UNIVERSIDAD DE SEVILLA
FACULTAD DE MATEMÁTICAS
DEPARTAMIENTO MATEMÁTICA APLICADA II

Drone coverage using deep reinforcement learning

Alina Kasiuk

UNIVERSIDAD DE SEVILLA
FACULTAD DE MATEMÁTICAS
DEPARTAMIENTO MATEMÁTICA APLICADA II

Drone coverage using deep reinforcement learning

Alina Kasiuk

Memory presented as part of the requirements to obtain a Master's degree in Mathematics from the University of Seville.

Supervised by: Prof. Dr. José Miguel Díaz Báñez

November 2020
Seville, Spain

Abstract

UAVs (Unmanned Aerial Vehicles) or drones have long been used to autonomously operate on a terrain and many strategies have been proposed when the environment is unknown. When the drone is tasked with a path planning problem in a unknown terrain, it must be able to correctly perceive its environment online and plan its path without human supervision. Particularly, covering or patrolling of a specific area has been a challenging optimization problem in robotics. This work addresses a particular covering task in unknown outdoor environments in which the drone has a limited power. The agent has to return to a base station when it is running out of battery. The problem is then to generate an optimal path that starts and ends at a base station and covers the target area through several tours. A covering planner based on Deep Reinforcement Learning is proposed where a Deep Q-network is trained to learn a control policy to approximate the optimal strategy at each step. Simulation results showed that the algorithm is able to learn and generalizes well to different types of environments. After multiple sequences of the training process, the virtual mobile drone gets information of the whole space with a coverage rate of over 80%. The experiments also demonstrate that the drone finds a trajectory balancing the goals of safe recharging and maximum coverage ratio.

Resumen

Los UAV (Vehículos Aéreo no Tripulados) o drones se vienen utilizando desde hace tiempo para operar de forma autónoma en un terreno y se han propuesto muchas estrategias para cuando el entorno es desconocido. Cuando el dron se le encomienda una tarea de planificación en un terreno desconocido, debe ser capaz de percibir correctamente su entorno en tiempo real y planificar su camino sin supervisión humana. En particular, cubrir o patrullar un área específica ha sido todo un desafío en el campo de la optimización en robótica. Este trabajo aborda una tarea de cobertura en entornos exteriores desconocidos en los que el dron tiene un tiempo de vida limitado. Por lo tanto, el agente tiene que regresar a una estación base cuando se está quedando sin batería. El problema es generar un camino óptimo que comience y termine en una estación base y cubra el objetivo usando para ello varios recorridos. En esta tesis se propone un planificador de cobertura basado en el aprendizaje reforzado profundo (Deep Reinforcement Learning) donde se usa un aprendizaje de una política de control para aproximar la estrategia óptima en cada paso. Los resultados de la simulación mostraron que el algoritmo es capaz de aprender y generaliza bien a diferentes tipos de ambiente. Después de múltiples secuencias del proceso de entrenamiento, el dron móvil virtual obtiene información de todo el espacio con una tasa de cobertura del 80%. Los experimentos también han demostrado que el dron es capaz de encontrar una trayectoría equilibrando los dos objetivos planteados: recarga segura y ratio de cobertura.

Acknowledgements

Throughout the writing of this thesis I have received a great deal of support and assistance. I would first like to thank my thesis advisor, Prof. Dr. José Miguel Díaz Báñez, whose dedicated support, guidance, and comments have been invaluable throughout this study. He paid close attention to the progress of my work. This project would not have been possible without his support.

I would also like to thank all the members of the Galgo research group for friendly discussions on the topic and Miguel Ángel Pérez Cutiño in particular for his help with programming implementation.

Finally, I must express my very profound gratitude to my family and to my boyfriend Alberto for providing me with unfailing support and continuous encouragement throughout this year in Spain, studying and through the process of researching and writing this thesis.

Contents

List of Figures

List of Algorithms

Introduction

Unmanned aerial vehicles (UAVs or drones) have been largely used in missions involving navigating through an unknown environment, as they can host a wide range of sensors to measure the environment with relative low operation costs and high flexibility. However, it is difficult to attain this in most realistic implementations, since the knowledge and data regarding the environment are normally limited or unavailable. Using Reinforcement Learning (RL) is a good approach to overcome this issue because it allows a UAV or a UAV team to learn and navigate through the changing environment without a model of the environment [24]. Deep reinforcement learning (DRL) is the combination of reinforcement learning and deep learning. This field of research has been able to solve a wide range of complex decision making tasks in robotics [14]. The main idea is that an agent may learn by interacting with its environment, similarly to a biological agent. Using the experience gathered, the agent should be able to optimize some objectives given in the form of cumulative rewards.

In this paper we use DRL to compute a tour that enables a mobile agent to cover points of an area of interest. Then the optimal tour is iteratively used to cover the overall required zone. Thus, our problem is relating with well known NP-hard problems: Travelling Salesman Problem, the Lawn Mowing Problem and the Milling problem [3]. In the robotics area, [11] provides a survey of general (ground robotics) approaches to coverage path planning. A recent surveys for UAVs is [8].

The main constraint in UAV path planning is the battery endurance. Despite recent improvements in battery technology, the maximum flying range of small UAVs is still a severe constraint. From a computational point of view, the problem is quite different when a limited lifetime is considered. Indeed, some planning problems can be solved in polynomial time when the battery endurance is considered unlimited but could be NP-hard when the power is a constraint, see [2] for an example. Then, the power constraint leads to challenging optimization problems when a drone or a team of drones are tasked to navigate in a specific scenario.

In the following we mention some related work. Collecting data from sensor devices in an outdoor environment imposes challenging constraints on the trajectories design for autonomous UAVs. Battery energy restricts mission duration for the drones severely, while the complex urban environment poses challenges in obstacle avoidance. A recent tutorial covering the paradigms of trajectory planning for data collection with drones is given in [26]. Bithaset al. [6] provide a survey on machine learning techniques, including but not limited to reinforcement learning(RL), for various UAV communications scenarios. Most existing approaches to UAV data collection are not based on RL and only find a solution for one set of scenario parameters at a time. For example, Esrafilianet al. [9] proposed a two-step algorithm to optimize a UAV's trajectory and its scheduling decisions in an urban data collection mission using a combination of dynamic and sequential convex programming. However, deep reinforcement learning has been explored in UAV

communication scenarios. In [5], a UAV base station serves two ground users and the goal is to show the advantages of neural networks over table-based Q-learning. However, any explicit assumptions about the environment is considered at the price of long training time. On the other hand, deep deterministic policy gradient was proposed by Qiet al. [22] to learn a continuous control policy for a UAV providing persistent communications coverage to a group of users in an environment without obstacles. A more related work is given by Liu et al. [16], where an RL multi-agent algorithm collecting data simultaneously with ground and aerial vehicles in an environment with obstacles and charging stations is proposed. Their approach also makes use of convolutional nertworks to exploit a map of the environment but, in contrast to our method, control policies have to be relearned entirely when scenario and environmental parameters change. Finally, we mention a recent paper that proposes the use of DRL to find an optimal path under the assumption that not all the areas have the same coverage requirements. Picarelli et al. [21] propose a reinforcement learning approach that, given a relevance map representing coverage requirements, a robot autonomously chooses the best actions to optimize the coverage. However, they do not consider the battery endurance constraint. To the best of our knowledge, DRL has not been considered for UAV for coverage path planning in a unknown environment under power constraints before.

The main goal of this thesis is serve a first step and a basis for path planners that are based on DRL, especially those under energetic constraints. This is particularly interesting for some type of applications with UAVs. The idea is to achieve sensor-based navigation in unknown scenarios with the help of the Reinforcement Learning framework in order to succeed in covering the ground with a good coverage rate which is evaluated with appropriately defined metrics. The simple problem formulation makes it possible to quickly generalize the solution concept to various domains with little changes or adjustments in the structure. This is made possible thanks to the mathematical tools used in the scope of this thesis.

The rest of the work is outlined as follows. In chapter 1, we gives an overview of the Reinforcement Learning and the underlying concepts, in Markov Decision Processes in particular. The learning problem is introduced as well as the mathematical framework and tools used to implement the proposed Deep Reinforcement Learning algorithm. Chapter 2 defines the type of neural networks used in this study as well the deep neural networks. Section 3 presents the method to solve the problem. Finally, in chapter 5, we summarize the work and draw directions for future work.

1 Reinforcement learning

Reinforcement learning is an area of machine learning, that becomes popular recently thanks to its capabilities in solving learning problem without relying on a model of the environment. It is a computational approach to understanding and automating goal-directed learning and decision making. The RL problem can be formalized as an *agent* that has to choose an *action* in an *environment* to maximize a numerical signal, called *reward*, that measures the performance of the agent (see Figure 1.1). A learning agent must be able to interact with environment and to take actions that affect the state. The agent does not know which action to take. It deals with the *exploration/exploitation* dilemma while learning. The agent must try a variety of actions and progressively favor those that appear to be best. Doing this the agent is seeking for a goal or goals relating to the state of the environment.

In this chapter we will introduce the basic concepts of the reinforcement learning algorithm and describe the mathematics behind it. We will base base this study on Sutton and Barto's RL book [24]. It covers RL fundamentals and reflects new progress, e.g., in deep Q-network, AlphaGo, policy gradient methods, as well as in psychology and neuroscience. Xintian Han gives a brief summary of the the mathematical approach of the problem [12]. [20] focus his paper "Autonomous uav navigation using reinforcement learning" on applying RL algorithm for drones. Many other papers also use RL concepts into different field of robotics ([21, 18, 19] and others).

1.1 Problem Setup

The general RL problem is formalized as a discrete time stochastic control process where an agent interacts with its environment in the following way: the agent starts, in a given state within its environment $s_0 \in S$. At each time step t, the agent receives a state $s_t \in S$ and selects an action $a_t \in \mathcal{A}$, following the policy $\pi(a_t|s_t)$, which is the agent's behavior.

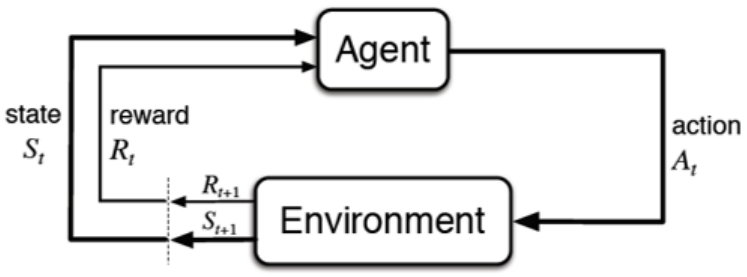


Figure 1.1: The agent–environment interaction in a Markov decision process[24].

The agent obtains a scalar reward r_t, and transitions to the next state s_{t+1}, according to the environment dynamics, or model, for reward function $\mathcal{A}(s, a)$ and state transition probability $\mathcal{P}(s_{t+1}|a_{t+1})$ respectively. In an episodic problem, this process continues until the agent reaches a terminal state and then it restarts. The return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the discounted, accumulated reward with the discount factor $\gamma \in (0, 1]$. The agent aims to maximize the expectation of such long term return from each state. The problem is set up in discrete state and action spaces. It is not hard to extend it to continuous spaces.

1.2 Markov Decision Processes

Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards.

First we will define the Markov process. A sequence of states is Markov if and only if the probability of moving to the next state S_{t+1} depends only on the present state s_t and not on the previous states $S_1, S_2, ..., S_{t-1}$. That is, for all t,

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, S_2, ..., S_t] \tag{1.1}$$

We always talk about time-homogeneous Markov chain in RL, in which the probability of the transition is independent of t:

$$\mathbb{P}[S_{t+1} = s'|S_t = s] = \mathbb{P}[S_t = s'|S_{t-1} = s] \tag{1.2}$$

Definition 1.1 (Markov Process). *A Markov Process (or Markov Chain) is a tuple $(\mathcal{S}, \mathcal{P})$, where*

- *$\mathcal{S}$ is a (finite) set of states.*
- *$\mathcal{P}$ state transition probability matrix. $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$.*

If we introduce reward, action and discount into a Markov process, we get a Markov decision process.

Definition 1.2 (Markov Decision Processes). *A Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma, \mathcal{R})$, where*

- *$\mathcal{S}$ is a finite set of states.*
- *$\mathcal{A}$ is a finite set of actions.*
- *$\mathcal{P}$ state transition probability matrix. $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$.*
- *$\gamma \in (0, 1]$ is a discount factor.*
- *$\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function.*

In the MDP, the transition to the next state $S_t + 1$ depends not only on the current state S_t, but also depends on the action A_t you make at the current state. Also, each state-action pair is attached with a reward function. MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made.

1.2.1 Returns and Episodes

We have said that the agent's goal is to maximize the cumulative reward it receives in the long run. If the sequence of rewards received after time step t is denoted R_{t+1}, R_{t+2}, R_{t+3}, ..., than we seek to maximize the expected return, where where the return, denoted G_t, is defined as some specific function of the reward sequence. For reinforcement learning tasks, which break naturally into sub-sequences, called episodes (*episodic task*), the return function is the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{1.3}$$

where T is a final time step. On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit.We call these *continuing tasks*. The final time step would be $T = \infty$, and the return could itself easily be infinite. To avoid it we add an additional parameter $\gamma \in [0, 1]$, called the *discount rate*. We can define the expected *discounted return* as:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.4}$$

Returns at successive time steps are related to each other:

$$G_t \doteq R_{t+1} + \gamma G_{t+1} \tag{1.5}$$

We can define the return, in general for both episodic and continuing tasks:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1.6}$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both).

1.2.2 Policies and Value Functions. Bellman equation

Formally, a *policy* is a mapping from states to probabilities of selecting each possible action:

$$\pi : \mathcal{S} \to p(\mathcal{A} = a|S) \tag{1.7}$$

If the MDP is episodic, i.e., the state is reset after each episode of length T, then the sequence of states, actions, and rewards in an episode constitutes a trajectory of the policy.

The *state-value function for policy* π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define $v_\pi(s)$ formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right], \text{ for all } s \in \mathcal{S} \tag{1.8}$$

where $\mathbb{E}$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step.

The *action-value function for policy* π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right]. \tag{1.9}$$

We can decompose the state-value function into two parts: the immediate reward R_{t+1} and discounted value of successor state $\gamma v_\pi(S_{t+1})$:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ &= \underbrace{\mathbb{E}_\pi[R_{t+1}|S_t = s]}_{\text{immediate reward}} + \underbrace{\mathbb{E}_\pi[\gamma v_\pi(S_{t+1})|S_t = s]}_{\text{discounted value of successor state}} \end{aligned} \tag{1.10}$$

Similarly, the action-value function can be decomposed as follows:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \tag{1.11}$$

To simplify notations, we define $\mathcal{R}_s^a = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$. We can see the relationship between $v_\pi(s)$ and $q_\pi(s,a)$:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a) \tag{1.12}$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \tag{1.13}$$

Expressing $q_\pi(s,a)$ in terms of $v_\pi(s)$ in the expression of $v_\pi(s)$, we get a *Bellman equation* for v_π,

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right) \tag{1.14}$$

The Bellman equation relates the state-value function of one state with that of other states. Similarly, we also have a Bellman equation for $q_\pi(s,a)$,

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s') q_\pi(s',a') \tag{1.15}$$

One use for the Bellman equation is to compute the value function for a given policy. If we combine all the Bellman equations in an MDP with n states, we get n linear equations for the n unknown value functions. We can get the value functions by solving linear equations ((recursion)). However, this step may take $O(n^3)$ time complexity. Other solution is to use the dynamic programming.

1.2.3 Optimal Policy and Optimal Value Functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. A policy π defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. The *optimal policy* is a policy that better than or equal to all other policies. We denote all the optimal policies by π_*. They share the same state-value function, called the optimal *state-value function*, denoted v_*, and defined as

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{1.16}$$

for all $s \in \mathcal{S}$.

The optimal value function specifies the best possible performance in the MDP. We say an MDP is "solved" when we know the optimal value function.

Similarly, we can define the action-value function as the maximum action-value function over all policies:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \tag{1.17}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

Theorem 1.1. *For any Markov Decision Process,*

- *There exists an optimal policy $\pi_* \geq \pi$, for all π*
- *All optimal policies achieve the optimal value function, $v_{\pi*}(s) = v_*(s)$.*
- *All optimal policies achieve the optimal action-value function, $q_{\pi*}(s,a) = q_*(s,a)$.*

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (1.14). We can find the relationship between the optimal state-value function and the optimal action-value function,

$$v_*(s) = \max_a q_*(s,a), \tag{1.18}$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \tag{1.19}$$

Expressing $q_*(s,a)$ in terms of $v_*(s)$ in the expression of $v_*(s)$, we get a Bellman optimality equation for v_*:

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right) \tag{1.20}$$

We also have a Bellman equation for q_*:

$$q_*(s) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a') \tag{1.21}$$

The Bellman optimality equations are non-linear and there is no closed form solution in general.

1.3 Dynamic Programming

The term dynamic programming (DP) refers to paradigm that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). We usually assume that the environment is a finite MDP: its state, action, and reward sets, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ are finite, and that its dynamics are given by a set of probabilities $p(s', r|s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$ $\mathcal{S}^+$ is $\mathcal{S}$ plus a terminal state if the problem is episodic). The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. We can write the Bellman optimality equations (1.20), (1.21) as:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_*(s') \right], \end{aligned} \tag{1.22}$$

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(S_{t+1}, a')\right],
\end{aligned} \tag{1.23}
$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$. We will introduce three paradigms of dynamic programming in reinforcement learning: policy evaluation, policy iteration and value iteration. Policy evaluation is used to find the value function of a certain policy. Policy iteration and value iteration are used to find the optimal value function and optimal policy.

1.3.1 Policy Evaluation

We consider how to compute the state-value function v_π for an arbitrary policy π. We start from an initial guess v_1 and then apply Bellman equation (1.14) iteratively to it: $v_1 \to v_2 \to v_3 \to \ldots \to v_\pi$. At each iteration $k+1$, for all states $s \in \mathcal{S}$, we update $v_{k+1}(s)$ from $v_k(s')$ according to Bellman equations, where s' is a successor state of s:

$$
\begin{aligned}
v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \\
&= \sum_{a\in\mathcal{A}} \pi(a|s)\left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_k(s')\right) \\
&= \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_k(s')\right],
\end{aligned} \tag{1.24}
$$

for all $s \in \mathcal{S}$. $\pi(a|s)$ is the probability of taking action a in state s under policy π. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called *iterative policy evaluation*. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. The pseudocode tests the quantity:

Algorithm 1: Iterative Policy Evaluation, for estimating $V \approx v_\pi$ [24]

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(S)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$.
$\Delta \leftarrow 0$
repeat
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

The iterative process stops when the maximum difference between value function at the current step and that at the previous step is smaller than some small positive constant.

1.3.2 Policy Iteration

Suppose we have determined the value function v_π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy, but to know is there a new policy that is better than the current one, we need to use policy iteration. If select a random action a in a state s and follow the existing policy the value:

$$q_\pi(s,a) \doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}|S_t = s, A_t = a] = \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right], \tag{1.25}$$

Theorem 1.2 (Policy Improvement Theorem). *Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,*

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{1.26}$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s) \tag{1.27}$$

Moreover, if there is strict inequality of (1.26) at any state, then there must be strict inequality of (1.27) at that state.

Proof. Starting from (1.26) we keep expanding the q_π side with (1.25) and reapplying (1.26) until we get $v_{\pi'}(s)$:

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = \pi'(s)] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}, \pi'(S_{t+1}))|S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma[R_{t+2} + \gamma v_\pi(S_{t+2})|S_{t+1}, A_{t+1} = \pi'(S_{t+1})])|S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2})|S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3})|S_t = s] \\
&\vdots \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3})|S_t = s]
\end{aligned}$$

Let us consider the new greedy policy, π', given by

$$\begin{aligned}
\pi'(s) &\doteq \operatorname*{argmax}_a q_\pi(s,a) \\
&= \operatorname*{argmax}_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \\
&= \operatorname*{argmax}_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right],
\end{aligned}$$

By construction, the greedy policy meets the conditions of the policy improvement theorem 1.2, so we know that it is as good as, or better than, the original policy. The process

of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*. Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

where $\xrightarrow{E}$ denotes a policy *evaluation* and $\xrightarrow{I}$ denotes a policy *improvement*. This way of finding an optimal policy is called policy *iteration*.

Algorithm 2: Policy Iteration for estimating $\pi \approx \pi_*$ [24]

1. Initialization
$V(S) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
$\Delta \leftarrow 0$
repeat
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V(s')\right]$,
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

3. Policy Improvement
policy-stable $\leftarrow$ *true*
for $s \in \mathcal{S}$ **do**
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
  if *old-action*$\neq \pi(s)$ **then**
    policy-stable $\leftarrow$ *false*
if *policy-stable* **then**
  return $V \approx v_*$ *and* $\pi \approx \pi_*$
else
  goto *(2)*

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy-evaluation and policyimprovement processes interact, independent of the granularity and other details of the two processes.

1.3.3 Value Iteration

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update

operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_t + 1) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \end{aligned}$$

for all $s \in \mathcal{S}$. For arbitrary v_0, the sequence v_k can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

A complete algorithm with this kind of termination condition is shown below.

Algorithm 3: Value Iteration, for estimating $\pi \approx \pi_*$ [24]

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$ $\Delta \leftarrow 0$

repeat
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$,
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep.

1.4 Different settings to learn a policy from data

Two classes of learning methods can be distinguished in Reinforcement Learning: model-based and model-free methods.

Model-based methods assume a representation of the environment where the transition function between states and the reward function are known and the agent plans accordingly. In these cases, the environment can be represented by an MDP and when the problem is well defined and its corresponding MDP is completely known, solving it is straightforward and can be done using Bellman equations leading to the optimal policy that maximizes the amount of reward the agent can expect to get.

In model-free methods, the agent learns with trail-and-error from experience explicitly. The model (state transition function) is not known or learned from experience. In this case, the MDP is not given beforehand and the goal is to solve it. More specifically, the environment is assumed to be partially observable. As a consequence, the equations governing the way the environment operates (especially when it is dynamic) are not fully known across the whole space.

In this thesis, the algorithm to be used is a model-free one which does not make use of the distribution of probabilities of transitions $\mathcal{P}$ between states associated with the MDP. The core idea of the algorithm is based on the trial-and-error paradigm.

Temporal difference learning methods are a fundamental component of RL algorithms that allows learning to occur directly from raw experience. TD learning often consists of on-policy and off-policy methods.

SARSA (so called because it uses state-action-reward-state-action experiences to update the Q-values) is an on-policy TD learning algorithm that learns the agent policy directly from raw experience.

In contrast, Q-learning is a popular off-policy algorithm where a target policy of the agent is learnt, but a different behaviour policy is used to generate the behaviour of the agent. An exploratory behaviour policy is often used to explore the full state and action space.

Both on-policy and off-policy methods in RL have their own advantages and limitations. On-policy methods such as Monte-Carlo policy gradient methods (also known as REINFORCE) often suffer from high variance estimates and requires large number of on-policy samples. Off-policy methods such as Q-learning and actor-critic methods, in contrast are sample efficient as they can use all samples including off-policy samples to explore the state space. The notion of on-policy and off-policy can be understood as same-policy and different-policy.

In this we will introduce this method and discuss its usage for the problem posed in this thesis.

1.4.1 Monte Carlo Methods

Any method which solves a problem by generating suitable random numbers, and observing that fraction of numbers obeying some property or properties, can be classified as a Monte Carlo (MC) method. The Monte Carlo method for reinforcement learning learns directly from episodes of experience without any prior knowledge of MDP transitions. Here, the random component is the return or reward. Similar to dynamic programming, there is a policy evaluation (finding the value function for a given random policy) and policy improvement step (finding the optimum policy).

Monte Carlo Policy Evaluation

The goal here, again, is to learn the value function $v_\pi(s)$ from episodes of experience under a policy pi. We know that we can estimate any expected value simply by adding up samples and dividing by the total number of samples:

$$\bar{V}_\pi(s) = \frac{1}{N} \sum_{i=1}^{N} G_{i,s} \tag{1.28}$$

There are two similar Monte Carlo (MC) methods that have different theoretical properties: *first-visit method* and *every-visit method*. The first-visit MC method estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the every-visit MC method averages the returns following all visits to s. First-visit MC is shown in procedural form in the box. Every-visit MC would be the same except without the check for S_t having occurred earlier in the episode.

Algorithm 4: First-visit MC prediction, for estimating $V \approx V_\pi$ [24]

Input: a policy π to be evaluated
Initialize $V(s)$, arbitrarily, for all $s \in \mathcal{S}$
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

repeat
  for *each episode* **do**
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
      $G \leftarrow \gamma G + R_{t+1}$
      while S_t *appears in* $S_0, S_1, ..., S_{t-1}$ **do**
        Append G to $Returns(S_t)$
        $V(St) \leftarrow$ average$(Returns(St))$
until *satisfied*

Monte Carlo Control

Similar to dynamic programming, once we have the value function for a random policy, the important task that still remains is that of finding the optimal policy using Monte Carlo. In the classical policy improvement method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

Since we do not know the state transition probabilities $p(s', r|s, a)$, we can't do a look-ahead search like DP. Hence, all the information is obtained via experience of playing the game or exploring the environment.

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:

$$\pi(s) \doteq \underset{a}{\text{argmax}}\, q(s, a) \tag{1.29}$$

A greedy policy (like the above mentioned one) will always favor a certain action if most actions are not explored properly. For Monte Carlo policy iteration it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. In *Monte Carlo ES* (Monte Carlo with Exploring Starts) Algorithm all the state action pairs have non-zero probability of being the starting pair. This will ensure each episode which is played will take the agent to new states and hence, there is more exploration of the environment.

Algorithm 5: Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$ [24]

Initialize $\pi(s) \in \mathcal{A}(s)$(arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}(s)$(arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}$
$Returns(s,a) \leftarrow$ an empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}$

repeat
  for *each episode* **do**
    Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}$ randomly such that all pairs have probability > 0
    Gener. an episode from S_0, A_0, following π: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$ **for** *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
      $G \leftarrow \gamma G + R_{t+1}$
      while S_t *appears in* $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$ **do**
        Append G to $Returns(S_t, A_t)$
        $Q(St, A_t) \leftarrow$ average $(Returns(S_t, A_t))$
        $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$
until *satisfied*

This algorithm does not work when there is a single start point for an environment. The simplest idea for ensuring continual exploration – epsilon choose the action which maximises the action value function and with probability epsilon choose an action at random. For any ε-soft policy, π, any ε-greedy policy with respect to q_π is guaranteed to be better than or equal to π. The complete algorithm is given in the box below.

Algorithm 6: Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$ [24]

Algorithm parameter: small $\varepsilon > 0$ **Initialize** $\pi(s)$ an arbitrarily ε - soft policy
$Q(s,a) \in \mathbb{R}(s)$(arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}$
$Returns(s,a) \leftarrow$ an empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}$

repeat
  for *each episode* **do**
    Generate an episode following π: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$ **for** *each step of episode,* $t = T-1, T-2, ..., 0$ **do**
      $G \leftarrow \gamma G + R_{t+1}$
      while S_t *appears in* $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$ **do**
        Append G to $Returns(S_t, A_t)$
        $Q(S_t, A_t) \leftarrow$ average $(Returns(S_t, A_t))$
        $A^* \leftarrow \text{argmax}_a\, Q(S_t, a)$
        for *all* $a \in \mathcal{A}(S_t)$ **do**
          :
        $$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t) & \text{if } a \neq A^* \end{cases}$$
until *satisfied*

1.4.2 Temporal-Difference Learning

TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. 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 other learned estimates, without waiting for a final outcome (they bootstrap).

Both TD and Monte Carlo methods use experience to solve the prediction problem. A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right], \tag{1.30}$$

where G_t is the actual return following time t, and α is a constant step-size parameter. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods immediately form a target at the next time step $t+1$ 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

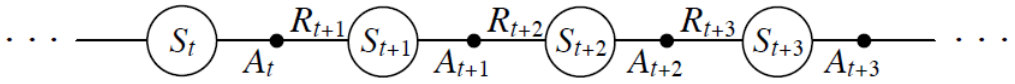
$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right], \tag{1.31}$$

immediately on transition to S_{t+1} and receiving R_{t+1}. This TD method is called TD(0), or one-step TD, because it is a special case of the TD(λ) and n-step TD methods. The difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$ calls the TD error and arises in various forms throughout reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{1.32}$$

Sarsa: On-policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. Recall that an episode consists of an alternating sequence of states and state–action pairs:



The convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right], \tag{1.33}$$

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_t, A_t)$ is defined as zero. This rule uses every element of the quintuple of events, $S_t, A_t, R_t, S_{t+1}, A_{t+1}$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name *SARSA* for the algorithm. It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π.

Algorithm 7: Sarsa (on-policy TD control) for estimating $Q \approx q_*$ [24]

Algorithm parameters:
step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(S, A)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}$,
arbitrarily except that $Q(terminal, \bullet) = 0$

for *each episode* **do**
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  repeat
    for *each step of episode* **do**
      Take action A, observe R, S'
      Choose A' from S' using policy derived from Q (e.g., ε-greedy)
      $Q(s, a) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$,
      $S \leftarrow S'$; $A \leftarrow A'$
  until S *is terminal*

Q-learning: Off-policy TD Control

Off-policy TD control algorithm known as Q-learning (Watkins, 1989), defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right], \tag{1.34}$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. 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 pairs continue to be updated. The Q-learning algorithm is shown below in procedural form.

Algorithm 8: Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$ [24]

Algorithm parameters:
step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(S, A)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}$,
arbitrarily except that $Q(terminal, \bullet) = 0$

for *each episode* **do**
  Initialize S
  repeat
    for *each step of episode* **do**
      Choose A from S using policy derived from Q (e.g., ε-greedy)
      Take action A, observe R, S'
      $Q(s, a) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', a) - Q(S, A)\right]$,
      $S \leftarrow S'$;
  until S *is terminal*

Fitted Q-learning

François-Lavet [10] in the book "An introduction to deep reinforcement learning" introduces us in using deep neural network for the Q-function approximation. The idea is based on the fitted Q-learning concept.

In fitted Q-learning, the algorithm starts with some random initialization of the Q-values $Q(s, a; \theta)$ where θ refers to the initial parameters (usually such that the initial Q-values should be relatively close to 0 so as to avoid slow learning). Then, an approximation of the Q-values at the k -th iteration $Q(s, a; \theta_k)$ is updated towards the target value

$$Y_k^Q = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k) \tag{1.35}$$

where $\theta_k)$ refers to some parameters that define the Q-values at the k -th iteration. In neural fitted Q-learning (NFQ), the state can be provided as an input to the Q-network and a different output is given for each of the possible actions. This provides an efficient structure that has the advantage of obtaining the computation of $\max_{a' \in \mathcal{A}} Q(s', a'; \theta_k$ in a single forward pass in the neural network for a given s_0. The Q -values are parameterized with a neural network $Q(s, a; \theta_k)$ where the parameters $\theta_k)$ are updated by stochastic gradient descent (or a variant) by minimizing the square loss:

$$L_{DQN} = \left(Q(s', a'; \theta_k) - Y_k^Q\right) \tag{1.36}$$

Thus, the Q-learning update amounts in updating the parameters:

$$\theta_{k+1} = \theta_k + \alpha(Y_k^Q - Q(s', a'; \theta_k))\nabla_{\theta_k} Q(s, a; \theta_k) \tag{1.37}$$

where α is a scalar step size called the learning rate. Using the square loss is not arbitrary. Indeed, it ensures that $Q(s, a; \theta_k)$ should tend without bias to the expected value of the random variable Y_k^Q. Hence, it ensures that $Q(s, a; \theta_k)$ should tend to $Q^*(s, a)$ after many iterations in the hypothesis that the neural network is well-suited for the task and that the experience gathered in the dataset D is sufficient (we describe the deep Q-networks with more details in the section 2.3).

2 | Deep neural networks

The area of Artificial Neural Networks has become a matter of engineering and achieving good results in Machine Learning tasks. Among them is the idea of Q-function approximation with Deep Q-Network in Reinforcement Learning. In this chapter we will give a brief introduction to the basic concepts of Neural Networks and Deep Neural Networks in it particular case basing the investigation on the Charu C. Aggarwal [1] and Sandro Skansi [23] books. We also touch the basics of Convolutional Neural Networks (CNN) explained on Stanford university course [13]. In the end of this chapter we go deeper to the convolutional networks application in Reinforcement learning and study the Deep Q-learning concepts. Vincent François-Lavet [12], Yuxi Li [15], Kai Arulkumaran [4] provide an introduction to deep reinforcement learning models, algorithms and techniques.

2.1 The Basic Architecture of Neural Networks

Artificial Neural Networks (ANNs) were inspired with the idea of modeling biological neural systems and proposed creating virtual neurons. As biological systems, the neurons are connected together to form a network of nodes. The type of information circulating in neural nets is of scalar type, analogously to electrical signals emitted in a brain of a living being. Any neural network is made of simple basic elements: neurons or perceptrons. In this section we see the structure of this elements. Also we will to combine them to construct a deep or multi layer networks.

2.1.1 Single Computational Layer: The Perceptron

The simplest neural network is referred to as the perceptron. This neural network contains a single input layer and an output node. The basic architecture of the perceptron is shown in Figure 2.1

Consider a situation where each training instance is of the form $(\mathbf{X}, y)$, where each $\mathbf{X} = \{x_1, ..., x_d\}$ contains d feature variables, and $y \in (-1, +1)$ contains the observed value of the binary class variable. The input layer contains d nodes that transmit the d features $\mathbf{X} = \{x_1, ..., x_d\}$ with edges of weight $\mathbf{W} = \{w_1, ..., w_d\}$ to an output node. The input layer does not perform any computation in its own right. The linear function $\mathbf{X} \bullet \mathbf{W} = \sum_{j=1}^{d} w_j x_j$ computed at the output node. Therefore, the prediction $\hat{y}$ is computed as follows:

$$\hat{y} = \text{sign}\{\mathbf{X} \bullet \mathbf{W}\} = \text{sign}\left\{\sum_{j=1}^{d} w_j x_j\right\} \tag{2.1}$$

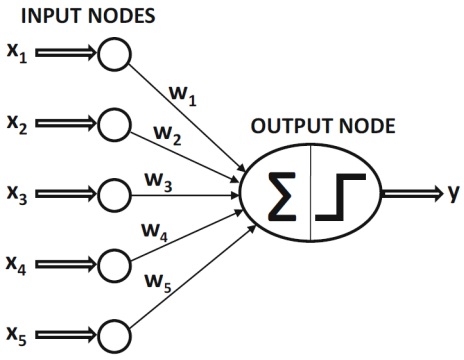


Figure 2.1: The basic architecture of the perceptron [1].

2.1.2 Choice of Activation Function

The choice of activation function is a critical part of neural network design. In the case of the perceptron, the choice of the sign activation function is motivated by the fact that a binary class label needs to be predicted. Every activation function (or non-linearity) takes a single number and performs a certain fixed mathematical operation on it. We use the notation Φ to denote the activation function:

$$\hat{y} = \Phi\{\mathbf{X} \bullet \mathbf{W}\} \tag{2.2}$$

The classical activation functions that were used early in the development of neural networks were the sign, sigmoid, and the hyperbolic tangent functions:

$$\Phi(v) = \text{sign}(v) \quad \text{(sign function)}$$
$$\Phi(v) = \frac{1}{1+e^{-v}} \quad \text{(sigmoid function)}$$
$$\Phi(v) = \frac{e^{2v}-1}{e^{2v}+1} \quad \text{(tanh function)}$$

The choice of the loss function is critical in defining the outputs in a way that is sensitive to the application at hand.

2.1.3 Loss function

The data loss in a supervised learning problem measures the compatibility between a prediction (e.g. the class scores in classification) and the ground truth label. The data loss takes the form of an average over the data losses for every individual example. That is, $L = 1/N \sum_i L_i$ where N is the number of training data.

In case of classification problem we assume a dataset of examples and a single correct label (out of a fixed set) for each example. One of two most commonly seen cost functions in this setting is the support vector machine (SVM):

$$L_i = \max\{0, 1 - y_i(\mathbf{X} \bullet \mathbf{W})\} \tag{2.3}$$

For example, least-squares regression with numeric outputs requires a simple squared loss of the form $(y - haty)^2$ for a single training instance with target y and prediction

$\hat{y}$. One can also use other types of loss like hinge loss for $y \in \{-1, +1\}$ and real-valued prediction $\hat{y}$ (with identity activation):

$$L = \max\{0, 1 - y \bullet \hat{y}\} \tag{2.4}$$

The second common choice is the Softmax classifier that uses the cross-entropy loss:
For multiway predictions (like predicting word identifiers or one of multiple classes), the softmax output is particularly useful. However, a softmax output is probabilistic, and therefore it requires a different type of loss function. In fact, for probabilistic predictions, two different types of loss functions are used, depending on whether the prediction is binary or whether it is multiway:

- *Binary targets* (logistic regression): In this case, it is assumed that the observed value y is drawn from $\{-1, +1\}$, and the prediction $\hat{y}$ is a an arbitrary numerical value on using the identity activation function. In such a case, the loss function for a single instance with observed value y and real-valued prediction $\hat{y}$ (with identity activation) is defined as follows:

  $$L = \log(1 + \exp(-y \bullet \hat{y})) \tag{2.5}$$

  This type of loss function implements a fundamental machine learning method, referred to as logistic regression. Alternatively, one can use a sigmoid activation function to output $\hat{y} \in (0, 1)$, which indicates the probability that the observed value y is 1. Then, the negative logarithm of $|y/2 - 0.5 + \hat{y}|$ provides the loss, assuming that y is coded from $\{-1, 1\}$. This is because $|y/2 - 0.5 + \hat{y}|$ indicates the probability that the prediction is correct. This observation illustrates that one can use various combinations of activation and loss functions to achieve the same result.
- *Categorical targets*: In this case, if $\hat{y}_1 \ldots \hat{y}_k$ are the probabilities of the k classes, and the r-th class is the ground-truth class, then the loss function for a single instance is defined as follows:

  $$L = -\log(\hat{y}_r) \tag{2.6}$$

  This type of loss function implements multinomial logistic regression, and it is referred to as the cross-entropy loss. Note that binary logistic regression is identical to multinomial logistic regression, when the value of k is set to 2 in the latter.

In practice, one rarely uses the perceptron criterion as the loss function. For discrete-valued outputs, it is common to use softmax activation with crossentropy loss. For real-valued outputs, it is common to use linear activation with squared loss. Generally, cross-entropy loss is easier to optimize than squared loss.

2.1.4 Multilayer Neural Networks

Multilayer neural networks contain multiple computational layers; the additional intermediate layers (between input and output) are referred to as hidden layers because the computations performed are not visible to the user. The specific architecture of multilayer neural networks is referred to as feed-forward networks, because successive layers feed into one another in the forward direction from input to output. An example of multilayer network is shown in Figure 2.2

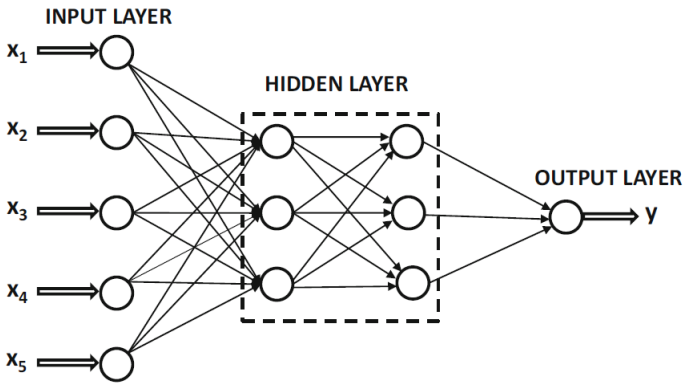


Figure 2.2: The basic architecture of a feed-forward network with two hidden layers and a single output layer [1].

The weights of the connections between the input layer and the first hidden layer are contained in a matrix $\mathbf{W}_1$ with size $d \times p_1$ whereas the weights between the r-th hidden layer and the $(r+1$-th hidden layer are denoted by the $p_r \times p_{r+1}$ matrix denoted by $\mathbf{W}_r$. If the output layer contains o nodes, then the final matrix $\mathbf{W}_{k+1}$ is of size $p_k \times o$ The d-dimensional input vector $\bar{x}$ is transformed into the outputs using the following recursive equations:

$$\bar{h}_1 = \Phi(\mathbf{W}_1^T \bar{x}) \qquad \text{(Input to Hidden Layer)}$$
$$\bar{h}_{p+1} = \Phi(\mathbf{W}_{p+1}^T \bar{h}_p) \; \forall p \in \{1 \ldots k-1\} \qquad \text{(Hidden to Hidden Layer)}$$
$$\bar{o} = \Phi(\mathbf{W}_{k+1}^T \bar{h}_k) \qquad \text{(Hidden to Output Layer])}$$

As we increase the size and number of layers in a Neural Network, the capacity of the network increases. Overfitting occurs when a model with high capacity fits the noise in the data instead of the (assumed) underlying relationship. It seems that smaller neural networks can be preferred if the data is not complex enough to prevent overfitting. However, this is incorrect – we should use as big of a neural network as your computational budget allows, and use other regularization techniques to control overfitting.

2.1.5 Training a Neural Network with Backpropagation

In the single-layer neural network, the training process is relatively straightforward because the error (or loss function) can be computed as a direct function of the weights, which allows easy gradient computation. In the case of multi-layer networks, the problem is that the loss is a complicated composition function of the weights in earlier layers. The gradient of a composition function is computed using the backpropagation algorithm. The backpropagation algorithm is a direct application of dynamic programming. It contains two main phases, referred to as the forward and backward phases, respectively. The forward phase is required to compute the output values and the local derivatives at various nodes, and the backward phase is required to accumulate the products of these local values over all paths from the node to the output.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs or ConvNets) are very similar to the neural networks that we have seen in the previous chapter: they are made up of neurons that have parameters in the form of weights and biases that can be learned. But a differential feature of CNNs is that they make the explicit assumption that the inputs are images, which allows us to encode certain properties in the architecture to recognize specific elements in the images.

An important property of image data is that it exhibits a certain level of translation invariance, which is not the case in many other types of grid-structured data. For example, a banana has the same interpretation, whether it is at the top or the bottom of an image. Convolutional neural networks tend to create similar feature values from local regions with similar patterns.

2.2.1 The basic elements of CNNs

A convolutional neural network is a neural network that has one or more convolutional layers. To understand let us see the 1D-convolutional layer structure. In this case the convolutional layer takes a 2D array and a small logistic regression with e.g. input size 4 (these sizes are usually 4 or 9, sometimes 16) and passes the logistic regression over the whole image. This means that the first input consists of components 1–9 of the flattened vector, the second input are the components 2–10, the third are components 3–11, and so on.You can see an overview of the process in the bottom of Figure 2.3

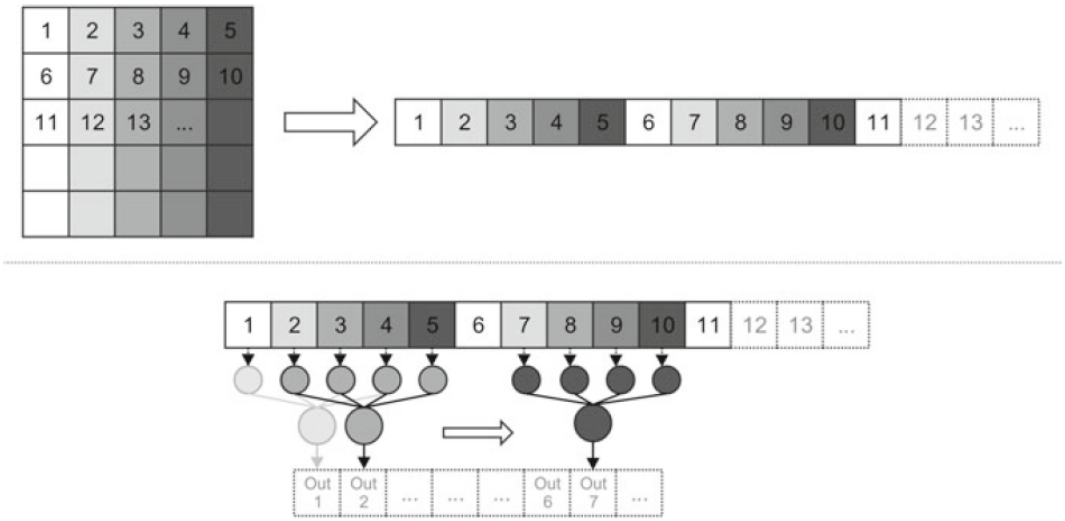


Figure 2.3: Building a 1D convolutional layer with a logistic regression [23].

In classical convolutional neural networks we deal with pictures. In this case the states in each layer are arranged according to a spatial grid structure. Each layer is a 3-dimensional grid structure, which has a height, width, and depth (Figure 2.4).

The convolutional neural network functions much like a traditional feed-forward neural network, except that the operations in its layers are spatially organized with sparse (and carefully designed) connections between layers. The three types of layers that are commonly present in a convolutional neural network are convolution, pooling, and ReLU. The ReLU activation is no different from a traditional neural network.

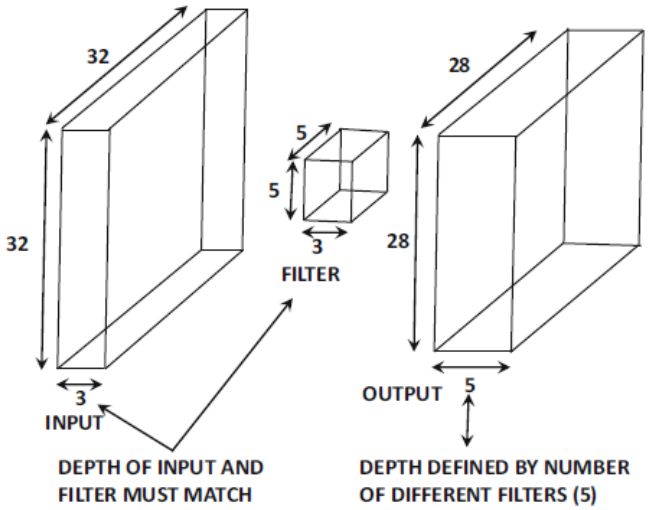


Figure 2.4: The convolution between an input layer of size 32 × 32 × 3 and a filter of size 5 × 5 × 3 produces an output layer with spatial dimensions 28 × 28. The depth of the resulting output depends on the number of distinct filters and not on the dimensions of the input layer or filter [1].

In the convolutional neural network, the parameters are organized into sets of 3-dimensional structural units, known as filters or kernels. The filter is usually square in terms of its spatial dimensions, which are typically much smaller than those of the layer the filter is applied to. On the other hand, the depth of a filter is always same is the same as that of the layer to which it is applied.

Assume that the dimensions of the filter in the q -th layer are $F_q \times F_q \times d_q$. The convolution operation places the filter at each possible position in the image (or hidden layer) so that the filter fully overlaps with the image, and performs a dot product between the $F_q \times F_q \times d_q$ parameters in the filter and the matching grid in the input volume (with same size $F_q \times F_q \times d_q$). Each filter position defines a spatial "pixel" (or, more accurately, a feature) in the next layer.

The p -th filter in the q -th layer has parameters denoted by the 3-dimensional tensor $W^{(p,q)} = [w^{(p,q)}_{i,j,k}]$. The indices i, j, k indicate the positions along the height, width, and depth of the filter. The feature maps in the q -th layer are represented by the 3-dimensional tensor $H^{(q)} = [h^{(q)}_{i,j,k}]$. When the value of q is 1, the special case corresponding to the notation H simply represents the input layer (which is not hidden). Then, the convolutional operations from the q -th layer to the $(q+1)$ -th layer are defined as follows:

$$h^{(q+1)}_{i,j,p} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w^{(p,q)}_{r,s,k} h^{(q)}_{i+r-1,j+s-1,k} \quad \forall i \in \{1..., L_q - F_q + 1\}$$
$$\forall j \in \{1..., B_q - F_q + 1\}$$
$$\forall p \in \{1..., d_q + 1\}$$

One property of convolution is that it shows equivariance to translation. In other words, if we shifted the pixel values in the input in any direction by one unit and then applied convolution, the corresponding feature values will shift with the input values. This is because of the shared parameters of the filter across the entire convolution.

Padding

One observation is that the convolution operation reduces the size of the (q + 1)th layer in comparison with the size of the q -th layer. This type of reduction in size is not desirable in general, because it tends to lose some information along the borders of the image (or of the feature map, in the case of hidden layers). This problem can be resolved by using padding. In padding, one adds $(F_q - 1)/2$ "pixels" all around the borders of the feature map in order to maintain the spatial footprint. Padding in 2D is simply a 'frame' of n pixels around the image (Figure 2.5). Note that it does not make much sense to use a padding of say 3 (pixels) if we use only a 3 by 3 local receptive field, since it will only go one pixel over the image border.

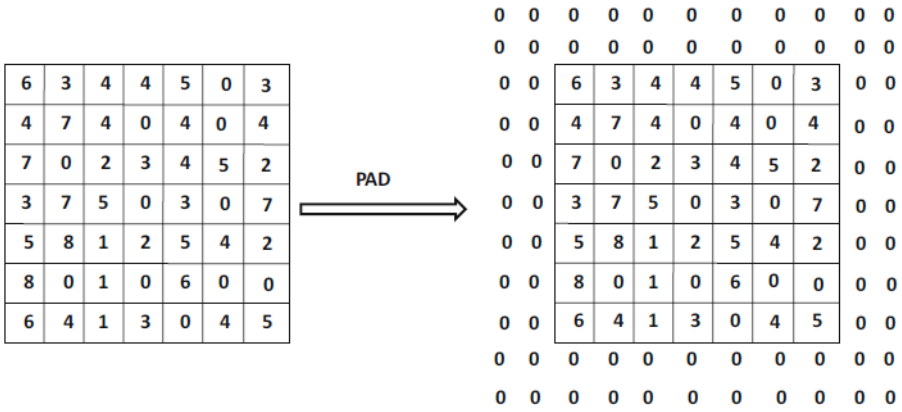


Figure 2.5: An example of padding. Each of the d_q activation maps in the entire depth of the q -th layer are padded in this way [1].

Strides

The parameter which says by how many components we move the receptive field between taking inputs is called the stride of the convolutional layer. When a stride of S_q is used in the q -th layer, the convolution is performed at the locations 1, S_q+1, $2S_q+1$, and so on along both spatial dimensions of the layer. The spatial size of the output on performing this convolution1 has height of $(L_q - F_q)/S_q + 1$ and a width of $(B_q - F_q)/S_q + 1$. As a result, the use of strides will result in a reduction of each spatial dimension of the layer by a factor of approximately S_q and the area by S_q^2, although the actual factor may vary because of edge effects.

The ReLU Layer

The convolution operation is interleaved with the pooling and ReLU operations. The ReLU activation is not very different from how it is applied in a traditional neural network. For each of the $L_q \times B_q \times d_q$ values in a layer, the ReLU activation function is applied to it to create $L_q \times B_q \times d_q$ thresholded values. These values are then passed on to the next layer. Therefore, applying the ReLU does not change the dimensions of a layer because it is a simple one-toone mapping of activation values.

Pooling

The pooling operation works on small grid regions of size $P_q \times P_q$ in each layer, and produces another layer with the same depth (unlike filters). For each square region of size $P_q \times P_q$ in each of the d_q activation maps, the maximum of these values is returned. This approach is referred to as max-pooling. If a stride of 1 is used, then this will produce a new layer of size $(L_q - P_q + 1) \times (B_q - P_q + 1) \times d_q$. However, it is more common to use a stride $S_q > 1$ in pooling. In such cases, the length of the new layer will be $(L_q - P_q)/S_q + 1$ and the breadth will be $(B_q - P_q)/S_q + 1$. Therefore, pooling drastically reduces the spatial dimensions of each activation map.

Unlike with convolution operations, pooling is done at the level of each activation map.

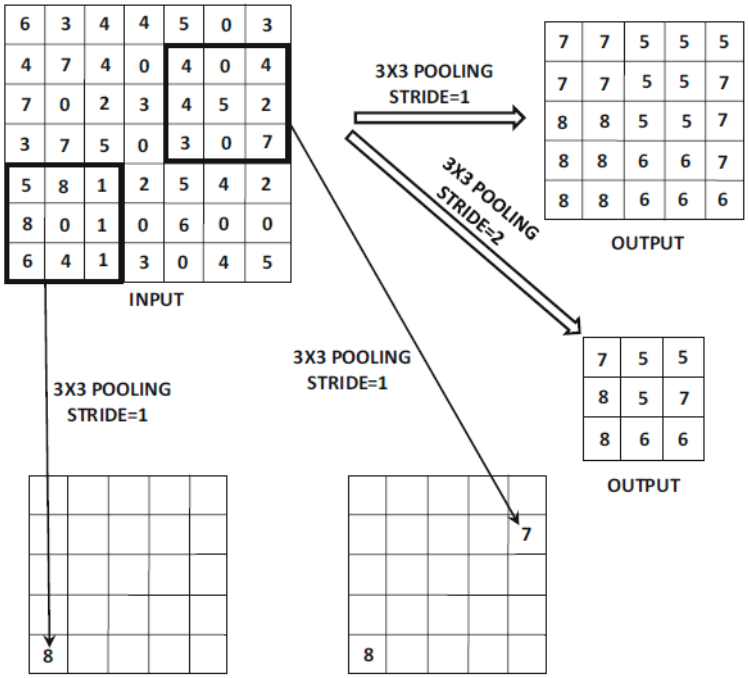


Figure 2.6: An example of a max-pooling of one activation map of size 7 × 7 with strides of 1 and 2. A stride of 1 creates a 5 × 5 activation map with heavily repeating elements because of maximization in overlapping regions. A stride of 2 creates a 3×3 activation map with less overlap. Unlike convolution, each activation map is independently processed and therefore the number of output activation maps is exactly equal to the number of input activation maps [1].

Other types of pooling (like average-pooling) are possible but rarely used. In the earliest convolutional network, referred to as LeNet-5, a variant of average pooling was used and was referred2 to as subsampling. In general, max-pooling remains more popular than average pooling.

Fully Connected Layers

Each feature in the final spatial layer is connected to each hidden state in the first fully connected layer. This layer functions in exactly the same way as a traditional feed-forward network. In most cases, one might use more than one fully connected layer to

increase the power of the computations towards the end.

As the image goes through the network, after a number of layers, we get a small image with a lot of channels. Then we can flatten this to a vector and use a simple logistic regression at the end to extract which parts are relevant for our classification problem. The logistic regression (this time with the logistic function) will pick out which parts of the representation will be used for classification and create a result which will be compared with the target and then the error will be backpropagated. This forms a complete convolutional neural network. A simple but fully functional convolutional network with four layers is shown in Figure 2.7

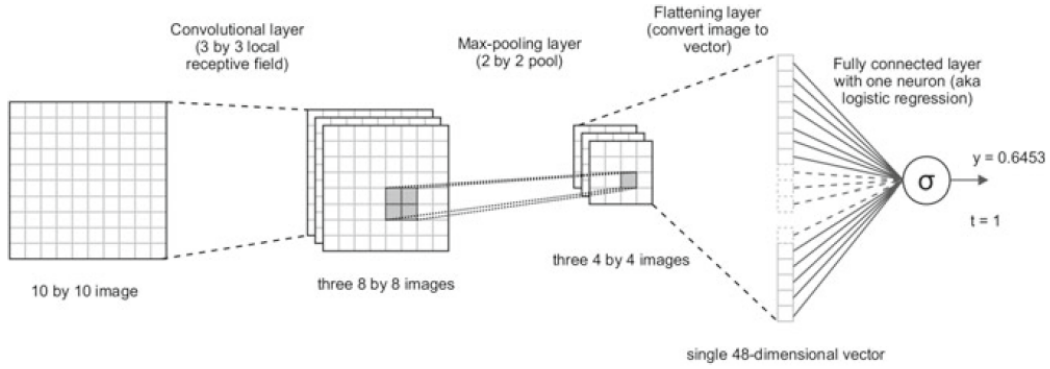


Figure 2.7: A convolutional neural network with a convolutional layer, a max-pooling layer, a flattening layer and a fully connected layer with one neuron [23].

2.2.2 Training a Convolutional Network

The process of training a convolutional neural network uses the backpropagation algorithm. The ReLU is relatively straightforward to backpropagate through because it is no different than a traditional neural network. For max-pooling with no overlap between pools, one only needs to identify which unit is the maximum value in a pool (with ties broken arbitrarily or divided proportionally). The partial derivative of the loss with respect to the pooled state flows back to the unit with maximum value. All entries other than the maximum entry in the grid will be assigned a value of 0.

The backpropagation through convolutions is also not very different from the backpropagation with linear transformations (i.e., matrix multiplications) in a feed-forward network. First, we describe a simple element-wise approach to backpropagation. Assume that the loss gradients of the cells in layer $(i+1)$ have already been computed. The loss derivative with respect to a cell in layer $(i+1)$ is defined as the partial derivative of the loss function with respect to the hidden variable in that cell. Convolutions multiply the activations in layer i with filter elements to create elements in the next layer. Therefore, a cell in layer $(i+1)$ receives aggregated contributions from a 3-dimensional volume of elements in the previous layer of filter size $F_i \times F_i \times d_i$. At the same time, a cell c in layer i contributes to multiple elements (denoted by set S_c) in layer $(i+1)$, although the number of elements to which it contributes depends on the depth of the next layer and the stride. Identifying this "forward set" is the key to the backpropagation. A key point is that the cell c contributes to each element in S_c in an additive way after multiplying the activation of cell c with a filter element. Therefore, backpropagation simply needs

to multiply the loss derivative of each element in S_c with respect to the corresponding filter element and aggregate in the backwards direction at c. For any particular cell c in layer i, the following pseudocode can be used to backpropagate the existing derivatives in layer-$(i+1)$ to cell c in layer-i:

Algorithm 9: Backpropagating Through Convolutions [1]

Identify all cells S_c in layer $(i+1)$ to which cell c in layer i
for *each cell* $r \in S_c$ **do**
    let δ_r be its (already backpropagated) loss-derivative with respect to cell r;
    let w_r be weight of filter element used for contributing from cell c to r;
$\delta_c = \sum_{r \in S_c} \delta_r \cdot w_r$

After the loss gradients have been computed, the values are multiplied with those of the hidden units of the $(i-1)$ -th layer to obtain the gradients with respect to the weights between the $(i-1)$ -th and i -th layer.

2.3 Deep Neural Networks and Reinforcement learning

We obtain deep reinforcement learning (deep RL) methods when we use deep neural networks to approximate any of the following components of reinforcement learning: value function $\hat{v}(s;\theta)$ or $\hat{q}(s,a;\theta)$, policy $\pi(a|s;\theta)$, and model (state transition function and reward function). Here, the parameters θ are the weights in deep neural networks.

Learning a sequential decision-making task appears in two cases: (i) in the offline learning case where only limited data on a given environment is available and (ii) in an online learning case where, in parallel to learning, the agent gradually gathers experience in the environment.

In the online setting, the learning problem is more intricate and learning without requiring a large amount of data (sample efficiency) is not only influenced by the capability of the learning algorithm to generalize well from the limited experience. Indeed, the agent has the possibility to gather experience via an *exploration/exploitation* strategy. In addition, it can use a replay memory to store its experience so that it can be reprocessed at a later time.

2.3.1 Deep Q-Networks

A deep Q network (DQN) is a multi-layered neural network that for a given state s outputs a vector of action values $Q(s,\cdot;\theta)$, where θ are the parameters of the network. For an n-dimensional state space and an action space containing m actions, the neural network is a function from $\mathbb{R}_n$ to $\mathbb{R}_m$.

In the section 1.4.2 we have given the brief introduction to the Q-function approximations and fitted Q-learning. Two important ingredients of the DQN algorithm as proposed by Mnih et al. [17] are the use of a target network, and the use of experience replay. This algorithm uses two heuristics to limit the instabilities:

- The target Q-network in Equation 1.35 is replaced by $Q(s', a'; \theta_k^-)$ where its parameters θ_k^- are updated only every $C \in \mathbb{N}$ iterations with the following assignment: $\theta_k^- = \theta^k$. This prevents the instabilities to propagate quickly and it reduces the risk of divergence as the target values Y_k^Q are kept fixed for C iterations. The idea of target networks can be seen as an instantiation of fitted Q-learning, where each period between target network updates corresponds to a single fitted Q-iteration. The target used by DQN is then

$$Y_k^Q = r + \gamma \max_{a \in \mathcal{A}} Q(s', a; \theta_k^-) \tag{2.7}$$

- In an online setting, the replay memory keeps all information for the last $N_replay \in \mathbb{N}$ time steps, where the experience is collected by following an ϵ-greedy policy. The updates are then made on a set of tuples $< s, a, r, s_0 >$ (called mini-batch) selected randomly within the replay memory. This technique allows for updates that cover a wide range of the stateaction space. In addition, one mini-batch update has less variance compared to a single tuple update. Consequently, it provides the possibility to make a larger update of the parameters, while having an efficient parallelization of the algorithm.

2.3.2 Double DQN

The max operation in Q-learning (Equations 1.34, 1.35) uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values in case of inaccuracies or noise, resulting in overoptimistic value estimates. Therefore, the DQN algorithm induces an upward bias. The double estimator method uses two estimates for each variable, which allows for the selection of an estimator and its value to be uncoupled (Hasselt, [25]). In the original Double Q-learning algorithm, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, θ and θ'. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. The target value Y_k^Q is replaced by

$$Y_k^{DoubleQ} = r + \gamma Q(s', \operatorname*{argmax}_{a \in \mathcal{A}} Q(s', a; \theta_k); \theta'_k) \tag{2.8}$$

The selection of the action, in the argmax, is still due to the online weights θ_k. This means that, as in Q-learning, we are still estimating the value of the greedy policy according to the current values, as defined by θ_k. However, we use the second set of weights θ'_k to fairly evaluate the value of this policy. This second set of weights can be updated symmetrically by switching the roles of θ and θ'.

3 A Deep RL algorithm for constrained drone coverage

3.1 Problem formulation

Suppose that we have a flying robot, for example a quadcopter-type UAV, to shot an area of interest with the onboard camera. We assume that at any position, the UAV can observe its state, i.e. its position. If we have full information about the environment, and the battery endurance is unlimited, a robot motion planning algorithm can be used and the problem becomes common.

In our model, the power is limited and the drone starts at a base station (BS) full of power, and it is allowed to travel a fixed amount of moves or steps. We use a relevance map partitioned into cells that describes the environment with costs associated to each cell, but the drone receives the information from the map partially. In this scenario, we consider the problem of finding a maximum cost tour, i.e. a path that starts and ends at BS such that the area collected is maximized. The task is then to learn a policy to optimize the partial coverage thought the fixed number of movements and come back to the base station for recharging. In the learning process, the drone needs to map the situations it faces to appropriate actions to maximize a numerical signal, called reward, that measures the performance of the drone.

3.1.1 Relevance map

A relevance map is a bi-dimensional, rectangular map of the environment to be monitored, whose values represent the relevance of a specific area, this is, the importance of its observation, or the cost for the system if that area is not observed. While this map could be dynamic throughout the mission, we focus on static maps for the duration of one mission in this paper.

We model the relevance map as a grid with weighted cells. In our case study we use the binary map, i.e. with values 0 and 1, to identify the zones that have any importance to be captured. Figure 3.1 illustrates an example. The relevance maps are used to define a priority in the need of visual coverage of parts of the environment by cameras mounted on UAVs.

3.1.2 Agent model

In order to solve the described coverage path planning problem with reinforcement learning, we introduce the control UAV that interacts with its environment by sampling its *state* (s), performing an *action* (a) and receiving a *reward* (r). Since the continuous

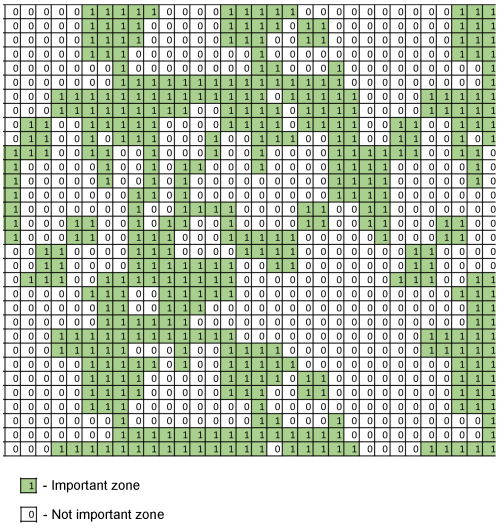


Figure 3.1: Binary map example.

space is too large to guarantee the convergence of the algorithm, in practice, normally these set will be represented as discrete finite sets approximately. Here we suppose that the camera is fixed in vertical position, in other words, the camera main axis matches Z axis of the drone. We also include the battery level p to the state. The *state* of the UAV is defined as the 4-tuple $s = [x, y, z, p] \in \mathcal{S}$ at the time step k:

- $x \in [0, m], Y \in [0, n], z \in [0, h]$ are the spatial coordinates of the UAV, assuming it cannot fly outside the range of the environment with size $m \times n$ and h is the maximum flying height;
- $p \in [0, b]$ is the current battery level (power). The drone stars from its base station fully charged and decreases the battery level one every time step.

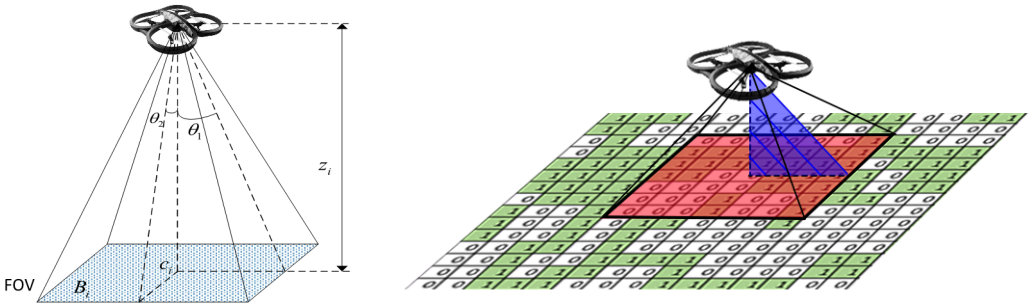
The drone can use the following set of actions $\mathcal{A}$ that decreases by one the power:

- *Forward, Backward, Left, Right*: move the agent to a neighboring cell in the four main directions.
- *Up, Down*: increment or decrease by one the flying height z.
- *ForwardLeft, ForwardRight, BackwardLeft, BackwardRight*: move the agent to a diagonal cell.

As we implement a Double DQN with soft update, as described in section 3.3, the network input consists in the relevance map and the current agent state. In this case, we cannot represent the state as a tuple as we need it as a relevance map size matrix. To fulfill this requirement we can get away from drone coordinates and start representing the current state basing on the drone camera's field of view (FOV).

If we accept an approximation letting the projection of the coordinates of the camera on the ground plane in the world reference system match the projection of the drone mass center coordinates, it is possible to define the ground projection of the image plane as a rectangular shape FOV enclosing the portion of ground plane observed by the agent, as shown in Figure 3.2 (a).

That is, the four corners of the images acquired by the camera can be represented as:



(a) Field of view of an UAV.

(b) Binary map example.

Figure 3.2: Modelling the covering at a state.

$$\hat{x} \in \left[\frac{-z}{\tan\theta_1}, \frac{z}{\tan\theta_1}\right], \hat{y} \in \left[\frac{-z}{\tan\theta_2}, \frac{z}{\tan\theta_2}\right]$$

If consider the simplest case and suppose that the camera is fixed with $\theta_1 = \theta_2 = 45°$ we receive:

$$\hat{x} \in [-z, z], \hat{y} \in [-z, z]$$

Other words, one step up expand the FOV on one cell each direction and one step down shrink the FOV on one cell each direction. The FOV with other actions changes analogously (see Figure 3.2(b)).

This way we represent the agent state as a matrix of the same size as the relevance map. The cells included in the FOV of the drone are represented by its power level on the moment. All the rest of the cells are represented by zeros, Figure 3.3.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	50	50	50	50	50	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	50	50	50	50	50	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	50	50	50	50	50	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	50	50	50	50	50	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	50	50	50	50	50	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

50 - Zone inside the FOV. The drone is 50% charged

0 - Zone outside the FOV

Figure 3.3: Drone state with its power level.

3.1.3 Reward function

Before defining the reward function let us define the total observed relevance ρ of an agent in state s as the sum of all the relevance, as defined by the relevance map $\mathbf{R}$, lying within the observed area FOV:

$$\rho(s) = \sum_{(x,y)\in \mathrm{FOV}} \mathbf{R}(x, y) \tag{3.1}$$

The reward function will be defined in terms of total observed relevance, meaning that actions leading to an increase of such a value will be positively rewarded. However, we require some additional constraints, otherwise reward maximization will always lead to "extreme" agent states where the entire relevance map is enclosed in FOV, e.g. by flying at very high altitude. We thus define the Constrained Total Observed Relevance (CTOR) as:

$$\hat{\rho}(s) = k(\mathrm{FOV})\rho(s) \tag{3.2}$$

where k is a penalty function with values in $[0, 1]$ penalizing observed areas with low resolution. We are ready to define the reward function of the Markov Decision Process that take into account the battery constraint.

If the action a, applied on state s, leads to a new state s', the local reward is:

$$r(s, a) = \hat{\rho}(s') - \hat{\rho}(s'), \tag{3.3}$$

thus the action is rewarded proportionally with the CTOR increment. Now, let $BL(s)$ be the battery level at the current state s, that is, the number of steps that could be done until the battery is dead. Let $dist(s)$ be the distance to the base station, that is, the minimum number of steps needed to reach the base station from the current state s.

The reward function that integrates battery constraint and local reward is defined as:

$$R(s, a) = (1 - P(s))[r(s, a)(1 - N(s, a)) - 100N(s, a)] + P(s)D(s, a), \tag{3.4}$$

where

$$\begin{aligned}
&P(s) = 1 \text{ if } BL(s) - dist(s) = 0\\
&P(s) = 0 \text{ if } BL(s) - dist(s) > 0\\
&N(s, a) = 1 \text{ if } BL(s) - dist(s') < 0\\
&N(s, a) = 0 \text{ if } BL(s) - dist(s') < 0\\
&D(s, a) = (dist(s') - dist(s))
\end{aligned}$$

3.2 A reinforcement learning algorithm for coverage task

The agent model, described in the previous section, can be interpreted like an agent-enforcement environment interaction (Figure 1.1). The agent has an objective to find a course of actions based on its states, called a policy, that ultimately maximizes its total amount of reward it receives over time.

The action value function $Q(s_t; a_t)$ is used to determine which action to take in a given state for each time step. The agent can iteratively compute the optimal value of this function, and from which derives an optimal policy.

In classic Q-learning algorithms, the Q-function updates with Bellman Equation (1.34). The optimal value functions computed this way are stored into tabular databases (Q-tables). In the coverage task we can deal with very big territories. Therefore, the usual approaches making use of tables storing values in large matrices are no longer practical because they do not scale-up especially in continuous (hence infinite) state spaces.

In this case it is convenient to use the deep Q-learning approach because it let us use Deep Networks as function approximators for Q (Deep Q-Networks). The implementation of Deep Q-Networks does not change the core concept of RL, with the exception that the the policy is now generated by a neural network.

To let the network explore different actions and allow it learn new things we should maintain the exploration-exploitation balance. The exploration is guaranteed with an ε - greedy policy. At each iteration, the agent picks a random action with probability ε and an action given by the network with probability $1 - \varepsilon$.

The problem of the classic Q-learning is that we can not use the same network parameters θ both for action selection and evaluation, because this could lead to biased estimates for Q. The algorithm we are proposing here adapts Double Deep Q-Network approach introduced in section 2.3.2. This way we train two independent networks for selection and evaluation tasks. In the box bellow we provide the pseudocode of the double deep Q-learning algorithm used in this work.

Algorithm 10: Double Deep Q-learning algorithm for drone coverage

Algorithm parameters:
small exploration probability $\varepsilon \in (0, 1]$
Initialize: relevance map;
base stations coordinates;
battery limit;
movement limit;
Initialize network parameters: replay memory;
main network parameters *theta* to random weights;
target network parameters $\tilde{\theta} \leftarrow \theta$;

for *each episode* **do**
  Choose initial state s_0 from a base station chosen randomly
  repeat
    if *the object state is inside the region and the battery is charged* **then**
      for *each step of episode* **do**
        With probability ε select random action a_t
        Otherwise select $a_t = \max_a Q(s_t, a)$
        Take action a_t, observe r_t, s_{t+1}
        Update exploration probability ε
        Backpropagate and update DQN with the minibatch
        if *C updates to DQN since last update to target network* **then**
          update the target Q-network $\hat{Q}(s, a) \leftarrow Q(S, A)$;
    else
      Finish the episode
  until *the territory is covered or movement limit is achieved*

3.3 Implementation of Neural Network

The input of the network is a tridimensional matrix of size (32, 32, 2). The depth 2 refers to two channels. The first channel contains the visual information of the current drone camera output. The second channel encapsulates the drone position in the environment, as well as the battery level and the base stations. Therefore, we construct a Q-Network based on convolutionals layers. Each layer reduce the two first dimensions of the matrix until we get a 1x1 matrix in the final layer. On the other hand, the number of channels is increased in the first layer and then we start to reduce it until we get as many channels as possible actions for the final layer. In this way we prevent from flattening the convolutional layer, which allows us to perform faster computations.

Figure 3.4 shows the basic structure of the network. We use three convolutional blocks, each one containing a convolutional layer, a batch normalization, a relu activation and a 10% dropout. Between these we have two maxpools operations. Then we have another convolutional layer as the model output.

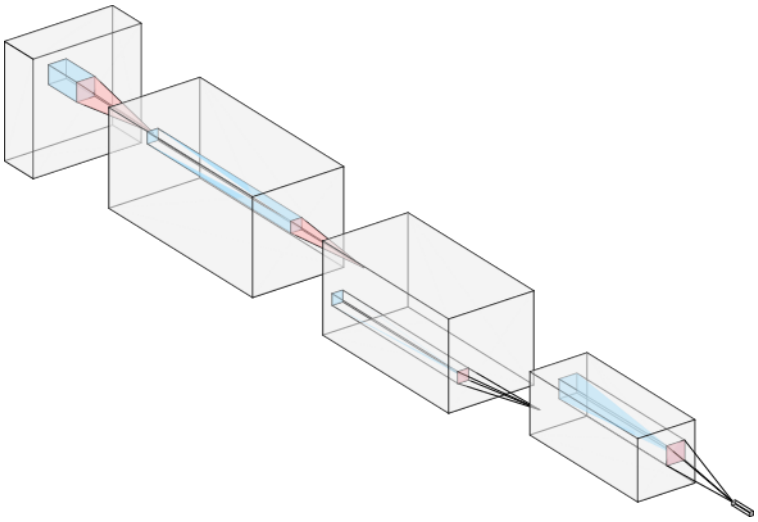


Figure 3.4: Network architecture.

The network training process is managed by the agent as defined for the environment. It uses a SmoothL1 loss criteria, and an Adam optimizer with learning rate of 0.001. In this agent, we define the current model Q and the target network Q' as in the q-learning technique described in [17]. We define the future reward as

$$f(s') = r + 0.8 * \arg\max(Q'(s'))$$

were 0.8 is the γ value that represents the importance of future rewards and s' is the state obtained from performing action a in state s following the policy π. Then we can define the loss function as

$$\text{Loss}(s, a) = \text{SmoothL1}(Q(s), f(\pi(s, a))). \tag{3.5}$$

3.4 Experimental results

3.4.1 Simulation environment

The coverage problem is formulated as a dynamic game. To implement successfully the described algorithm a custom environment was created using python libraries. The library OpenAI Gym [7] specifically made for RL applications was used to interact with the environment, start game episodes, observe states, collect rewards and perform actions.

The agent can move in a two dimensional grid through action commands. Each action consumes one unit of movement budget. The initial step consists of a fixed map, a zero-initialized coverage rate, a position at a base station and full power. Additionally, the initial movement budget is uniformly sampled from a movement budget range which is set to 200-400 for the purpose of this evaluation. The UAV's camera field of view (FoV) depends on the height and it centered underneath the drone. After each step of the mission, algorithm marks the FoV as seen in the coverage grid map (Figure 3.3).

During the training and simulation we used several binary maps on a 32×32 grid as relevance maps. The same algorithm can also be applied for bigger maps. To avoid the recovering and give the robot some motivation to discover new zones, the ones from the covered zones are converted to zero. When the robot leaves the area permitted by the map it receives the negative reward. Negative rewards encourage the robot to come back as quickly as possible because it makes the agent lose points when the game is still being played. At the beginning, when the agent is no trained enough, it can go too far outside and spend too much time there that slows down the learning process. To resolve this issue we decided to end the episode each time the robot goes more than two steps out the boundary any direction (x, y or z) or when the battery is less than -100. When the robot achieves the base station its battery level changes by 100. When the agent learns good to charge and do not leave the box, the episode can last the unlimited number of steps. The visual representation of the environment is shown in Figure 3.5.

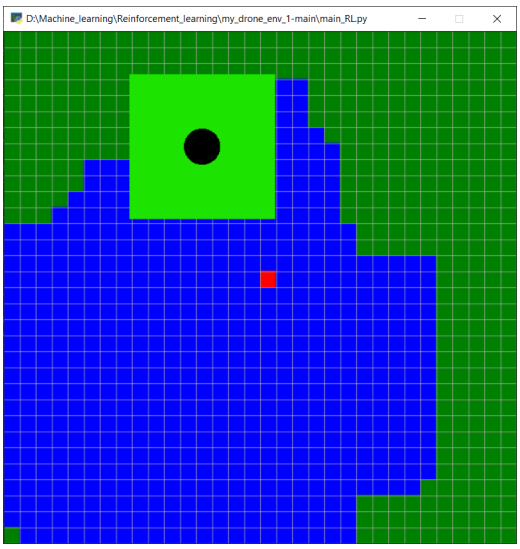


Figure 3.5: The simulation environment. A 32×32 grid map showing a base station in red, the charged agent (big green square), covered area (in blue cells) and not covered area (green cells).

3.4.2 Training

To train the model, the input is not taken directly from the states of the observed environment. Instead, tuples (state, reward, action, next state) are collected after each interaction with the environment and stored in a replay memory which is sampled. This approach helps increase the performance of the learning by reducing the correlation between individual samples in a training batch.

The training hyper-parameters used are listed in Table 3.1. They were obtained after a series of experimentations with the goal of determining the best combination of parameters.

During training, the exploration factor ε is varying, starting from 0.6 at the beginning and decreasing throughout the learning process until it reaches 0.1 using the following proportion:

$$\begin{aligned} &\varepsilon = 0.6 \quad \text{for} \quad i = 0 \\ &\varepsilon = 0.6 \cdot 0.9^{i/300} \quad \text{for} \quad 0 < i < 5000 \\ &\varepsilon = 0.1 \quad \text{for} \quad i > 5000 \end{aligned}$$

This decreasing is motivated by the fact that the agent initially has not acquired any knowledge, which forces the exploration of the free space in priority. With ongoing training, the agent will gradually exploit and rely on the accumulated knowledge and less on the exploration. The presented proportion had shown good results during the experiment but can be not the optimal one.

Parameter	Value	Signification
M	5000	maximum number of training episodes
$H \times W$	32×32	length and width of the workspace
Z	5	hight of the workspace
m	32	minibatch size
$\mathfrak{B}$	400	maximum movement budget
BL	200	maximum battary level
ϵ	0.1 .. 0.6	exploration rate
γ	0.9	discount factor

Table 3.1: Hyper-parameters for the DQN training.

Figure 3.6 describes the evolution of the total reward (the sum of the rewards) per episode in function of the number of training episodes. We can see that at the beginning the agent gets a lot of negative reward as making random steps. It is almost guaranteed leaves the environment and or goes opposite direction from the base when the battery is low. Throughout the training, the agent manages to collect positive reward which means it learns to stay in the restricted zone, charge the battery and discover the new territories. This shows that the learning improves and the agent gets better at covering the workspace.

Figure 3.7 illustrates the progress in average number of steps per episode during training. Every episode the agent could live more steps without going out the boundary and charging the battery on time. Note that even after 5000 episodes, the average number

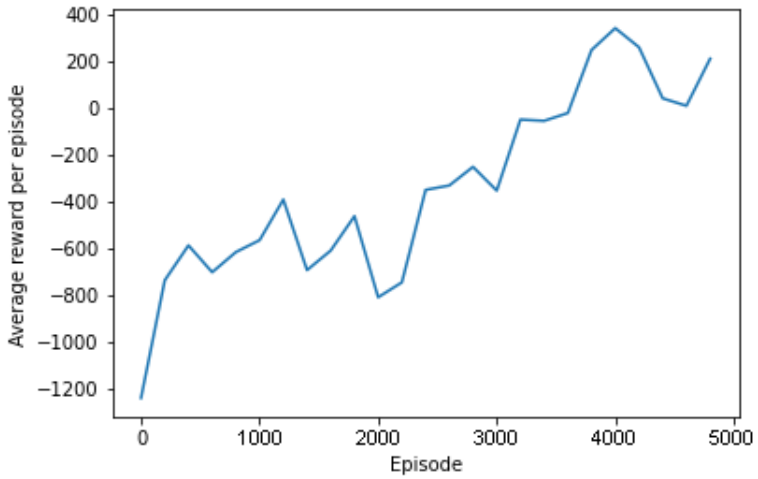


Figure 3.6: Average reward per episode during training. The statistics were computed by running an ϵ-greedy policy.

of steps is not equal to 400 (the maximum permitted number of steps). It means that the network is still not very good trained and it needs more training. But also we should take in account that the agent also can go wrong with the random steps permitted by ϵ.

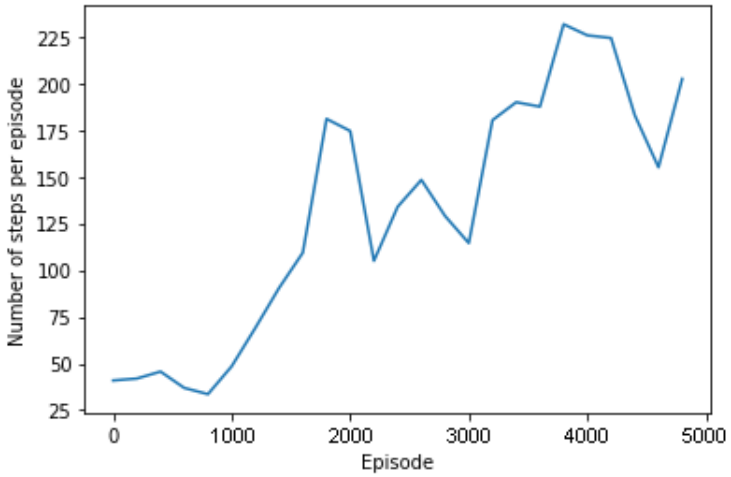


Figure 3.7: Average number of steps per episode during training.

3.4.3 Simulations and Discussion

For simulation purposes we have chosen two different maps to evaluate the coverage rate made by the model. In this experiment we want to see the ability of the agent to cover the field limiting the number time steps.

Figure 3.8 shows the percentage of the area that was covered for two different maps in a 32×32 grid. The first map is full of ones (we consider that all the zones of the map have the same importance). In the second map we have the square objects 8×8 cells size staggered like a checkerboard.

It can be clearly seen that the covered area increases with movement budget, which is expected as the drone is able to do more steps in different directions. We can also

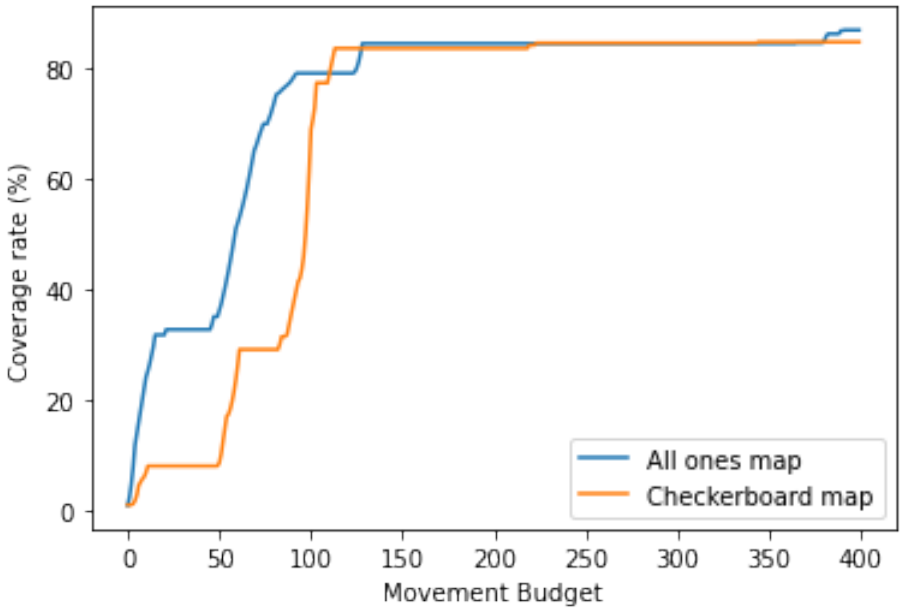


Figure 3.8: Coverage rate for two maps with different movement budget.

note that the model perfectly adapts to the different type of the maps. Even so as the model is still not very good trained, the robot does to much inefficient steps to and back and it can not cover the whole territory with an adequate movement budget. It is important to say that the agent does not necessarily utilize the whole allocated budget if it determines that there is a risk of not returning to the base in time or the coverage goal is already fulfilled. Thus, the drone finds a tour balancing the goals of safe recharging and maximum coverage ratio.

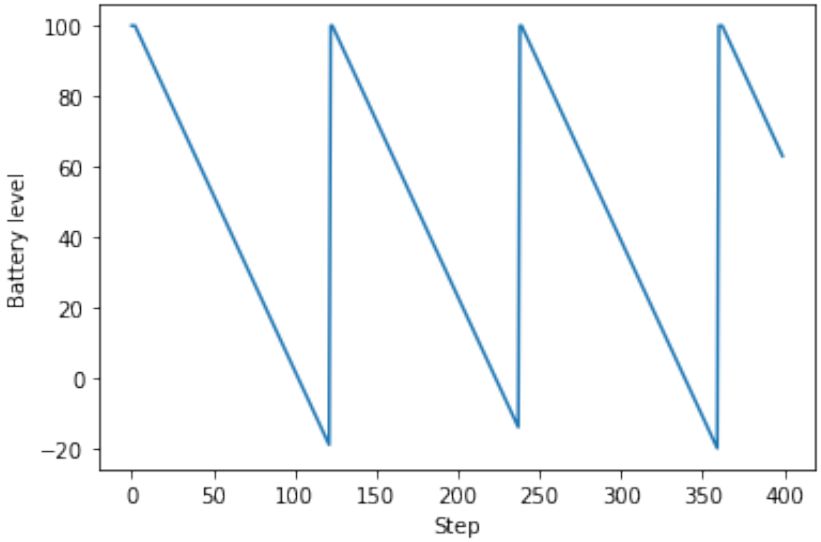


Figure 3.9: Battery level per step.

Finally, the drone's ability to plan a trajectory that comes back to the base station over the full movement budget range is evaluated through Figure 3.9, showing the battery level per step in the test mode. As we can see, when the battery level falls down to zero the robot immediately goes to recharge. The periodicity of the graphic shows us that

the drone learned well do not exceed the fixed number of steps in coverage mode (the power p is stated as 100 steps). As a consequence, the drone goes to charge in time and does not go around the base with random movements.

3.5 Conclusion and future work

This thesis focuses the problem of coverage path planning in unknown environments using a drone with limited battery endurance, with the aim of maximizing the covered area. The task is expressed as a game played by the drone that seeks to maximize a carefully chosen reward.

Although there exist similar approaches in the literature, the case in which the available power is limited is usually not considered and then it is a challenge to design a learning algorithm with this realistic constraint. In this work, we have introduced a new deep reinforcement learning approach for that optimization problem.

By feeding spatial information through map, we train a Q-network to learn a UAV control policy that generalizes over varying starting positions and varying power constraints. Using this method, we observed an incremental learning process that successfully balances safe landing and coverage of the target area on two different maps. In the first environment all cells have to be visited and in the second scenario, only a distinguished area is considered as target. The first results are promising as the agent learn to come back to the base station to refuel and to avoid crossing the environment boundaries.

The main drawback of the proposed method is that the training process is time consuming and the reward function could be tuned to get better results. In any case, the proposed approach can be seen as an initial step for handling the battery constraint in coverage problems on unknown maps.

To complete this ongoin work, we plan to experimental show the relevance of our method compared with other naive coverage paths, for instance the zig-zag patterns. In the future we also want to investigate the possibilities of transfer learning for this problem. At first we will train the drone with easier problems, for example, considering four actions, to further accelerate the training process described in this work. From there we will examine approaches to transfer the agent learning to greater dimensions and dynamics. Finally, another interesting direction for future research is to adapt our method for a more realistic scenario in which the power consumption is variable (depending on the wind, for example), the map has different topological altitudes, there exist obstacles or forbidden zones, etc.

Bibliography

[1] Charu C Aggarwal et al. *Neural networks and deep learning*. Springer, 2018.

[2] Oswin Aichholzer et al. "Scheduling drones to cover outdoor events". In: *Proc. European Workshop on Computational Geometry, EuroCG*. 2020.

[3] Esther M Arkin, Sándor P Fekete, and Joseph SB Mitchell. "Approximation algorithms for lawn mowing and milling". In: *Computational Geometry* 17.1-2 (2000), pp. 25–50.

[4] Kai Arulkumaran et al. "Deep reinforcement learning: A brief survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38.

[5] Harald Bayerlein, Paul De Kerret, and David Gesbert. "Trajectory optimization for autonomous flying base station via reinforcement learning". In: *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. IEEE. 2018, pp. 1–5.

[6] Petros S Bithas et al. "A survey on machine-learning techniques for UAV-based communications". In: *Sensors* 19.23 (2019), p. 5170.

[7] G. Brockman et al. "OpenAI Gym". In: *ArXiv* abs/1606.01540 (2016).

[8] Tauã M Cabreira, Lisane B Brisolara, and Paulo R Ferreira Jr. "Survey on coverage path planning with unmanned aerial vehicles". In: *Drones* 3.1 (2019), p. 4.

[9] Omid Esrafilian, Rajeev Gangula, and David Gesbert. "Learning to communicate in UAV-aided wireless networks: Map-based approaches". In: *IEEE Internet of Things Journal* 6.2 (2018), pp. 1791–1802.

[10] Vincent François-Lavet et al. "An introduction to deep reinforcement learning". In: *arXiv preprint arXiv:1811.12560* (2018).

[11] Enric Galceran and Marc Carreras. "A survey on coverage path planning for robotics". In: *Robotics and Autonomous systems* 61.12 (2013), pp. 1258–1276.

[12] Xintian Han. "A Mathematical Introduction to Reinforcement Learning". In: 2018.

[13] Andrej Karpathy. "Stanford university cs231n: Convolutional neural networks for visual recognition". In: *URL: http://cs231n. stanford. edu/syllabus. html* (2018).

[14] Sergey Levine et al. "End-to-end training of deep visuomotor policies". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1334–1373.

[15] Yuxi Li. "Deep reinforcement learning: An overview". In: *arXiv preprint arXiv:1701.07274* (2017).

[16] Chi Harold Liu et al. "Distributed and energy-efficient mobile crowdsensing with charging stations by deep reinforcement learning". In: *IEEE Transactions on Mobile Computing* (2019).

[17] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[18] Guillem Muñoz et al. “Deep reinforcement learning for drone delivery”. In: *Drones* 3.3 (2019), p. 72.

[19] Guido Novati, Lakshminarayanan Mahadevan, and Petros Koumoutsakos. “Deep-Reinforcement-Learning for Gliding and Perching Bodies”. In: *arXiv preprint arXiv:1807.03671* (2018).

[20] Huy X Pham et al. “Autonomous uav navigation using reinforcement learning”. In: *arXiv preprint arXiv:1801.05086* (2018).

[21] Claudio Piciarelli and Gian Luca Foresti. “Drone patrolling with reinforcement learning”. In: *Proceedings of the 13th International Conference on Distributed Smart Cameras*. 2019, pp. 1–6.

[22] Hang Qi et al. “Energy Efficient 3-D UAV Control for Persistent Communication Service and Fairness: A Deep Reinforcement Learning Approach”. In: *IEEE Access* 8 (2020), pp. 53172–53184.

[23] Filip Šoljić. “Sandro Skansi: Introduction to Deep Learning. From Logical Calculus to Artificial Intelligence”. In: *Synthesis philosophica* 34.2 (2019), pp. 477–479.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction. Second edition.* Adaptive computation and machine learning series. Cambridge, MA: The MIT Press, 2018. ISBN: 9780262039246.

[25] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *arXiv preprint arXiv:1509.06461* (2015).

[26] Yongs Zeng, Qingqing Wu, and Rui Zhang. “Accessing from the sky: A tutorial on UAV communications for 5G and beyond”. In: *Proceedings of the IEEE* 107.12 (2019), pp. 2327–2375.