9:** Next 6 steps

Figure 4.8 and 4.9 show a successful example of the model. Even if in the first steps it is near a target, it is able to recognize that it corresponds to the *Low* targets that it has seen during training. For this reason it starts to explore the unknowns regions until the discovery of the *High* target.

### Look-ahead (heuristic) comparison

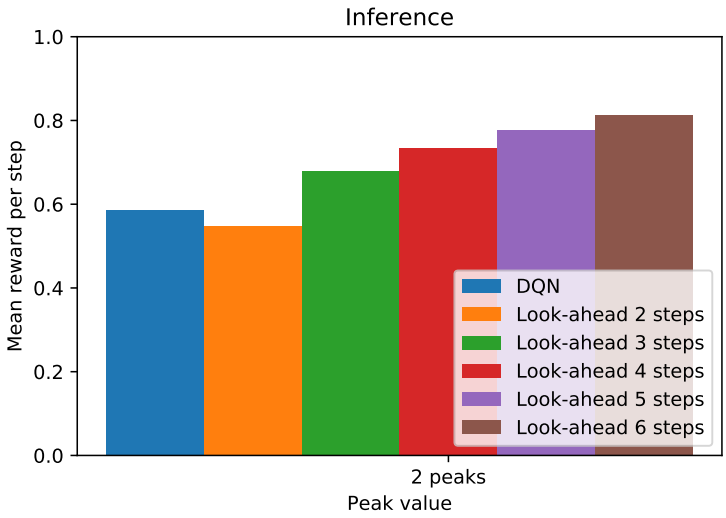


**Figure 4.10:** Look-ahead outperforms the neural network

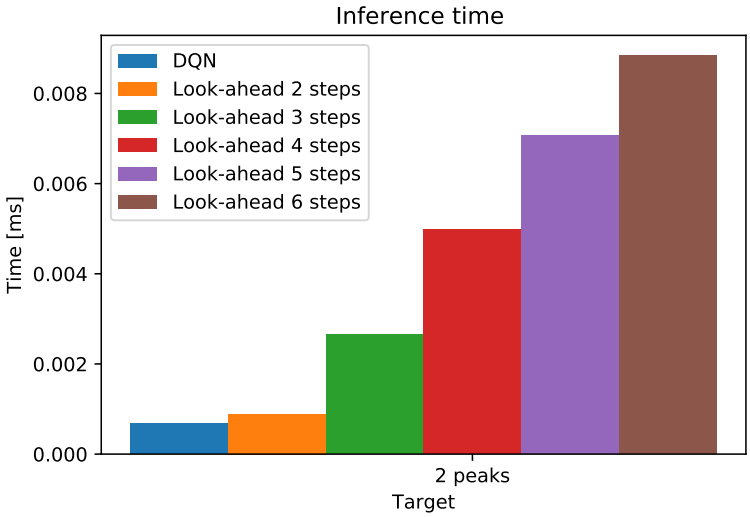


**Figure 4.11:** An heuristic approach is infeasible in a real world scenario

Here we compare the performances of the Neural Network with the heuristic approach described in Chapter III. In Figure 4.10 it is possible to see that the heuristic approach performs

better. Ideally, we would expect that the DQN should perform at least as well as *Look-ahead* if trained in a proper way, because the heuristic approach could be trapped in the lower target after the exploration of the map. However the world might be too small to observe these situations and the model has not trained sufficiently.
Instead, Figure 4.11 demonstrates our hypothesis about the time required from the heuristic approach to compute the next action. When we increase the steps of the *Look-ahead* algorithm, forcing it to compute an higher amount of possible action-combinations, we increase too much the time required. In a real world scenario, where the map is bigger, this approach becomes infeasible.

# 5

# Results -swarm scenario

In the swarm scenario, we have always used 6x6 grids, with targets with the same height and trying to avoid their overlap, dividing the map in 4 regions and assigning a target to each of them.

## 5.1 Distributed version

In the distributed version, we performed a hyper-parameter optimization search on 2 parameters: $\gamma$ and the malus corresponding to the overlapping drones situation. We then studied the 4 types of memory replay described in Chapter III and trained 2 models: the first one with 2 drones and 2 targets, the second one with 2 drones and 3 targets. We then compared their performances in different scenarios.

### 5.1.1 Hyper-parameters optimization search

An optimization on some hyper-parameters has been performed. In particular, we found that the best value of $\gamma$ was 0.8, and the best malus corresponding to the situation of at least 2 overlapping drones was -0.4. Figure 5.1 shows the training performances of all the possible combinations of the hyper-parameters studied. The 2 optimal values obtained for $\gamma$ and the malus are used in all the studies of this multi-drone scenario.

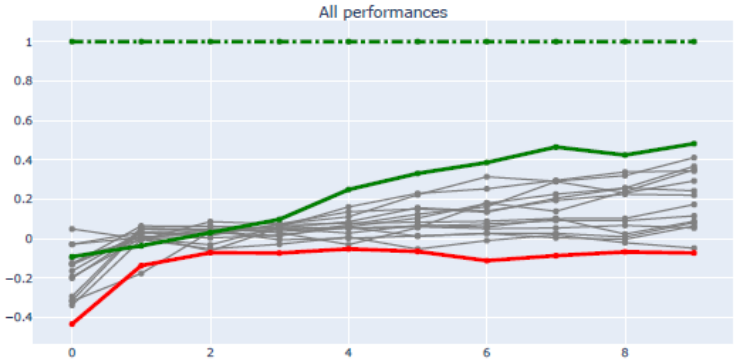


**Figure 5.1:** Mean reward per step of all the parameters combinations studied.

### 5.1.2 How to learn - memory replay settings

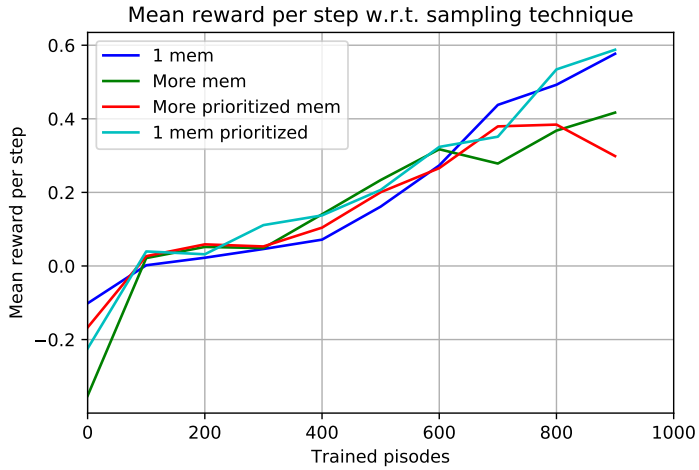


**Figure 5.2:** Single memory replay seems to be a better solution.

We also studied different sampling strategies on the creation of the mini-batch. Among the 4 different kinds of sampling proposed in Chapter III, we found that the best strategies seem to be the ones with a unique memory replay. This could be due to the fact that our attempt to guarantee a more frequent training on transitions with a positive reward could damage the training of transitions with negative rewards, especially the one involving overlapping. Indeed, if in this kind of environment is quite simple to compute the probability of an action that takes a negative reward trying to exit from the map, it is instead more difficult to compute the probabilities of a negative reward due to an overlapping. So, we could have

set a probability sampling on those transitions lower with respect to a random sampling, negatively affecting the learning on those transitions. Since the final performances between the single memory-replay and the one prioritized are not so different, for the following tests we used the first one, because the second one is more computationally demanding.

### 5.1.3 2 TARGETS VS 3 TARGETS MODELS

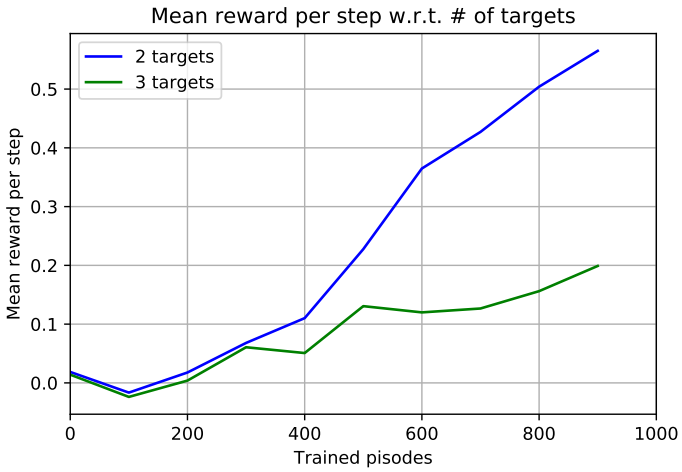


**Figure 5.3:** Model trained on 2 targets vs model trained on 3 targets.

Then we trained 2 models: one in a scenario with 2 targets, and one with 3. Since more targets are present in the map, we can expect an higher mean reward if the model is trained properly, because the mean distance between a drone and a target should be lower. Instead, looking at Figure 5.3, we can see that the second model converges slower with respect to the first one. We can make 2 hypothesis to explain that:

- the second scenario is more difficult to learn, so more training is required
- the map is too small and our technique to avoid the overlap of the targets is not perfect: there are more overlapping situations. In that case, the drones could have more difficulties, since their targets are near but they must also try to avoid the overlapping.

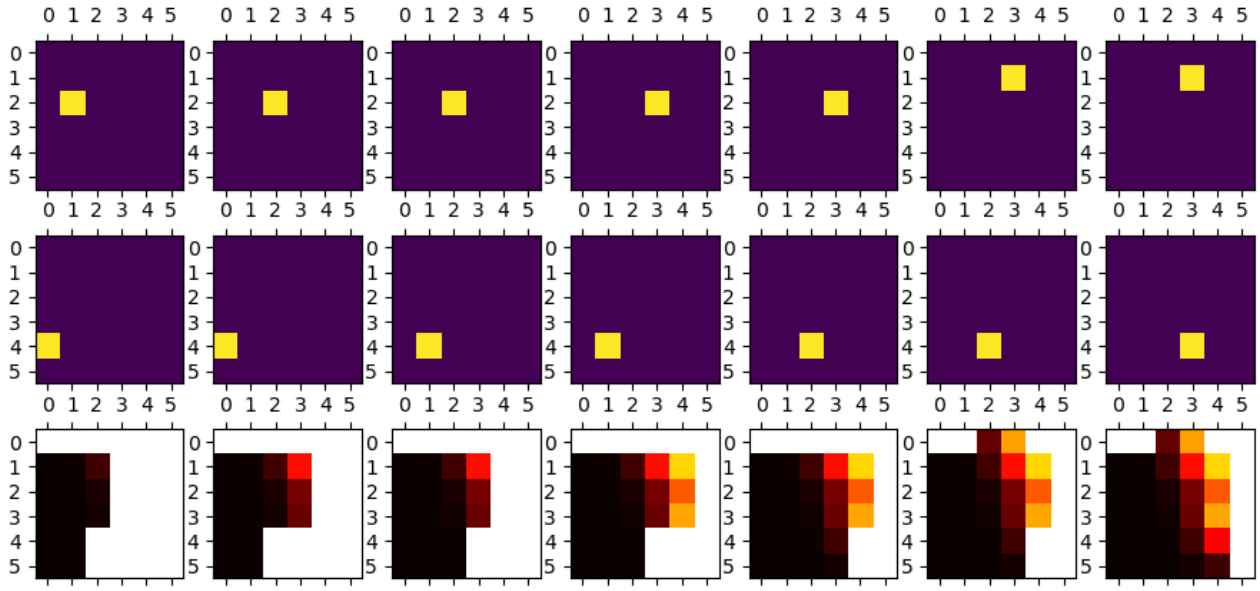

**Figure 5.4:** First 7 steps (from left to right) of a successful test episode of the model trained on 2 targets.

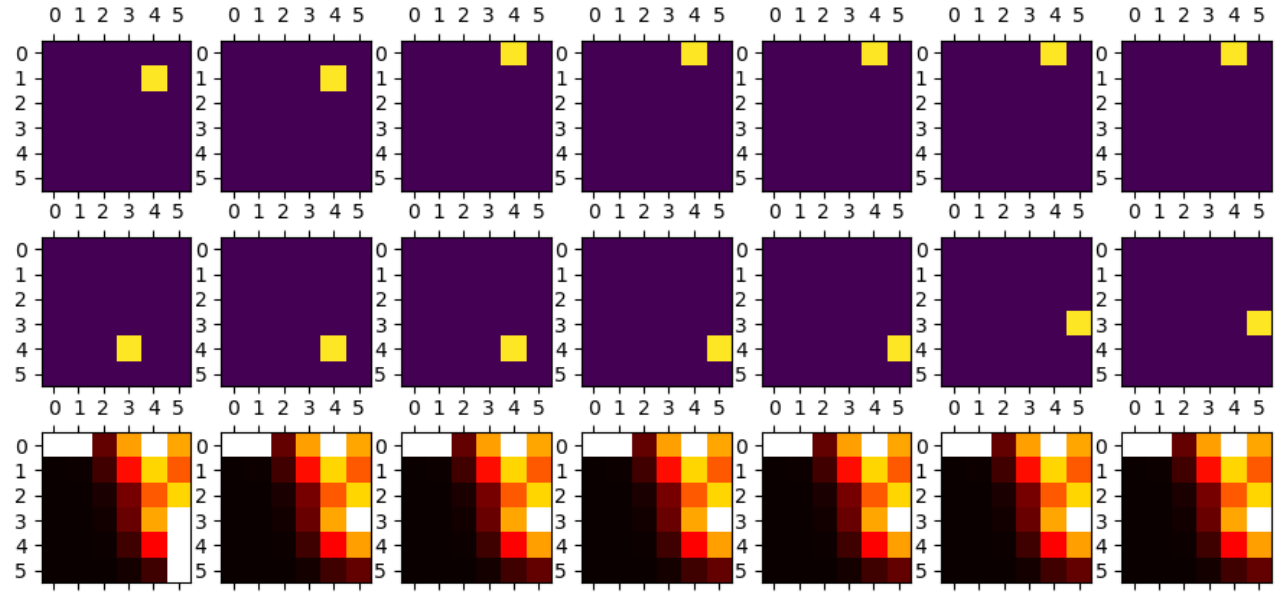

**Figure 5.5:** The following 7 steps: remember that at each time step only a drone is moved.

### 5.1.4 Inference - more targets

In Figure 5.6 we plotted how the 2 models studied in the previous paragraph perform on different scenarios. The number of times both drones reach their targets is still too slow for both models, even in the scenario in which they have been trained. From this plot you can see how much more difficult is this task to be learned, with respect to the single drone scenario.

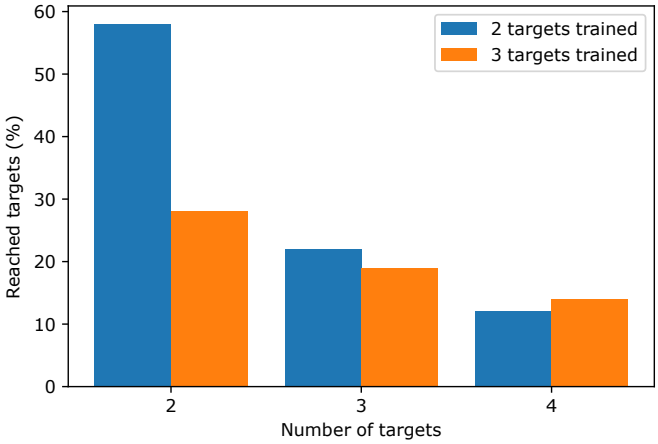


**Figure 5.6:** Generalization on the number of targets.

### 5.1.5 Inference - 1 drone

Then we removed the assumption of a multi-drone scenario and we tested the 2 models on a single-drone situation, comparing their performances with the model trained in the single-agent scenario on 2 targets. The first observation that we can make is that all the 3 models perform worse when tested on scenarios different from which they have been trained. When only 1 target is present, the task is simpler, but it is a task on which they have been not trained. We can say that in that situation the performance decrease also because the mean distance between the drone and a target is bigger than the situations in which more targets are present. So the model must be able to capture farther rewards. But the model trained on a single agent scenario seems to be able to adapt itself to this different environment. When the number of targets present increases, *Single 2 targets* seems to be more robust.

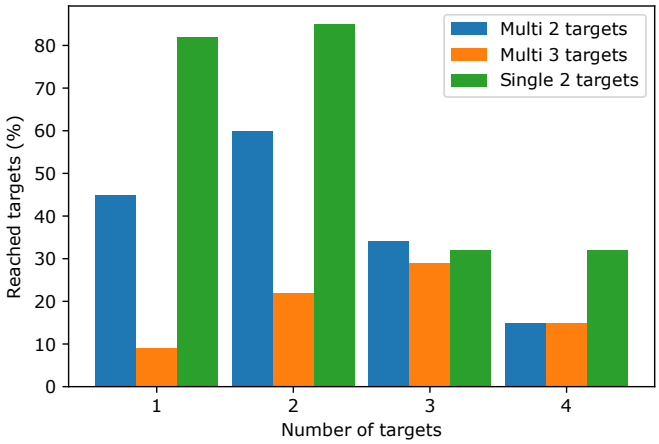


**Figure 5.7:** Generalization on the number of targets.

## 5.2 Centralized version

### 5.2.1 Computation of the centralized reward

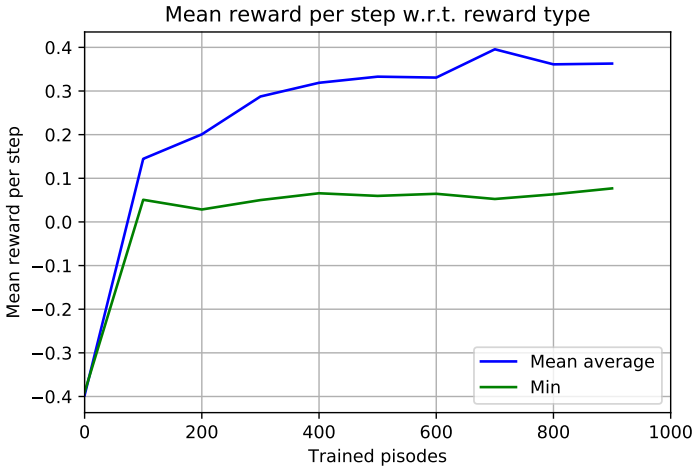


**Figure 5.8:** Performance with respect to the way centralized reward is computed.

For the centralized approach, we made a study on the way the centralized reward is computed. We have tested 2 techniques: the mean average of the rewards of both the drones, and the minimum value of them. The first resulted the best, maybe because the latter classified as null rewards transitions that in the first one are classified with a positive rewards. We can

think that the first one starts to converge sooner, but the latter could be more robust with respect to negative actions. A linear combination of them could be an interesting study.

### 5.2.2 Comparison with distributed

Looking at Figure 5.8, we can see that the final performance of the *Mean average* case is lower than the one obtained with the distributed version. We also observe a slower convergence of the model. This validates our intuition about the higher amount of training necessary for the centralized approach. Finally, we compared the performances of the model trained with

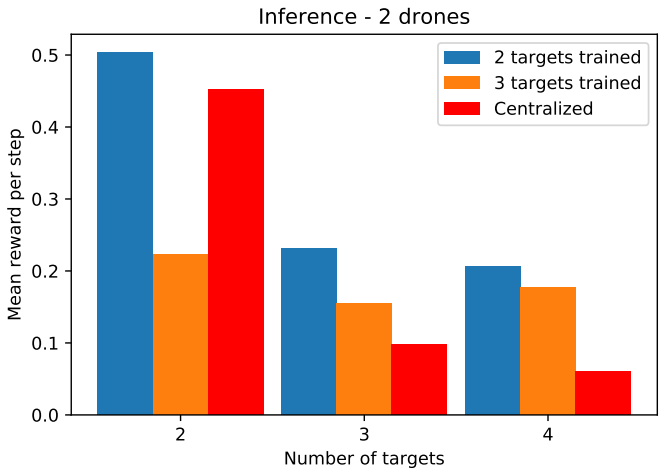


**Figure 5.9:** Performances on different situations of different models.

the centralized approach and the 2 models trained in the distributed scenario with 2 and 3 targets. They have received the same amount of training and this is clearly seen in Figure 5.9: the *Centralized* model should be the best one on 2 targets and the *3 targets trained* model the best one in the case of 3 targets. Instead we are not able to see this, probably because these 2 models require more training with respect to the one trained on 2 targets of the distributed approach.

# 6
# Conclusion

In Chapter I we introduced the goal of this project, highlighting the importance of Reinforcement Learning in complex situations and illustrating some of the possible applications of a drone swarm in a real world scenario. In Chapter II we introduced the theory on which RL is based, from the MDPs to Temporal Difference Learning and its extension to the Deep Learning approach. In particular, we focused on Deep-Q-Learning and the use of CNNs to implement it. In Chapter III we described our system model and the different environments we have considered, i.e. the single-drone and the multi-drone scenarios. In the same Chapter, we presented the learning strategies that we implemented to reach the project's objectives. Finally, in chapter IV an V we analyzed the results of our simulations, highlighting the benefits and the drawbacks of the proposed strategy over conventional approaches.

From the analysis of the results, we have verified our predictions on the models' performances. Models that we have thought to be the best, require more training for their complexity. If we train them with an insufficient amount of training, they are not the best ones. The centralized approach, for example, was not able to outperform the distributed one. Similarly, a distributed model trained on 3 targets was not able to outperform the one trained on 2 targets when tested on a scenario with 3 targets.
In the single drone scenario, we trained three different models, able to reach the target or the highest target when more than one is present. We proved the impracticability of a heuristic approach in a real-world scenario justifying the importance of a learning strategy. The latter

should be able to generalize to different scenarios from the ones seen during training: we have shown that the model trained in the single-agent scenario with 2 targets is able to outperform the two models trained in a single-target environment when only one peak in the map is present. This kind of generalization is not possible with a heuristic approach, because it must be re-designed for each different scenario.
We said that the centralized approach should outperform the distributed one because it is able to jointly optimize the action space of the controlled drones, but it could have required more training since the larger action space. We failed to demonstrate the first hypothesis: perhaps not only more training is required, but also the adjustment of other parameters and the use of different learning strategies. Instead, to help the distributed approach to converge faster, it could be possible to add some extra information in input to the CNNs, such as the recent positions of other drones, and not only their actual positions. This could encourage the use of LSTMs and of Recurrent Neural Networks in general.

In this project, we have faced all the typical difficulties of a Reinforcement Learning framework. The amount of time spent on the search of the best parameters is the most important one: a heuristic approach as the one applied in the multi-drone scenario is very useful for the task's learning. Even an incorrect selection on a unique parameter can prevent our goal's achievement. More studied should be done on the NN architectures: it could be interesting to see which parameters of the NNs can improve the performance and how, such as the size of the network, the loss type and the activation function used. Due to the small size of the grid, no pooling layers have been used in the CNN: their use must be studied and tuned. We highlight that CNNs can have a 3D structure. Therefore, with some modifications, our work can be extended in a 3D scenario where drones are able to move along three directions. Also, Dropout could be a useful extension, helping the NNs to generalize better and creating a Bayesian approximation model if used also in the prediction phase. [26] [27][28].

From the RL point of view, some possible extensions could be the use of Transfer Learning strategies [29][30], Teacher-Student Curriculum Learning[31] and of Curiosity [32]. Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. The second task is usually more difficult to be learned: in this way, it may be possible to train a model over simple policies and than make it learn more difficult tasks. For example, if we want that our drones are able to manage also the energy of their batteries, we could start to train them using the neural networks ob-

tained at the end of the training of this project [29]. Teacher-Student Curriculum Learning (TSCL) instead, is a framework for automatic curriculum learning, where the Student tries to learn a complex task and the Teacher automatically chooses subtasks from a given set for the Student to train on. The Student should practice more of the tasks on which it makes the fastest progress, i.e. where the slope of the learning curve is highest [31]. Finally, Curiosity could be used to incentive the exploration. For well-explored trajectories, the loss should be small while in less-explored trajectories, the loss is supposed to be large. For this reason, it is possible to create a new reward function (called "intrinsic reward") that provides rewards proportional to the loss of the predictive model. Thus, the agent receives a strong reward signal when exploring new trajectories [32].

Instead, from a theoretical point of view, with a high number of agents, the application of Game Theory can be useful[33]. When multiple learners simultaneously apply reinforcement learning in a shared environment, the traditional approaches often fail. In the multi-agent setting, the assumptions that are needed to guarantee convergence are often violated and many new complexities arise. When agent objectives are aligned and all agents try to maximize the same reward signal, coordination is required to reach the global optimum. When agents have opposing goals, a clear optimal solution may no longer exist. In this case, an equilibrium between agent strategies is usually searched for [33]. To search such equilibrium, Game Theory strategies could be very useful.

# References


[1] Y. Zhang, S. Wang, G. Ji, and P. Phillips, "Fruit classification using computer vision and feedforward neural network," *Journal of Food Engineering*, vol. 143, pp. 167–177, 2014.

[2] C.-H. Li, S.-L. Wu, C.-L. Liu, and H.-y. Lee, "Spoken squad: A study of mitigating the impact of speech recognition errors on listening comprehension," *arXiv preprint arXiv:1804.00320*, 2018.

[3] T. Sainath and C. Parada, "Convolutional neural networks for small-footprint keyword spotting," 2015.

[4] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[8] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[9] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intelligent Transport Systems*, vol. 4, no. 2, pp. 128–135, 2010.

[10] Z. Zhou, X. Li, and R. N. Zare, "Optimizing chemical reactions with deep reinforcement learning," *ACS central science*, vol. 3, no. 12, pp. 1337–1344, 2017.

[11] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li, "Drn: A deep reinforcement learning framework for news recommendation," in *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee, 2018, pp. 167–176.

[12] M. Erdelj and E. Natalizio, "Uav-assisted disaster management: Applications and open issues," in *2016 international conference on computing, networking and communications (ICNC)*. IEEE, 2016, pp. 1–5.

[13] M. Campion, P. Ranganathan, and S. Faruque, "Uav swarm communication and control architectures: a review," *Journal of Unmanned Vehicle Systems*, vol. 7, no. 2, pp. 93–106, 2018.

[14] M. Egorov, "Multi-agent deep reinforcement learning," *CS231n: Convolutional Neural Networks for Visual Recognition*, 2016.

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

[16] Y.-H. Wang, T.-H. S. Li, and C.-J. Lin, "Backward q-learning: The combination of sarsa algorithm and q-learning," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 9, pp. 2184–2193, 2013.

[17] P. Dayan and T. J. Sejnowski, "Td ($\lambda$) converges with probability 1," *Machine Learning*, vol. 14, no. 3, pp. 295–301, 1994.

[18] I. STEPHEN, "Perceptron-based learning algorithms," *IEEE Transactions on neural networks*, vol. 50, no. 2, p. 179, 1990.

[19] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[22] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "Boa: The bayesian optimization algorithm," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*. Morgan Kaufmann Publishers Inc., 1999, pp. 525–532.

[23] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[24] C. K. Goh, Y.-S. Ong, K. C. Tan, and E. J. Teoh, "An investigation on evolutionary gradient search for multi-objective optimization," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 2008, pp. 3741–3746.

[25] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," *arXiv preprint arXiv:1908.03265*, 2019.

[26] T. T. Sung, D. Kim, S. J. Park, and C.-B. Sohn, "Dropout acts as auxiliary exploration," *International Journal of Applied Engineering Research*, vol. 13, no. 10, pp. 7977–7982, 2018.

[27] H. Wu and X. Gu, "Towards dropout training for convolutional neural networks," *Neural Networks*, vol. 71, pp. 1–10, 2015.

[28] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International conference on machine learning*, 2013, pp. 1058–1066.

[29] S. Gamrian and Y. Goldberg, "Transfer learning for related reinforcement learning tasks via image-to-image translation," *arXiv preprint arXiv:1806.07377*, 2018.

[30] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," *arXiv preprint arXiv:1811.04551*, 2018.

[31] M. Eppe, S. Magg, and S. Wermter, "Curriculum goal masking for continuous deep reinforcement learning," in *2019 Joint IEEE 9th International Conference on Devel-*

*opment and Learning and Epigenetic Robotics (ICDL-EpiRob)*. IEEE, 2019, pp. 183–188.

[32] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks," *arXiv preprint arXiv:1605.09674*, 2016.

[33] A. Nowe, P. Vrancx, and Y.-M. De Hauwere, *Game Theory and Multi-agent Reinforcement Learning*, 01 2012, p. 30.

# Acknowledgments

I have to thank my supervisor, professor Alberto Testolin, for the assistance and the advice on Neural Networks and on the Reinforcement Learning methodologies. He also inspired me on this topic and gave me the tools to *explore* it. My co-supervisor instead, professor Andrea Zanella, showed me which were the most interesting studies and comparisons to do. I have to thank also Dr. Federico Chiariotti who helped me with the theoretical part of Reinforcement Learning and Dr. Federico Mason for the creation of the entire environment on which this project this based. They both helped me on the analysis of the results.