SAPIENZA
UNIVERSITÀ DI ROMA

# An experimental study on the effectiveness of planning-based approaches in improving sample efficiency in RL


Facoltà di Ingegneria dell'informazione, informatica e statistica
Master Degree in Artificial Intelligence and Robotics

**Andrea Bonaiuti**
ID number 1618825

Advisor
Prof. Fabio Patrizi

Co-Advisor
Prof. Luca Iocchi



Academic Year 2023/2024

Thesis not yet defended

---

**An experimental study on the effectiveness of planning-based approaches in improving sample efficiency in RL**

Tesi di Laurea Magistrale. Sapienza University of Rome



This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: bonaiuti.1618825@studenti.uniroma1.it

*to all*
*my beloved*

# Abstract

This thesis is an explorative study regarding the effectiveness of symbolic planning-based approaches as support for typical Reinforcement Learning algorithms.
Planning-based approaches deal in similar contexts as Reinforcement Learning's ones, but generally on a higher level of abstraction. They both define their problem with MDP assumptions and both aim to do the same thing, to teach an agent what action/move to select for each given state in order to reach a goal.
Motivated by that, the study on this thesis generates a plan with a symbolic planning-based approach and uses it as a starting point/initialization for Reinforcement Learning algorithms.
The selected case study is the Lunar Lander environment offered by Gymnasium's OpenAI, a minigame in which a spacecraft has to land safely on the ground.
The planning-based method involves an initial discretization of the domain to transpose the original environment in finite domain variables, a domain and problem definition in FOND (Fully Observable Non-Deterministic) settings to generate a plan that will be used to initially guide the reinforcement learning model.
The experiment with the proposed methods confirm the struggle of tabular versions of RL with this kind of environment, but show promising results with the Deep Q-Learning version methods that can solve the environment in a shorter time.

# Contents

# Chapter 1

# Introduction

Reinforcement Learning is one of the most successful branches in Machine Learning research field, along with Supervised, Unsupervised and Semi-Supervised Learning. This relevance is motivated by its capability to make an agent able to act and/or react instantaneously in a given environment.
Furthermore, thanks to recent technological advances and the spread of Deep Learning methods, in the last decade Reinforcement Learning methods have been able to deal with big and complex environments.

Historically notable achievements have been reached by systems developed to play board games like Backgammon, with TD-Gammon that reached the rank of 'master'[35] chess and in the chinese game of Go, a more complex game with a very large state space with respect to chess and backgammon. In this scenario the famous AlphaGo by Google [6] took the title of world champion in 2016.
Moreover, still in the gaming sector, the latest successful application focus on videogames with even more complex state spaces and the property of real time setting.
The most prominent example of it is the OpenAI Five model [4] that was able to beat pro players in the DOTA 2 game, a multiplayer online battle arena (MOBA) that is played in matches between two teams of five players each, with each team that has to conquer enemy bases and simultaneously defend their own.

Nowadays, Reinforcement Learning is a widely studied field of research with numerous applications, from autonomous driving and industry automation to finance, trading and healthcare sectors.
Reinforcement Learning can be deployed in different metodologies and different tools. But the overall structure results in a model that, at every step, receives as input a state space and gives as output the best action to take in that circumstance.
This model can come in multiple ways, as a table that records each state and the respective best action to execute or as a neural network that predicts action values of each input state. This argument will be described in more detail in the next chapters.

Automated/Classical Planning is another branch of Artificial Intelligence, and shares several properties with Reinforcement Learning scenarios and applications.

As it can be guessed by its name, Planning's aim is to generate a plan, that consists in a strategy or a sequence of actions worked out beforehand for reaching a predefined goal.
And, already by its more generic definition, the most important overlap between Automated Planning and Reinforcement Learning can be derived. They share the same setting, that is the ability to tell an agent what to do in an environment.
Automated Planning is typically done offline and requires a prior description of the environment/domain dynamics, constraints and goals in order to generate a plan. This is instead one of the biggest differences between the two fields: Automated Planning strongly relies on predefined domain and problem while Reinforcement Learning does not require prior knowledge and learns to adapt to the environment by interacting with it.

These complementarities and similarities together motivate the study of this thesis in exploring the efficiency of symbolic techniques of planning as support for reinforcement learning ones.
The proposed method generates a strategy/plan using Classical Planning techniques that will be used as a basis for the Reinforcement Learning training.
As a case study for this thesis, the selected one is the Lunar Lander environment offered by OpenAI's Gymnasium, a typical rocket trajectory optimization problem, where the goal is to safely land a spacecraft on a landing pad on the ground. The spacecraft has 3 different engines that can be fired to adjust its trajectory, that can also be perturbated by wind.
The observation space is composed by 8 variables, 2 booleans that represent whether each leg is in contact with the ground and the other 6 are floats that describe the lander coordinates and velocities.

Due to the difficulties that planners have in handling continuous environments and their limited computational efficiency with a big state spaces, a domain abstraction has been used to transpose the domain in the PDDL language by a proper discretization that introduces another crucial property of this planning setting: Non-Determinism.
For this reason, a FOND (Fully Observable Non-Deterministic) planner is used to generate plans for the discretized domain, called PRP (Planner for Relevant Policies) [28] that supports also conditional effects.

The next chapters articulates in a literature review of related works, an introducton on the background of planning and reinforcement learning used methods.
Finally a description of the domain construction for the planning phase, followed by the modality of conjunction between planning and reinforcement learning methods and then a final chapter with the results and their discussion.

# Chapter 2

# Background

## 2.1 Markov Decision Processes

A common and standard formalization of the planning problem is the Markov Decision Process, a class of stochastic sequential decision processes in which the cost and transition functions depend only on the current state of the environment and the current action. [32]

Markov Decision Process is a tuple $D = (S, A, f, r)$ where.:

- $S$ is the set of states.
- $A$ is the set of possible actions with $A(s)$ the set of available actions in a given state $s \in S$.
- $f : S \times A \times S \to [0, 1]$ is the transition function,
  with $f(s, a, s^{'}) = Pr(S_{t+1} = s^{'} | S_t = s, A_t = a)$ that is the probability that the state $s^{'}$ is reached from state $s$ with the execution of action $a$.
- $r : S \times A \times S \times \mathbb{R}$ is the reward function.,
  with $r(s, a, s^{'}, g) = Pr(Rt + 1 = g | S_t = s, A_t = a, St + 1 = s^{'})$ that is the probability of getting a reward $g$ after execution of action $a$ that reaches state $s^{'}$ from state $s$.

Considering the environment at discrete timesteps, at time $t$ the current state is $S_t$, the agent interacts with the environment by selecting an action $A_t$ and reaching new state $S_{t+1}$:

$$S_{t+1} \sim f(S_t, A_t, \cdot) = Pr(S_{t+1} = s^{'} | S_t = s, A_t = a)$$

And obtaining the reward $R_{t+1}$:

$$R_{t+1} = r(S_t, A_t, S_{t+1}, \dot{)}$$

If a reached state cannot be left, it's called *terminal* state and then the MDP is said to be *episodic*.

The agent's behaviour is a function called *policy* $\pi : S \times A \rightarrow [0, 1]$ where $\pi(s, a) = Pr(A_t = a|S_t = s)$ is the probability of selecting action $a$ when current state at time $t$ is $s$.
A policy $\pi$ along with an initial state $s_0$ determines a probability distribution over the possible sequences from which it's defined the *cumulative discounted reward*:

$$G = \sum_{t \geq 0} \gamma^t R_{t+1}$$

where $\gamma$ is the *discount factor* and $0 < \gamma \leq 1$. This value represents the preference towards recent rewards over later ones.

The computation of policies in MDP relies on the estimation of the expected cumulative reward, i.e. the reward expected by following a policy $\pi$ at each state:

$$v_\pi(s) = \mathbb{E}[\sum_{t \geq 0} \gamma^t R_{t+1}|S_0 = s]$$

The function $v$ is called *value function*. In the same manner an *action value* function is defined as the expected cumulative discounted reward for the execution of each action from each state:

$$q_\pi(s, a) = \mathbb{E}[\sum_{t \geq 0} \gamma^t R_{t+1}|S_0 = s, A_0 = a]$$

Where the reward is obtained by following the policy $\pi$ after execution of action $a$ in state $s$.

Algorithms that deal with MDPs try to find the optimal policy $\pi^*$ that would correspond with optimal value and action functions:

$$v^*(s) = max_{\pi \in \Pi} v_\pi(s) = max_{a \in A} q^*(s, a), s \in S$$

Where $\Pi$ represents the set of all policies.

$$q^*(s, a) = \sum_{s' \in S} f(s, a, s^{'})(r_{s'} + \gamma v * (s^{'}))$$

Those functions can be stored explicitly only in some environments, as the state space and action space grow, storing all the possible combinations in a tabular form could be impractical and for continuous state spaces is trivial why it is not even possible at all. But discretize the domain can be a useful workaround as will be described later.

## 2.2 Classical Planning

Classical Planning is a field in Artificial Intelligence that studies the computation of ordered sets of actions from a given initial state to solve a task, i.e. reach a predefined goal state.
First researches appeared in the 50s with studies about state-space search and control

theory to solve tasks for robotic and automatic systems.

### 2.2.1 Planning languages and PDDL

A first standard for this problem is the STanford Research Institute Problem Solver (STRIPS) [11], developed for an autonomous robot controller.
Another notable standard is $SAS^+$ [2], thanks to its rich structural information. It represents a planning problem using multi-valued state variables instead of the propositional facts in STRIPS [2].
ADL formalism [31] then was proposed to allow more expressiveness and a good computational tractability, combining aspects from STRIPS and situation calculus, it will also be extended with conditional effects or state-dependent action costs [15].

From these works derive the actual most common language for modeling planning tasks, PDDL (Planning Domain Definition Language) [1]. This language, originally developed for the AIPS-98 planning competition, intended to express the physics of a domain and explicitly declare what predicates there are, what actions are possible and what effects these actions produce.
The syntax is inspired by Lisp, either for the structure of the domain definition and for the parenthesis expression [25]. While in the first version PDDL already supported many syntactic features like STRIP-style actions, conditional effects, safety constraints, universal quantification over dynamic universes and the specification of hierarchical actions composed by subactions and subgoals; some extensions have been introduced as a standard to include more features like temporal and numeric support [13], timed initial literals [9], soft goals [16] and processes and events [12].

The main concept of PDDL is the definition of a domain. A file in which the environment is described with its objects, predicates, and the available actions.
Objects are the basic blocks of the domain and parameters for predicates, they can be typed as shown in Figure 2.1 in a structured way and instantiated as constants, i.e. objects that are present across all instances of a problem.

```
(:types
    site material - object
    bricks cables windows - material
)
```

**Figure 2.1.** Example of object type definition in PDDL.

Predicates can be simple atoms or expressions associated to objects and usually the goal of a planning task is to reach a specific set of true predicates.

Actions are the functions able to modify predicates, they consists of a declaration of parameters (objects), preconditions (a proposition, set of true predicates) and

```
(:predicates
    (walls-built ?s - site)
    (windows-fitted ?s - site)
    (foundations-set ?s - site)
```

**Figure 2.2.** Example of predicate definition in PDDL language.

effects (negation of some predicates and others to be true).

```
(:action <action_name>
    :parameters (<argument_1> ... <argument_n>)
    :precondition (<logical_expression>)
    :effect (<logical_expression>)
    ; :expansion
)
```

**Figure 2.3.** Typical structure of action definition in PDDL language, parameters can be defined by object type and constants while preconditions are predicates made by these predefined parameters, as well as the effects.

Over the following decades, planning has been exploited and applied in multiple real-world tasks with success, from space missions [29] to underwater vehicles controllers [3].
But, despite its popularity, wide application of planning systems into real-world problems is far from being easily handled.
Planning techniques are limited in the state space they can deal with, and also, even with the recent numeric support of some language and planners, solving a task in a continuous environment still requires some workaround like discretization.

### 2.2.2 FOND planning

Discretizing a domain it's an usual way to simplify a problem and reduce the computational load, but when the discretization is still an abstraction but not in a so-high level, i.e. keeping the same action space, and not introducing macro-actions and subgoals, it is important to introduce the non-determinism in the equation.

When discretizing, a variable's domain is divided in different "bins", intervals of values that the variable can assume but that will be interpreted as the same value in the discretized domain.
Doing so, it is straightforward to reason over that the execution of an action that modifies that variable's value could modify that in a new value still in the same bins as the previous value.
So, in the original continuous domain, the variable's value has changed but in the discretized one that variable's value has not.

This behaviour of the discretized transposition of the environment has to be described in the planning domain, that requires discretized domains.
Fortunately, extension of the PDDL domains has brought the support of Fully Observable Non Deterministic (FOND) domains.

FOND planning [7] is a way to introducing non-determinism in planning along whit probabilistic planning [39]. The former deals with finding a contingent plan or a policy, while the latter primarily focuses on maximizing the probability to reach the goal.
Both approaches' aim is to *determinize* the actions by replacing every non-deterministic action with a set of deterministic ones and then using classical planning algorithm to solve for the solution.
The similarities continue, as both build an incomplete/partial policy to map states with actions and then simulate that policy. During this simulation, if a new state is encountered, the planner classifies that as a new planning problem and updates its policy with a new computed plan.

A first difference between FOND and probabilistic planning are primarily in the definition of action effects: in FOND planning it is required to just define all the possible outcomes as it can be seen in Figure 2.4, while in probabilistic planning each outcome has to be defined with an associated probability.
The non-determinism in FOND planning is introduced by the keywork *oneof*, a clause to be used in the effects section of an action in which it is possible to declare multiple mutually exclusive effects.
But the most significant difference is that FOND techniques simulate every potential outcome of a non-deterministic action while probabilistic approach consider only one randomly selected action outcome for each action.

```
:effect (and
    (when
        (positive ?vx) (oneof (and (not (current_x ?x)) (current_x ?x_next)) (and))
    )
    (when
        (negative ?vx) (oneof (and (not (current_x ?x)) (current_x ?x_prev)) (and))
    )
)
```

**Figure 2.4.** Example of action definition in the study case settings. the "when" clause is used for conditional effects, while the "oneof" clause is used to represent two or more possible effects. The fluent "(and)" indicates a possible effect in which nothing changes.

### Weak, Strong and Strong Cyclic plans

A solution to a FOND planning task is a policy that maps a state to an appropriate action such that the agent eventually reaches the goal. A policy is *closed* if it returns an action for every non-goal state a policy reaches and a state $s$ is said to be *reachable* by a policy if there is a chance that following policy leads the agent to $s$.
When the agent executes an action, the effect is chosen randomly and so, a closed

policy has to handle every possible outcome of an action may return.
There are three types of plans for a FOND problem according to Daniele at al.[8].

- **Weak Plan**: A weak plan is a policy that achieves the goal with non-zero probability. A weak plan may be as simple as a sequence of actions that achieves the goal with assumed non-deterministic action outcomes. A policy for a weak plan it's not closed.
- **Strong Plan**: A strong plan is a closed policy that achieves the goal and never visits the same state twice. A strong plan provides a guarantee on the maximum number of steps to achieve the goal but is often too restrictive.
- **Strong Cyclic Plan**: A strong cyclic plan is a closed policy that achieves the goal and every reachable state can reach the goal using the policy. A strong cyclic plan guarantees that the agent eventually reaches the goal, but does not guarantee the agent can do so in a fixed number of steps.

A FOND planner, as state previously in this chapter, aims to *determinize* all actions, i.e. to modify set of actions $A$ in $A^{'}$ such that every action in $A^{'}$ is deterministic. The single outcome determinization creates $A^{'}$ by selecting a single outcome for every action in $A$. So, the all outcomes determinization creates $A^{'}$ bt creating a new action for every non-deterministic outcome of an action in $A$.
Finding a classical plan to a determinization provides a weak plan for the non-deterministic planning task. And this approach is used repeatedly to build a strong cyclic plan. [14]
This planning method involves a loop in which, when an unvisited reachable state is found, a weak plan in the all outcomes determinization if computed and incorporated to the policy. The planner repeats this process until the policy is strong cyclic or it finds a deadend and backtracks to replan. [27]

## 2.3 Reinforcement Learning

An RL (Reinforcement Learning) agent interacts with an environment over time and from that it learns the best way to interact with it in order to reach a goal.
At each timestep $t$, the agent receives a state $s_t \in S$ from a state space $S$ and selects an action $a_t \in A$ from an action space $A$ according to a policy $\pi(a_t|s_t)$, which defines the agent behaviour.
After the action execution, it receives a scalar reward $r_t$ and a new landing state $s_{t+1}$ according to the domain's reward function $R(s, a)$ and state transition probability $P(s_{t+1}|s_t, a_t)$ respectively.
In an episodic problem, this process continues until the agent reaches the goal or another terminal state. Then it restarts a new episode with the same logic.

As RL problems can be formulated as Markovian Decision Processes (MPDs), it shares the same concepts and formulas like the discounted accumulated reward $R_t = \sum_{k=0} \gamma^k r_{t+k}$. The agent aims to maximize the expectation of such long term return from each state.

Again, as in MDPs, this is exploited by reasoning over a state value function and/or an action value function. These functions can be expressed and decomposed into the Bellman equation, a recursive equation that consider the value of any state as the immediate reward $r_{t+1}$ plus the discounted value of the state that follows.
The value and optimal value function are written respectively as:

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

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

In the same way action value function and its optimal version are:

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

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

### 2.3.1 Tabular Q-Learning

In order to reach optimal values and consequentially the optimal policy $\pi^*$, typically when no model is available, Temporal Difference (TD) is used as learning method for value function evaluation [34] that comes in different versions as SARSA or Q-Learning.
It aims to learn value function directly from experience with temporal difference error, online and in a fully incremental way. It can be seen as a prediction problem. All variant come with a slightly different update rule that originally is:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$$

The following algorithms show the pseudocode of the most famous approaches following an $\epsilon$-greedy approach, in which the trade-off between exploration (search on the state space for unvisited states) and exploitation (use of the so far learned policy to reach the goal and update the value function) is defined by a (often decreasing) threshold in order to prefer exploration at the beginning of the learning process and exploitation at the end for updating correctly the learned state action pairs values.

SARSA, that stands for State Action Reward (next)State (next) Action, is an on-policy control method to find the optimal policy, that evaluates the policy based on samples from the same policy and then refines it greedily wrt. action values. The following Algorithm 2 shows its pseudocode for a tabular version, in which the policy is a sequence of rules for each state.

On the contrary, Q-Learning is an off-policy method to find the optimal policy, where the agent may follow an unrelated behavioural policy. Essentially it learns

**Data:** policy $\pi$ to be evaluated
**Result:** value function V
initialize V arbitrarily;
**for** *each episode* **do**
  initialize state $s$;
  **for** *each step, state s not terminal* **do**
    $a \leftarrow$ action given by $\pi$ for $s$;
    take action $a$, observe $r, s^{'}$;
    $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s^{'}) - V(s)]$;
    $s \leftarrow s^{'}$;
  **end**
**end**

**Algorithm 1:** Classic TD learning

**Result:** action value function $Q$
initialize $Q$ arbitrarily;
**for** *each episode* **do**
  initialize state $s$;
  **for** *each step, state s not terminal* **do**
    $a \leftarrow$ action for $s$ derived by $Q$, e.g. $\epsilon$-greedy;
    take action $a$, observe $r, s^{'}$;
    $a^{'} \leftarrow$ action for $s^{'}$ derived by $Q$, e.g. $\epsilon$-greedy;
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s^{'}, a^{'}) - Q(s,a)]$;
    $s \leftarrow s^{'}$;
    $a \leftarrow a^{'}$;
  **end**
**end**

**Algorithm 2:** Sarsa

action value function with an update rule slightly different from the SARSA one and can be seen in the following Algorithm 3, where it greedily refines the policy wrt. action values by the max operator.

### 2.3.2 Deep Q-Network

All the previously shown methods rely on recording each state of the studied domain with their corresponding Q-values. Despite it being a precise and correct method to build an easy to handle policy, with continuous domains or with just domains with a big state space, tabular methods become quite expensive and slow to learn.

To solve this issue, algorithms can introduce deep neural networks to appproximate any component of reinforcement learning, from value and action functions to policies and models (like state transition function and reward function).
This is called Deep Reinforcement Learning and can essentially be seen as a difference in the way to store the policy and to retrieve actual Q-values.

**Result:** action value function $Q$
initialize $Q$ arbitrarily;
**for** *each episode* **do**
  initialize state $s$;
  **for** *each step, state s not terminal* **do**
    $a \leftarrow$ action for $s$ derived by $Q$, e.g. $\epsilon$-greedy;
    take action $a$, observe $r, s^{'}$;
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s^{'}, a^{'}) - Q(s,a)]$;
    $s \leftarrow s^{'}$;
  **end**
**end**

**Algorithm 3:** Q-Learning

DQN, Deep Q-Networks, are able to approximate the action value function by training on past experience and, important factor, with unrelated samples of transitions as we are still facing a problem described as an MDP.
During the learning process, as the agent keeps interacting with the environment, a experience replay buffer is built with transitions[23] and from that, at every step, a batch of samples is retrieved and a gradient descent step on the model is performed. Another important property of DQN is the introduction of a target network [36] to evaluate the greedy policy according to the online network, but to use the target network to estimate its value.
An example of Deep Q-Learning can be seen in the following Algorithm 4 proposed by Mnih [26].

**Data:** the pixels and the game score
**Result:** $Q$ action value function
initialize $Q$ with random weights $\theta$;
Initialize replay memory $D$;
Initialize target $Q^{'}$ with weights $\theta^{-} = \theta$;
**for** *each episode* **do**
  initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
  **for** *each step, state s not terminal* **do**
    Following $\epsilon$-greedy policy select $a$ = random action or $argmax_a Q(\phi(s_t), a; \theta)$;
    take action $a$, observe $r$ and image $x_{t+1}$;
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$;
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$;
    **if** *episode terminates* **then**
      $y_j = r_j$
    **end**
    **else**
      $y_j = r_j + \gamma max_{a'} Q^{'}(\phi_{j+1}, a^{'}; \theta^{-})$
    **end**
    ;
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ wrt network parameter $\theta$;
    Every $C$ steps reset $Q^{'} = Q$ i.e., set $\theta^{-} = \theta$;
  **end**
**end**

**Algorithm 4:** Deep Q-Network

# Chapter 3

# Related Work

There is a wide literature regarding Reinforcement Learning techniques, benchmarks and many tools and framework have been developed for this sector.
OpenAI's Gymnasium [5] is the most used open source Python library for developing and comparing reinforcement learning algorithms, with a lot of environments built in order to test different types of challenge.
In recent past Reinforcement Learning works make use of QLearning [38], a model-free algorithm in conjunction with Deep Learning, using a neural network to predict q-values. This new methodology called Deep Q-Learning [19] is now the standard way to deal with Reinforcement Learning expecially in large state space environments.

Despite the similarities and complementarities between Reinforcement Learning and Automated Planning are well known in the scientific community, there are not so many works that study the possibility and the effectiveness of using both methods in conjunction to improve the efficiency of known Reinforcement Learning algorithms. Lin's work [24] has been one of the first to propose a combination effort of the two methods, by learning action models from past experiences that can be used for planning and speed up the learning process. Additionally, the term '*planning*' is used to represent different methodologies, settings and context. For example, the term trajectory planning refers to the task of computing a correct trajectory.
In this study, the term planning is referring to the field of Artificial Intelligence which explores the process of using autonomous techniques to solve planning and scheduling problems, with a more detailed description that will follow in the next chapters.
With a similar meaning of planning, MuZero [33] has reached state-of-the-art performances in exploiting Atari, chess and Go games thanks to a trained network that is called at every step of a Tree Search planning method.
From this study, some analysis have been done on the role of planning along with reinforcement learning. Results tell that planning is most useful in learning and that planning alone is not sufficient to drive strong generalizations. [18]
MuZero algorithm has inspired Reanalyse study in computing new targets on existing data point for both offline and online reinforcement learning.
A similar idea is proposed by Hoel at al. [20] using a modification of MonteCarlo Tree Search method which constructs a search tree based on random sampling, with

a neural network that biases the sampling towards the most relevant part of the search tree.
Another example of reasoning has been done in DARLING [22], that generates multiple plans to constraint the behaviour of the agent to reasonable choices in a deterministic domain.
Planning and Reinforcement Learning can be used in combination to exploit long-term reasoning capability of the former that lacks in the latter. This metodology is proposed by several works like Gieselmann et al. [17] that uses planning for long-term decision making in hierarchical reinforcement learning, where agents are trained in parallel for each subgoal and hierarchy level.
Work by Eysenbach et al. [10] propose a similar method in a general purpose control algorithm that decomposes a task in a sequence of subtasks computed by a graph constructed with reinforcement learning, with a goal conditioned value-function that provides nodes and weights taken from a replay buffer.
Walsh et al. [37] make use of sample-based planners to overcome the difficulty of classical planners in dealing with too large state spaces.

# Chapter 4

# Lunar Lander Environment

Now that the background has been described, the presentation of the study can start with a description of the study case environment and its properties that will influence the methodology and some important choices in order to deal with it properly.

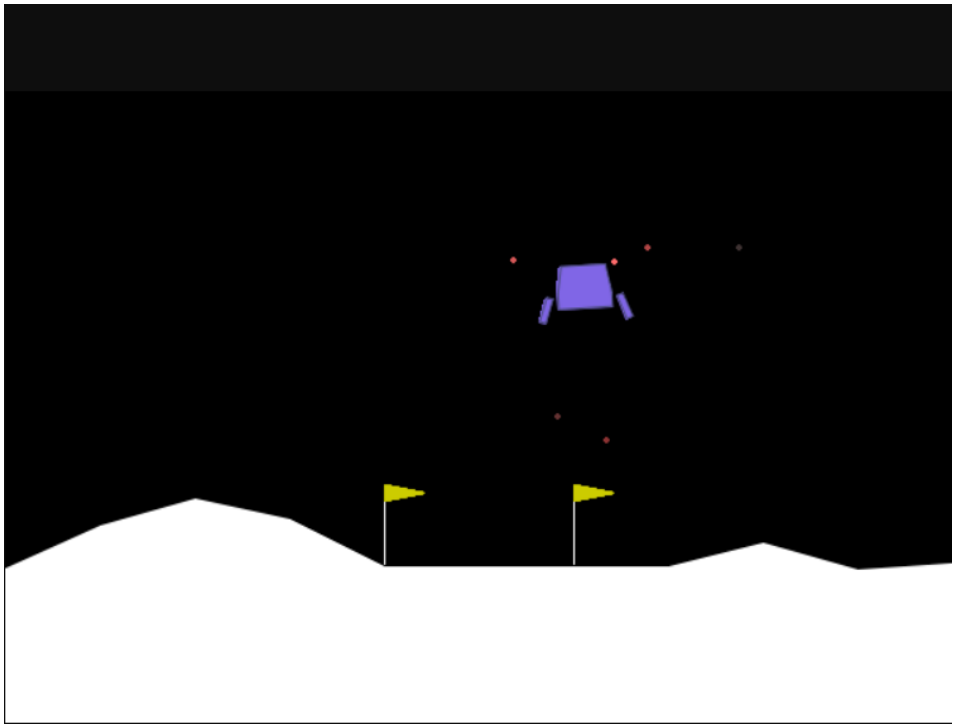

**Figure 4.1.** An example of the Lunar Lander environment in action, the white ground is generated differently at every episode, but the yellow flags are always in the same position and in an plain ground. The red points around the rocket represent the execution of the engines.

The environment selected for research and experiments is the Lunar Lander environment offered by Farams's Gymnasium library [5].
It is a maintained fork of OpenAI's Gym library, an API standard for reinforcement learning with a diverse collection of reference environments.

Lunar Lander is part of the Box2D environment, a 2D physics engine for games [30]. This environment is a classic rocket trajectory optimization problem in which the spacecraft's goal is to safely land on the landing pad by using its 3 engines (a main central one and two side engines) to correct its trajectory.

As shown in Figure 4.1, the environment is divided in a black section that represents the free space, a region in which the rocket can freely fly, and a white section that represents the ground. The ground composition is different in each episode, and can have multiple hills or plains, but the landing pad i.e. the space between the two yellow flags is always at the center of the viewport and composed by plain ground.

## 4.1 Observation/State space

This environment's state space has a relatively small number of state variables, but all of them are continuous except the last two that will be described.
The state comes in an 8-dimensional vector that consists in the coordinates $x$ and $y$, its corresponding linear velocities that we will denote as $v_x$ and $v_y$, its angle with respect to the ground $theta$ as well as the angular velocity denoted as $v_{theta}$.
All these variables are typed as floats and have have their own domain:

- $x = [-1.5, 1.5]$
- $y = [-1.5, 1.5]$
- $v_x = [-5, 5]$
- $v_y = [-5, 5]$
- $theta = [-3.1415927, 3.1415827]$
- $v_{theta} = [-5, 5]$

The last two variables are booleans that represent whether each leg is in contact with the ground.
Given the ground that fills around the 25% of the viewport, not all the state space is accessible and reachable from the rocket (i.e. the agent), almost the entirely negative domain of the coordinate $y$ is unattainable but still defined.
Additionally, the same viewport don't show the entirety of the possible values. Reaching the edges of the viewport doesn't mean reaching the limits of the $x$ or $y$ variables. Sometimes the rocket can move outside the visible area but still in an admissible state, but going too much far away from that will result in reaching a terminal state and in a restart of the environment.
The landing pad is always at coordinates $(0, 0)$

## 4.2 Action space

The action space is quite straightforward. It comprises the possibility of execute 4 actions, defined by simple integer values.
The available actions are:

- 0 = *Idle*: do nothing. The lander doesn't fire any engine, let the gravity do it's job along with the current velocities.
- 1 = *Left engine*: The left sided engine of the rocket is fired, it can adjust the current lateral velocity ($v_x$) as well as the angular one ($v_{theta}$).
- 2 = *Main engine*: The main engine of the rocket is fired, it is positioned at the bottom of the rocket's body and can counterbalance the gravitational acceleration. Consecutive execution of this action can even put the rocket in a positive $v_y$ velocity, making it able to fly up.
- 3 = *Right engine*: Same as action 1 but on the other side.

The use of the sided engine is crucial to keep the engine well orientated and on the right $x$ coordinate. But the execution priority is not so trivial because the angle domain is positive counterclockwise.
This means that to rectify a negative value of $x$ coordinate, the execution of the left side engine would push the rocket to the negative part of the angle ($theta$) coordinate.
And the angle wrt. to the ground is crucial, as an execution of the main engine in an tilted (inclined) positioning of the rocket would not push the rocket up, but will push it along its axis resulting in an increased angular velocities that would make the rocket's position more oblique on the next steps.
The power of the engines is fixed, and it would represent the engine at full power, according to the Pontryagin's maximum principle [21], that says it is optimal to fire the engine at full throttle or turn it off.

## 4.3 Rewards

The reward function, as most of the environments from the Gymnasium library, is already defined at every step.
For each step, the reward:

- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased the slower/faster the lander is moving.
- is decreased the more the lander is tilted.
- is increased by 10 points for each leg that is in contact with the ground.

- is decreased by 0.03 point each frame a side engine is firing.
- is decreased by 0.3 points each frame the main engine is firing.

The total reward of an episode is the sum of the rewards for all the steps within that episode.
The episode receives an additional reward of -100 for crashing on the ground, i.e. the rocket makes contact with the ground with its body. On the contrary, it receives +100 points when safely landing.
An episode is considered solved if at the end its cumulative reward is at least 200 points.
Multiple experiments have shown that the reward function is not so perfect, sometime resulting in very high episode rewards (above 200 points) for trajectories that finish in a rough landing outside the flags. That is because the +100 points for landing are given anyway and the reward given for going down is higher that the penalty for going down rapidly.

## 4.4 Episodes and Seeding

Every episode starts with the rocket at the top center of the viewport.
What is different for each episode, other than the ground profile as previously said, is the random initial force applied to its center of mass.
Despite the initial position and angle are always the same, velocities can have almost any value in their domain at the start of an episode. This is what makes the environment challenging, episode started with high value of some lateral or angular velocities are hard to adjust, as in few steps, if not adjusted in time, the trajectory can become impossible to correct.
An episode finishes if: the lander crashes (the lander body gets in contact with the gounrd), the lander ges outside of the viewport by far, the lander is not awake. According to the Box2D documentation, a body which is not awake is a body which doesn't move and doesn't collide with any other body.

A single specific starting observation can be exploited by manually selecting a seed. In the latest version of Lunar Lander (LunarLander-v2) selecting a seed doesn't mean to select a set, or a predefined sequence of starting observation, but only one.
A model is said to be able to resolve the environment if it is able to achieve an average reward of at least 200 points in a span of 100 episodes, without a specific seed set.
The non-selection is crucial, because for some seed with low if not even zero initial velocities, the learning process could be very short but almost insignificant.
Furthermore, the environment comes with additional options that can be selected and adjusted like the gravity magnitude, the possibility to introduce wind in the environment and modify its power in two distinct ways: wind power to modify the magnitude of linear wind applied to the craft and the turbulence power that indicates the magnitude of rotational wind applied to it.

The creation and use of the environment is quite straightforward thank to the Gymnasium APIs. The environment is created by the command make:

```
import gymnasium as gym
env = gym.make('LunarLander-v2', render_mode='rgb_array')
```

With the *rendermode* parameter it is possible to select a visual rendering of the environment as seen in Figure 4.1 with the value "*human*" or as a vector with the value "*rgbarray*".

In addition, the use of the environment can be restricted in just other two commands, the reset of the environment, in which one could specify a seed as argument:

```
state, info = env.reset(seed=<seed>)
```

The above command resets the environment to the start of an episode and returns the initial observation (*state*).
The last main method to deal with the environment is the step function, that is the main method to execute an action and observe the outcome:

```
obs, r, terminated, truncated, info = env.step(<action>)
```

The *step* method accepts as argument the action, in this case an integer from 0 to 3 that represent the action as described above, and returns the next state (obs, observation), the associated reward (r) and two booleans that represent if the returned state is a terminal one or not.

# Chapter 5

# Discretization and Planning

The first challenge to face in order to be able to generate a plan for every environment, and in particular for the study case of Lunar Lander is the transposition of the environment structure in a planning domain language.
The definition of a planning problem implicitly consists in a domain abstraction, in which symbolic representations of the domain properties and dynamics are defined and does not often stand in a 1 to 1 correspondence.
As previously stated, the handling of continuous variables is not supported by most planners and this require a proper discretization over the state variables with the conseguence of the introduction of non-determinism in the action effects.
This chapter describe the methodology for the discretization and the writing of the environment's dynamics in the PDDL formalism, proposing different adaptions and their challenges.

## 5.1 From continuous to discrete values

The discretization of an domain is quite straightforward.
Each domain variable is divided in intervals/bins that are represented as a single value, in the case of a planning domain in a single fluent/object/constant.
In order to make the planning algorithm work in an admissible time, the number of bins should not be too big, but at the same time, to make the generated plan be consistent and logically correct when used in the original domain, the number of bins should not be too small.
In the Lunar Lander environment there are six continuous state variables and two booleans, so the complexity of the state space is on the order of $O(n^6 \times 2 \times 2)$, where $n$ is the number of discretized values for each state variable.
Thus, even discretizing each state variable into 10 values can produce over 400,000 different states, which is far too large to explore even for a tabular reinforcement learning algorithm and for a planner is almost intractable.

As it will be described in a later section, the computational load of a FOND planner depends not only on the number of states, but also on the number of different outcomes defined in the actions.

After simulations and results, admissible number of bins found can vary between a minimum of 3 bins (ex. $x = -1, 0, 1$) to a maximum of 7 bins (ex. $x = -3, -2, -1, 0, 1, 2, 3$).
All the bins are defined in PDDL domain as typed constants, because they will be present in all the problem of the same domains, as shown in Listing 5.1 and in Listing 5.1

**Listing 5.1.** Types of PDDL domain

```
(:types
    value
    t_value vt_value vx_value vy_value x_value y_value - value
)
```

**Listing 5.2.** Constants definition

```
(:constants t_-1 t_0 t_1 - t_value
    vt_-1 vt_0 vt_1 - vt_value
    vx_-1 vx_0 vx_1 - vx_value
    vy_-2 vy_-1 vy_0 vy_1 - vy_value
    x_-1 x_0 x_1 - x_value
    y_0 y_1 y_2 - y_value
)
```

With this reasoning we can take advance of the Lunar Lander structure. Given that the lander cannot go underground and so cannot result in negative values for the $y$ coordinate, it is fair to create bins on just the positive half of the $y$ domain.

With the same logic, vertical velocity $v_y$ can be defined just for the negative values, because the lander will quite always go downway. At best conditions just a single bin for positive values can be defined.
In Listing 5.1 it is shown the most compact way to represent all the environment's variable taking advantage of the Lunar Lander structure. The $y$ coordinate are defined over only the positive values (with the 0 value), and the $v_y$ velocity can assume integer values from -2 to 1.

With the state variables defined, the domain needs the logic to handle those values. The whole logic in a PDDL domain, as well as in other planning formalisms, is defined with predicates.
In a discretized domain, the first logic to implement is the contiguity of bins, to make the planner able to switch from a value to the next/previous one.
So, two main predicates are defined: *(next ?n ?v - value)* and *(prev ?p ?v - value)* predicates.
These two predicates are responsible to define contiguity over the values and can be used by all different variables, this is the reason for the hierarchical typing in the Listing 5.1, where all the state variables types are subtypes of the type *value*. the syntax is quite simple, the first argument value is the next/previous value of the second argument.

Another important logic to define is the sign of velocities. As will be described in the action section, the physics of the environment, as simulation of the reality, changes the coordinate values according to the velocities so, in order to change the coordinate bins, the model should know in which direction to go.
For this purpose, two predicates are defined as *(positive ?v - value)* and *(negative ?v - value)*, that assign a sign to a value object.

The last logic to implement is the representation of the actual current state. that would correspond to the observation of the environment returned by the *step* function in the Gymnasium APIs.
A predicate is then defined for each state variables with the corresponding type as expected.
They are written as *(current<type> ?v - <type>value)*.
An example over all the described predicated is shown in the Listing 5.1
The last two domain variables, the two booleans that represent the contact with the ground by the two rocket's leg, can be omitted from the planning domain definition.

**Listing 5.3.** Predicates representation for the Lunar Lander environment.

```
(:predicates
    ;current state definition x,y = cartesian position
    ;t(theta)=angle wrt x axis
    ;next and prev predicates used to connect domain values
    ;ex. from x_10 and x_9 -> (x_prev x_9 x_10) and (x_next x_10 x_9)
    (current_t ?t - t_value)
    (current_vt ?vt - vt_value)
    (current_vx ?vx - vx_value)
    (current_vy ?vy - vy_value)
    (current_x ?x - x_value)
    (current_y ?y - y_value)
    ;predicates to define relations between constants
    (negative ?v - value)
    (next ?n ?v - value)
    (positive ?v - value)
    (prev ?p ?v - value)
)
```

## 5.2 Actions

The behaviour of the environment, i.e. the dynamics, that in this specific case correspond to the physics, is defined by the actions.
As the aim of the whole planning phase is to bring the result (the planned policy) as a support for the reinforcement learning process at a sample level, the action space cannot be modified or abstracted. The policy should tell which exact action to execute for each state defined in it.
For that reason, the actions defined in the PDDL domains are the same available in

the original environment: idle, left engine, main engine and right engine.
Here is where the non-determinism is expressed with the use of the keyword *(oneof)* in the effects section, that enables to specify multiple mutually exclusive outcomes.

The actions can be formulated in multiple ways, from the one most faithful to the real physics dynamics to the one that doesn't even consider the velocities, and all of those versions have been defined and tested to assess the both the planning time computation and also the result of the resulting policies in the real environment.
For the remaining description of the actions, the most faithful to the physics will be used as example in order to describe the whole complexity and all the possibilities of abstracting in the planning PDDL domain.
The main effect of each action is the (possible) change of current values for some velocities, while the (again possible) change of bins for the coordinates is possible in all action as effect of the current velocities, as it is in the real world.
Let's take as example the *Idle* action as shown in the following Listings.

**Listing 5.4.** Idle action parameters in the most complete dynamics domain.

```
:parameters (?y_prev ?y ?y_next - y_value
    ?vy_prev ?vy - vy_value ?vx - vx_value ?vt - vt_value
    ?x_prev ?x ?x_next - x_value ?t_prev ?t ?t_next - t_value)
```

As shown in the Listing 5.2 above, parameters are numerous and they always comprehend the actual current value with its contiguous values for the coordinates $x, y, t$, where $t$ stands for *theta*. And the velocities comes with just the current values except for the variable that could be actually modified by this action.
In this particular example, the *Idle* action lets the gravity do its work so the only velocity that could change is the $v_y$ one, and precisely it could only go towards the negative part (as the coordinates are negative towards the bottom of the environment).

**Listing 5.5.** Idle action precondition in the same setting as above.

```
:precondition (and
        (current_y ?y) (current_vy ?vy) (current_t ?t)
        (current_vt ?vt) (current_x ?x) (current_vx ?vx)
        (prev ?y_prev ?y)
        (next ?y_next ?y)
        (prev ?vy_prev ?vy)
        (prev ?x_prev ?x)
        (next ?x_next ?x)
        (prev ?t_prev ?t)
        (next ?t_next ?t)
    )
```

In the above Listing 5.2 the preconditions of the *Idle* action are shown.
In this section the meaning of the before mentioned parameters is defined. All the current state variables are defined and the contiguity of all the coordinate values and for the only $v_y$ velocity are ensured.

The most interesting part of the action is the effects section in which the possible actions are defined.

**Listing 5.6.** Idle action effects.

```
:effect (and
        (oneof (and (not (current_vy ?vy)) (current_vy ?vy_prev)) (and))
        (when
            (positive ?vy)
            (oneof
                (and (not (current_y ?y)) (current_y ?y_next)) (and)
            )
        )
        (when
            (negative ?vy)
            (oneof
                (and (not (current_y ?y)) (current_y ?y_prev)) (and)
            )
        )
        (when
            (positive ?vx)
            (oneof (and (not (current_x ?x)) (current_x ?x_next)) (and)
            )
        )
        (when
            (negative ?vx)
            (oneof (and (not (current_x ?x)) (current_x ?x_prev)) (and)
            )
        )
        (when
            (positive ?vt)
            (oneof (and (not (current_t ?t)) (current_t ?t_next)) (and)
            )
        )
        (when
            (negative ?vt)
            (oneof (and (not (current_t ?t)) (current_t ?t_prev)) (and)
            )
        )
    )
```

As shown in the Listing 5.2, the effects are numerous and consider all variables in the domain.
The logic is quite straightforward: The first effect is the actual contribution of the *Idle* action, the possibility to change bin for the $v_y$ velocity.
The other effects are consequences of the environment physics: for each coordinate let's say $x$, there is a check on its corresponding velocity, let's say $v_x$, with the keyword *when*. This is called a conditional effects: if the first proposition if of true

value, the next proposition is applied as an effect.
Going on with the example of $x$ and $v_x$, if the velocity $v_x$ is positive, then the current coordinate $x$ may change bin in favor for the next one, on the contrary, if negative, the new bin may be the previous one.
It is important to say that it may change, this is what is called non-determinism in this domain. The keyword *(oneof (<effect 1>) (effect 2))* tells that an action can have one of the two defined effects, and the keyword *(and)* is the atom that contains no proposition and means that nothing changes. This last outcome would certainly correspond to a change of the value in the real environment, but in the discretized domain, the new value will still be in the same bin as the previous one.

The other three action follow the same logic, with the same velocity check and the possible corresponding outcome for all the coordinates and with a different possible change of the velocities that the action directly affect.

## 5.3 Problem Definition

Until now, only the domain part of a planning task has been described. What is still missing is the definition of the goal.
In PDDL formalism, this is done in a different file that will actually define the problem. This is a common way to make the defined domain able to be used with different tasks/problems.

The problem file has to describe the initial state, with all the initial True predicates, and the desired goal state to reach, expressed again in the desired final True predicates.
Having modeled the contiguity and sign of velocities as predicates in the domain, this is the right place to actually define them.
This will result in expressing all the contiguity predicates for the defined constants as described in the example of the Listing 5.3.

**Listing 5.7.** Example of contiguity construction in the problem file.

```
(prev y_0 y_0) (prev y_0 y_1) (prev y_1 y_2)
```

The first predicate in the example above is needed due to the definition of the actions, when is mandatory to define in some action a previous value even if it is the lower one.
In the same manner, the sign of velocities is defined with the corresponding predicates: *(negative ?v - value)* and *(positive ?v - value)*
The last values to define in the initial state are the actual current values of the state variables. Again, this is done by using the corresponding predicates as shown in Listing 5.3.

**Listing 5.8.** Initial state variables values for the planning problem.

```
(current_x x_-1) (current_y y_2) (current_t t_-1)
```

```
(current_vx vx_-1) (current_vy vy_-1) (current_vt vt_1)
```

The last definition needed in the problem file, is the goal state, this will be defined as the desired current state variables values using the same predicates in the Listing 5.3 but with the final desired values. For the Lunar Lander problem, to represent the landing the value of $y$ coordinate will be the constant $y_0$ and the one for the $x$ coordinate will be $x_0$ as well.

## 5.4 Abstracted Dynamics

In the previous section an example of how to represent at best the real world physics has been shown. The *Idle* action is composed by a high number of variables and has a high number of possible effects, and this is the case for the other three action as well.

During planning, this high number of different outcomes means that each state would have numerous successors.
Furthermore, the logic of FOND planners is to "determinize" the model, that means to generate multiple deterministic actions from a non-deterministic one.
This would generate lot of new actions that would enlarge the action space too much and will result in an intractable domain for the planner.
The effects of the *Idle* action contains 7 *(oneof)* statements with two possible outcomes, that would mean to generate 128 deterministic actions from just the mentioned action. In addition, sided engine actions contain an additional statement that would result in 265 new deterministic actions.

The experiments done with the domain in which the physics are completely modeled, that has also the most possible minimal discretization (in order to reduce as possible the domain size), resulted in the impossibility of finding a strong cyclic plan with a required time for just instantiating the problem that exceeds the minute, while the extensive search for the strong cyclic plan, as described in Chapter 2, could continue indefinitely for hours without an end.
Just this time issue could be a problem in our scenario. In order to be a valid support for the reinforcement learning process, the time required for generating a policy should be very short if not almost instantaneous and should not extend too much the total time of the whole process.

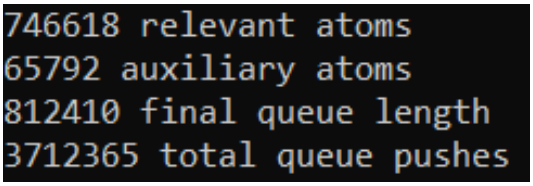


**Figure 5.1.** A screenshot showing the size of the instantiated planning problem for the described domain with the whole physics dynamics modeled. The numbers describe the total amount of rules, determinized actions, state variables and possible propositions, all encoded in $SAS^+$ formalism with boolean variables and clauses.

```
Dumping the policy and fsaps...
Plan found, but not strong cyclic.
Peak memory: 913076 KB
```

**Figure 5.2.** The encountered output message at the end of all planning runs of a too big and complex domain that runs out of time.

## 5.5 Single Plan Limitations

In addition to the challenge of defining a domain that can be easily handled from the planner and that outputs a strong cyclic plan, another challenge is to obtain a policy that can behave promisingly in the actual Gymnasium environment.
The planned policy is not expected to solve the environment of course, if so it would not be necessary to even use reinforcement learning after. But the resulting policy should at least perform better than the random agent (agent that returns always a random action), and that can have a promising behaviour in the environment by a visual inspection.

Inspecting a policy computed by the planning in some experiments, it can be seen that the FOND planner doesn't explore the whole state space defined in the domain, but only the relevant states for the current problem, i.e. the state that will certainly occur in the trajectory from the initial state to the goal one. This is of course an expected behaviour of the planner [28, 27].
But at the same time, this is an important limitation as the policy, in order to solve the environment that can have multiple different initial states, should be able to generalize the task.

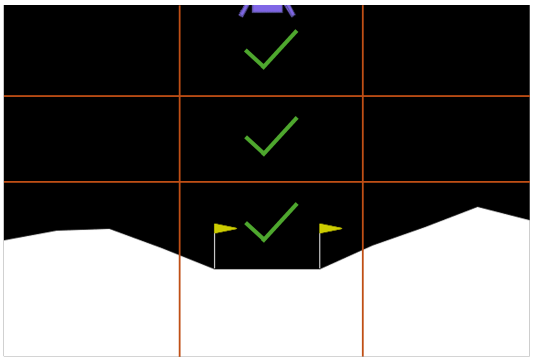

**Figure 5.3.** Example of state space explored by a single plan, with initial state as $x = 0$ and $y = 2$.

As it can be seen in Figures 5.4 and 5.3, planning will only plan and explore for relevant states according to the current problem the planner is fed with.
The example in the figures show only a projection of $x$ and $y$ coordinate for a better understanding, but the same limited exploration is done for *theta* angle too, and also for the velocities variables. All the images refer to a domain with the minimum number of bins. In addition, that is not the real mapping between the Lunar Lander

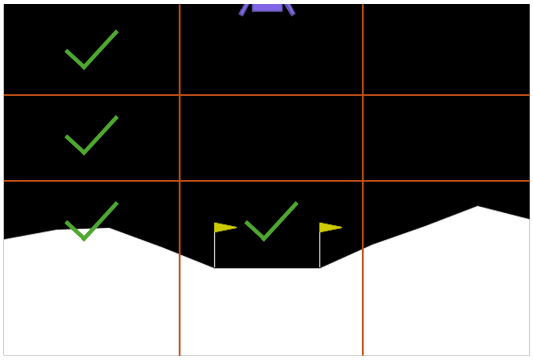

**Figure 5.4.** Example of state space explored by a single plan, with initial state as $x = -1$ and $y = 2$.

environment and the PDDL bins, in the figures the lines are "equally" distributed for a better visualization.
The used mapping between all the domains bins and the real environment follows the same reasoning for all the tested domains that will be described in a later section that is: all the coordinates that are far from the center (the goal values) can be generalized into one single state, while coordinates near the center are modeled with more precision.
The approach is showed in Figure 5.5, with an example for the $x$ coordinate discretization. The same discretization is also used for all the other state variables.

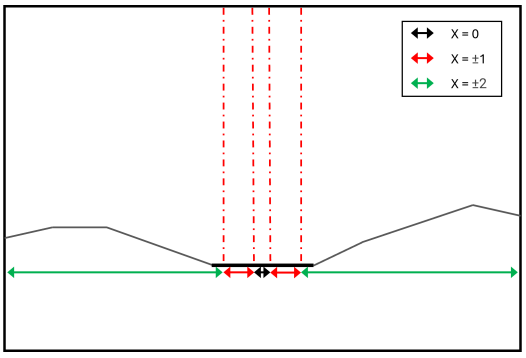


**Figure 5.5.** An example of the mapping between discretized domain and the actual environment for the $x$ coordinate. Values at the center are more important for guiding the trajectory corrections while going to the sides it can be more generalized.

## 5.6 Expanded Planning

This restricted state space search of a single planning run make any defined domain almost useless for the real environment because more than the 50% of the times the policy will have no rule defined for the current state, and as a consequence no policy will be of any support for the reinforcement learning process.

The only reasonable and simple workaround to expand the explored state is an

iterative planning on the same domain with different starting states that will try to cover most of the actual state space.
The approach is straightforward as the idea is: multiple problems with all the possible combination of the initial state are sequentially fed to the planner and the computed policies are all merged into one, without recording duplicate states.

The need of planning multiple times to obtain a policy with the most possible explored state space, is another important factor in the reasoning of the domain definition along with its inner complexity that has been discussed previously.
The whole planning process should not take too much time. It should be almost irrelevant compared to the time needed for the reinforcement learning process.
The next section will describe some domains tested with their properties and their performances.

## 5.7 Versions

Due to the planners limits and the complexity of the complete domain, various modification of the latter has been done in order to find the better trade-off between computational load, correctness of the dynamics and promising performances when simulating the planned policy in the real Gymnasium environment.
A list of developed and tested versions of the domain will follow.

- **No Velocity**: This is the highest abstraction trial for the environment, as the name says, the velocities are not modeled. This resulted in a very low computational load for the planner, as the remaining action effects are just few. But on the other hand the computed policy is very inconsistent and poorly performs in the real environment, failing every time from a reward point of view and from a visual one, even with a higher number of bins.

- **Mini**: This domain is the one already mentioned that models the physics dynamics in the most complete way. The name comes from the selection of the bins, with the lowest number of them in order to make the model as light as possible. But as previously stated, it's not enough to generate a strong cyclic policy. For this reason, and also for the long time to only instantiate the problem during the planning process, it will be left aside.

- **No_x**: Experiments show that the most compromising situation for an episode is when the rocket becomes too much tilted that also the main engine firing contribute to the rotational movement. This is the reason why, during the decision of what to remove from the domain definition to make it lighter for the planner, this domain has been defined. The removal of $x$ coordinate from the domain seemed a more reasonable choice wrt. the *theta* removal.
  In this setting, with just $y$ and *theta* coordinates defined (with their respective velocities), the lander should learn how to land with the right orientation. And this is exactly what it does, when using the (expanded) plan in simulation in the Gymnasium environments.
  When the starting horizontal velocity is near zero, it also succeeds. but in all

the other cases it lands while shifting horizontally.
The time needed for the extended planning is less than 4 minutes.

- **No_theta**: This is a similar domain as the previous one, but with the $x$ coordinate in place of the *theta* one. The complexity is similar to the previous one, but the behaviour is worse as the rocket always becomes tilted and miserably fails.

- **Combined**: After many experiments and trials on how to introduce the reasoning over the $x$ coordinate starting from the most promising domain, *No x*, the solution was to create multiple domains in a "Divide et Impera" way.
  Three different domains have been defined with each one modeling just one coordinate (with its respective velocity), so a domain to model the $x$ coordinate (*"x_only"*), one for the $y$ coordinate (*"y_only"*) and the last one for the *theta* coordinate (*"t_only"*).
  These three domains are planned with the extended method as explained before, but the policies remain separate.
  At simulation time, the three policies are queried and the most suggested action is selected.
  These domains are very light and the whole process of extended planning of all the three domains requires only a minute or two.
  The results in the simulation are remarkable, it often lands in a quite correct way, except for episodes starting with high velocities in $x$ and *theta*, but this model is still the most promising and the one with the higher average reward in the Gymnasium environment.

**Table 5.1.** Table for domain comparison on planning and performance on environment simulation.

| Domain | Planning time | Strong Cyclic | Avg. Reward | Visual Behaviour |
|---|---|---|---|---|
| **No vel.** | $< 1min$ | Yes | -300 | Very bad |
| **Mini** | $\infty$ | No | Irrelevant | Good until explored |
| **No_x** | $< 4min$ | Yes | -160 | Sometime good |
| **No_t** | $< 4min$ | Yes | -480 | Worst |
| **Combined** | $< 2min$ | Yes | **1.4** | **Very good** |
| **Random** | // | // | -180 | Very bad |

The Table 5.7 summarizes the performances of the tested domains.
The last row is not a plan, but its just the random agent taken as a baseline for the average reward comparison.

The planning time refers to the total time of extended planning described in Section 5.6.
Average rewards are very low values, this is expected as positive values would mean that the domain is able to solve the environment on more than 50% of the time. For

that reason the *Combined* domains is the most promising and will be selected for the integration as support in the reinforcement learning described in the next chapter.

# Chapter 6

# Planning integration in RL Methods

This section will describe the different ways of integration of the planned policies as support for the reinforcement learning training.
Reinforcement learning's used methods are tabular Q-Learning, that has been used as a basis for the research, and Deep Q-Learning.
The proposed integration methods are similar for the two variants of reinforcement learning training.

## 6.1 Tabular Reinforcement Learning

As described in the Background chapter, tabular reinforcement learning involves the use of a "*table*" to record each state with its associated action values.
The process is the same as described in Algorithms 3 and/or 2, differentiating in the update rule.

Being that this is not an approximation function method, but a recording of rules for each state, we need to define a set of different states derived from the actual environment.
In order to deal with the continuous environment of Lunar Lander, a discretization of the state variables is needed as in the planning process described previously.

The *table* used is a python *defaultdict* from the collections library. It is the best choice as it enables to not raise the KeyError exception when searching for a key not already in the dictionary. that is common when exploring.
In that situation, the defaultdict object permits to immediately create a default value for the missing key and add the pair in the dictionary. In this scenario, the key corresponds to the discretized state while the value is the vector of action values, initialized with a default value of zero.

The baseline method used is the classic Q-Learning algorithm described in the Algorithm 3.

### 6.1.1 Initialization Method

The *table* used in the Q-Learning method described in the previous section shares a similar structure with the planned policy from the PDDL domain.
Both are based on recording a specific state and the relative value.
The planned output, after appropriate reconstruction of the state variables from the output generated in $SAS^+$ formalism, is composed by a list of *rules* with each one that corresponds to a specific state of the planned discretized domain, and the action planned to be executed in that same state.

This policy is built using the defined *Rule* class, and aggregated in a list saved using pickle and json formats.
The *table* used for the reinforcement learning process is a *defaultdict* as previously said, and the only real difference is that it doesn't explicitly tell the action to execute at a given state, but instead it stores the action value, called Q-value, for each action. It is trivial to derive that the action to execute is the one with the highest Q-value.

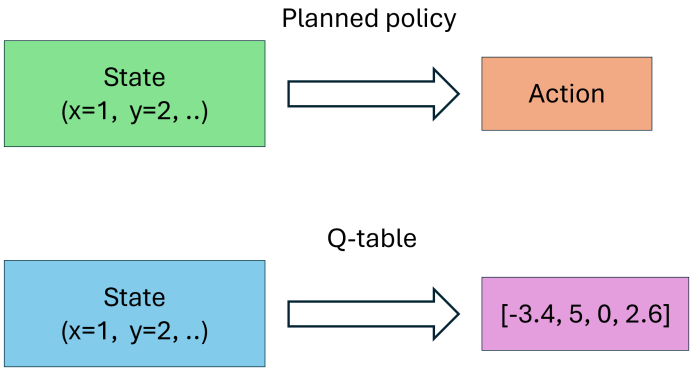


**Figure 6.1.** Graphic example of "value" difference from the planned policy, that tells what action to execute, and the Q-table, that stores each action's Q-value.

This similar structure suggests the idea to use the planned policy as a starting dictionary for the reinforcement learning's table.
Before the start of the training loop, the planned policy is loaded, each rule is formatted to be a key according to the dict formalism and stored into the Q-table. What is left is the Q-values vector. The planned policy actually doesn't know anything about them, but only that one particular action, the one it suggests to execute, should have a positive value as it should be the best to lead the agent to the goal.
So, the resulting Q-value vector is a vector with all zeros, as the default value of the *defaultdict*, and an arbitrary positive value for the action suggested by the planned policy.

$$\forall s \in S,\ Qv_s[i] = \left\{ \begin{array}{ll} c & if\ \pi(s) = i \\ 0 & otherwise \end{array} \right\}\ with\ c > 0. \tag{6.1}$$

The Equation 6.1.1 tells how the Q-values are initialized, with $Qv$ that is the Q-values vector, $s$ a state in the state space $S$ and $\pi$ the planned policy.

After this initialization, the standard Q-Learning algorithm can begin, using the just initialized Q-table with no additional modifications.

### 6.1.2 During Training

Another way to integrate the planned policy into the reinforcement learning training is to use it at learning time, inside the training loop.

As already explained in the background chapter, the Q-Learning algorithm chooses how to interact with the environment between two different ways: exploiting the policy it has learned at that time or exploring for new state taking a random action from the ones available.

As the most promising policy planned reaches positive values in simulations on the actual environment, although low values, the same values are reached from the training agent around half the training process.
The idea is to use the planned actions in the explore-exploit choice, in order to guide the agent towards the "good" actions and learn in the first episodes of the run the Q-values of relevant states.

This could be concretely done in some different ways.

- **Replacing exploration** A first one is by replacing the exploration choice with the "*exploiting*" of the planned policy in an arbitrary number of starting episode. By doing that, the agent could still exploit itself and, when not, the planned policy could suggests some better options.
  The action suggested by the planned policy are said to be better because at the start, the training agent is ignorant about the environment, while the planned policy is surely more informed.
  This modification of the Q-Learning algorithm is used for some arbitrary number of initial episodes to then continue with the classical algorithm version.

- **Replacing exploitation** Another option is to replace not the exploration this time, but the agent exploitation instead. As the agent is ignorant at the start of the training process, in this way the agent could already see from the start an alternation of good actions (from exploiting the planned policy) and exploration.
  This replacement is done for some arbitrary number of initial episodes to then continue with the classical algorithm version.

- **Double-Epsilon** An alternative and a sort of middle-way solution between the previous two. Instead of having two choices of action selection, the planned policy is integrated as a third choice.
  The three options enables to explore and exploit the learned agent while also receiving some hints from the planned policy.
  These three options last only for the first episodes as the previous methods, in order to let the training finish as the classic Q-Learning.

- **Pre-planning** This could be seen as a transposition of the initialization method described in the previous section during training time, without recording an arbitrary Q-value for the action suggested by the planned policy but instead already retrieving the Q-values from interacting with the environment.
  For some arbitrary number of initial episode, the choice between exploration and exploitation during the action selection is replaced with only a query to the planned policy.
  After these initial episodes, the training continues with classical Q-Learning reinforcement learning.

**Data:** state $s$, threshold $\epsilon$, initial episodes $first\_iters$, current episode $ep$
$random \leftarrow$ random number in [0,1];
**if** $random > \epsilon$ **then**
  $action \leftarrow exploit\_agent(state)$;
**end**
**else**
  **if** $ep < first\_iters$ **then**
    $action \leftarrow action\_from\_plan(s)$;
  **end**
  **else**
    $action \leftarrow$ random number in {0,1,2,3};
  **end**
**end**
**return** $action$;

**Algorithm 5:** Action selection in replacing exploration

**Data:** state $s$, threshold $\epsilon$, initial episodes $first\_iters$, current episode $ep$
$random \leftarrow$ random number in [0,1];
**if** $random > \epsilon$ **then**
  **if** $ep < first\_iters$ **then**
    $action \leftarrow action\_from\_plan(s)$;
  **end**
  **else**
    $action \leftarrow exploit\_agent(state)$;
  **end**
**end**
**else**
  $action \leftarrow$ random number in {0,1,2,3};
**end**
**return** *action*;

**Algorithm 6:** Action selection in replacing exploitation

**Data:** state $s$, first threshold $\epsilon_1$, second threshold $\epsilon_2$ with $\epsilon_1 > \epsilon_2$, initial episodes $first\_iters$, current episode $ep$
$random \leftarrow$ random number in [0,1];
**if** $random > \epsilon_1$ **then**
  $action \leftarrow exploit\_agent(state)$;
**end**
**else if** $random > \epsilon_2$ **then**
  $action \leftarrow action\_from\_plan(s)$;
**end**
**else**
  $action \leftarrow$ random number in {0,1,2,3};
**end**
**return** *action*;

**Algorithm 7:** Action selection with two epsilons

## 6.2 Deep Q-Learning

Methods proposed for integration of the planned policy in the Deep Q-Learning training follow the same idea of the integration in tabular reinforcement learning. The next sections will describe how the same ideas have been adapted to be compatible with a neural network, especially with the initialization method.
Pytorch is the selected library used to build and train the neural network involved.

### 6.2.1 Pre-training the model

This is where there is the biggest difference between tabular and Deep Q-Learning. Even if the input and the output of the network is the same as the *table* used in tabular learning, state as input and action Q-values as output, the inner structure of a neural network makes it impossible to directly change its predicted values like it has been described in the initialization method.

What can be done to replicate a similar behaviour, that is to prepare at the best possible the agent for the classical Deep Q-Learning, is to train the network before the reinforcement learning loop.

As in the tabular method, there is a difference between the planned policy output and the one expected from the network.
In order to do so, data is needed to train the network. Samples for the training are retrieved from the planned policy itself.
At each training iteration, a batch of random rules is chosen as input for the network, and the corresponding labels are built as done in the initialization method described in Equation 6.1.1, as it is impossible to know a priori the correct Q-values for each action in each state.

After this process, the network should have learned some hints for the environment and can then be used in the classical Deep Q-Learning algorithm.

### 6.2.2 During training

The other way to use the planned policy as a support for reinforcement learning training is to use it during the actual training loop.
The methods described here are the same exact transpositions of the ideas proposed for the tabular version of Q-Learning, three ways to make use of the good hints of the planned policy during the action selection at each step.

- **Replacing exploration** When rolling a random number to decide whether explore or exploit the agent, the planned policy is queried instead of the exploration choice.
  This modification of the Q-Learning algorithm is used for some arbitrary number of initial episodes to then continue with the classical algorithm version.

- **Replacing exploitation** At the same time as the previous one, the choice between exploration and exploitation is replaced with a choice between exploration and planned policy exploitation.
  This replacement is done for some arbitrary number of initial episodes to then continue with the classical algorithm version.

- **Double-Epsilon** The available choices at selection times become three, exploration, agent exploitation and planned policy exploitation.
  These three options last only for the first episodes as the previous methods, in order to let the training finish as the classic Q-Learning, with only the choice of exploitation and a decaying probability of exploration.

- **Pre-planning** As in the tabular version, the action selection choice between random choice or exploiting the learning agent, is replaced by retrieving the

action from the planned policy.
This is done for some arbitrary number of initial episodes.

# Chapter 7

# Results

## 7.1 Planned Policy

Planning a reliable policy in an acceptable time is the first cornerstone to make the integration of it in the reinforcement learning process.
Given that the total time of Q-Learning for Gymnasium's Lunar Lander can vary from 30 minutes to 1 hour and a half, depending on if using tabular or DQN, or if using early stopping at some points, the admissible time for computing the planned policy with the extended planning described in Section 5.6 is set to a maximum of 5 minutes.
Fortunately, except for the domain that defined the physics dynamics at its best (that requires too much time), all other domains require a lower total time than the selected threshold.
The Table 5.7 summarizes these values.
In addition, a planned policy should also bring a considerable knowledge of the environment, to be useful for reinforcement learning, otherwise it would be useless.

To determine whether and how much a planned policy is useful, the same test of the trained agent is performed. A simulation over 100 episodes in the environment is done using only the policy to guide the rocket. Rewards of each episode are recorder and then the average is computed.
An agent is said to be able to solve the environment if it reaches an average reward over 100 episodes of at least 200 points.
Although it is not expected the planned policy to solve the environment, it is required that a "*good*" one would at least achieve a better average reward than the baseline. The baseline for this comparison is a random policy/agent, where at each step an action is randomly selected. This could be also seen as the starting agent of the Q-Learning algorithm, where at the first episode the Q-table contains all zeros for each action if in tabular settings, and where the neural network has its parameters initialized at random values.

As reported in Table 7.1, the random agent achieves an average reward of -180 points. Two planned domains are able to outmatch it: the **No_x** domain and the **Combined** one.

**Table 7.1.** Table for domain comparison on planning and performance on environment simulation.

| Domain | Planning time | Avg. Reward | Visual Behaviour |
|---|---|---|---|
| **No_x** | $< 4min$ | -160 | Sometime good |
| **Combined** | $< 2min$ | **1.4** | **Very good** |
| **Random** | // | -180 | Very bad |

As the **Combined** domain outperforms the other one by a large amount, it will be selected as planned policy to integrate in the reinforcement learning process.

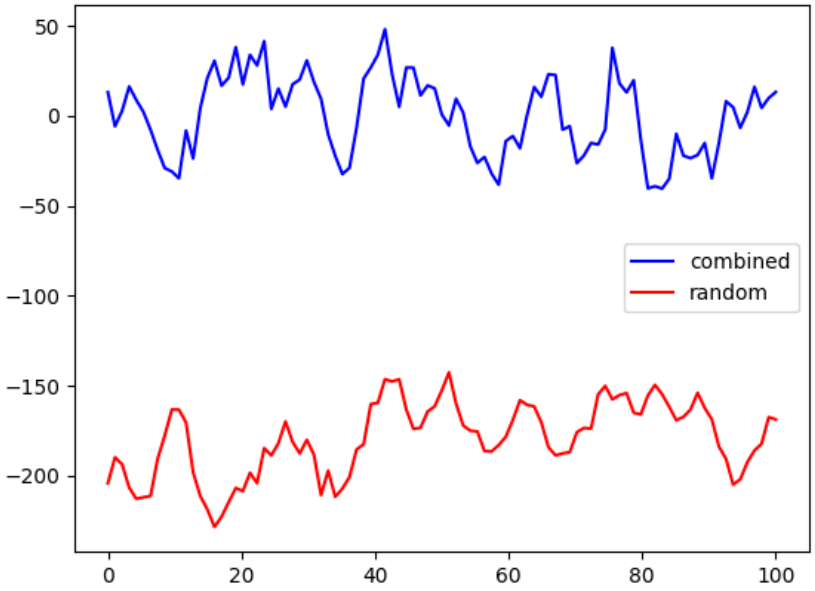


**Figure 7.1.** Graphic representation of simulation rewards by the baseline random agent and the combined planned domain. The plot doesn't represent each single reward, it is a moving average in order to make it easily readable.

As it is confirmed in Figure 7.1, the **Combined** domain' reward is way higher than the random agent.
Knowing that the goal is to reach 200 points, it would be at middle-way from it, compared to the baseline.

## 7.2 Tabular Q-Learning

Tabular reinforcement learning is a simpler but also weaker method in comparison with its Deep Learning version, it is known of its limits in facing large and complex environments.

This is also proved for the Lunar Lander environment, where the results are poor and not so relevant. The algorithm struggles in learning how to act in the whole environment, being capable of achieving the goal only for the easiest starting points, i.e. where initial velocities have values near zero and the rocket is moving only downwards.
Runs of training with no specified seeds in the environment, so with all the possible initial states, results in all failures. The algorithm never reaches an average reward of 200 point over 100 episodes. All the runs have been done with a maximum of 10000 episodes.
This is also the case of all the proposed methods. Multiple runs of each proposed methods have been done from scratch and none of them achieves the target average reward in less than the maximum number of episodes. All runs terminate after reaching the maximum number of episodes.
The maximum number of episodes for each method including the baseline has been tested equal to 10.000 and then 20.000 with the same results.

**Table 7.2.** Table showing the average results of agents trained in all tabular methods.

| Method | Avg. Reward |
|---|---|
| **Baseline** | -100 |
| **Replaced Exploration** | -140 |
| **Replaced Exploitation** | -120 |
| **Double Epsilon** | -105 |
| **Pre-planning** | -142 |
| **Initialization** | **-50** |

The Table 7.2 above, show the average results over 100 episodes of simulation for each method. All values are negative, indicating that the maximum number of episodes is not enough for the tabular version of Q-Learning to learn the environment. The worse results of the proposed methods compared to the baseline can be explained by the amount exploration reduced in the algorithm, a quantity that even in the baseline is not enough.
In addition, the replaced exploitation results can be explained with the same reasoning but talking about the reduced exploitation and resulting update of the learned policy.

The initialization method fails as well in solving the environment, despite an

initial table that is able to achieve some successful episode (the performances inherited from the planned policy).
Nevertheless, it is able to achieve the better performances after 10.000 trained episodes.

## 7.3 Deep Q-Learning

Deep Q-Learning has become the best way to deploy reinforcement learning algorithms, achieving the best results mostly in continuous or complex environments, as this one.
And, given this method as the state-of-the-art for this kind of problems, this is the most relevant method to test the proposed integrations.

As baseline, a simple feed forward neural network is defined with 3 layers, an input dimension of the state space size, 8, an output dimension of the number of actions, 4, and hidden dimension of 128. ReLU has been selected as activation function.
The network's output is to predict the Q-values of each function.
The batch size used is of 64 and learning rate is $1e-4$. The action selection is done by an epsilon greedy policy as described in the Background chapter. The discount factor is set to 0.99 and the replay memory buffer for the experience replay is set to a maximum size of 10000.

The same network with the same hyperparameters will be used also for all the other methods in order to make fair comparisons.
The maximum number of episode is set to 1000 with an early stopping mechanism that, each 10 training episodes, simulates the model for 100 episodes and, if it reaches an average value of 200 points (i.e. solves the environment), the training stops and the model is saved.
The early stopping mechanism is developed for all the other methods for comparison as well.
This is done because the sample efficiency and comparison with the baseline method, for the different methods, can be evaluated in the number of episodes needed to train an agent able to solve the environment.

In order to make a comparison as fair as possible, multiple training runs are done for each method, even the baseline, and the average number of steps needed is computed.
This is crucial to be done, because the Q-Learning with epsilon-greedy policy relies on random selection of exploration and exploitation and this results in different number of episodes needed in different runs.

The Table 7.3 shows, for each method, the average number of episodes needed by the training process to solve the environment (again, achieving an average reward of at least 200 points over a span of 100 episodes.) and its standard deviation over multiple runs from scratch.

**Table 7.3.** Table for comparison of average episodes needed for training an agent able to solve the environment, i.e. reach an average reward of 200 points over a span of 100 episodes., the last column represents the standard deviation of the training runs.

| Method | Avg. Ep. | Std. Dev. |
|---|---|---|
| **Baseline** | 700 | 166 |
| **Replaced Exploration** | **439** | **150** |
| **Replaced Exploitation** | 670 | 242 |
| **Double Epsilon** | 600 | 198 |
| **Pre-planning** | 655 | 186 |
| **Pre-Training** | 730 | 200 |

The results tell that the integration of the planned policy in the training process can eventually speed up the learning.

While all the proposed methods give a little improvement compared to the baseline, the **Replaced Exploration** one have a great impact, reducing the average number of episodes needed to solve the environment by 38%.
Other methods achieves little improvements as shown in Table 7.3.
It is to be considered that for all the methods, including the baseline method, the standard deviation of all the considered test runs is quite high, but the performances of the **Replaced Exploration** method show also a slightly reduced standard deviation.
In general, this high value of standard deviation for all method is to be explained by the high dependency of Q-Learning method on the randomness involved in it, that is present in the action selection and in the retrieving of samples from the memory for the experience replay (i.e. the training of the neural network).

The Figure 7.2 show a pointwise average of all the methods learning curves. **Replaced Exploration** confirms its best results in terms of early learning and termination. Pre Planning and Replaced Exploitation, despite its initial good values - that comes from the initial exploitation of the planned policy, and from it, the planned policy performances are confirmed - then suffer for the biggest instability. The double epsilon curve is also promising, being almost above the baseline method curve, but seems to stall at a slightly lower value than the target 200 point, eventually reaching the threshold in time equal to the best method or later.

It is interesting to note that the partial results of the Initialization method (named **Pre-Training** in the Figures) after the "*pre-training*" (i.e. initial training before Q-Learning with samples retrieved from the planned policy) achieves actually very good results, around **140** average reward over a span of 100 simulated episodes. As the policy reaches values around +1-4 points, it means that the network is able

**Table 7.4.** Table showing the percentage of improvements for all the tested methods in comparison with the baseline performances.

| Method | % Improvement |
|---|---|
| **Replaced Exploration** | **38%** |
| **Replaced Exploitation** | 4.3% |
| **Double Epsilon** | 14.3% |
| **Pre-planning** | 6.5% |

to learn and generalize well the environment, but unfortunately not enough.
The pre-training has been done with the same network as the other methods, for 100 iterations and with a slightly higher learning rate.
Despite the very good results of the pretrained network, using it for a Q-Learning training doesn't maintain the premises, with results similar to baseline or other models.
But this behaviour can be explained by the fact that the labels generated for the pre-training are in the same form of the initialized policy in the tabular initialization method, with an arbitrary value for the suggested action from the planned policy and zero otherwise. This values of course don't reflect actual Q-values, that could be positive values for multiple action in a state or even multiple negatives. This discrepancy between used labels and Q-values computed during training can cause an initial confusion for the network that has to correct the majority of its prediction in addition to learn the rest of the environment dynamics.
In addition, despite the pretrained network achieves good results, it's simulation result below 200 points means that some of its prediction are incorrect, as well as different from actual Q-values.

For a better understanding of the training curves follow two more figures, the first with the promising methods and the final one with all the worse ones, all compared to the baseline.

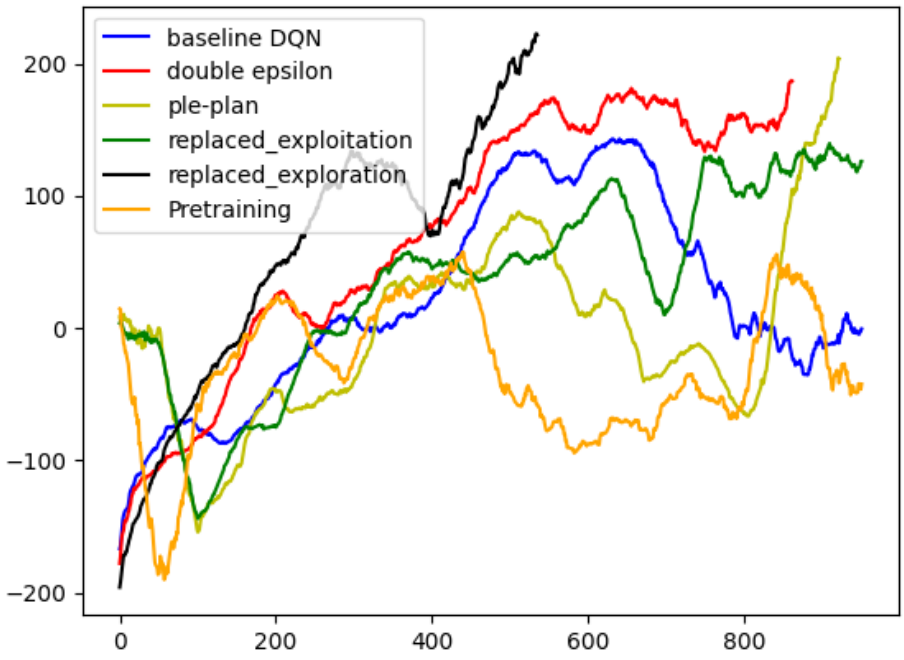


**Figure 7.2.** Pointwise average of learning curves of all experiment runs for each method. It is relevant to note that each run terminates at different episodes, so the beginning of the curves are good average of all runs for each method, while going to the right, some runs finish earlier than other and the average is computed between the remaining ones or even just one. This is the reason why some curves end in low values. That means these methods had some or just one failing run that reached the maximum number of episodes (1000) without success, so far right values are no more averages, but just the reamaining failing run(s).

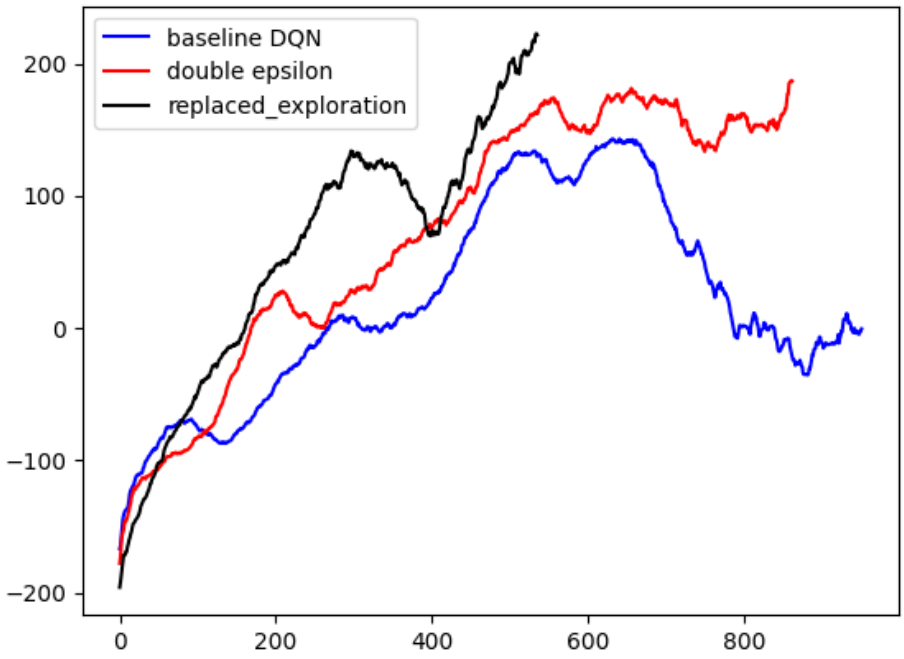


**Figure 7.3.** Pointwise average of learning curves of all experiment runs for promising methods and baseline.

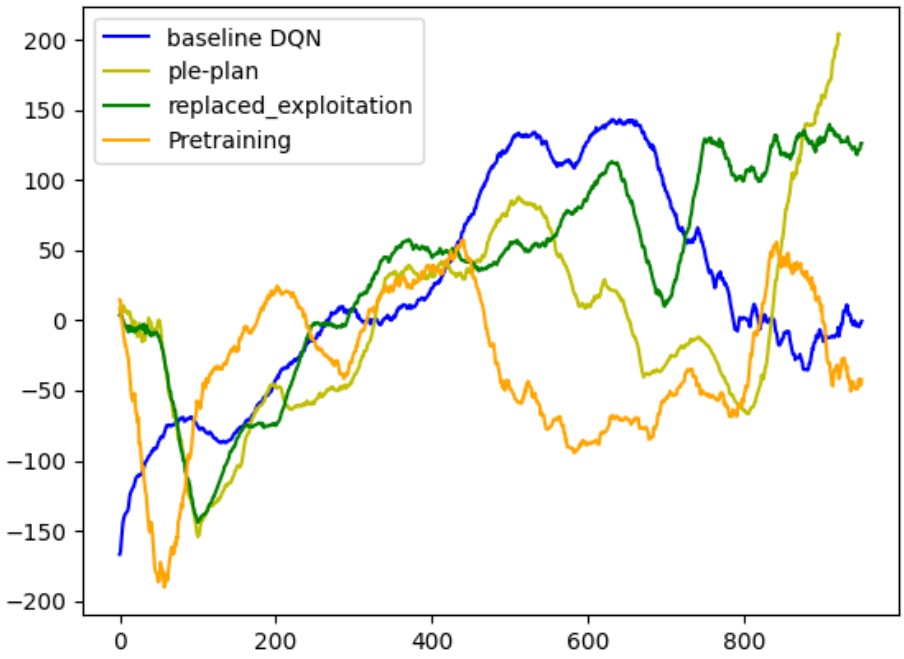


**Figure 7.4.** Pointwise average of learning curves of all experiment runs for unstable methods and baseline.

# Chapter 8

# Conclusions and Future Work

This thesis has explored the modalities and possibilities over the integration of symbolic planning techniques in the reinforcement learning training process.
It has been shown that, despite its potentialities and its proved optimization, planning techniques are not enough to solve on their own complex and continuous environment like the Lunar Lander, even with a proper discretization, multiple exploration and combination of different domain definitions.
The computational power is not enough for complex FOND planning, that in this study case has leveraged the need of defining different domains to lessen the single computational complexity.
But, despite all, the developed strategy still achieves good results in half of the simulation, being able to solve the environment in the episodes with the "easiest" starting states.
In addition, the tabular initialization method achieves a better, or let's say less bad, than the baseline tabular Q-Learning.
This can suggest that in a longer training process it may solve the environment in less time.

The integration in the reinforcement learning process has shown some improvements, in particular regarding the Deep Q-Learning integrations.
Some methods excel more than other, with the most relevant improvement with the **Replaced Exploration** method that achieves an improvement of 38% in terms of average episodes needed in order to solve the complete environment.
The other tested methods also show an improvement but of little magnitude.
All the tested methods, even the baseline one, have shown a high standard deviation in terms of episodes needed to solve the environment in multiple test runs. This is a proof that the Q-Learning algorithm, in particular the Deep Learning version relies on the randomness present in the $\epsilon$-greedy action selection and in the sampling of the experience replay process.

Another relevant partial result is the performance of the *pretrained* model, that achieves interesting and promising results, this method could be investigated deeper in order to achieve better performances and, eventually make it able to solve the environment at its own. This method could eventually replace the entire reinforce-

ment learning process in this environment.
The Deep Q-learning algorithm could be modified in order to predict vectors with similar values as the planned policy, building similar labels in the experience replay stage. This could refine the initial "confusion" of the *pretrained* model that switches from learning to predict the right action only to predict the actual Q-values of all the actions.

Another interesting study is the applicability and the results of these methodologies in others Gymnasium environments, that come in more complicated or easier tasks.

In addition, all the methods can be deeper exploited with a more extended search in the hyperparameter space to assess if hyperparameter changes from the baseline can improve the performances.
Other strategies of domain definition for the planning stage could also be tested.

# Bibliography


[1] Constructions Aeronautiques et al. “Pddl| the planning domain definition language”. In: *Technical Report, Tech. Rep.* (1998).

[2] Christer Bäckström and Bernhard Nebel. “Complexity results for SAS+ planning”. In: *Computational Intelligence* 11.4 (1995), pp. 625–655.

[3] James G Bellingham and Kanna Rajan. “Robotics in remote and hostile environments”. In: *science* 318.5853 (2007), pp. 1098–1102.

[4] Christopher Berner et al. “Dota 2 with large scale deep reinforcement learning”. In: *arXiv preprint arXiv:1912.06680* (2019).

[5] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[6] Jim X Chen. “The evolution of computing: AlphaGo”. In: *Computing in Science & Engineering* 18.4 (2016), pp. 4–7.

[7] Marco Daniele, Paolo Traverso, and Moshe Y Vardi. “Strong cyclic planning revisited”. In: *European Conference on Planning*. Springer. 1999, pp. 35–48.

[8] Marco Daniele, Paolo Traverso, and Moshe Y Vardi. “Strong cyclic planning revisited”. In: *European Conference on Planning*. Springer. 1999, pp. 35–48.

[9] Stefan Edelkamp and Jörg Hoffmann. *PDDL2. 2: The language for the classical part of the 4th international planning competition*. Tech. rep. Technical Report 195, University of Freiburg, 2004.

[10] Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. “Search on the Replay Buffer: Bridging Planning and Reinforcement Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/5c48ff18e0a47baaf81d8b8ea51eec92-Paper.pdf.

[11] Richard E Fikes and Nils J Nilsson. “STRIPS: A new approach to the application of theorem proving to problem solving”. In: *Artificial intelligence* 2.3-4 (1971), pp. 189–208.

[12] Maria Fox and Derek Long. “PDDL+: Modeling continuous time dependent effects”. In: *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*. Vol. 4. 2002, p. 34.

[13] Maria Fox and Derek Long. “PDDL2. 1: An extension to PDDL for expressing temporal planning domains”. In: *Journal of artificial intelligence research* 20 (2003), pp. 61–124.

[14] Jicheng Fu et al. “Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems”. In: *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. Vol. 22. 3. 2011, p. 1949.

[15] Florian Geißer, Thomas Keller, and Robert Mattmüller. “Delete relaxations for planning with state-dependent action costs”. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 6. 1. 2015, pp. 228–229.

[16] Alfonso Gerevini and Derek Long. *Plan constraints and preferences in PDDL3*. Tech. rep. Technical Report 2005-08-07, Department of Electronics for Automation . . ., 2005.

[17] Robert Gieselmann and Florian T. Pokorny. “Planning-Augmented Hierarchical Reinforcement Learning”. In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 5097–5104. DOI: 10.1109/LRA.2021.3071062.

[18] Jessica B. Hamrick et al. *On the role of planning in model-based deep reinforcement learning*. 2021. arXiv: 2011.04021 [cs.AI]. URL: https://arxiv.org/abs/2011.04021.

[19] Todd Hester et al. “Deep q-learning from demonstrations”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[20] Carl-Johan Hoel et al. “Combining Planning and Deep Reinforcement Learning in Tactical Decision Making for Autonomous Driving”. In: *IEEE Transactions on Intelligent Vehicles* 5.2 (2020), pp. 294–305. DOI: 10.1109/TIV.2019.2955905.

[21] Richard E Kopp. “Pontryagin maximum principle”. In: *Mathematics in Science and Engineering*. Vol. 5. Elsevier, 1962, pp. 255–279.

[22] Matteo Leonetti, Luca Iocchi, and Peter Stone. “A synthesis of automated planning and reinforcement learning for efficient, robust decision-making”. In: *Artificial Intelligence* 241 (2016), pp. 103–130. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2016.07.004. URL: https://www.sciencedirect.com/science/article/pii/S0004370216300819.

[23] Long-Ji Lin. “Self-improving reactive agents based on reinforcement learning, planning and teaching”. In: *Machine learning* 8 (1992), pp. 293–321.

[24] Long-Ji Lin. “Self-improving reactive agents based on reinforcement learning, planning and teaching”. In: *Machine learning* 8 (1992), pp. 293–321.

[25] John McCarthy. “History of LISP”. In: *History of programming languages*. 1978, pp. 173–185.

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

[27] Christian Muise, Sheila McIlraith, and Christopher Beck. “Improved nondeterministic planning by exploiting state relevance”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 22. 2012, pp. 172–180.

[28] Christian Muise, Sheila McIlraith, and Vaishak Belle. "Non-deterministic planning with conditional effects". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 24. 2014, pp. 370–374.

[29] Pandu Nayak et al. "Validating the ds-1 remote agent experiment". In: *Artificial intelligence, robotics and automation in space*. Vol. 440. 1999, p. 349.

[30] Ian Parberry. *Introduction to Game Physics with Box2D*. CRC Press, 2017.

[31] Edwin PD Pednault. "ADL: exploring the middle ground between STRIPS and the situation calculus." In: *Kr* 89 (1989), pp. 324–332.

[32] Martin L Puterman. "Markov decision processes". In: *Handbooks in operations research and management science* 2 (1990), pp. 331–434.

[33] Julian Schrittwieser et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[34] Richard S Sutton and Andrew G Barto. "6. Temporal-Difference Learning 133". In: *MIT Press* (1998).

[35] Gerald Tesauro. "TD-Gammon, a self-teaching backgammon program, achieves master-level play". In: *Neural computation* 6.2 (1994), pp. 215–219.

[36] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[37] Thomas Walsh, Sergiu Goschin, and Michael Littman. "Integrating Sample-Based Planning and Model-Based Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 2010), pp. 612–617. DOI: 10.1609/aaai.v24i1.7689. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7689.

[38] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[39] Sung Wook Yoon, Alan Fern, and Robert Givan. "FF-Replan: A Baseline for Probabilistic Planning." In: *ICAPS*. Vol. 7. 2007, pp. 352–359.

# Acknowledgments

*Despite my well-known dishabituation in expressing feelings and thoughts and my below-average writing skill, this is an attempt to acknowledge everyone that has come with me so far.*
*This will not be an explicit one by one citing, if you were interested enough to read this, you are for sure one of the people considered here.*
*You should know well about my regrets on this 'adventure', through all my lows (many) and highs (few).*
*I often felt sadness, loneliness and distrust, headache and heartache, almost because of myself.*
*But if you're reading this, I just want to thank you, from the bottom of my heart. I've been able to keep standing on my screen, to sit up from the bed or the couch after hours of nothingness only because of you and the other readers.*
*Whether it was just a coffee together, a drink, a little chat or something deeper like your thoughtfulness, serious discussions, economic support too.., you supported me to keep going and not give up. There's no way I could have ever done this without you.*
*Again, thank you. I hope I have already showed it to you, if not, I'm sorry. I'll do my best to prove it.*

*P.s. Mom, Dad, I have to explicitly mention and thank you. I could have not asked for better parents.*
*If you're not mom or dad, I hope you weren't offended by that. :)*