Single-partition adaptive Q-learning: algorithm and applications

João Pedro Estácio Gaspar Gonçalves de Araújo

Thesis to obtain the Master of Science Degree in

Mechanical Engineering

Supervisors: Prof. Miguel Afonso Dias de Ayala Botto
Prof. Mário Alexandre Teles de Figueiredo

July 2020

Dedicated to my mother and grandmother.

Acknowledgments

First and foremost, an acknowledgment is due to my supervisors, Professors Miguel Ayala Botto and Mário Figueiredo, for their immense support and help over the past five months. This thesis would not have been possible without their supervision.

I have a lot to thank my family for. First, to my mother, who spent countless hours studying with me when I was little, laying the foundations of my success in College. Then, to my father, who fostered in me the curiosity for investigation, and endured countless hours of non-sensical rambling (mainly composed of half sentences) about this thesis. My brothers and sisters, in particular Francisco, who helped me proofread this text several times, and for always being available for random questions about abstract maths. Finally, my grandparents, for all their support.

Two persons who had a major impact on my life and to whom I owe much are Professors Yann Ollivier and Isabelle Guyon, who were the first ones to introduce me to the field of Artificial Intelligence. A word of appreciation is also due to the Calouste Gulbenkian Foundation, particularly to the members of the scientific committee of the “Novos Talentos em Inteligência Artificial” Fellowship, who were among the first to believe in me, and to my former supervisor, Professor Manuel Lopes, who guided my first steps into Reinforcement Learning.

On an institutional level, I would like to thank CMA - *Centro de Matemática e Aplicações, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa*, for providing access to their computational facilities, which were used to carry the scariest calculations presented in this thesis.

Last, but far from least, to my very good friends, who were always there to support me (without any particular ordering; they know who they are): P.HS, ARS, P.CS, PGR, SJm, AdG, SJ, NS, ES, F, P, my friends and colleagues from the University of Lisbon, from the volunteering camps, from the JdA, and many others who made my university experience unforgettable. To all who inspired me to be who I am today, thank you!

A special word of appreciation is also due to Miguel Rocha and Rafael Andrade, who in the middle of a hard exam season helped me to revise this text.

Resumo

A aprendizagem por reforço (AR) é uma área dentro da aprendizagem automática que estuda como agentes podem aprender a levar a cabo uma tarefa sem serem explicitamente programados para o fazer. Um conceito importante em AR é o de eficiência amostral: um algoritmo é eficiente se precisa de poucas amostras para aprender uma determinada tarefa. Até há pouco tempo, admitia-se que os algoritmos *model-based* eram mais eficientes que os *model-free*. Recentemente, foi demonstrado que os algoritmos *model-free* também podem ser eficientes. Um dos últimos algoritmos desenvolvidos é o *adaptive Q-learning* (AQL), o qual lida com espaços de estados e ações contínuos ao dividi-los adaptativamente consoante as amostras recolhidas. O AQL é projectado para aprender políticas que variam com o tempo. No entanto, muitos problemas (como o controlo de sistemas invariantes no tempo) podem ser resolvidos satisfatoriamente com políticas invariantes no tempo. Esta dissertação introduz o *single-partition adaptive Q-learning* (SPAQL), uma versão melhorada do AQL projetada para aprender políticas invariantes no tempo. O SPAQL é avaliado empiricamente em quatro problemas, dois dos quais da área do controlo. Os agentes SPAQL exibem melhor desempenho que os AQL, aprendendo inclusive políticas mais simples. Para os problemas de controlo, o SPAQL com estado terminal (SPAQL-TS, de *terminal state*) é introduzido e comparado juntamente com o SPAQL face ao método *trust region policy optimization* (TRPO), um algoritmo de AR padrão para resolver problemas de controlo. Num dos problemas (`CartPole`), o SPAQL e o SPAQL-TS demonstram uma maior eficiência amostral que o TRPO.

Palavras-chave: aprendizagem por reforço, Q-learning, eficiência amostral, controlo

Abstract

Reinforcement learning (RL) is an area within machine learning that studies how agents can learn to perform their tasks without being explicitly told how to do so. An important concept in RL is sample efficiency: an algorithm is sample-efficient if it requires a low amount of samples to learn its task. Until recently, it was thought that model-based RL algorithms were more sample-efficient than model-free ones. This changed with recent developments in provably efficient model-free algorithms. One of the latest algorithms developed is *adaptive Q-learning* (AQL), an efficient model-free algorithm that handles continuous state and action spaces by adaptively partitioning them in a data-driven manner. By design, AQL learns time-variant policies. However, many problems (such as control of time-invariant systems) can be solved satisfactorily using time-invariant policies. This thesis introduces *single-partition adaptive Q-learning* (SPAQL), an improved version of AQL designed to learn time-invariant policies. SPAQL is evaluated empirically on four different problems, out of which two are control problems. SPAQL agents perform better than AQL ones, while at the same time learning simpler policies. For the control problems, *SPAQL with terminal state* (SPAQL-TS) is introduced, and, along with SPAQL, is compared to *trust region policy optimization* (TRPO), an RL algorithm known to perform well in control problems. In one of the control problems (`CartPole`), SPAQL and SPAQL-TS display a higher sample-efficiency than TRPO.

Keywords: reinforcement learning, Q-learning, sample efficiency, control

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

α	Learning rate.
λ	Error tolerance (SPAQL-TS).
δ	Probability of the regret bound failing.
π	Policy
$\pi_V^\star, \pi_Q^\star, \pi^\star$	Optimal policy.
τ	Temperature parameter for Boltzmann exploration.
τ_{min}	Temperature reset value (SPAQL).
ε	Probability of picking an action at random in ε-greedy exploration.
ξ	Scaling parameter of the upper confidence bounds.

Roman symbols

$\mathcal{A}$	Set of actions available to an agent interacting with a system.
$\mathcal{D}$	Distance function.
$\texttt{dom}\,(B)$	Domain of ball B.
$\mathcal{P}$	Partition.
$\mathbb{P}$	Transition kernel.
$\mathbf{Q}_h(x,a)$	Estimate of the Q-function value of pair (x,a) at time instant h.
$\mathcal{S}$	Set of system states.
$\mathbf{V}_h(x)$	Estimate of the value function of state x at time instant h.
a	Action picked by the agent.
B	Ball.
$b, b(x,a)$	Bonus term of the upper confidence bound.

$b_\xi(v)$	Bonus term of the upper confidence bound for AQL and SPAQL.
d	Temperature doubling factor parameter (SPAQL).
d_{max}	Maximum of all distances between all points in a given ball.
H	Horizon (number of steps in one episode of a Markov decision process).
h	Time step.
K	Total number of training episodes.
N	Number of evaluation rollouts.
n	Number of times the parents of a ball have been split.
$n_{(x,a)}$	Number of times the state-action pair (x, a) has been visited.
$Q_h^\pi(x,a)$	Q function (evaluated at pair (x, a)) at time instant h under policy π.
$Q_h^\star(x,a)$	Optimal Q function (evaluated at pair (x, a)) at time instant h under policy π.
R	Reward function range.
r	Reward.
$r(B)$	Radius of ball B.
$r(x,a)$	Reward function.
u	Temperature increase parameter (SPAQL).
$v, n(B)$	Number of times ball B has been visited.
$V_h^\pi(x)$	Value function (evaluated at state x) at time instant h under policy π.
$V_h^\star(x)$	Optimal value function (evaluated at state x) at time instant h under policy π.
x	System state.
x'	State reached by choosing action a while at state x under transition kernel $\mathbb{P}$.
x_1	Initial state of an MDP.
x_{ref}	Reference state to track.
$\texttt{RELEVANT}(x)$	Set of relevant balls for state x.

Subscripts

h	Time step.
v	Number of times a ball has been visited.

Superscripts

k	Training episode.

Acronyms

AQL	Adaptive Q-learning
MDP	Markov decision process
NBQL	Net-based Q-learning
RL	Reinforcement learning
RLSVI	Randomized least-squares value iteration
SPAQL	Single partition adaptive Q-learning
SPAQL-TS	Single partition adaptive Q-learning with terminal state
TRPO	Trust region policy optimization
UCB	Upper confidence bound

Chapter 1

Introduction

Reinforcement learning (RL) is an area within machine learning that studies how agents (such as humanoid robots, self-driving cars, or computer programs that play chess) can learn to perform their tasks without being explicitly told how to do so. The problem can be posed as that of learning a mapping from system states to agent actions. In order to do this, the agent observes and interacts with the system, choosing which action to perform given its current state. By choosing a certain action, the system transitions to a new state, and the agent may receive a reward or a penalty. Based on these rewards and penalties, the agent learns to assess the quality of actions. By setting the agent's objective to be reward maximization, it will learn to prefer good actions over bad ones. As an example, consider a robot that stacks boxes. The system is the environment where the robot works, and the actions correspond to moving, grabbing, and releasing the boxes. The robot may receive rewards for each box successfully stacked, and may receive penalties for failing to stack a box, or knocking down the existing box stack. The sum of all rewards and penalties received by the robot during its task is called the cumulative reward. Maximizing this cumulative reward is the objective of RL [1]. This approach has led to many successful applications of RL in areas such as control of autonomous vehicles [2], game playing [3, 4], healthcare [5], and education [6], to name a few.

Existing RL methods can be divided into two main families: model-free and model-based. In model-based methods, the agent either has access to or learns a model of the system, which it uses to plan future actions. Although model-free methods are applicable to a wider array of problems, concerns with the empirical performance of model-free methods have been raised in the past due to their sample complexity [7, 8]. Two additional problems, besides sample complexity, are: the trade-off between exploration and exploitation, and dealing with *Markov decision processes* (MDP) with continuous state and action spaces. Several solutions to deal with these two latter problems have been proposed. For example, exploration and exploitation can be balanced using a stochastic policy, such as ε-greedy [1]. MDPs with continuous state and action spaces can be dealt with by using function approximators, such as neural networks.

1.1 Motivation

Recently, some theoretical work has addressed the sample complexity of model-free methods. Informally, sample complexity can be defined as the number of samples that an algorithm requires in order to learn. Sample-efficient algorithms require fewer samples to learn, and thus are more desirable to develop and use. There are several approaches to sample-efficient RL [9]. However, until recently, it was not known whether model-free methods could be proved to be sample-efficient. This changed when Jin et al. [10] proposed a Q-learning algorithm with *upper confidence bound* (UCB) exploration for discrete tabular MDPs, and showed that it had a sample efficiency comparable to *randomized least-squares value iteration* (RLSVI), a model-based method proposed by Osband et al. [11]. Later, Song and Sun [12] extended that algorithm to continuous state-action spaces. Their algorithm, *net-based Q-learning* (NBQL), requires discretizing the state-action space with a grid of fixed step size. This creates a trade-off between memory requirements and algorithm performance. Coarser discretizations are easier to store, but the algorithm may not have sufficient resolution to achieve satisfactory performance. Finer discretizations yield better performance, but involve higher memory requirements. In order to address this trade-off, Sinclair et al. [13] proposed *adaptive Q-learning* (AQL). AQL introduces adaptive discretization into NBQL, starting with a single ball covering the entire state-action space, and then adaptively discretizing it in a data-driven manner. This adaptivity ensures that the relevant parts of the state-action space are adequately partitioned, while keeping coarse discretizations in regions that are not so relevant.

The papers mentioned in the previous paragraph consider time-variant value functions and learn time-variant policies. This means that a state-action space partition needs to be kept for each (discrete) time step. However, for many practical purposes, time-invariant policies are sufficient to solve the problem satisfactorily (albeit not optimally). This is particularly true in problems with time-invariant dynamics, such as the classical `Pendulum` problem.

1.2 Contributions

With this motivation, this thesis proposes *single-partition adaptive Q-learning* (SPAQL), an improved version of AQL specifically tailored to learn time-invariant policies. SPAQL is evaluated on two simple example problems, as proof of concept. It is then evaluated in two classical control problems, the `Pendulum` and the `CartPole`, which are harder due to their more complex state and action spaces. This thesis also introduces *single-partition adaptive Q-learning with terminal state* (SPAQL-TS), an improved version of SPAQL, which uses concepts from control theory to achieve a better performance in the problems under study. Both SPAQL and SPAQL-TS perform better than AQL on both problems. Furthermore, SPAQL-TS manages to solve the `CartPole` problem, thus earning a place in the `OpenAI Gym` Leaderboard, alongside other *state-of-the-art* methods.

In the `CartPole` problem, both SPAQL and SPAQL-TS show higher sample efficiency than *trust region policy optimization* (TRPO), a standard RL algorithm known for satisfactorily solving a wide variety of control tasks [14]. Furthermore, SPAQL policies are tables that map states to actions, thus being clearly interpretable. TRPO uses neural networks, which means that the learned policy is a black box. The difficulty in interpreting neural networks, as well as their large number of parameters, have stimulated the scientific community to develop more interpretable and compact policy representations (see, for example, recent work by Kubalík et al. [15]). The empirical results presented in this thesis show that SPAQL and SPAQL-TS contribute to this effort of developing algorithms that efficiently learn interpretable control policies.

To the best of the author's knowledge, this is the first time that efficient Q-learning algorithms (*i. e.*, those similar to the ones developed by Jin et al. [10], Song and Sun [12], and Sinclair et al. [13]) are evaluated in classic control problems.

1.3 Objectives

The objectives of this thesis are

- to introduce single-partition adaptive Q-learning (SPAQL), an efficient model-free episodic RL algorithm designed and tailored to learn time-invariant policies;
- to introduce SPAQL with terminal state (SPAQL-TS), an improved version of SPAQL that leverages concepts from control theory in order to achieve a better performance in control tasks;
- to evaluate SPAQL by comparing it to existing baselines (random policy and AQL) in two example problems (oil discovery and ambulance routing), as proof of concept;
- to evaluate SPAQL and SPAQL-TS by comparing it to existing baselines (random policy and TRPO) in two control problems (`Pendulum` and `CartPole`);
- to provide a set of guidelines for reporting the results of RL experiments.

1.4 Thesis Outline

Chapter 2 provides the background for this thesis. It recalls some background concepts in RL (Markov decision processes, Q-learning, and experiment reporting) and metric spaces, and closes with a literature review on recent advances in efficient model-free RL algorithms.

Chapter 3 contains descriptions of the algorithms studied and developed in this thesis. It starts by briefly describing AQL, introducing concepts and definitions which are used to introduce SPAQL, the main contribution of this thesis. The chapter closes with the description of SPAQL-TS.

Chapter 4 describes the experiments performed to empirically evaluate SPAQL and SPAQL-TS. It is divided into two sections. In the first one, SPAQL is evaluated in two example problems, as proof of concept. The second one regards the evaluation of SPAQL and SPAQL-TS in two control problems. Each section contains a high-level description of the experiments' implementation, the experimental procedure followed, and the parameters used. Results are then presented and analyzed.

Finally, the main results are summarized and conclusions drawn in Chapter 5. Further work directions are also proposed.

1.5 Notation

Although they work essentially on the same problems, the RL and control communities use different notations [16]. This thesis builds upon RL work (particularly that by Sinclair et al. [13]), and therefore the RL notation is kept for consistency in Chapters 2 (Background), 3 (Algorithms), and Section 4.1 (oil discovery and ambulance routing problems). However, since its main audience is the control community, Section 4.2 (`Pendulum` and `CartPole` problems) uses the control notation. Table 1.1 lists the symbols used to denote the problem variables in the following sections, along with their usual control counterparts.

RL **Chapters 2 and 3, and Section 4.1**	**Control** **Section 4.2**	**Definition**
x	x	State vector of the system / Linear position (when ambiguous, a distinction will be made)
	o	Observation vector of the system (simulates the sensors available). In the RL sections, it is assumed that $o := x$
x_{ref}	x_{ref}	Reference state being tracked in a control problem
a	u	Control action
$\sum_h r_h(x,a)$	$-J$	Cumulative reward (in control, it is more common to refer to cost J)
v	NC	Number of times a ball has been visited
u	NC	Temperature increase factor in SPAQL
d	NC	Temperature doubling period factor in SPAQL
h	t	Discrete time instant

Table 1.1: Correspondence between the notation used in the following sections. "NC" (No Correspondence) means that the respective symbols only appear in Chapters 2 and 3, and Section 4.1 (RL).

Chapter 2

Background

This chapter recalls some basic concepts of RL and metric spaces, relevant for the subsequent presentation. For a more detailed background in RL, the reader is referred to the classic book by Sutton and Barto [1] (or the notes by Szepesvári [17]). The chapter closes with a literature review on sample-efficient model-free RL algorithms.

2.1 Reinforcement Learning

2.1.1 Markov Decision Processes

This thesis adopts the Markov decision process (MDP) framework for modeling problems. All MDPs considered have finite horizon, meaning that episodes (a simulation of the MDP for a certain number of steps) terminate after a fixed number of discrete time steps.

Formally, an MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, H, \mathbb{P}, r)$, where:

- $\mathcal{S}$ denotes the set of *system states*;
- $\mathcal{A}$ is the *set of actions* of the agent interacting with the system;
- H is the *number of steps* in each episode (also called the *horizon*);
- $\mathbb{P}$ is the *transition kernel*, which assigns to each triple (x, a, x') the probability of reaching state $x' \in \mathcal{S}$, given that action $a \in \mathcal{A}$ was chosen while in state $x \in \mathcal{S}$; this is denoted as $x' \sim \mathbb{P}(\cdot \mid x,\ a)$ (unless otherwise stated, x' represents the state to which the system transitions when action a is chosen while in state x under transition kernel $\mathbb{P}$);
- $r : \mathcal{S} \times \mathcal{A} \to R \subseteq \mathbb{R}$ is the *reward function*, which assigns a reward (or a cost) to each state-action pair (Sinclair et al. [13] use $R = [0, 1]$).

In this thesis, the only limitation imposed on the state and action spaces is that they are bounded. Furthermore, it is assumed that the MDP has time-invariant dynamics, meaning that neither the transition kernel nor the reward function vary with time. There is previous work, such as that by Lecarpentier and Rachelson [18], that uses the term “stationary” when referring to time-invariant MDPs.

The system starts at the initial state x_1. At each time step $h \in \{1, ..., H\}$ of the MDP, the agent receives an observation $x_h \in \mathcal{S}$, chooses an action $a_h \in \mathcal{A}$, receives a reward $r_h = r(x_h, a_h)$, and transitions to state $x_{h+1} \sim \mathbb{P}(\cdot \mid x_h,\ a_h)$. The objective of the agent is to maximize the cumulative reward $\sum_{h=1}^{H} r_h$ received throughout the H steps of the MDP. This is achieved by learning a function $\pi : \mathcal{S} \to \mathcal{A}$ (called a *policy*) that maps states to actions in a way that maximizes the accumulated rewards. This function is then used by the agent when interacting with the system. If the policy is independent of the time step, it is said to be time-invariant.

2.1.2 Q-learning

One possible approach for learning a good (eventually optimal) policy is *Q-learning*. The idea is to associate with each state-action pair (x, a) a number that expresses the **q**uality (hence the name Q-learning) of choosing action a given that the agent is in state x. The value of a state x at time step h is defined as the expected cumulative reward that can be obtained from that state onward, under a given policy π:

$$V_h^{\pi}(x) := \mathbb{E}\left[\sum_{i=h}^{H} r(x_i, \pi(x_i)) \;\Big|\; x_h = x\right]. \tag{2.1}$$

This is equivalent to averaging all cumulative rewards that can be obtained under policy π until the end of the MDP, given that at time step h the system is in state x. By taking into account all rewards until the end of the MDP, the value function provides the agent with information regarding the rewards in the long run. This is relevant, since the state-action pairs yielding the highest rewards in the short run may not be those that yield the highest cumulative reward at the end of the MDP. However, if H is large enough and there is a large number of state-action pairs, it may be impossible (or impractical) to compute the actual value of $V_h^{\pi}(x)$ for all $1 \leq h \leq H$ and $x \in \mathcal{S}$, and some approximation has to be used instead. For example, a possible estimator samples several paths at random, and averages the cumulative rewards obtained [1].

The *value function* $V_h^{\pi} : \mathcal{S} \to \mathbb{R}$ provides information regarding which states are more desirable. However, it does not take into account the actions that the agent might choose in a given state. The *Q function*,

$$Q_h^{\pi}(x, a) := r(x, a) + \mathbb{E}\left[\sum_{i=h+1}^{H} r(x_i, \pi(x_i)) \;\Big|\; x_h = x,\ a_h = a\right] = r(x, a) + \mathbb{E}\Big[V_h^{\pi}(x') \;\Big|\; x,\ a\Big], \tag{2.2}$$

takes this information into account since its domain is the set of all possible state and action pairs. This allows the agent to rank all possible actions at state x according to the corresponding values of Q at time step h, and then make a decision regarding which action to choose. Notice that the second expectation in the previous equation is with respect to $x' \sim \mathbb{P}(\cdot \mid x,\ a)$.

The functions $V_h^\star(x) = \sup_\pi V_h^\pi(x)$ and $Q_h^\star(x,a) = \sup_\pi Q_h^\pi(x,a)$ are called the *optimal value function* and the *optimal Q function*, respectively. The policies $\pi_V^\star$ and $\pi_Q^\star$ associated with $V_h^\star(x)$ and $Q_h^\star(x,a)$ are one and the same, $\pi_V^\star = \pi_Q^\star = \pi^\star$; this policy is called the *optimal policy* [1]. There may be more than one optimal policy, but they will always share the same optimal value and optimal Q function. These functions satisfy the so-called *Bellman equation*

$$Q_h^\star(x,a) = r(x,a) + \mathbb{E}[V_h^\star(x') \mid x, a]. \tag{2.3}$$

Q-learning is an algorithm for computing estimates $\mathbf{Q}_h$ and $\mathbf{V}_h$ of the Q function and the value function, respectively. The updates to the estimates at each time step are based on Equation 2.3, according to

$$\mathbf{Q}_h(x,a) \leftarrow (1-\alpha)\mathbf{Q}_h(x,a) + \alpha(r(x,a) + \mathbf{V}_h(x')), \tag{2.4}$$

where $\alpha \in [0,1]$ is the *learning rate*. This update rule reaches a fixed point (stops updating) when the Bellman equation is satisfied, meaning that the optimal functions were found. Actions are chosen greedily according to the `argmax` policy

$$\pi(x) = \mathtt{argmax}_a \mathbf{Q}_h(x,a). \tag{2.5}$$

The greediness of the `argmax` policy may lead the agent to become trapped in local optima. To escape from these local optima, stochastic policies such as ε-greedy or Boltzmann exploration can be used. The ε-greedy policy chooses the greedy action with probability $1-\varepsilon$ (where ε is a small positive number), and picks any action uniformly at random with probability ε [1]. Boltzmann exploration transforms the $\mathbf{Q}_h$ estimates into a probability distribution (using softmax), and then draws an action at random. It is parametrized by a temperature parameter τ, such that, when $\tau \to 0$, the policy tends to `argmax`, whereas, for $\tau \to +\infty$, the policy tends to one that picks an action uniformly at random from the set of all possible actions.

Finding and escaping from local optima is part of the exploration/exploitation trade-off. Another way to deal with this trade-off is to use *upper confidence bounds* (UCB). Algorithms that use UCB add an extra term $b(x,a)$ to the update rule

$$\mathbf{Q}_h(x,a) \leftarrow (1-\alpha)\mathbf{Q}_h(x,a) + \alpha(r(x,a) + \mathbf{V}_h(x') + b(x,a)), \tag{2.6}$$

which models the uncertainty of the Q function estimate. An intuitive way of explaining UCB is to say that the goal is to choose a value of $b(x,a)$ such that, with high probability, the actual value of $Q_h^\star(x,a)$ lies within the interval $[\mathbf{Q}_h(x,a), \mathbf{Q}_h(x,a) + b(x,a)]$. A very simple example is $b(x,a) = 1/n_{(x,a)}$, where $n_{(x,a)}$ is the number of times action a was chosen while the agent was in state x. As training progresses and $n_{(x,a)}$ increases, the uncertainty associated with the estimate of $\mathbf{Q}_h(x,a)$ decreases. This decrease affects the following training iterations since, eventually, actions that have been less explored will have higher $\mathbf{Q}_h(x,a)$ due to the influence of term $b(x,a)$, and will be chosen by the greedy policy. This results in exploration.

In the finite-horizon case (finite H), the optimal policy usually depends on the time step [19]. One of the causes for this dependency is the constraint $V^{\pi}_{H+1}(x) = 0$ for all $x \in \mathcal{S}$ (as mentioned by Sinclair et al. [13] and Sutton and Barto [1, Section 6.5]), since it forces the value function to be time-variant. The SPAQL algorithm, which learns time-invariant policies, deals with this by ignoring this constraint. For more information, refer to Section 3.2.

2.1.3 Reporting experiments

Recently, Henderson et al. [20] called for attention to the significance and reproducibility of results in RL. In their paper, they comment on the lack of standardization in reporting Deep Reinforcement Learning results, enumerate the sources of variance, and suggest guidelines for future research. Colas et al. [21] suggest the use of statistical power analysis to compare two different RL algorithms, and also to choose the number of different random seeds to use. This thesis follows the recommendations from both papers, to the extent allowed by the available computational resources.

When presenting learning curves, the average cumulative reward is used as a measure of learning. The 95% confidence interval is plotted around the average curve. A large number of tests is carried to narrow the confidence intervals, and ensure statistical significance of the results. The same standard applies when plotting the number of arms used by the agents.

When a relation of order is claimed between the performance of two individual algorithms, it is based on the result of a Welch t-test with 5% significance level [14, 21].

2.2 Metric Spaces

A metric space is a pair $(X, \mathcal{D})$ where X is a set and $\mathcal{D} : X \times X \to \mathbb{R}$ is a function (called the *distance function*) satisfying the following properties:

$$\begin{aligned}
&(\forall x, y \in X)\ \mathcal{D}(x,y) = 0 \Leftrightarrow x = y, \\
&(\forall x, y \in X)\ \mathcal{D}(x,y) = \mathcal{D}(y,x), \\
&(\forall x, y, z \in X)\ \mathcal{D}(x,z) \leq \mathcal{D}(x,y) + \mathcal{D}(y,z), \\
&(\forall x,\ y \in X)\ \mathcal{D}(x,y) \geq 0.
\end{aligned} \tag{2.7}$$

A ball B with center x and radius r is the set of all points in X which are at a distance strictly lower than r from x, $B(x,r) = \{b \in X : \mathcal{D}(x,b) < r\}$. The diameter of a ball is defined as diam$(B) = \sup_{x,y \in B} \mathcal{D}(x,y)$. The diameter of the entire space is denoted $d_{max} =$ diam(X). Next, the concepts of *covering*, *packing*, and *net* are recalled.

Definition 2.2.1 (Sinclair et al. [13]). *An r-covering of X is a collection of subsets of X that covers X (i.e., any element of X belongs to the union of the collection of subsets) and such that each subset has diameter strictly less than r.*

Definition 2.2.2 (Sinclair et al. [13]). *A set of points $\mathcal{P} \subset X$ is an r-packing if the distance between any two points in $\mathcal{P}$ is at least r. An r-net of X is an r-packing such that $X \subseteq \cup_{x \in \mathcal{P}} B(x, r)$.*

2.3 Literature Review

There has been a considerable amount of work on sample-efficient model-free RL algorithms. For a thorough list of references, the reader is referred to the introductory sections of the papers by Sinclair et al. [13] and Touati et al. [22]. More recently, Neustroev and de Weerdt [23] introduced a unified framework for studying sample-efficient UCB-based algorithms in the tabular setting, which includes some of the previous algorithms as particular cases. With a few exceptions (notably the works of Sinclair et al. [13] and Neustroev and de Weerdt [23]), this line of work has been mostly theoretical, with little to no empirical validation.

An efficient Q-learning algorithm with UCB in the infinite-horizon setting has been proposed by Wang et al. [24]. Their algorithm learns time-invariant policies, but only considers the tabular case, which means that it does not work with continuous state-action spaces, unlike the algorithm proposed in this thesis.

Previous works have used UCB to deal with exploration. However, early tests with SPAQL showed that it was not enough, and Boltzmann exploration was introduced. This exploration technique is a standard tool in RL [25], and it has been seen performing better than other strategies, such as UCB, in selected problems [26, 27].

Boltzmann exploration has its drawbacks, namely the existence of multiple fixed points [28]. To deal with this issue, alternative softmax-based operators have been proposed by Asadi and Littman [29] and Pan et al. [30]. The approach followed in this thesis uses the classical softmax function, while keeping track of the best policy found so far, therefore clipping the instability associated with Boltzmann exploration.

The main challenge involved with the use of Boltzmann exploration is setting the temperature schedule. A straightforward approach, used by Tijsma et al. [27], is to set a constant temperature, which can be found using grid search or any other tuning method. Another approach is to decrease (anneal) the temperature as some function of the training iteration [30]. The approach herein followed is the opposite one: start with a low temperature, and gradually increase it if the performance is not improving. This allows a gradual shift from exploitation to exploration. Once the policy improves, the temperature is reset to a low value, and the procedure is repeated. This is similar to cyclical annealing schedules, which have already been used in supervised learning applications [31–36]. In all these methods, the cycle length is controlled by at least one user-defined parameter. In the algorithm proposed in this thesis, the cycle reset occurs automatically, when an increase in performance is detected. The user only has to provide two parameters: one parameter controlling the rate at which the temperature heads towards the maximum value; another that works similarly to a doubling period factor. It also makes more sense to talk about cold restarts instead of warm restarts, since at each restart the temperature is set to the minimum value. Yamaguchi et al. [37] use a combination of UCB and Boltzmann exploration as exploration strategy. Unlike the approach followed in this thesis, they use a monotonically decreasing schedule for the temperature.

There exists abundant literature on the connections and applications of RL to control. For a collection of references on the topic, the reader is referred to the surveys by Kaelbling et al. [38], Polydoros and Nalpantidis [39], and Busoniu et al. [40].

The previously mentioned papers by Jin et al. [10], Song and Sun [12], Sinclair et al. [13], Touati et al. [22], Neustroev and de Weerdt [23], and Wang et al. [24] either lack experimental validation or evaluate their algorithms on other types of problems (such as finding the maximum of a function, or navigating a grid world [13, 23]). As stated in Section 1.2, this thesis innovates by evaluating efficient Q-learning algorithms in control problems.

Chapter 3

Algorithms

This chapter introduces *single-partition adaptive Q-learning* (SPAQL), the main contribution of this thesis. It starts with a brief description of *adaptive Q-learning* (AQL), the algorithm on which SPAQL is based. For a detailed description, please refer to the original paper by Sinclair et al. [13].

SPAQL with terminal state (SPAQL-TS) is described at the end of the chapter.

3.1 Adaptive Q-learning

Adaptive Q-learning (AQL) is a Q-learning algorithm for finite-horizon MDPs with continuous state and action spaces. It keeps a state-action space partition for each time step. While this is fundamental when dealing with time-variant MDPs, for time-invariant ones it may result in an excessive use of memory resources.

The pseudocode for AQL is shown in Algorithm 1. For each variable, superscripts denote the current episode (training iteration) k, and subscripts denote the time step h. Intuitively, the algorithm partitions a collection $\mathcal{P}^1 = \{\mathcal{P}^1_h : h = 1, \ldots, H\}$ of initial balls (one per time step h), each containing the entire state-action space, into smaller balls. For each ball B, it keeps an estimate of the value of the Q-function, denoted by $\mathbf{Q}^k_h(B)$ (estimate of $\mathbf{Q}$ at time step h of episode k).

A ball B_i is said to be relevant for the current state x^k_h (state at time step h of episode k) if there exists at least one pair $(x^k_h, a) \in \mathcal{S} \times A$ such that $(x^k_h, a) \in \texttt{dom}\,(B_i)$, where $\texttt{dom}\,(B_i)$ is the domain of ball B_i. The domain of a ball is defined as

$$\texttt{dom}\,(B) = B \setminus \left(\bigcup_{B' \in \mathcal{P} : r(B') < r(B)} B' \right). \tag{3.1}$$

In other words, the domain of a ball B is the set of all points $b \in B$ which are not contained inside any other ball of strictly smaller radius than $r(B)$. This concept is illustrated in Appendix B. In the examples presented in this thesis, it is ensured that either $\texttt{dom}\,(B) = B$, or $\texttt{dom}\,(B) = \varnothing$. Each case corresponds to before and after splitting ball B, respectively.

The set of all relevant balls at a given time step h of a given episode k is denoted by $\texttt{RELEVANT}^k_h(x^k_h)$

Algorithm 1 Adaptive Q-learning ([13])

```
1: procedure ADAPTIVE Q-LEARNING($\mathcal{S}, \mathcal{A}, \mathcal{D}, H, K, \delta$)
2:     Initiate $H$ partitions $\mathcal{P}_h^1$ for $h = 1, \ldots, H$ each containing a single ball with radius $d_{max}$ and $\mathbf{Q}_h^1$ estimate $H$
3:     for each episode $k \leftarrow 1, \ldots K$ do
4:         Receive initial state $x_1^k$
5:         for each step $h \leftarrow 1, \ldots, H$ do
6:             Select the ball $B_{sel}$ by the selection rule $B_{sel} = \texttt{argmax}_{B \in \texttt{RELEVANT}_h^k(x_h^k)} \mathbf{Q}_h^k(B)$
7:             Select action $a_h^k = a$ for some $(x_h^k, a) \in \texttt{dom}(B_{sel})$
8:             Play action $a_h^k$, receive reward $r_h^k$ and transition to new state $x_{h+1}^k$
9:             Update Parameters: $v = n_h^{k+1}(B_{sel}) \leftarrow n_h^k(B_{sel}) + 1$
10:
11:            $\mathbf{Q}_h^{k+1}(B_{sel}) \leftarrow (1-\alpha_v)\mathbf{Q}_h^k(B_{sel}) + \alpha_v(r_h^k + \mathbf{V}_{h+1}^k(x_{h+1}^k) + b_\xi(v))$ where
12:
13:            $\mathbf{V}_{h+1}^k(x_{h+1}^k) = \min(H, \max_{B \in \texttt{RELEVANT}_{h+1}^k(x_{h+1}^k)} \mathbf{Q}_{h+1}^k(B))$ (see Section 3.1, Equation 3.3)
14:            if $n_h^{k+1}(B_{sel}) \geq \left(\frac{d_{max}}{r(B_{sel})}\right)^2$ then SPLIT BALL($B_{sel}, h, k$)
15: procedure SPLIT BALL($B$, $h$, $k$)
16:     Set $B_1, \ldots B_n$ to be an $\frac{1}{2}r(B)$-packing of $\texttt{dom}(B)$, and add each ball to the partition $\mathcal{P}_h^{k+1}$ (see Definition 2.2.2)
17:     Initialize parameters $\mathbf{Q}_h^{k+1}(B_i)$ and $n_h^{k+1}(B_i)$ for each new ball $B_i$ to inherit values from the parent ball $B$
```

(or simply $\texttt{RELEVANT}(x_h^k)$). Given the current state x_h^k, the algorithm selects the relevant ball with the highest estimate of the Q-function, denoted by B_{sel}. It then picks an action uniformly at random from within that ball. The number of times a ball has been visited at time step h is denoted n_h^k. This number carries across training iterations, as per line 9 of Algorithm 1. If the number of times a ball has been visited is above a certain threshold of times, it is then partitioned into smaller balls, thus refining the state-action space partition in the regions where most visits have occurred. This threshold is defined as $(d_{max}/r(B_{sel}))^2$, where d_{max} is the radius of the initial ball covering the entire state-action space, and $r(B_{sel})$ is the radius of the currently selected ball. If each child ball has half the radius of its parent, then $r(B) = d_{max}/2^n$, where n is the number of times that ball B's parents have been split. Substituting in the threshold formula, the equivalent expression 4^n is obtained [13].

The update rule for the estimates of $\mathbf{Q}_h^k$ is given by

$$\mathbf{Q}_h^{k+1}(B_{sel}) \leftarrow (1-\alpha_v)\mathbf{Q}_h^k(B_{sel}) + \alpha_v(r_h^k + \mathbf{V}_{h+1}^k(x_{h+1}^k) + b_\xi(v)), \tag{3.2}$$

where $b_\xi(v)$ is the UCB bonus term, v is a simpler way to denote the number of times that the currently selected ball B_{sel} has been **v**isited, and α_v is the learning rate. Both $b_\xi(v)$ and α_v are defined below. The $k+1$ superscript in $\mathbf{Q}_h^{k+1}$ means that the update to the estimate carries to the next episode, not having any effect on the current one (similar to what was previously explained for n_h^k).

The value function is defined as

$$\mathbf{V}_{h+1}^k(x_{h+1}^k) = \begin{cases} \min(H, \max_{B \in \texttt{RELEVANT}_{h+1}^k(x_{h+1}^k)} \mathbf{Q}_{h+1}^k(B)) & 1 \leq h < H \\ 0 & h = H \end{cases}. \tag{3.3}$$

The value of the terminal state, $\mathbf{V}_{H+1}^{k}(x)$ (where H is the horizon), is defined to be 0 for all states $x \in \mathcal{S}$.

The trade-off between exploration and exploitation is handled using an upper confidence bound (UCB) of the estimate of the Q-function, $b_{\xi}(v)$, defined as

$$b_{\xi}(v) = \frac{\xi}{\sqrt{v}}, \tag{3.4}$$

where ξ is called the *scaling parameter* of the upper confidence bounds, and is a user-defined parameter. Sinclair et al. [13] provide an expression for the value of ξ,

$$\xi = 2\sqrt{H^3 \log\left(\frac{4HK}{\delta}\right)} + 4Ld_{max}, \tag{3.5}$$

where K is the number of training iterations, δ is a parameter related to the high-probability regret bound [13], L is the Lipschitz constant of the Q-function, and d_{max} is the radius of the initial ball covering the entire state-action space.

The learning rate α_v is set according to

$$\alpha_v = \frac{H+1}{H+v}. \tag{3.6}$$

The previous description of the algorithm can be summed up as the application of three rules:

- **Selection rule**: given the current state x_h^k, select the ball with the highest value of $\mathbf{Q}_h^k$ (out of the ones which are relevant), and from within that ball pick an action a_h^k uniformly at random.

- **Update parameters**: increment $n_h(B)$, the number of times ball B has been visited (the training iteration k is not relevant and can be dropped), and update $\mathbf{Q}_h^{k+1}(B)$ according to Equation 3.2 ($k+1$ is notation for carrying the Q-function estimate over to the next episode).

- **Re-partition the space**: when a ball B has been visited more than $(d_{max}/r(B))^2$ times, cover it with a $\frac{1}{2}r(B)$-Net of B; each new ball inherits the number of visits and the Q-function estimate from its parent ball.

3.2 Single partition adaptive Q-learning

The proposed SPAQL algorithm builds upon AQL [13]. The change proposed aims at tailoring the algorithm to learn time-invariant policies. The main difference is that only one state-action space partition is kept, instead of one per time step; *i.e.*, $\mathbf{Q}_h^k := \mathbf{Q}^k$ and $\mathbf{V}_h^k := \mathbf{V}^k$, where k denotes the current training iteration. The superscript k is used to distinguish between the estimates being used in the update rules (denoted $\mathbf{Q}^k$ and $\mathbf{V}^k$) and the updated estimates (denoted $\mathbf{Q}^{k+1}$ and $\mathbf{V}^{k+1}$). In order to simplify the notation, the superscript may be dropped when referring to the current estimate (denoted $\mathbf{Q}$ and $\mathbf{V}$).

Algorithm 2 Auxiliary functions for SPAQL

```
1: procedure EVALUATE AGENT($\mathcal{P}, \mathcal{S}, \mathcal{A}, H, N$)
2:     Collect $N$ cumulative rewards from ROLLOUT($\mathcal{P}, \mathcal{S}, \mathcal{A}, H$)
3:     Return the average cumulative reward
4: procedure ROLLOUT($\mathcal{P}, \mathcal{S}, \mathcal{A}, H$)
5:     Receive initial state $x_1$
6:     for each step $h \leftarrow 1, \ldots, H$ do
7:         Pick the ball $B_{sel} \in \mathcal{P}$ by the selection rule $B_{sel} = \texttt{argmax}_{B \in \texttt{RELEVANT}(x_h)} \mathbf{Q}(B)$
8:         Select action $a_h = a$ for some $(x_h, a) \in \texttt{dom}(B_{sel})$
9:         Play action $a_h$, record reward $r_h$, and transition to new state $x_{h+1}$
10:    Return cumulative reward $\sum_h r_h$
11: procedure SPLIT BALL($B$)
12:    Set $B_1, \ldots B_n$ to be an $\frac{1}{2}r(B)$-packing of $\texttt{dom}(B)$, and add each ball to the partition $\mathcal{P}$ (see Definition 2.2.2)
13:    Initialize parameters $\mathbf{Q}(B_i)$ and $n(B_i)$ for each new ball $B_i$ to inherit values from the parent ball $B$
14: procedure BOLTZMANN SAMPLE($\mathbf{B}$, $\tau$)
15:    Normalize values in $\mathbf{B}$ by dividing by $\max(\mathbf{B})$ (this helps to prevent overflows)
16:    Sample a ball $B$ from $\mathbf{B}$ by drawing a ball at random with probabilities following the distribution
           $$P(B = B_i) \sim \exp(\mathbf{Q}(B_i)/\tau)$$
```

Algorithm 3 Single-partition adaptive Q-learning

```
1: procedure SINGLE-PARTITION ADAPTIVE $Q$-LEARNING($\mathcal{S}, \mathcal{A}, \mathcal{D}, H, K, N, \xi, \tau_{\text{min}}, u, d$)
2:     Initialize partitions $\mathcal{P}$ and $\mathcal{P}'$ containing a single ball with radius $d_{max}$ and $\mathbf{Q} = H$
3:     Initialize $\tau$ to $\tau_{\text{min}}$
4:     Calculate agent performance using EVALUATE AGENT($\mathcal{P}, \mathcal{S}, \mathcal{A}, H, N$) (Algorithm 2)
5:     for each episode $k = 1, \ldots, K$ do
6:         Receive initial state $x_1^k$
7:         for each step $h = 1, \ldots, H$ do
8:             Get a list $\mathbf{B}$ with all the balls $B \in \mathcal{P}'$ that contain $x_h^k$
9:             Sample the ball $B_{sel}$ using BOLTZMANN SAMPLE($\mathbf{B}, \tau$) (Algorithm 2)
10:            Select action $a_h^k = a$ for some $(x_h^k, a) \in \texttt{dom}(B_{sel})$
11:            Play action $a_h^k$, receive reward $r_h^k$, and transition to new state $x_{h+1}^k$
12:            Update Parameters: $v = n(B_{sel}) \leftarrow n(B_{sel}) + 1$
13:
14:            $\mathbf{Q}(B_{sel}) \leftarrow (1 - \alpha_v)\mathbf{Q}(B_{sel}) + \alpha_v(r_h^k + \mathbf{V}(x_{h+1}^k) + b_\xi(v))$ where
15:
16:                $\mathbf{V}(x_{h+1}^k) = \min(H, \max_{B \in \texttt{RELEVANT}(x_{h+1}^k)} \mathbf{Q}(B))$ (see Section 3.2)
17:            if $n(B_{sel}) \geq \left(\frac{d_{max}}{r(B_{sel})}\right)^2$ then SPLIT BALL($B_{sel}$) (Algorithm 2)
18:        Evaluate the agent using EVALUATE AGENT($\mathcal{P}', \mathcal{S}, \mathcal{A}, H, N$) (Algorithm 2)
19:        if agent performance improved then
20:            Copy $\mathcal{P}'$ to $\mathcal{P}$ (keep the best agent)
21:            Reset $\tau$ to $\tau_{\text{min}}$
22:            Decrease $u$ using some function of $d$ (for example, $u \leftarrow u^d$, assuming $d < 1$)
23:        else
24:            Increase $\tau$ using some function of $u$ (for example, $\tau \leftarrow u\tau$)
25:            if more than two splits occurred then
26:                Copy $\mathcal{P}$ to $\mathcal{P}'$ (reset the agent)
27:                Reset $\tau$ to $\tau_{\text{min}}$
```

3.2.1 Auxiliary procedures

Before describing the algorithm, it is necessary to define some auxiliary functions (Algorithm 2). A measure of an agent's performance is obtained by performing a full episode under the current policy, and recording the cumulative reward. This is called a rollout, and is performed using function ROLLOUT. The arguments that are given as input to this function are the ones describing the problem (state and action spaces $\mathcal{S}$ and $\mathcal{A}$, and horizon H) and the agent (partition $\mathcal{P}$, with estimates $\mathbf{Q}$). Since there might be random elements in the environment (stochastic transitions, for example) and in the agent (stochastic policies), it is necessary to estimate the average performance of the agent by doing several rollouts and averaging the cumulative rewards. This estimation is done by function EVALUATE AGENT. Besides the input arguments required for the ROLLOUT function, EVALUATE AGENT also requires the number of rollouts N to perform.

Functions BOLTZMANN SAMPLE and SPLIT BALL are used during training for action selection and ball splitting, respectively. Function BOLTZMANN SAMPLE implements Boltzmann exploration. It has as arguments a list $\mathbf{B}$ of balls which contain the given state $x \in \mathcal{S}$, and a temperature parameter τ. It then draws a ball at random from $\mathbf{B}$ according to the distribution induced by the values of $\mathbf{Q}$ and temperature τ. Function SPLIT BALL receives as input a ball B and covers it with smaller balls of radius $r(B)/2$, where $r(B)$ is the radius of the ball being split.

3.2.2 Main algorithm

The algorithm (Algorithm 3) keeps two copies of the state-action space partition. One ($\mathcal{P}$) is used to store the best performing agent found so far (performance is defined as the average cumulative reward obtained by the agent). The other copy of the partition ($\mathcal{P}'$) is modified during training. At the end of each training iteration, the performance of the agent with partition $\mathcal{P}'$ is evaluated. If it is better than the performance of the previous best agent (with partition $\mathcal{P}$), the algorithm keeps the new partition ($\mathcal{P} \leftarrow \mathcal{P}'$) and continues training. However, if performance decreases, then, at any moment, it is able to restart the agent and retrain from the previously found best one. In this way, the algorithm ensures that at the end of training the best agent which was found is returned. Another advantage of keeping only the partition associated with the best performance is that it forces an increase in the number of arms to correspond to an improvement in performance, thus preventing over-partitioning of the state-action space. At first, both partitions contain a single ball B with radius d_{max} (which ensures it covers the entire state-action space). The value of $\mathbf{Q}(B)$ is optimistically initialized to H, the episode length.

Each training iteration is divided into two parts. In the first one, a full episode (consisting of H time steps) is played. The values of $\mathbf{Q}$ are updated in each time step, and splitting occurs every time the criterion is met. At the end of the episode, the agent is evaluated over N runs, and the average cumulative reward computed. The second part of the training iteration modifies the policy according to the performance of the agent after training for an episode. To balance exploration with exploitation, a Boltzmann exploration scheme with an adaptive schedule is used. A temperature parameter τ allows varying the policy between a deterministic `argmax` policy (for $\tau \to 0$) and a purely random one (for

$\tau \to +\infty$). If the agent currently being trained achieved a better performance than previous agents, it may be somewhere worth exploiting. The value of τ is reset to a user-defined τ_{min} (≈ 0) in order to ensure that the policy becomes greedy (`argmax`). If the agent performs worse than the best agent, it may be because it is stuck in a local optimum, or simply in an uninteresting region. The value of τ is thus increased to make the policy behave in a more exploratory way. Empirical tests show that increasing τ by multiplying it by a factor $u > 1$ works well, although other schemes may also yield good results. The same tests also show that it is recommendable to normalize the values of $\mathbf{Q}$ before generating the probability distribution (see the definition of function BOLTZMANN SAMPLE in Algorithm 2), and vary τ between a minimum of $\tau_{\text{min}} = 0.01$ (greedy) and a maximum of 10 (random), at which updates to τ saturate.

Updates to the Q-function estimate $\mathbf{Q}$ are done according to

$$\mathbf{Q}^{k+1}(B_{sel}) \leftarrow (1-\alpha_v)\mathbf{Q}^k(B_{sel}) + \alpha_v\big(r_h^k + \mathbf{V}^k(x_{h+1}^k) + b_\xi(v)\big). \tag{3.7}$$

The main difference between Equation 3.7 and Equation 3.2 is that the estimates of the value function $\mathbf{V}^k(x_{h+1}^k)$ and Q-function $\mathbf{Q}^{k+1}(B_{sel})$ are time-invariant in Equation 3.7. The remaining terms retain their meaning from Equation 3.2: r_h^k is the reward obtained during training iteration k on time step h, α_v is the learning rate (Equation 3.6), $b_\xi(v)$ is the bonus term related with the upper confidence bound (defined according to Equation 3.4, with the value of ξ set by the user), and v is the number of times ball B_{sel} has been visited.

The value function is defined according to

$$\mathbf{V}^k(x_{h+1}^k) = \min(H, \max_{B \in \texttt{RELEVANT}(x_{h+1}^k)} \mathbf{Q}^k(B)). \tag{3.8}$$

To simplify the notation, the subscript and superscript in $\texttt{RELEVANT}_h^k$ were omitted in Equation 3.8 and in Algorithm 3. There are two differences from Equation 3.3 to Equation 3.8. The first one is that the value function in Equation 3.8 is time-invariant. The second one is that this definition also holds for the value of the final state, while Equation 3.3 sets $\mathbf{V}_{H+1}^k(x) = 0$, for all x. The reason for this second difference is that this boundary condition in Equation 3.3 generates a dependency of the value function on the time step, that should be avoided when learning a time-invariant policy.

Summarizing, the SPAQL algorithm includes the three rules used by the original AQL algorithm:

- **Selection rule**: select a relevant ball B for x_h^k by drawing a sample from a distribution induced by the values of $\mathbf{Q}(B)$ and τ (breaking ties arbitrarily). Select any action a_h^k to play such that $(x_h^k, a_h^k) \in \texttt{dom}\,(B)$.

- **Update parameters**: increment $n(B)$ and update $\mathbf{Q}(B)$ according to Equation 3.7 (with the difference that the same $\mathbf{Q}$ function is considered at all time steps).

- **Re-partition the space**: if $n(B) \geq (d_{max}/r(B))^2$, cover $\texttt{dom}\,(B)$ with a $\frac{1}{2}r(B)$-Net of $\texttt{dom}\,(B)$. Each new ball B_i in the net inherits the $\mathbf{Q}(B_i)$ and $n(B_i)$ values from its parent ball B. This rule is kept equal to the original one proposed by Sinclair et al. [13].

Two new rules are introduced in SPAQL to balance exploration and exploitation:

- **Adapt the temperature**: if the agent's performance did not improve, the temperature τ is increased. If the performance improved, the policy is reset to greedy. As training progresses, and more visits are required to split existing balls, the temperature increase rate factor u is decreased. This keeps the policy greedy for a longer time, and allows more splits to occur.
- **Reset the agent**: as training progresses, existing balls will be split. If a ball contains an optimum, further splits of that ball are required in order for the agent to get closer to it. If the agent performance has not increased after two splits (either in the same ball or on two different balls), then the agent and the policy are reset ($\mathcal{P}' \leftarrow \mathcal{P}$, and τ is set to the minimum).

3.3 Single partition adaptive Q-learning with terminal state

When designing a controller, the goal is to have the final state of the system to be as close as possible to a user-defined state (reference). Denoting this reference state by x_{ref}, it is possible to modify the value function used in SPAQL (Equation 3.8) to take this information into account. The intuition behind this idea is to decrease the value of the states based on their distance to the reference state. There are several possible ways to do this, depending on the properties sought for the new value function. One such possibility is to weight the value function with a Gaussian with mean x_{ref} and user-defined standard deviation λ. The resulting value function,

$$\mathbf{V}^k(x_{h+1}^k) = \exp\left(-\left(\frac{\mathcal{D}(x_h^k, x_{ref})}{\lambda}\right)^2\right) \min\left(H, \max_{B \in \texttt{RELEVANT}(x_{h+1}^k)} \mathbf{Q}(B)\right), \tag{3.9}$$

weights the SPAQL value function according to the distance $\mathcal{D}(x_h^k, x_{ref})$ to the reference state (error). The weight given to the error is controlled by the parameter $\lambda > 0$. Setting λ to a low value forces the algorithm to search for policies that minimize the error as much as possible, while setting λ to a large value allows the algorithm to search for more tolerant policies.

This modified version of SPAQL is referred to as SPAQL with terminal state (SPAQL-TS).

This very simple idea allows the introduction of some domain knowledge into the concept of quality. Instead of defining quality solely as a blind search for high cumulative rewards, this modified value function introduces the notion of error, and correlates high-quality states with low error values. In principle, this should allow for more efficient training.

Chapter 4

Experiments

In this chapter, SPAQL is tested in the oil and ambulance example problems (used by Sinclair et al. [13]) as proof of concept. Along with SPAQL-TS, it is then tested in two control problems. A random policy and AQL agents are used as baseline for comparison in all problems. In the control problems, *trust region policy optimization* (TRPO) is also used as baseline.

The code associated with all the experiments presented in this thesis is available at `https://github.com/jaraujo98/SinglePartitionAdaptiveQLearning`

4.1 Proof of concept

In this section SPAQL is compared to AQL in the Oil Discovery and Ambulance Relocation problems, used by Sinclair et al. [13]. The state and action space in both problems is $\mathcal{S} = \mathcal{A} = [0, 1]$. The metric used is the ∞ product metric $\mathcal{D}((x, a), (x', a')) = \max\{|x - x'|, |a - a'|\}$.

A paper describing SPAQL, along with this experimental evaluation, is available on the arXiv [41].

4.1.1 Implementation

For SPAQL agents, the implementation of the algorithm uses the tree data structure used by Sinclair et al. [13] to implement partition $\mathcal{P}_h^k$, but only keeps one tree per agent, instead of one per time step. The initial ball has center $(0.5, 0.5)$ and radius[1] 0.5 (a square of side 1 centered at $(0.5, 0.5)$), allowing it to cover tightly the entire state-action space. Each split divides each ball into four equal balls, each with half of the radius of the parent ball (*i. e.*, each square is split into four equal smaller squares).

Ball selection is done using a recursive algorithm. Since AQL uses `argmax` as policy, it only needs to check whether the state is contained in the ball, while keeping track of the ball which has the highest $\mathbf{Q}$ value. SPAQL uses a stochastic policy, and therefore it needs to check if the state-action pair is contained in the ball.

[1]Although the algorithm specifies that the initial radius should be $d_{max} = 1$, this would end up covering a lot of space outside the state-action space. Experiments show that initializing the radius to the tighter 0.5 value results in faster learning.

The temperature parameter τ schedule is controlled by parameters u and d. The temperature is increased by multiplying by u (a real number greater than 1), and clipped at 10 to avoid numerical problems. When a better agent is found, u is updated to be u^d (where d is slightly smaller than 1). In this way, more iterations are required before the policy becomes random, allowing for more exploitation and splitting.

4.1.2 Procedure and parameters

The parameters that are common to both algorithms are the number of training iterations (episodes) K, the episode length H, and the scaling value of the confidence bounds ξ. For each environment, the number of agents trained (to estimate average cumulative rewards) was the one found in the code provided with [13] (25 agents in the oil problem, and 50 in the ambulance one).

The SPAQL algorithm has two additional parameters, u and d, which control the temperature schedule. These were set to be 2 and 0.8, respectively, since they are seen to be acceptable over a wide range of experiments. The value of τ_{min} was set to 0.01. For a value of $u = 2$, this means that 10 iterations are enough to turn this policy into a random one (since $0.01 \times 2^{10} \approx 10$, value at which the temperature saturates).

In order to compare both algorithms, the impact of the scaling parameter ξ is studied. A fixed set of 13 scaling values

$$\xi \in \{0.01, 0.1, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 3, 4, 5\}, \tag{4.1}$$

is used for both scenarios. Table 4.1 contains the parameters used in the experiments whose results are reported in Figures 4.1 and 4.3. The average cumulative reward at the end of training was calculated over the several agents for each algorithm, along with the 95% confidence intervals. The trials that resulted in higher average cumulative rewards are used for the remaining analysis.

Parameter	Value
Episode length H	5
Number of Oil episodes (training iterations) K	5000
Number of Oil agents	25
Number of Ambulance episodes (training iterations) K	2000
Number of Ambulance agents	50
Number of evaluation Rollouts N	20

Table 4.1: Parameters for the scaling experiments in the oil and ambulance problems.

4.1.3 Oil discovery

Setup

In this problem, described by Mason and Watts [42], an agent surveys a 1D map in search of hidden "oil deposits". It is similar to a function maximization problem, where a cost is incurred every time a new estimate is made. This cost is proportional to the distance travelled. The state space is the set of locations that the agent has access to ($\mathcal{S} = [0, 1]$). Since each action corresponds to the position of the agent in the next time step, the state space will match the action space ($\mathcal{A} = [0, 1]$). The transition kernel is $\mathbb{P}_h(x' \mid x, a) = \mathbb{1}_{[x'=a]}$, where $\mathbb{1}_{[A]}$ is the indicator function (evaluates to 1 if condition A is true, and to 0 otherwise). The reward function is $r_h(x, a) = \max\{0, f(a) - |x - a|\}$, where $f(a) \in [0, 1]$ is called the *survey function*. This survey function encodes the location of the deposits ($f(a) = 1$ means that the exact location has been found). The same survey functions considered by Sinclair et al. [13] are considered in this thesis:

- quadratic survey function, $f(x) = 1 - \lambda(x - c)^2$, with $\lambda \in \{1, 10, 50\}$;
- Laplace survey function, $f(x) = e^{-\lambda|x-c|}$, with $\lambda \in \{1, 10, 50\}$.

The deposit is placed at approximately $c \approx 0.75$ (the actual location is $0.7 + \pi/60$). It is clear that the optimal policy is to choose the location of the deposit at every time step. Following this policy, an initial penalty is paid when moving from 0 to c, but in the remaining steps the maximum reward is always obtained. Using this optimal policy, the cumulative reward can be bounded by

$$1 - |0 - c| + (H - 1) \times 1 = H - c. \tag{4.2}$$

Applying this formula to the case under study (with $H = 5$ and $c \approx 0.75$), the maximum reward is found to be approximately 4.25.

Results

The effect of the scaling parameter ξ on both AQL and SPAQL agents trained in the oil problem is shown in Figure 4.1.

The Laplace survey function results in more concentrated rewards. As the value of λ is increased, the rewards become even more concentrated. This makes it harder for both types of agents to approach the maximum cumulative reward estimated previously (4.25). The AQL agents outperform the SPAQL agents for every value of ξ when $\lambda \in [10, 50]$. When $\lambda = 1$, half of the ξ values result in higher rewards for the AQL agents, while the other half results in higher rewards for the SPAQL agents.

The quadratic survey function can also concentrate rewards, but does so in a more relaxed way than the Laplace survey function. This allows both types of agents to approach the maximum cumulative reward, even when $\lambda = 50$. On average, SPAQL agents perform better than the AQL ones (higher average cumulative rewards and lower standard deviations). However, there always exists at least one scaling value for which the AQL agents achieve higher cumulative rewards than the SPAQL ones.

Two main conclusions can be drawn from Figure 4.1. The first one is that, for both types of agents, the value of ξ should be picked from the interval $[0, H/3]$. The second one is that SPAQL agents are less sensitive to changes in ξ than AQL agents (as seen by the size of the shaded regions). This is not surprising, since AQL agents rely solely on UCB for exploration, while SPAQL agents use UCB plus Boltzmann exploration.

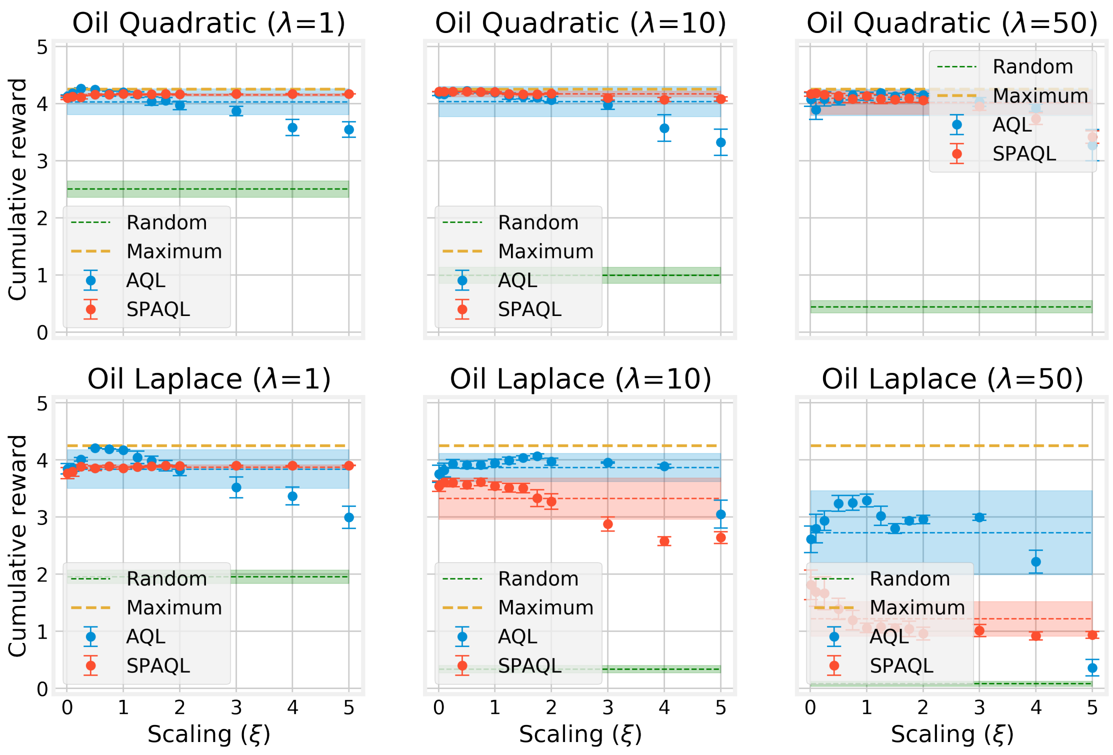

Figure 4.1: Comparison of the effect of different scaling parameter values on the average cumulative reward for the oil discovery problem with survey functions described in Section 4.1.3. Each dot corresponds to the average of the rewards obtained by the 25 agents after 5000 training iterations, and the error bars display the corresponding 95% confidence interval. Dashed lines represent the average cumulative reward calculated over the 13 scaling values listed in Section 4.1.2, and the shaded areas represent the corresponding standard deviation.

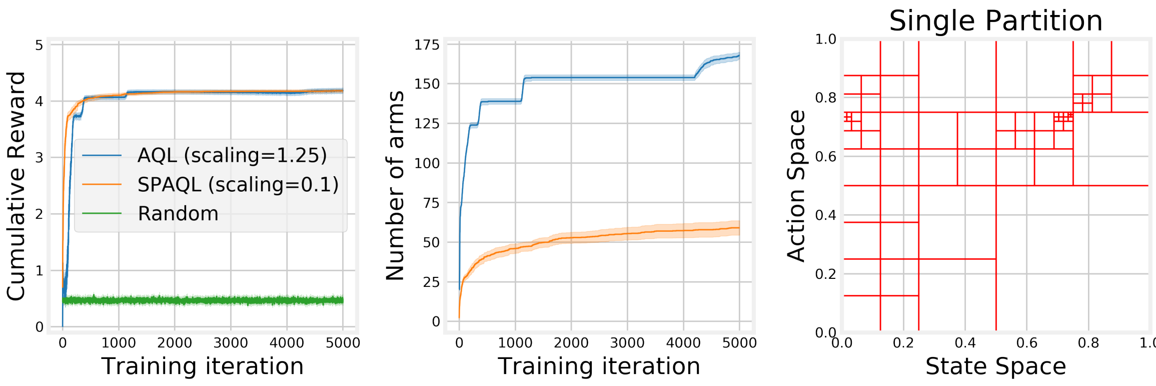

Figure 4.2: Average cumulative rewards, number of arms, and best SPAQL agent partition for the agents trained in the oil problem with reward function $r(x, a) = \max\{0, 1 - 50(x - c)^2 - |x - a|\}$. Shaded areas around the solid lines represent the 95% confidence interval.

Shifting the focus from the scaling parameter to the learning process, Figure 4.2 shows the average cumulative rewards obtained by the SPAQL and the AQL agents for the oil problem with quadratic survey function ($\lambda = 50$), along with the partition learned by the best SPAQL agent. Looking at this partition, it can be seen that the neighborhoods of points $(0, 0.75)$ and $(0.75, 0.75)$, which correspond to the optimal policy, have been thoroughly partitioned, meaning that the agent located and exploited the location of

		λ (Quadratic)		
		1	10	50
Average cumulative reward	Random	2.50 ± 0.06	0.99 ± 0.06	0.44 ± 0.04
	AQL	$\mathbf{4.26 \pm 0.01}$	$\mathbf{4.22 \pm 0.01}$	$\mathbf{4.19 \pm 0.04}$
	SPAQL	4.17 ± 0.00	4.21 ± 0.00	$\mathbf{4.18 \pm 0.03}$
Average number of arms	AQL	155.72 ± 4.47	140.60 ± 2.86	167.84 ± 2.09
	SPAQL	42.04 ± 1.90	35.08 ± 1.10	59.08 ± 4.52

		λ (Laplace)		
		1	10	50
Average cumulative reward	Random	1.95 ± 0.05	0.33 ± 0.03	0.08 ± 0.02
	AQL	$\mathbf{4.21 \pm 0.01}$	$\mathbf{4.07 \pm 0.04}$	$\mathbf{3.29 \pm 0.11}$
	SPAQL	3.90 ± 0.00	3.61 ± 0.07	1.81 ± 0.26
Average number of arms	AQL	158.36 ± 2.83	195.08 ± 2.70	357.08 ± 7.12
	SPAQL	39.28 ± 1.89	67.12 ± 4.89	57.28 ± 7.55

Table 4.2: Average cumulative rewards and number of arms ($\pm 95\%$ confidence interval) in the oil discovery problem. The best performing agent for each value of λ is shown in bold. When the Welch t-test does not find a significant statistical difference between two performances, both are shown in bold.

the oil deposit. In the first training iterations, the SPAQL agents increase the cumulative rewards faster than the AQL agents. Both types of agents stabilize at the same level of cumulative reward, with the difference that the SPAQL agents use around one third of the arms of the AQL agents. The results of the other tests in the oil problem are presented in Table 4.2, and the respective figures are shown in Appendix A. The SPAQL agents always increase their cumulative rewards faster than the AQL agents during the initial training iterations. However, in most cases (especially for the Laplace survey function, Figures A.4, A.5, and A.6), they are eventually surpassed by the AQL agents. Looking at the associated partitions, it is clear that this happens due to a lack of exploitation by the SPAQL agents, especially when the value of λ is low (meaning that the rewards are high over a wide area). Tuning the values of u and d may be a way to increase the cumulative rewards in SPAQL. However, it is clear in every case that the SPAQL agents find the location of the oil deposit. The difference between AQL cumulative rewards and SPAQL cumulative rewards at the end of training is due to the coarseness of the SPAQL partition and the structure of the reward function, which may greatly amplify very small differences in actions. For example, when using the Laplace survey function with $\lambda = 50$, choosing action c results in reward 1, while choosing action $0.99c$ (an error of 1%) results in a reward of ≈ 0.61, a decrease of almost 40%. With the quadratic survey function, the same relative error of 1% in the action leads to a 0.5% reduction in reward. This explains why SPAQL agents perform as well as AQL agents when using the quadratic survey function with $\lambda = 50$, but only achieve half of the average cumulative reward when using the Laplace survey function. These relatively small differences in rewards, however, are compensated by the lower number of arms.

4.1.4 Ambulance routing

Setup

This problem, described by Brotcorne et al. [43], is a stochastic variant of the previous one. The agent controls an ambulance that, at every time step, has to travel to where it is being requested. The agent is also given the option to relocate after fulfilling the request, paying a cost to do so. Sinclair et al. [13] use a transition kernel defined by $\mathbb{P}_h(x'|x,a) \sim \mathcal{F}_h$, where $\mathcal{F}_h$ denotes the request distribution for time step h. The reward function is $r_h(x'|x,a) = 1 - [c|x-a| + (1-c)|x'-a|]$, where $c \in [0,1]$ models the trade-off between the cost of relocation and the cost of traveling to serve the request.

It is not mandatory that $\mathcal{F}_h$ varies with the time step. However, if that is so, then in principle a time-invariant policy would not be a good choice for solving the problem. In this thesis only time-invariant scenarios ($\mathcal{F}_h := \mathcal{F}$) are considered. The experimental setups considered are

- $\mathcal{F} =$ Uniform$(0,1)$, for $c \in \{0, 0.25, 1\}$ (modelling disperse request distributions);
- $\mathcal{F} =$ Beta$(5,2)$, for $c \in \{0, 0.25, 1\}$ (where Beta(a,b) is the Beta probability distribution, modelling concentrated request distributions).

The optimal policy depends on the value of c. Sinclair et al. [13] suggest two heuristics for both extreme cases ($c \in \{0,1\}$). The "No Movement" heuristic is optimal when $c = 1$. In this case, the cost paid is only the cost to relocate, and therefore if the agent does not relocate it does not incur on any cost. This policy corresponds to the line $x = a$ in the state-action space. The "Mean"[2] heuristic is optimal when $c = 0$. In this case, the cost paid is only the cost of traveling to meet a request. Therefore, the agent should relocate to where the next request is most likely to appear. The empirical mean $\hat{\mu}$ of distribution $\mathcal{F}$ is a good estimator of this location. This policy corresponds to the horizontal line $a = \hat{\mu}$ in the state-action space.

Intuitively, the optimal policies for the values of c in between 0 and 1 will be a mix of these two optimal policies.

Results

The effect of the scaling parameter ξ in the ambulance problem is seen in Figure 4.3. Unlike the oil problem (Figure 4.1), SPAQL agents perform better than AQL ones, independently of the value of ξ. The exception is the case $c = 1$, where tuning of ξ allows the AQL agents to match the cumulative rewards of SPAQL agents. The average cumulative reward over the scaling values is always higher for the SPAQL agents, and the corresponding standard deviations are negligible when compared to those of AQL agents, meaning that in this problem the SPAQL agents have a very low sensitivity to the value of ξ.

The full experimental results are presented in Table 4.3, and the respective images in Appendix A. Unlike the oil problem, in the ambulance problem the SPAQL agents are clearly and consistently better than the AQL ones, reaching higher cumulative rewards earlier in training, with much fewer arms (in

[2]The original authors probably meant "Mean" instead of "Median".

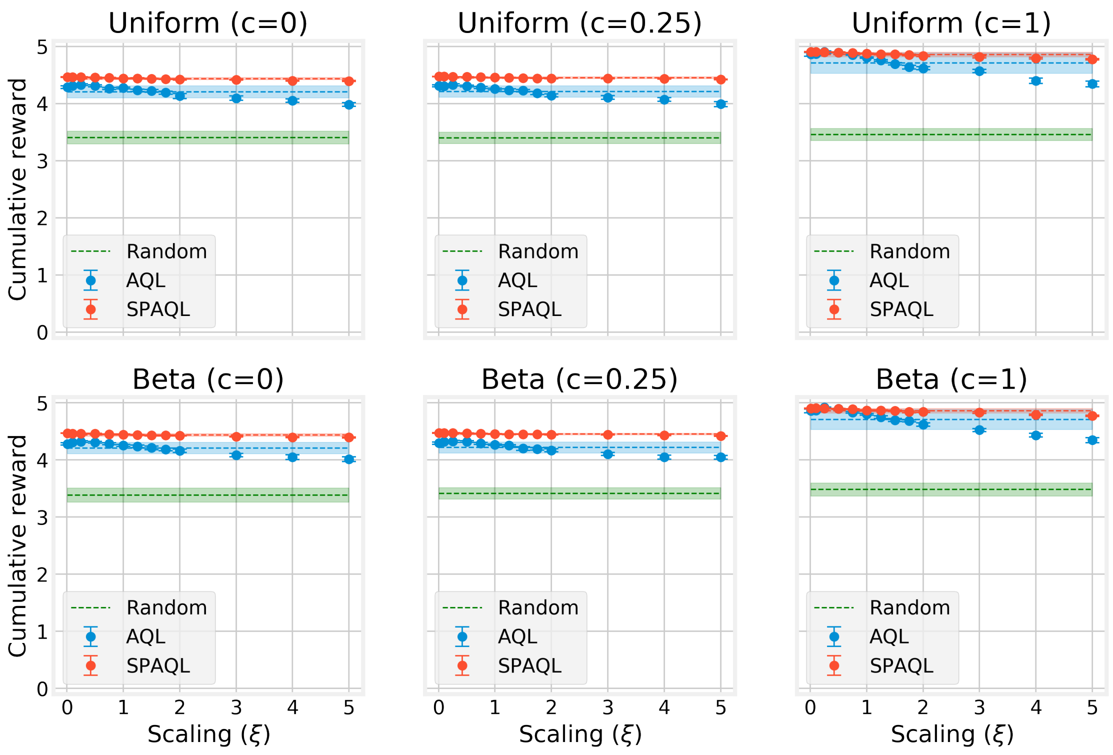

Figure 4.3: Comparison of the effect of different scaling parameter values on the average cumulative reward for the ambulance problem with arrival distributions described in Section 4.1.4. Each dot corresponds to the average of the rewards obtained by the 50 agents after 2000 training iterations, and the error bars display the corresponding 95% confidence interval. Dashed lines represent the average cumulative reward calculated over the 13 scaling values listed in Section 4.1.2, and the shaded areas represent the corresponding standard deviation.

some cases with only one fifth of the arms). This is consistent with the results in [27], who observed that Boltzmann exploration outperforms other exploration strategies in stochastic problems with rewards in the range $[0, 1]$. Figure 4.4 shows the cumulative rewards for an ambulance problem with a uniform arrival distribution and $c = 1$ (only relocation is penalized). Recall that the optimal policy is the “No Movement” one, which corresponds to the line $a = x$. This line appears finely partitioned, as would be expected. Keeping $c = 1$, but changing to a Beta$(5, 2)$ distribution (Figure A.12), it can be seen that the partition is now focused on the top right quadrant, where new arrivals are most likely to appear.

For this heuristic, the objective is to zero in on the diagonal as precisely as possible. As was already seen in the oil problem, AQL agents are better than SPAQL ones at this zeroing in. This is particularly seen in Figure A.12, where the SPAQL partition is coarser than the AQL partitions in time steps 3 and 5. It is this difference that allows AQL agents to catch up to SPAQL agents. However, despite both types of agents stabilizing at the same cumulative reward level, SPAQL agents use much fewer arms, which gives them a competitive advantage over AQL agents.

For $c = 0$, the action under the “Mean” heuristic would be $a = 0.5$ for the uniform arrival case, and $a \approx 0.7$ for the Beta$(5, 2)$ arrivals. These two policies correspond to horizontal lines, which could have been expected to be seen in the partitions in Figures A.7 and A.10, respectively. The neighborhoods

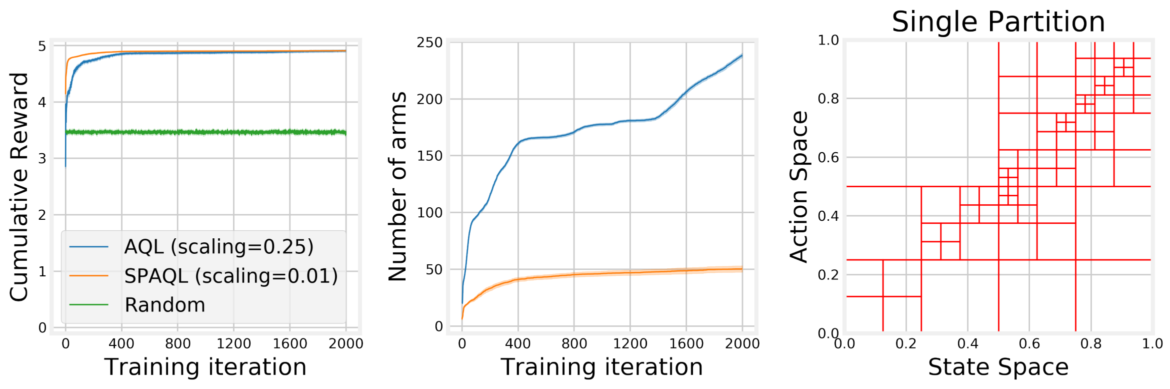

Figure 4.4: Average cumulative rewards, number of arms, and best SPAQL agent partition for the agents trained in the ambulance problem with uniform arrival distribution and reward function $r(x, a) = \max\{0, 1 - |x - a|\}$ ($c = 1$). Shaded areas around the solid lines represent the 95% confidence interval.

		c (Uniform)		
		0	0.25	1
Average cumulative reward	Random	3.41 ± 0.03	3.40 ± 0.03	3.46 ± 0.03
	AQL	4.33 ± 0.01	4.33 ± 0.02	$\mathbf{4.90 \pm 0.02}$
	SPAQL	$\mathbf{4.46 \pm 0.00}$	$\mathbf{4.47 \pm 0.00}$	$\mathbf{4.91 \pm 0.00}$
Average number of arms	AQL	244.10 ± 2.51	248.24 ± 2.18	238.40 ± 1.84
	SPAQL	32.26 ± 2.11	29.44 ± 2.03	50.32 ± 2.76

		c (Beta)		
		0	0.25	1
Average cumulative reward	Random	3.38 ± 0.03	3.41 ± 0.03	3.48 ± 0.03
	AQL	4.32 ± 0.02	4.32 ± 0.02	$\mathbf{4.92 \pm 0.01}$
	SPAQL	$\mathbf{4.47 \pm 0.01}$	$\mathbf{4.47 \pm 0.00}$	4.91 ± 0.00
Average number of arms	AQL	244.04 ± 3.14	250.1 ± 2.30	239.54 ± 1.95
	SPAQL	31.96 ± 2.21	29.56 ± 1.97	50.02 ± 1.28

Table 4.3: Average cumulative rewards and number of arms ($\pm 95\%$ confidence interval) in the ambulance problem. The best performing agent for each value of c is shown in bold. When the Welch t-test does not find a significant statistical difference between two performances, both are shown in bold.

of those lines are more partitioned than other areas of the state-action space, but not as much as the diagonal line in Figure 4.4. Even though the partitions shown are coarser than the corresponding ones shown in [13], the SPAQL agents reach higher rewards. This means that further partitions of the state-action space do not increase the average cumulative reward, and AQL agents may be needlessly over-partitioning the state-action space. This illustrates the remark made in Section 3.2.2 regarding prevention of over-partitioning.

Increasing the episode length

With an episode length of $H = 5$, it is easy for small differences in actions to yield noticeable differences in cumulative reward, as noted previously in the oil problem with Laplace survey function. Therefore, two agents of each algorithm were trained in an oil experiment with Laplace survey function with $\lambda = 1$, but

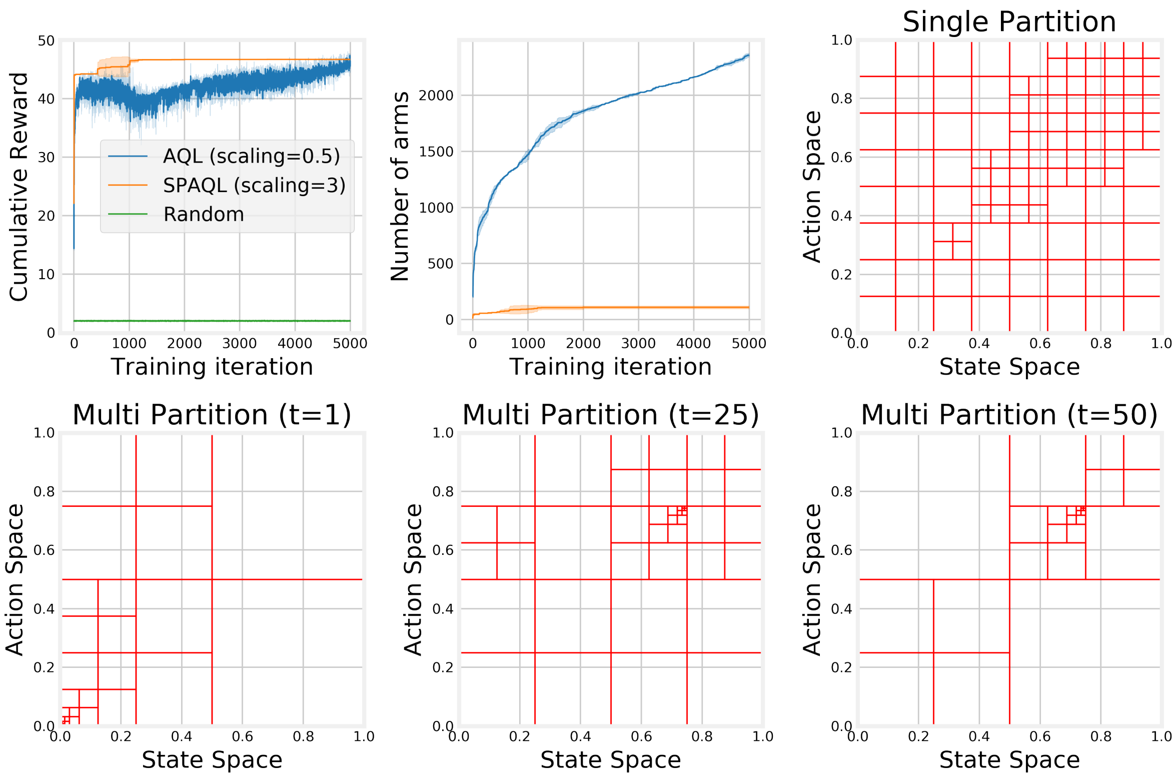

Figure 4.5: Average cumulative rewards, number of arms, and partitions for the best SPAQL and AQL agents trained in the oil problem with Laplace survey function ($\lambda = 1$) and horizon $H = 50$. Shaded areas around the solid lines represent the 95% confidence interval.

with $H = 50$ time steps. This means that the maximum cumulative reward scales from 4.25 to 49.25, and that the SPAQL policy will be greedy for more time steps, allowing for existing balls to split further. The result is shown in Figure 4.5. Both SPAQL agents stabilize at a maximum before 2000 training iterations. The AQL agents, on the other hand, have a much fuzzier training curve, take longer to reach the best rewards, and use almost ten times more arms. Furthermore, looking at the partition for the first time step, it is clear that the action therein taken is not the optimal one. Even so, it should also be noticed that, given enough training iterations, the information about the global optimum (which is already available at time step 25) would eventually propagate all way until time step 1, thus yielding the optimal policy. This is a clear advantage of AQL over SPAQL. Given a time-invariant problem, if one of the time steps finds the optimal actions, the update rule will eventually propagate this information to all other time steps. However, this comes at a high cost in terms of memory (number of arms) and time (training iterations).

It is important to highlight that, in the first 1000 training iterations, the cumulative reward for the AQL agents decreases steadily, while the number of arms is increasing. Even if given enough training iterations to allow the cumulative rewards to reach the maximum level, the arms associated with suboptimal policies will remain inside the partition, wasting memory and computational resources. This shows that, even for medium-sized episodes, AQL agents may not be a good choice for time-invariant problems, as growth in the number of arms is not necessarily followed by an increase in cumulative reward, and suboptimal arms remain in the policy even after a better one has been found.

4.1.5 Discussion

In almost all experiments, the SPAQL agents are less sensitive to the scaling parameter ξ (the exception being the oil problem with Laplace survey function). Recalling that the scaling parameter controls the value of the bonus term, which is introduced to deal with the exploitation/exploration trade-off, this lower sensitivity to the scaling parameter is probably due to SPAQL using Boltzmann exploration to deal with the same trade-off. The lower sensitivity to this parameter, along with the fact that most environments shown tend to favor lower scaling values, suggest that the scaling term could probably be removed from SPAQL (by setting it to 0). However, this would reduce SPAQL to Q-learning with Boltzmann exploration, which has not been proved to be sample-efficient. Therefore, the final recommendation of this section with regards to the choice of the scaling value ξ is to choose a non-zero value lower than $H/3$.

As a final note on the scaling parameter, recall Equation 3.5 for the expression of ξ in terms of the other training variables. Neglecting the term proportional to the Lipschitz constant L, setting $H = 5$, $K = 1000$, and considering a low value of δ (for example 0.1), the value obtained for the scaling parameter is larger than 80. This goes against the experimental evidence presented, which points towards smaller values of the scaling parameter being preferable over large ones, even in the AQL agents. It also exceeds H, which is the maximum value that $\mathbf{Q}(x, a)$ is allowed to have. A bonus term of 80 in an environment where the episode length is 5 would have no effect other than saturating the updates to the value function. In future work, it would be interesting to understand why this happens, and if the expression of the bonus term can be modified in order for it to be bounded by H.

4.2 Control problems

In this section, AQL, SPAQL, and SPAQL-TS are tested and compared on the `Pendulum` and `CartPole` problems of `OpenAI Gym` [44]. To close the section, these three algorithms are compared against TRPO [8], an RL algorithm known to perform well on these two problems [14].

Throughout this section, the control notation is adopted. Refer to Table 1.1 in Section 1.5 for the correspondence between the notation used in the previous sections and the notation used in the ones that follow.

4.2.1 Implementation

The state-action spaces for the `Pendulum` and `CartPole` problems are described in Sections 4.2.3 and 4.2.4, respectively. The data structures implemented to store the partitions used by the agents assume that the state-action pairs would be converted into a standard space (namely $[-1, 1]^3 \times [-1, 1]$ for the `Pendulum`, and $[-1, 1]^4 \times \{0, 1\}$ for the `CartPole`). The reason for using these standard spaces is that both the state-action space of the `Pendulum` and the state space of the `CartPole` problems can be tightly covered by a ball with radius 1 centered at the origin. The finite action set of the `CartPole` problem, $\{0, 1\}$, has to be dealt with separately. After checking if the state is within the selected ball, the implementation asserts that the action chosen is contained within the action set associated with the same ball. If the ball only has one possible action (the action set is a singleton), the state-action pair can only be contained within that ball if the action is the one associated with that ball. As in Section 4.1.1, a tree data structure is used to store the partitions, and ball selection is done using a recursive algorithm.

4.2.2 Procedure and parameters

Similarly to what was done in for the oil discovery and ambulance routing problems, the effect of the scaling parameter ξ on AQL and SPAQL is studied for the `Pendulum` and `CartPole` problems. For each algorithm and scaling value, 20 agents were trained for 100 iterations. By definition, the episode length on both systems is 200 [44]. The number of rollouts N used to evaluate each agent was set to 20. The scaling values used are based on the ones used in Section 4.1.2, scaled according to the change in horizon length (from $H = 5$ to $H = 200$),

$$\xi \in \{0.4, 4, 10, 20, 30, 40, 50, 60, 70, 80, 120, 160\}. \tag{4.3}$$

As a baseline, the effect of setting the scaling parameter ξ to 0 is also studied. This disables the upper confidence bounds, turning both algorithms into AQL algorithms with Boltzmann exploration.

For SPAQL and SPAQL-TS, the values of τ_{min}, u (SPAQL parameter, not to be confused with the control action), and d are set to 0.01, 2, and 0.8, respectively. For SPAQL-TS, the value of λ is set to 1.2. Table 4.4 contains the parameters used in the experiments reported in Figures 4.6 and 4.8. The average cumulative reward at the end of training was calculated over the several agents, along with the 95% confidence intervals.

After the values of ξ which resulted on higher average cumulative rewards were identified, a respective number of 20 agents were trained for 2000 iterations. In order to check for problem resolution (refer to Section 4.2.4), the `CartPole` agents were evaluated 100 times instead of 20. A summary of these parameters is listed in Table 4.5.

Parameter	Value
Episode length H	200
Number of scaling episodes (training iterations) K	100
Number of agents	20
Number of evaluation rollouts N	20

Table 4.4: Parameters for the scaling experiments in the `Pendulum` and `CartPole` problems.

Parameter	Value
Episode length H	200
Number of episodes (training iterations) K	2000
Number of agents	20
Number of evaluation rollouts N (`Pendulum`)	20
Number of evaluation rollouts N (`CartPole`)	100

Table 4.5: Parameters for the `Pendulum` and `CartPole` experiments after choosing ξ.

4.2.3 Pendulum

Setup

The objective is to drive a pendulum with length $l = 1$m and mass $m = 1$kg to the upright position by applying a torque on a joint at one of its ends. The state of the system is the angular position of the pendulum with the vertical axis, θ, and the angular velocity $\dot{\theta}$ of the pendulum. This can be written compactly as $x = [\theta, \dot{\theta}]^T$. The control action u is the value of the torque applied (which can be positive or negative, to indicate direction). The observation $o = [c_\theta, s_\theta, \dot{\theta}]^T$ is a $3D$ vector whose entries are the cosine (c_θ) and sine (s_θ) of the pendulum's angle, and the angular velocity [44]. The observation vector simulates the sensors available to measure the state variables of the system, and is part of the definition of the `OpenAI Gym` problem. There is no distinction between state and observation spaces in Chapter 3, but it is clear that Sinclair et al. [13] have the observation space in mind when they mention state space (since the samples used by the algorithms must necessarily come from the observation space). The angular velocity saturates at -8 and 8, and the control action saturates at -2 and 2. The cost to be minimized (which, from an RL point of view, is equivalent to a negative reward) is given by [44]

$$J = \sum_{t=1}^{H} \text{normalize}(\theta_t)^2 + 0.1\dot{\theta}_t^2 + 0.001(u_t^2), \tag{4.4}$$

where normalize$(\cdot)$ is a function that maps an angle into its principal argument (*i. e.*, a value in $]-\pi, \pi]$).

AQL and SPAQL were designed with the assumption that the rewards were in the interval $[0, 1]$ (which implies that $\mathbf{V}^k(o) \in [0, H]$, for all observations o). The effect of different cost structures is assessed by training the algorithms on the actual system (with rewards ranging from approximately -16 to 0) and on a system with rewards scaled to be in the interval $[0, 1]$. The UCB bonus term (Equation 3.4) can be expected to play an important part in the original problem, since it may balance the negative reward by making the term $r_t^k + b_\xi(v)$ in Equations 3.2 and 3.7 always positive during training.

The state-action space of this problem (as observed by the agent) is

$$\mathcal{S} \times \mathcal{A} = ([-1, 1] \times [-1, 1] \times [-8, 8]) \times [-2, 2]. \tag{4.5}$$

In order to simplify the implementation (Section 4.2.1), this space is mapped to a standard space ($[-1, 1]^4$). The angular velocity ($\dot{\theta} \in [-8, 8]$) is divided by 8, and then passed to the agent. The agent then picks a control action $u \in [-1, 1]$. Before this action is sent to the simulation, it is multiplied by 2.

The ∞ product metric (used by the agent) is written as

$$\mathcal{D}((x, u), (x', u')) = \max\{|c_\theta - c_{\theta'}|, |s_\theta - s_{\theta'}|, \frac{|\dot{\theta} - \dot{\theta}'|}{8}, |u - u'|\}. \tag{4.6}$$

Since the agent only sees the state-action pairs in the standard space $[-1, 1]^4$, the control action it picks is already contained in the interval $[-1, 1]$. It is the interface's task to scale this control action back to the action space of the problem. Since the control action is already correctly scaled, the component associated with the control action u in Equation 4.6 does not have to be divided by 2.

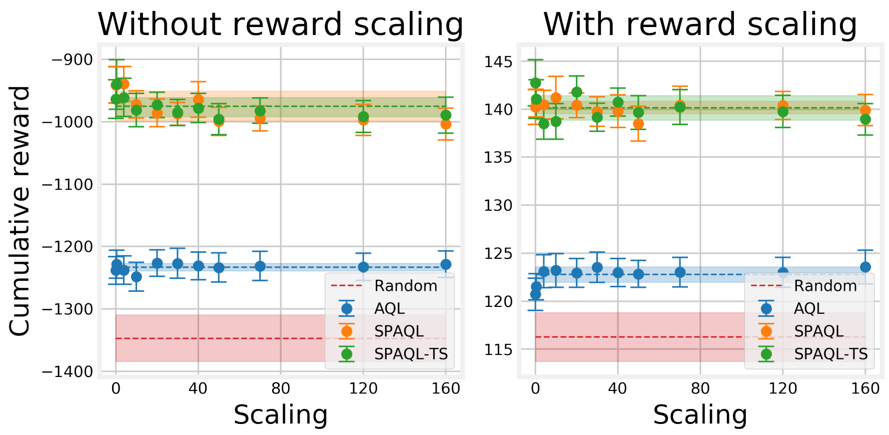

Figure 4.6: Comparison of different scaling parameter values on the average cumulative reward for the `Pendulum` system (with and without reward scaling). Each dot corresponds to the average of the rewards obtained by the 20 agents after 100 training iterations, and the error bars display the corresponding 95% confidence interval. Dashed lines represent the average cumulative reward calculated over the scaling values listed in Section 4.2.2, and the shaded areas represent the corresponding standard deviation.

For SPAQL-TS, the reference observation o_{ref} (denoted x_{ref} in Section 3.3) is set to $[1, 0, 0]^T$, corresponding to reference state $[\theta_{ref}, \dot{\theta}_{ref}]^T = [0, 0]^T$.

Results

The results in Figure 4.6 show that the scaling parameter has little impact on the three algorithms. Setting the scaling values ξ to 0 seems to be beneficial to both SPAQL and SPAQL-TS, meaning that Boltzmann exploration suffices to handle this problem. Although 100 iterations is a small number, it is already clear that both algorithms are learning (*i. e.*, the policies perform better than random ones), with SPAQL learning more than AQL. There is no difference between the two variants of SPAQL. Reward scaling does not greatly impact the results. Despite the difference in cumulative reward axes scales, the relative positions of the different algorithms in the plots with and without reward scaling remain the same. A possible explanation for the similarity between the learning curves with and without reward scaling is that the optimistic initialization of the Q-function manages to absorb the effect of the negative rewards.

The result of training for 2000 iterations without reward scaling is shown in Figure 4.7. The average performance and number of arms at the end of training are recorded in Table 4.6. Both AQL and SPAQL distance themselves from the random policy, with SPAQL and SPAQL-TS performing better than AQL. The average number of arms of the AQL agents (1.95×10^5) is two orders of magnitude higher than the average number of arms of the SPAQL and SPAQL-TS agents (1.28×10^3 and 1.08×10^3, respectively), despite its lower performance. This exemplifies the problem of using time-variant policies to deal with time-invariant problems. Applying the Welch test [21] to SPAQL and SPAQL-TS with a significance level of 5%, there is not enough evidence to support the claim that SPAQL is better than SPAQL-TS, and vice versa.

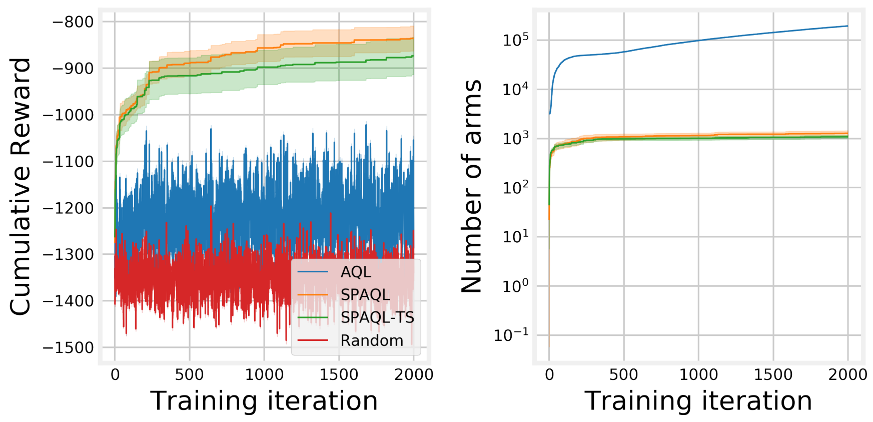

Figure 4.7: Average cumulative rewards and number of arms for the agents trained in the `Pendulum` system without reward scaling. Shaded areas around the solid lines represent the 95% confidence interval.

4.2.4 CartPole

Setup

The cartpole system consists of a pole with length $l = 1$m and mass $m = 0.1$kg attached to a cart of mass $m = 1$kg [45]. The state vector considered is the cart's position x and velocity $\dot{x}$, and the pole's angular position with the vertical axis θ and angular velocity $\dot{\theta}$. In order to distinguish the position x from the state vector, in this section the state vector is denoted as $\mathbf{x} = [x, \dot{x}, \theta, \dot{\theta}]^T$. Unlike the `Pendulum` system, in the `CartPole` system, the agent has direct access to the state variables. A simulation of the problem is termed an episode. An episode terminates when one of the following conditions is met [44]: the episode length (200) is reached; the cart position leaves the interval $[-2.4, 2.4]$ meters; the pole angle leaves the interval $[-12, 12]$ degrees.

The goal is to keep the simulation running for as long as possible (the 200 time steps). Only two actions are allowed ("push left" and "push right", with a force of 10N). The cost J is $-\sum_{t=1}^{T} 1$, where $T \leq H$ is the terminal time step [44]. This is equivalent to a reward of 1 for each time step, including the terminal one (T). According to the `OpenAI Gym` documentation, the problem is considered solved when the average cost is lower than or equal to -195 (or the cumulative reward is greater than or equal to 195) over 100 consecutive trials. The concept of "solved" is used as an entry requirement for the `OpenAI Gym` Leaderboard, which serves as a tool for comparing RL algorithms [44].

The state-action space of this problem (as observed by the agent) is [44]

$$\mathcal{S} \times \mathcal{A} = \left([-4.8, 4.8] \times \mathbb{R} \times \left[-\frac{24\pi}{180}, \frac{24\pi}{180} \right] \times \mathbb{R} \right) \times \{0, 1\}. \tag{4.7}$$

This space is mapped to a standard $[-1, 1]^4 \times \{0, 1\}$ space. For the cart position x and pole angle θ (first and third state variables, respectively), the mapping is done in a similar way as was done with the angular velocity of the `Pendulum` (divide by the maximum value). Their range is twice the one mentioned in the termination conditions ($[-2.4, 2.4]$ meters and $[-12\pi/180, 12\pi/180]$ radians, respectively), in order to capture the terminal observation [44]. Cart and pole velocities are allowed to assume any real value.

To bound them, the sigmoid function (shifted and rescaled) is used

$$\phi_m(y) = \frac{2}{1+\exp(-2\,m\,y)} - 1. \tag{4.8}$$

Parameter m controls the slope at the origin, and can be used to include some domain knowledge. For example, the largest linear distance that the cart can travel without terminating an episode is $2.4 - (-2.4) = 4.8$m, and the time discretization step for this system is 0.02. Therefore, the maximum absolute velocity which is reasonable to expect the cart to have is $4.8/0.02 = 240$m/s. This leads to $m_{\dot{x}} = 1/240$ as a reasonable value for m when bounding the cart velocity. Following a similar reasoning, a reasonable value for m when bounding the pole's angular velocity is $m_{\dot{\theta}} = 1/21$.

The ∞ product as metric for the `CartPole` problem is written as

$$\begin{aligned}\mathcal{D}((\mathbf{x}, u), (\mathbf{x}', u')) &= \\ &= \max\left\{ \frac{|x-x'|}{4.8}, |\phi_{m_{\dot{x}}}(\dot{x}) - \phi_{m_{\dot{x}}}(\dot{x'})|, \frac{|\theta-\theta'|}{24\pi/180}, |\phi_{m_{\dot{\theta}}}(\dot{\theta}) - \phi_{m_{\dot{\theta}}}(\dot{\theta}')|, 2 \times \mathbb{1}_{[u \neq u']} \right\},\end{aligned} \tag{4.9}$$

where $\mathbb{1}_{[A]}$ is the indicator function (defined previously in Section 4.1.3). For SPAQL-TS, the reference state $\mathbf{x}_{ref}$ is set to $[0, 0, 0, 0]^T$.

This problem differs from the previous one in two important aspects. First, the rewards are already normalized. Second, while in the `Pendulum` problem the set of possible actions is a real interval ($[-2, 2]$), in this problem the set of possible actions is finite (there are only two possible actions). In a certain sense, the `Pendulum` problem is akin to a regression problem, while the `CartPole` is akin to a classification one. While in regression problems the objective is to learn a continuous mapping from inputs to outputs, in classification problems the objective is to learn a partition of an input space. Given the nature of AQL and SPAQL, it would be reasonable to assume that they would perform better in the `CartPole` problem than in the `Pendulum` one. However, the `CartPole` problem also has a higher dimensional state-action space, which poses an additional challenge.

Results

The effect of the scaling parameter ξ on the `CartPole` problem is shown in Figure 4.8, and leads to two observations. The first one is that disabling the upper confidence bounds (setting $\xi = 0$) results in a great decrease in SPAQL performance, highlighted by the wide orange area. This negative effect is also seen in AQL. When $\xi = 0$, AQL performs worse than a random policy, and SPAQL performs as well as a random policy. Setting the scaling parameter to a relatively low value (such as 0.4) immediately solves this problem. SPAQL-TS is not affected by this. Besides this first outlier ($\xi = 0$) in AQL and SPAQL, the effect of the scaling parameter on the three algorithms is not significant. The experimental evidence presented here supports the suggestion from Section 4.1.5 of using non-zero scaling values lower than $H/3$.

The second observation regarding Figure 4.8 is that the AQL policies are barely distinguishable from

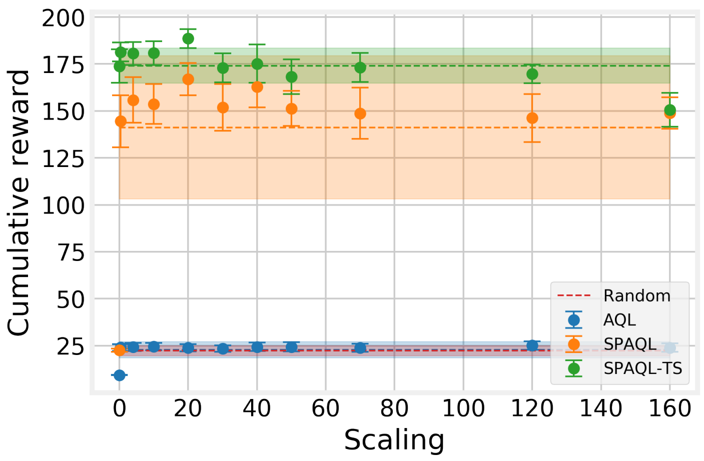

Figure 4.8: Comparison of different scaling parameter values on the average cumulative reward for the `CartPole` system. Each dot corresponds to the average of the rewards obtained by the 20 agents after 100 training iterations, and the error bars display the corresponding 95% confidence interval. Dashed lines represent the average cumulative reward calculated over the scaling values listed in Section 4.2.2, and the shaded areas represent the corresponding standard deviation.

random ones, independently of ξ. This may be due to the curse of dimensionality. With a 5D state-action space, every time a split occurs, 32 new balls are created[3]. The updates to the values of the Q-function of these balls become independent. Updating one of those 32 balls will lead to learning something related to that state. However, the remaining 31 balls will remain random (equally likely to pick one of the two actions). This problem can be considered as a variant of the Coupon Collector Problem [46]. Assuming that the state-action space is visited uniformly at random, we wish to know how many visits are needed in order to be sure that each ball was visited at least once. This number is bounded[4] by $\Theta(n \log(n))$, where n is the number of balls in the partition. Given that n grows exponentially with the dimension of the state-action space, the number of visits required will also grow exponentially. In AQL, we need to multiply this number of visits by H, since a different partition is kept for each time step, and they are updated independently. This argument seems to indicate that the practical sample efficiency of AQL does not scale well with the dimension of state-action space.

The training curves for the three algorithms are shown in Figure 4.9. The average performance and number of arms at the end of training are recorded in Table 4.7. While in the `Pendulum` system the AQL agents were able to distinguish themselves from the random policy, for the `CartPole` system, this does not happen. There are fluctuations in the performance of AQL, although no improvement is permanent. On the other hand, the SPAQL and SPAQL-TS agents quickly achieve average cumulative rewards of approximately 193 and 199, respectively (out of 200.00). Considering that these agents were evaluated 100 times, the system is solved by SPAQL-TS. Manually inspecting the performances of the twenty individual agents stored at the end of training, it is seen that only one SPAQL agent solved the system (average cumulative reward higher than 195). On the other hand, seventeen SPAQL-TS agents solved it (out of which fourteen scored the maximum average performance of 200). The Welch test concludes

[3]In fact, since there are only two possible actions, after the first split, the child balls will only be split into 16 new ones.
[4]$f(n) = \Theta(g(n))$ means that f is asymptotically bounded above and below by g.

that SPAQL-TS is better than SPAQL at a significance level of 5% with a p-value of the order of 10^{-6}. SPAQL-TS agents end the training with around twice the number of arms of SPAQL (1.29×10^3 and 5.58×10^2, respectively). The AQL agents finish training with 25 times more arms (3.30×10^4).

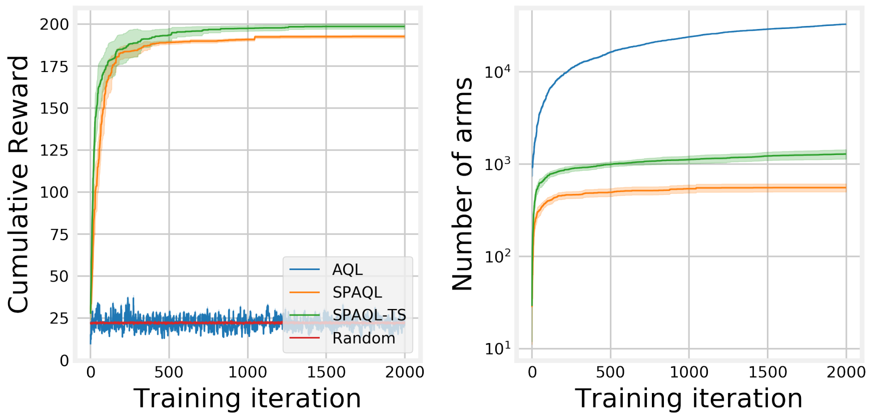

Figure 4.9: Average cumulative rewards and number of arms for the agents trained in the `CartPole` system. At the end of each iteration the agents were evaluated 100 times. Shaded areas around the solid lines represent the 95% confidence interval.

4.2.5 Comparing AQL, SPAQL, and TRPO

	`Pendulum`	
	Average cumulative reward at the end of training	Number of arms at the end of training
AQL	-1287.31 ± 5.79	195032.75 ± 575.81
SPAQL	-835.99 ± 26.91	1277.50 ± 168.74
SPAQL-TS	-873.40 ± 39.94	1082.50 ± 91.42
Random	-1340.43 ± 6.24	
TRPO	$\mathbf{-176.76 \pm 15.79}$	

Table 4.6: Average cumulative rewards and number of arms ($\pm 95\%$ confidence interval) for the different agents at the end of training in the `Pendulum` system. The best performance is shown in bold.

	`CartPole`	
	Average cumulative reward at the end of training	Number of arms at the end of training
AQL	22.40 ± 0.60	32988.9 ± 186.61
SPAQL	192.57 ± 0.94	557.75 ± 55.10
SPAQL-TS	$\mathbf{198.53 \pm 1.55}$	1289.75 ± 151.22
Random	22.15 ± 0.64	
TRPO	$\mathbf{197.27 \pm 3.17}$	

Table 4.7: Average cumulative rewards and number of arms ($\pm 95\%$ confidence interval) for the different agents at the end of training in the `CartPole` system. The best performance is shown in bold. When the Welch t-test does not find a significant statistical difference between two performances, both are shown in bold.

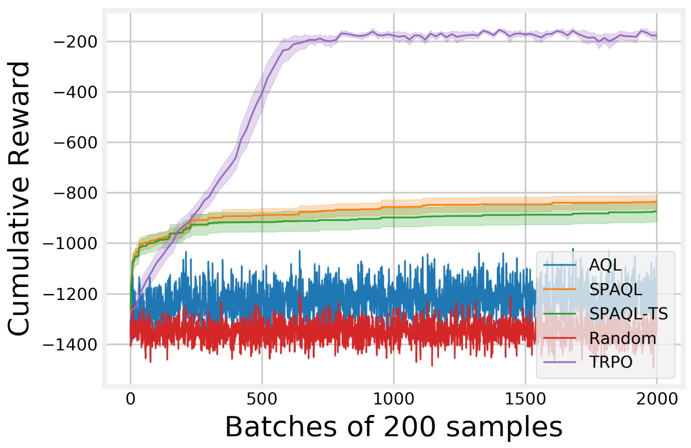

Figure 4.10: Average cumulative rewards and number of arms for the agents trained in the `Pendulum` system without reward scaling. Shaded areas around the solid lines represent the 95% confidence interval.

Trust region policy optimization (TRPO) is an RL algorithm that is known to perform well in control problems [8, 14]. Instead of learning a value function and then choosing actions greedily, it learns a policy directly using neural networks. While this allows TRPO to perform well, and to generalize to unseen situations, the policies that it learns are not interpretable, unlike the ones learned by SPAQL (which are tables of actions associated with states). This trade-off between performance and interpretability justifies the comparison between these two algorithms.

The implementation of TRPO provided by `Garage` [47] was used. The parameters used in the examples provided with the source were kept. Average cumulative reward estimates were computed over 20 different random seeds, with policies trained for 100 iterations, using 4000 samples per iteration. This gives a total of 400 thousand training samples, the same amount used when training AQL and SPAQL agents (2000 training iterations, with 200 samples collected in each). Figures 4.10 and 4.11 show the learning curves for random, AQL, SPAQL, SPAQL-TS, and TRPO agents, as a function of the number of samples used (in batches of 200 samples).

For the `Pendulum` system (Figure 4.10, Table 4.6) the SPAQL variants take the lead in cumulative reward increase, but are outrun by TRPO after 200 batches (the number of samples exceeds 40 thousand). The TRPO agents continue learning until stabilizing the cumulative reward around -200.

In the `CartPole` system (Figure 4.11, Table 4.7), both SPAQL variants use the initial batches more efficiently. TRPO catches up around batch 200 (40 thousand samples). In the end, TRPO agents perform as well as SPAQL-TS agents (the Welch test does not find enough evidence to prove that one is better than the other).

TRPO uses a neural network to approximate the policy. Since neural networks are universal function approximators, it was expectable to see them perform better than AQL and SPAQL in the `Pendulum` problem, which is similar to a regression problem. Trying to solve the `Pendulum` using AQL or SPAQL is similar to solving a regression problem using a decision tree. Although possible, the resulting tree may be very large.

For the `CartPole` problem, which is similar to a classification problem, the neural network requires

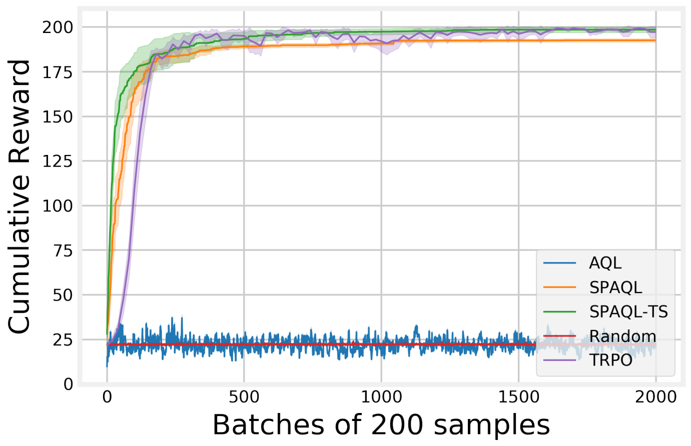

Figure 4.11: Average cumulative rewards and number of arms for the agents trained in the `CartPole` system. Shaded areas around the solid lines represent the 95% confidence interval.

more samples than SPAQL to learn an adequate state-action space partition. However, even though it manages to learn this partition, it is not straightforward to convert the neural network to an interpretable representation such as a table. SPAQL policies are tables, and therefore the policy obtained at the end of training is already interpretable, should it be required.

Chapter 5

Conclusions

This thesis introduces single-partition adaptive Q-learning (SPAQL), an improved version of adaptive Q-learning (AQL) tailored for learning time-invariant policies in reinforcement learning (RL) problems. In order to balance exploration and exploitation, SPAQL uses Boltzmann exploration with a cyclic temperature schedule in addition to upper confidence bounds (UCB). As proof of concept, SPAQL is first evaluated in two example problems: oil discovery and ambulance routing.

SPAQL with terminal state (SPAQL-TS), an improved version of SPAQL that borrows concepts from control theory, is also introduced. Both SPAQL and SPAQL-TS are evaluated in two control problems: the `Pendulum` and the `CartPole`.

5.1 Achievements

Experiments show that, with very little parameter tuning, SPAQL performs satisfactorily in the problems where AQL was originally tested (oil discovery and ambulance routing), resulting in partitions with a lower number of arms, and requiring fewer training iterations to converge. The two problems studied can be seen as a deterministic and a stochastic variant of the same problem, with SPAQL performing as well as or better than AQL on half of the problems.

AQL and SPAQL are also tested on two classical control problems: the `Pendulum` and the `CartPole`. Both SPAQL and SPAQL-TS outperform AQL in the control problems. Moreover, SPAQL-TS manages to solve the `CartPole` problem, showing higher sample-efficiency than trust region policy optimization (TRPO, a standard RL method for solving control problems) when processing the first batches of samples.

Combining the empirical evidence presented, it can be claimed that SPAQL and SPAQL-TS have a higher sample-efficiency than AQL in time-invariant problems, and thus are a relevant contribution to the field of sample efficient model-free RL algorithms.

5.2 Future Work

There are several possible directions for further work. Both AQL and SPAQL can be further modified in several ways. The next natural modification would be to run several episodes during each training iteration, instead of only one. In SPAQL, this would increase the exploitatory behavior of the algorithm, possibly solving the cumulative reward difference seen in the oil problem with Laplace survey function. However, this would introduce another parameter (the number of episodes to run during each training iteration) into the algorithm, thus increasing the effort required for tuning. Currently, the parameters of SPAQL are highly conjugate with the episode length. It would also be interesting to study automatic ways of setting u and d given H. Similarly to what was done in this thesis for ξ, an empirical study regarding the effect of tuning parameters u and d on SPAQL performance could be done.

Another parameter that may have a relevant impact is the number of evaluation rollouts, N. If this number is too low, the performance estimates may be inaccurate, and lead SPAQL to bad decisions when updating the stored partition. However, a larger N also means more training time. On the other hand, as the state-action space is partitioned and the radius of the balls decreases, lower values of N are required to obtain accurate estimates. Therefore, it would also be interesting to see if it is possible to set N automatically based on the rollouts themselves.

Finally (from an algorithm design perspective), a formal analysis of the complexity of SPAQL would allow a rigorous comparison in terms of sample efficiency to AQL or similar algorithms.

Regarding experimental further work, the empirical evaluation in this thesis is not exhaustive. The `OpenAI Gym` implements many more control problems, with more challenging state-action spaces. A possible direction for further work is to test SPAQL and SPAQL-TS on more problems. This would result in more interpretable policies for these problems, that could provide insights for future research in control. Another possible direction is to study the causes behind the higher sample-efficiency of SPAQL and SPAQL-TS when processing the first training batches, compared to TRPO (refer to Figures 4.10 and 4.11). This may help pave the road for more sample-efficient control algorithms.

The `Pendulum` system has a continuous action space, while the `CartPole` system has a finite one. The results in the `Pendulum` show that, given their current status, neither SPAQL nor SPAQL-TS are good choices when controlling systems with a continuous action space. It is preferable to use them in control problems where there is a finite set of control actions, such as the `CartPole`. One of the reasons for this is that updates to a ball do not affect its neighbors. Propagating updates to the neighboring balls could be a way to improve generalization of SPAQL policies. Another possible approach to get SPAQL to work on continuous action spaces is to partition only the state space, and for each state space part learn a linear mapping from states to actions. Another possible line of work is to combine the adaptive partitioning of SPAQL with other approaches that use interpretable nonlinear function approximators, like the symbolic regression methodology proposed by Kubalík et al. [15].

Finally, although it is straightforward to convert the partitions learned by SPAQL into tables, it would be interesting to implement an automatic translator from partition to decision tree, which would enable a clear and simplified setting to analyze the policies.

Bibliography

[1] R. Sutton and A. Barto. *Reinforcement learning: an introduction*. The MIT Press, Cambridge, Massachusetts, 2nd edition, 2018. ISBN 978-0262039246.

[2] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah. A survey of deep learning applications to autonomous vehicle control. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–22, 2020. doi: 10.1109/TITS.2019.2962338. URL `https://doi.org/10.1109/TITS.2019.2962338`.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404. URL `https://doi.org/10.1126/science.aar6404`.

[4] OpenAI, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019. URL `http://arxiv.org/abs/1912.06680`.

[5] S. M. Shortreed, E. B. Laber, D. J. Lizotte, T. S. Stroup, J. Pineau, and S. A. Murphy. Informing sequential clinical decision-making through reinforcement learning: an empirical study. *Mach. Learn.*, 84(1-2):109–136, 2011. doi: 10.1007/s10994-010-5229-0. URL `https://doi.org/10.1007/s10994-010-5229-0`.

[6] T. Mu, S. Wang, E. Andersen, and E. Brunskill. Combining adaptivity with progression ordering for intelligent tutoring systems. In R. Luckin, S. Klemmer, and K. R. Koedinger, editors, *Proceedings of the Fifth Annual ACM Conference on Learning at Scale, London, UK, June 26-28, 2018*, pages 15:1–15:4. ACM, 2018. doi: 10.1145/3231644.3231672. URL `https://doi.org/10.1145/3231644.3231672`.

[7] M. P. Deisenroth and C. E. Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 465–472. Omnipress, 2011. URL `https://icml.cc/2011/papers/323_icmlpaper.pdf`.

[8] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In F. R. Bach and D. M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1889–1897. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/schulman15.html.

[9] Y. Yu. Towards sample efficient reinforcement learning. In J. Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5739–5743. ijcai.org, 2018. doi: 10.24963/ijcai.2018/820. URL https://doi.org/10.24963/ijcai.2018/820.

[10] C. Jin, Z. Allen-Zhu, S. Bubeck, and M. I. Jordan. Is Q-learning provably efficient? In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 4868–4878, 2018. URL http://papers.nips.cc/paper/7735-is-q-learning-provably-efficient.

[11] I. Osband, B. V. Roy, and Z. Wen. Generalization and exploration via randomized value functions. In M. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2377–2386. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/osband16.html.

[12] Z. Song and W. Sun. Efficient model-free reinforcement learning in metric spaces. *CoRR*, abs/1905.00475, 2019. URL http://arxiv.org/abs/1905.00475.

[13] S. R. Sinclair, S. Banerjee, and C. L. Yu. Adaptive discretization for episodic reinforcement learning in metric spaces. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3):55:1–55:44, 2019. doi: 10.1145/3366703. URL https://doi.org/10.1145/3366703.

[14] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. In M. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1329–1338. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/duan16.html.

[15] J. Kubalík, J. Zegklitz, E. Derner, and R. Babuska. Symbolic regression methods for reinforcement learning. *CoRR*, abs/1903.09688, 2019. URL http://arxiv.org/abs/1903.09688.

[16] W. B. Powell. AI, OR and Control Theory: A Rosetta Stone for Stochastic Optimization. Technical report, Princeton University, 2012.

[17] C. Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

doi: 10.2200/S00268ED1V01Y201005AIM009. URL https://doi.org/10.2200/S00268ED1V01Y201005AIM009.

[18] E. Lecarpentier and E. Rachelson. Non-stationary markov decision processes, a worst-case approach using model-based reinforcement learning. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 7214–7223, 2019. URL http://papers.nips.cc/paper/8942-non-stationary-markov-decision-processes-a-worst-case-approach-using-model-based-reinforcement-learning.

[19] L. Busoniu, R. Babuska, D. Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Automation and Control Engineering. CRC Press, 2010. ISBN 9781439821091.

[20] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In S. A. McIlraith and K. Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3207–3214. AAAI Press, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16669.

[21] C. Colas, O. Sigaud, and P. Oudeyer. How many random seeds? Statistical power analysis in deep reinforcement learning experiments. *CoRR*, abs/1806.08295, 2018. URL http://arxiv.org/abs/1806.08295.

[22] A. Touati, A. A. Taïga, and M. G. Bellemare. Zooming for efficient model-free reinforcement learning in metric spaces. *CoRR*, abs/2003.04069, 2020. URL https://arxiv.org/abs/2003.04069.

[23] G. Neustroev and M. M. de Weerdt. Generalized optimistic Q-learning with provable efficiency. In A. E. F. Seghrouchni, G. Sukthankar, B. An, and N. Yorke-Smith, editors, *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, pages 913–921. International Foundation for Autonomous Agents and Multiagent Systems, 2020. URL https://dl.acm.org/doi/abs/10.5555/3398761.3398868.

[24] Y. Wang, K. Dong, X. Chen, and L. Wang. Q-learning with UCB exploration is sample efficient for infinite-horizon MDP. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=BkglSTNFDB.

[25] N. Cesa-Bianchi, C. Gentile, G. Neu, and G. Lugosi. Boltzmann exploration done right. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural*

Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA, pages 6284–6293, 2017. URL http://papers.nips.cc/paper/7208-boltzmann-exploration-done-right.

[26] V. Kuleshov and D. Precup. Algorithms for multi-armed bandit problems. *CoRR*, abs/1402.6028, 2014. URL http://arxiv.org/abs/1402.6028.

[27] A. D. Tijsma, M. M. Drugan, and M. A. Wiering. Comparing exploration strategies for Q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016, Athens, Greece, December 6-9, 2016*, pages 1–8. IEEE, 2016. doi: 10.1109/SSCI.2016.7849366. URL https://doi.org/10.1109/SSCI.2016.7849366.

[28] M. L. Littman. *Algorithms for Sequential Decision-Making*. PhD thesis, Brown University, USA, 1996. AAI9709069.

[29] K. Asadi and M. L. Littman. An alternative softmax operator for reinforcement learning. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 243–252. PMLR, 2017. URL http://proceedings.mlr.press/v70/asadi17a.html.

[30] L. Pan, Q. Cai, Q. Meng, W. Chen, and L. Huang. Reinforcement learning with dynamic boltzmann softmax updates. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020 [scheduled for July 2020, Yokohama, Japan, postponed due to the Corona pandemic]*, pages 1992–1998. ijcai.org, 2020. doi: 10.24963/ijcai.2020/276. URL https://doi.org/10.24963/ijcai.2020/276.

[31] L. N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017, Santa Rosa, CA, USA, March 24-31, 2017*, pages 464–472. IEEE Computer Society, 2017. doi: 10.1109/WACV.2017.58. URL https://doi.org/10.1109/WACV.2017.58.

[32] L. N. Smith and N. Topin. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120, 2017. URL http://arxiv.org/abs/1708.07120.

[33] G. Huang, Y. Li, G. Pleiss, Z. Liu, J. E. Hopcroft, and K. Q. Weinberger. Snapshot ensembles: Train 1, get M for free. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=BJYwwY9ll.

[34] H. Fu, C. Li, X. Liu, J. Gao, A. Çelikyilmaz, and L. Carin. Cyclical annealing schedule: A simple approach to mitigating KL vanishing. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 240–250. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1021. URL https://doi.org/10.18653/v1/n19-1021.

[35] N. Mu, Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney. Parameter re-initialization through cyclical batch size schedules. *CoRR*, abs/1812.01216, 2018. URL http://arxiv.org/abs/1812.01216.

[36] I. Loshchilov and F. Hutter. SGDR: stochastic gradient descent with warm restarts. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=Skq89Scxx.

[37] A. Yamaguchi, J. Takamatsu, and T. Ogasawara. Learning strategy fusion to acquire dynamic motion. In *11th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2011), Bled, Slovenia, October 26-28, 2011*, pages 247–254. IEEE, 2011. doi: 10.1109/Humanoids.2011.6100853. URL https://doi.org/10.1109/Humanoids.2011.6100853.

[38] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res.*, 4:237–285, 1996. doi: 10.1613/jair.301. URL https://doi.org/10.1613/jair.301.

[39] A. S. Polydoros and L. Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *J. Intell. Robotic Syst.*, 86(2):153–173, 2017. doi: 10.1007/s10846-017-0468-y. URL https://doi.org/10.1007/s10846-017-0468-y.

[40] L. Busoniu, T. de Bruin, D. Tolic, J. Kober, and I. Palunko. Reinforcement learning for control: Performance, stability, and deep approximators. *Annu. Rev. Control.*, 46:8–28, 2018. doi: 10.1016/j.arcontrol.2018.09.005. URL https://doi.org/10.1016/j.arcontrol.2018.09.005.

[41] J. P. Araújo, M. Figueiredo, and M. A. Botto. Single-partition adaptive Q-learning. *CoRR*, abs/2007.06741, 2020. URL https://arxiv.org/abs/2007.06741.

[42] W. Mason and D. J. Watts. Collaborative learning in networks. *Proceedings of the National Academy of Sciences*, 109(3):764–769, 2011. doi: 10.1073/pnas.1110069108. URL https://doi.org/10.1073/pnas.1110069108.

[43] L. Brotcorne, G. Laporte, and F. Semet. Ambulance location and relocation models. *Eur. J. Oper. Res.*, 147(3):451–463, 2003. doi: 10.1016/S0377-2217(02)00364-8. URL https://doi.org/10.1016/S0377-2217(02)00364-8.

[44] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

[45] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Syst. Man Cybern.*, 13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077. URL https://doi.org/10.1109/TSMC.1983.6313077.

[46] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discret. Appl. Math.*, 39(3):207–229, 1992. doi: 10.1016/0166-218X(92)90177-C. URL https://doi.org/10.1016/0166-218X(92)90177-C.

[47] The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. `https://github.com/rlworkgroup/garage`, 2019.

Appendix A

Experimental results and figures for the oil and ambulance problems

In this section, the learning curves for the best agents found during the scaling experiments are shown. Each curve shows the average cumulative reward and the 95% confidence interval. The scaling values are identified in the respective legend. The average number of arms is also shown, along with the state-action space partition for the best SPAQL agent at the end of training. Finally, the AQL agent partition at time steps 1, 3, and 5 is also shown.

Each section contains some brief comments to the images, which complement the analysis in Sections 4.1.3 and 4.1.4.

A.1 Oil Problem with Quadratic Survey Function

Starting with $\lambda = 1$ (Figure A.1), it is clear that the best SPAQL agent did not exploit the optimal point of 0.75, and over partitioned uninteresting regions of the space. Even so, the SPAQL agents managed to reach a cumulative reward similar to that of AQL, with one third of the number of arms of the AQL agents. The high rewards over a large area are probably a cause for this lack of exploitation, since, when the value of λ is increased (concentrating the rewards), SPAQL agents partition the neighborhood of the optimal point more finely. Another possible cause is that the best performance currently stored happened by chance to be unexpectedly high. A way to prevent this from happening is to increase the number of evaluation rollouts N. In this experiment, $N = 20$ rollouts were used to evaluate the agents, but a higher number would lead to better estimates. The AQL agent learned that the optimal action given initial state $x = 0$ is $a = 0.75$, and then learned to hold the action $a = 0.75$. The partitions at time instants 3 and 5 are very similar, as would be expected given that this is a stationary problem. It is this sort of partition duplication that is avoided by keeping only one partition.

Moving to the case when $\lambda = 10$ (Figure A.2), the average cumulative rewards at the end of training for both algorithms are barely distinguishable. The SPAQL agents converge earlier than the AQL agents, and with fewer arms. The best SPAQL partition shows a better exploitation of the optimal point, compared with the case when $\lambda = 1$. Once again very similar partitions are seen at time steps 3 and 5 in the best AQL agent. The increase in the number of AQL arms around training iteration 4100 might seem to indicate that AQL agents have still not converged. However, the training curves indicate that both agents are very close to the maximum, meaning that the extra arms in the AQL agent correspond to new splits which are approximating the location of the optimal point to an accuracy of millimeters. The number of arms can grow indefinitely in order to approximate all of the decimal places in the floating point representation of $0.7 + \pi/60$, and therefore should not be considered when assessing convergence of the algorithm.

Finally, Figure A.3 shows the results when $\lambda = 50$. These results are similar to the case of $\lambda = 10$. The concentrated rewards have lead to an even better exploitation of the optimal point by the SPAQL agent.

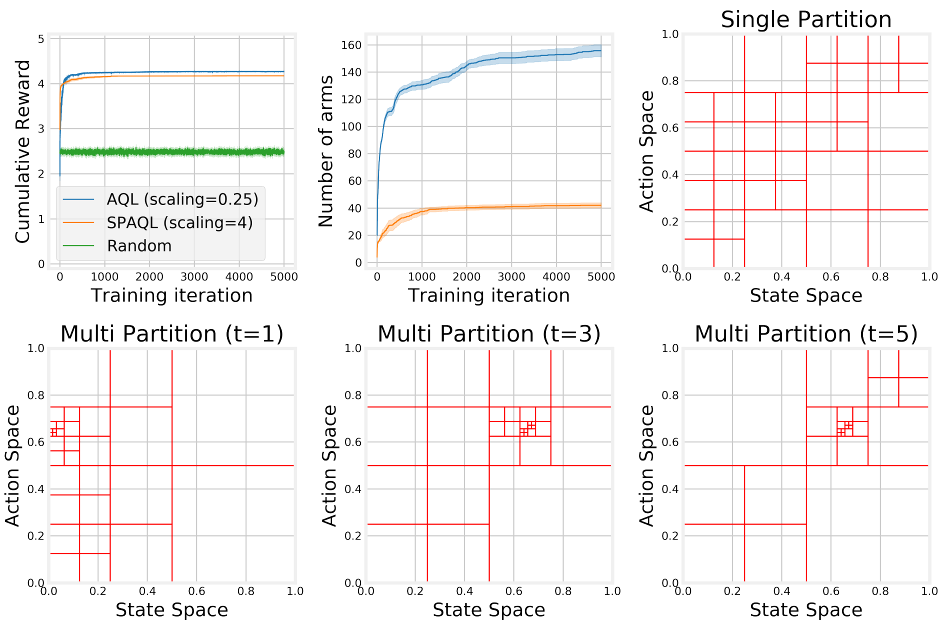

Figure A.1: Comparison of the algorithms on the oil problem with quadratic survey function ($\lambda = 1$).

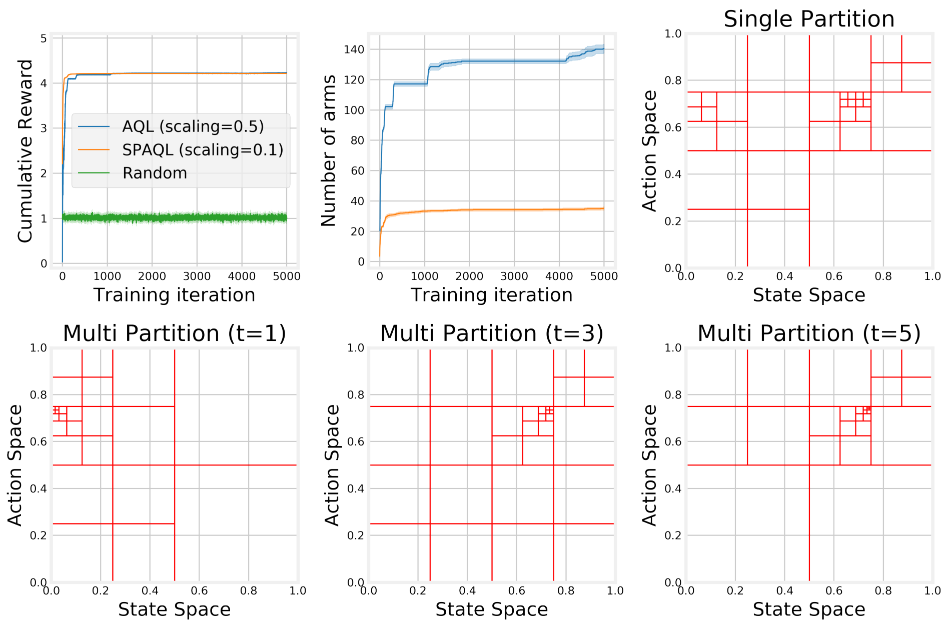

Figure A.2: Comparison of the algorithms on the oil problem with quadratic survey function ($\lambda = 10$).

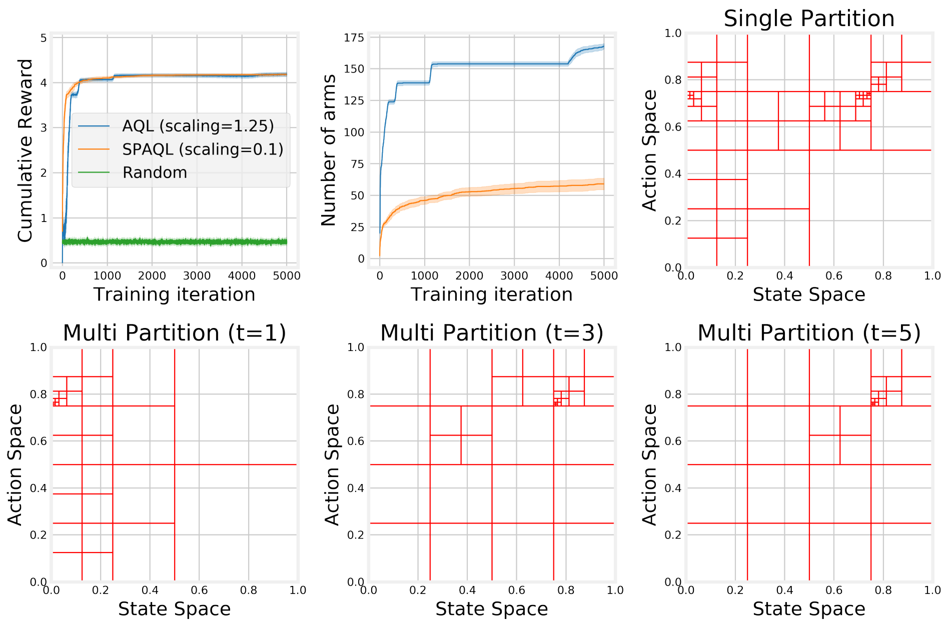

Figure A.3: Comparison of the algorithms on the oil problem with quadratic survey function ($\lambda = 50$).

A.2 Oil Problem with Laplace Survey Function

When using the Laplace survey function rewards become more concentrated than when using the quadratic survey function. This is the main reason behind the differences in cumulative reward at the end of training. Considering the case with $\lambda = 1$ (Figure A.4), although the SPAQL agents stabilize at slightly lower rewards, they still manage to achieve them with a much smaller number of arms. The partition for the best SPAQL agent shows a lack of exploitation of the optimal point, as was the case with the quadratic survey function with $\lambda = 1$ (Figure A.1). The partitions for time steps 3 and 5 in the AQL agent are once again barely distinguishable.

Moving to $\lambda = 10$ (Figure A.5), training has become more difficult for both types of agents, when compared with the case when $\lambda = 1$. The partition of the SPAQL agent indicates a lot of exploration from the initial state ($x = 0$), along with a fine partition around the optimal point, meaning that, once found, it was exploited. Since the rewards are more concentrated, and the AQL agent partitions the neighborhood of the optimal point more finely, it ends up collecting higher rewards than the SPAQL agent, which keeps a coarser partition around the optimal point. However, it should be noted that, for all practical purposes, the SPAQL agent managed to find the approximate location of the optimal point, using fewer arms than the AQL agent.

Finally, Figure A.6 shows the results when $\lambda = 50$. This is the hardest problem, with the most concentrated rewards. Both algorithms locate the optimal point, but the AQL agents exploit it much more, leading to higher rewards. However, in the process, it partitioned a lot the individual partitions, leading to an average number of arms six times higher than the ones in the SPAQL partition, which has implications regarding the amount of resources required to store all the partitions of the agent.

In the two latter cases ($\lambda \in \{10, 50\}$), the 5000 iterations are clearly not enough for the AQL and SPAQL agents to converge. However, if finding the approximate location of the oil deposit is considered as the objective of this problem, then the 5000 iterations were enough. The cumulative rewards are lower when compared to other episodes due to the high concentration of rewards around the optimal point, and the lack of reward signal in the remaining state-action space.

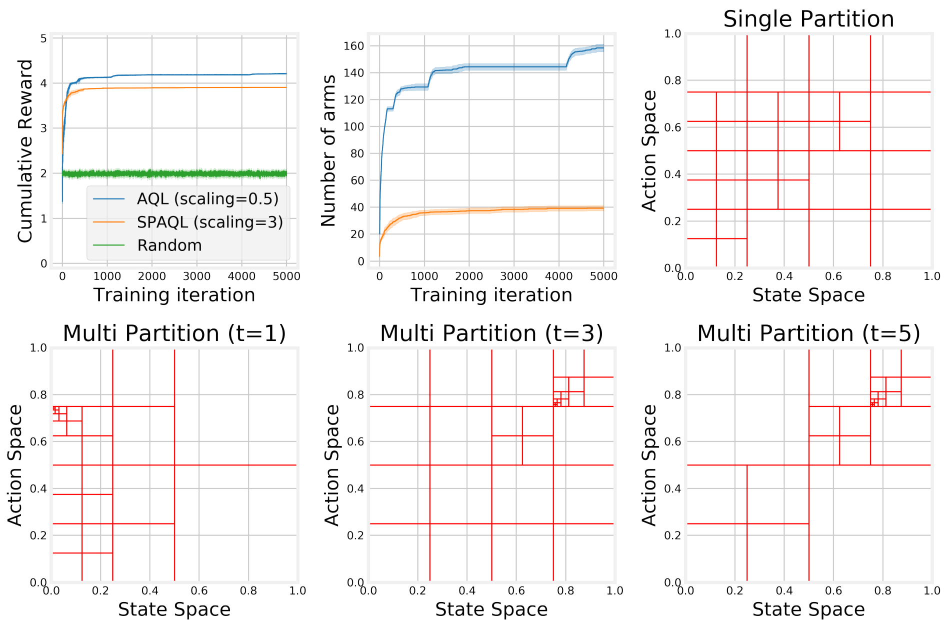

Figure A.4: Comparison of the algorithms on the oil problem with Laplace survey function ($\lambda = 1$).

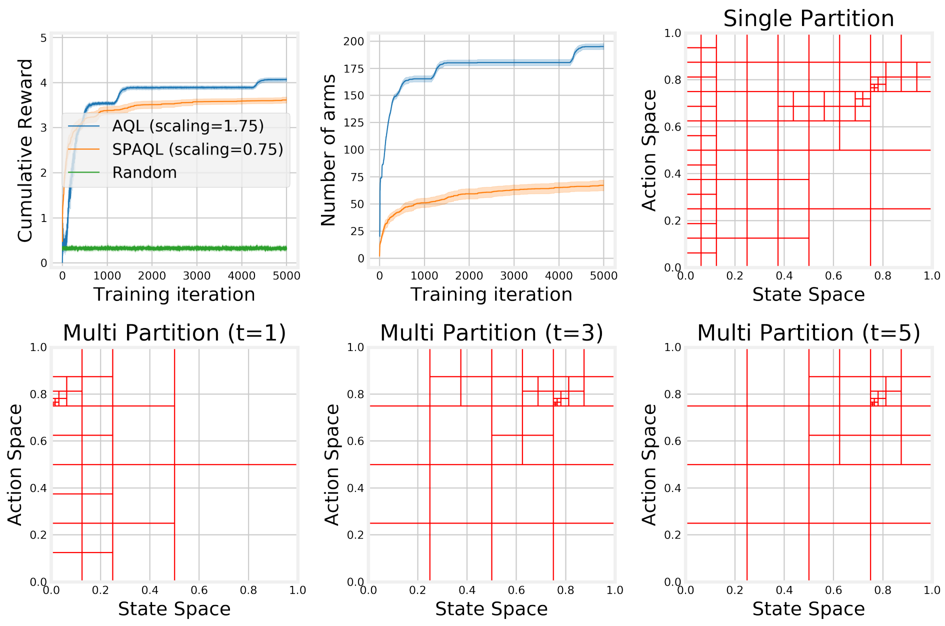

Figure A.5: Comparison of the algorithms on the oil problem with Laplace survey function ($\lambda = 10$).

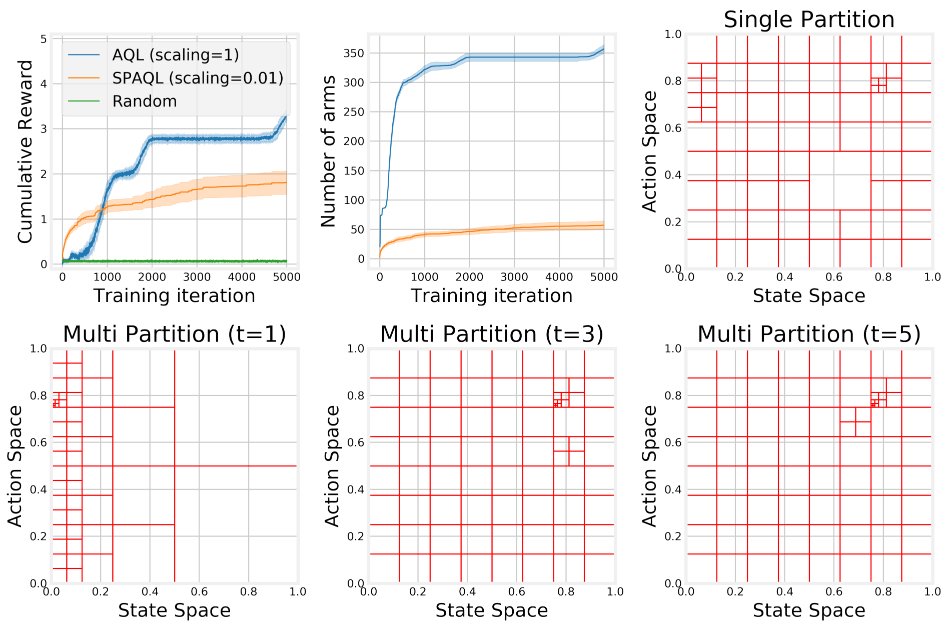

Figure A.6: Comparison of the algorithms on the oil problem with Laplace survey function ($\lambda = 50$).

A.3 Ambulance Problem with Uniform Arrivals

For the case when $c = 0$, shown in Figure A.7, the heuristic solution to this problem is the "Mean" heuristic, which corresponds to the line $a = 0.5$. The SPAQL agents managed to achieve higher rewards with a coarser partition and about one fifth of the arms of the AQL agents.

Moving to Figure A.8, corresponding to $c = 0.25$, the optimal policy would be a mix of the "Mean" (horizontal line) and the "No Movement" (diagonal line) heuristic. The SPAQL agents managed to achieve higher rewards with a coarser partition and less than one fifth of the arms of the AQL agents, a reduction even greater than the one in the case $c = 0$.

Finally, Figure A.9 shows the results when $c = 1$. The optimal policy would be the "No Movement" (diagonal line) heuristic. The diagonal line appears finely partitioned in both types of agents. However, there was clearly a shortage of samples of states within $[0, 0.25]$ in time steps 3 and 5, which may bring a problem in deployment. The SPAQL agents managed to achieve higher rewards with around one fifth of the arms of the AQL agents.

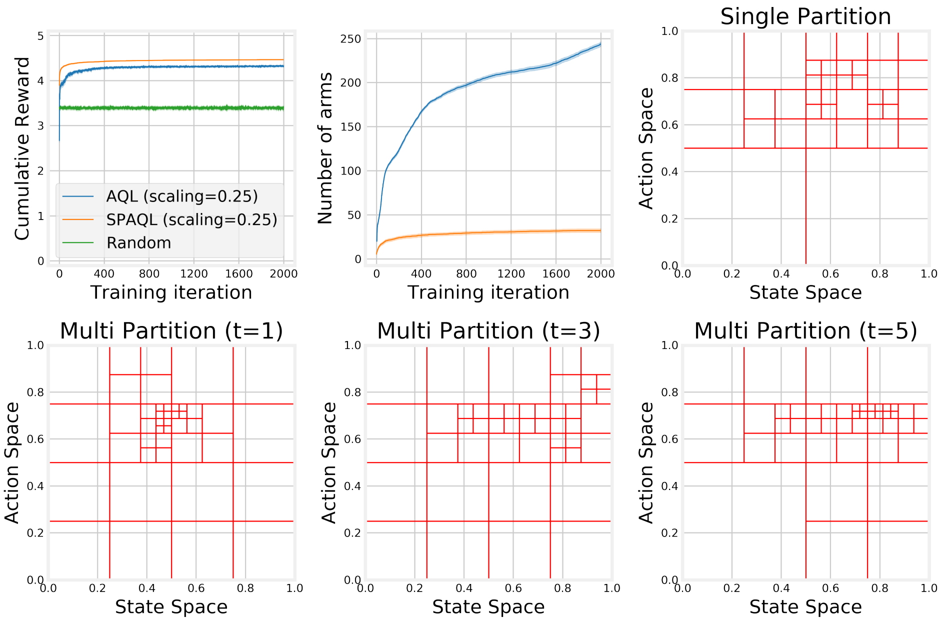

Figure A.7: Comparison of the algorithms on the ambulance problem with uniform arrival distribution and only paying the cost to go ($c = 0$).

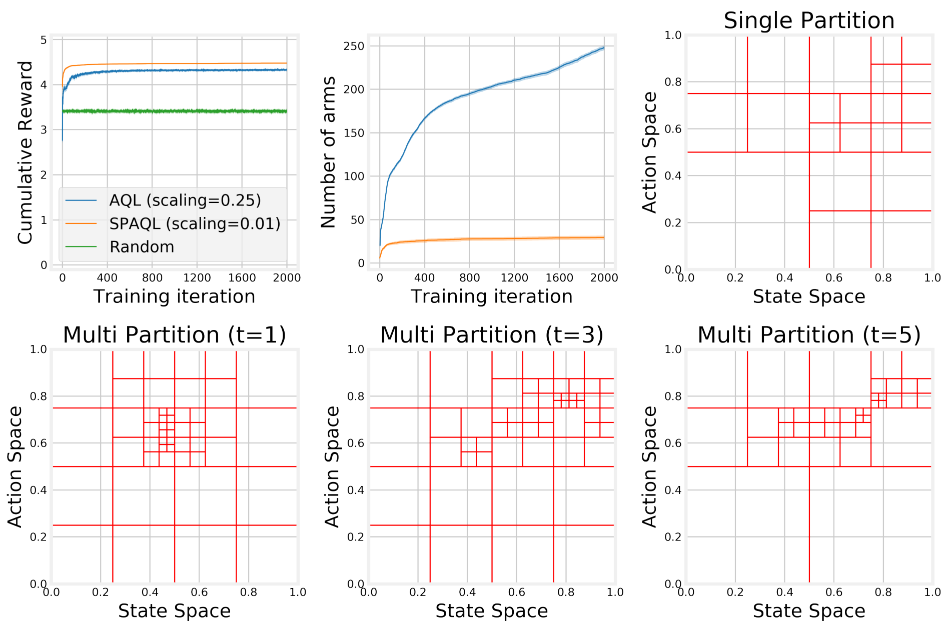

Figure A.8: Comparison of the algorithms on the ambulance problem with uniform arrival distribution and paying a mix between the cost to relocate and the cost to go ($c = 0.25$).

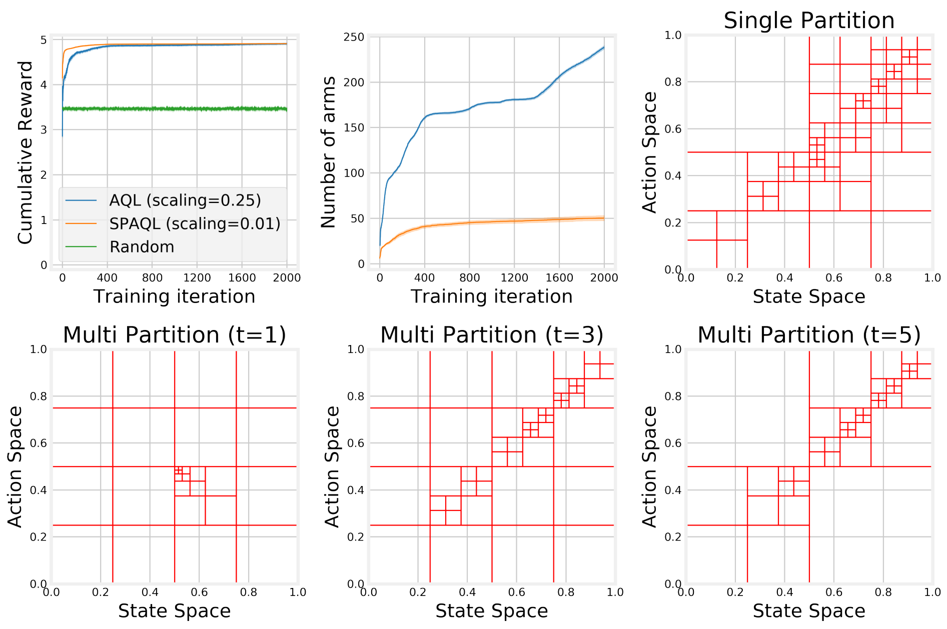

Figure A.9: Comparison of the algorithms on the ambulance problem with uniform arrival distribution and paying only the cost to relocate ($c = 1$).

A.4 Ambulance Problem with Beta Arrivals

For the case when $c = 0$, shown in Figure A.10, the heuristic solution to this problem is the "Mean" heuristic, which corresponds to the line $a \approx 0.7$. The SPAQL agent managed to achieve higher rewards with a coarser partition and less than one fifth of the arms of the AQL agents.

Moving to Figure A.11, corresponding to $c = 0.25$, the optimal policy would be a mix of the "Mean" (horizontal line) and the "No Movement" (diagonal line) heuristic. The SPAQL agents managed to achieve higher rewards with a coarser partition and less than one fifth of the arms of the AQL agents.

Finally, Figure A.12 shows the results when $c = 1$. The optimal policy would be the "No Movement" (diagonal line) heuristic. Since arrivals are now concentrated around 0.7 (the mean of the distribution), the diagonal appears more partitioned for states within $[0.5, 1]$ than for states within $[0, 0.5]$. The AQL agents do a finer partition of the diagonal, but this still does not prevent the SPAQL agents from converging to better rewards earlier, and with around one fifth of the number of arms of the AQL agents.

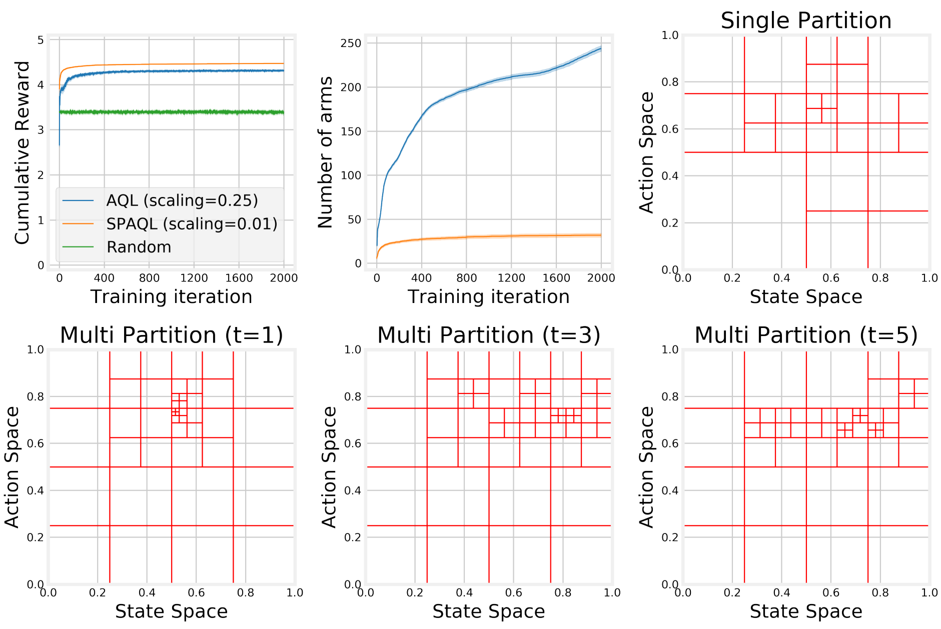

Figure A.10: Comparison of the algorithms on the ambulance problem with Beta$(5, 2)$ arrival distribution and only paying the cost to go ($c = 0$).

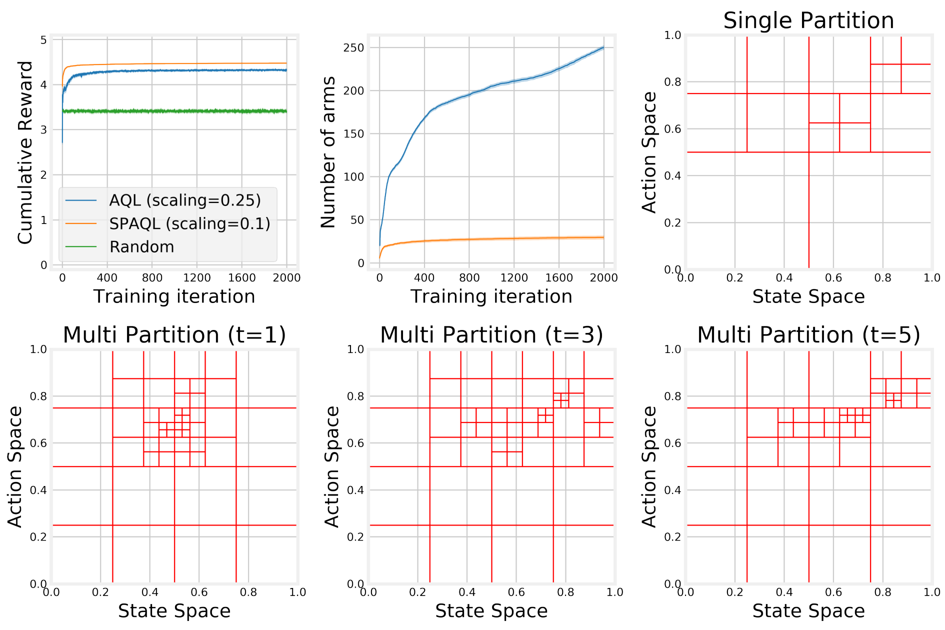

Figure A.11: Comparison of the algorithms on the ambulance problem with Beta$(5, 2)$ arrival distribution and paying a mix between the cost to relocate and the cost to go ($c = 0.25$).

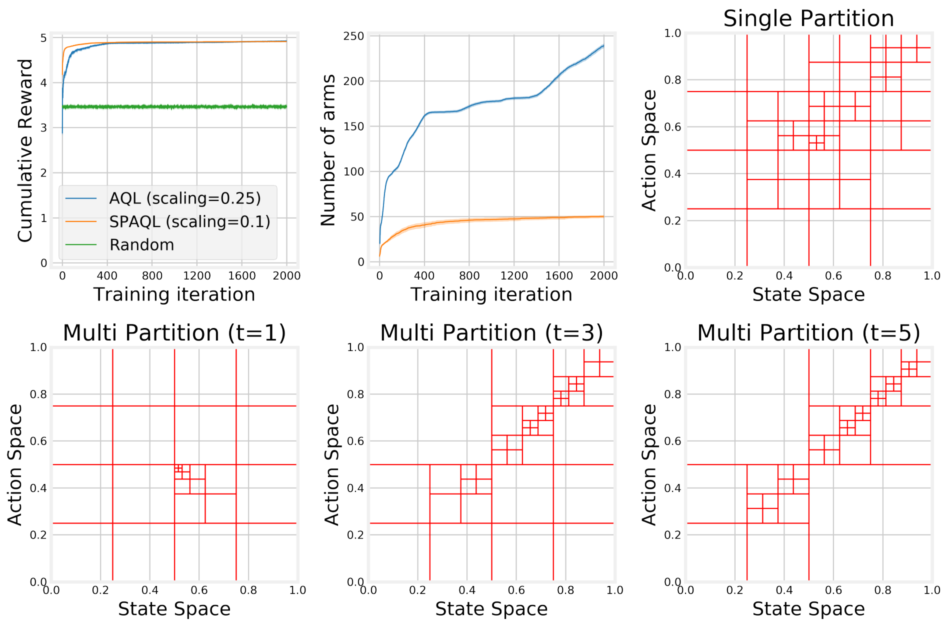

Figure A.12: Comparison of the algorithms on the ambulance problem with Beta$(5, 2)$ arrival distribution and paying only the cost to relocate ($c = 1$).

Appendix B

Illustration of the concept of domain

Consider the yellow ball in Figure B.1 as the initial ball covering the entire state-action space. The blue balls on the right form a covering of the yellow ball. Since the radius of the blue balls is half of the radius of the yellow ball, and the blue balls cover the yellow ball completely, the domain of the yellow ball is the empty set. In Figure B.2, the red balls on the left image form a covering of the rightmost blue ball. Following a line of reasoning equal to the one used for the yellow ball, the domain of the blue ball covered by the red balls is the empty set. For the neighboring blue balls, that are not covered by the smaller red balls but intersect some of them, the domain is the set of points which are not contained inside any red ball. The domains are highlighted by the blue regions with full opacity seen on the image on the right. The points in the blue balls occluded by the red balls do not belong to the domain of the respective blue ball.

Figure B.3 illustrates a case where a disjoint covering can be found for every ball. The domain for each ball is either the ball itself (if it has never been split), or the empty set $\varnothing$ (if it has been split once).

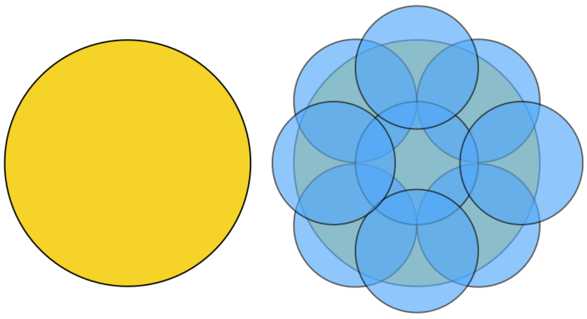

Figure B.1: The initial ball (yellow ball on the left), and a blue covering of it (image on the right).

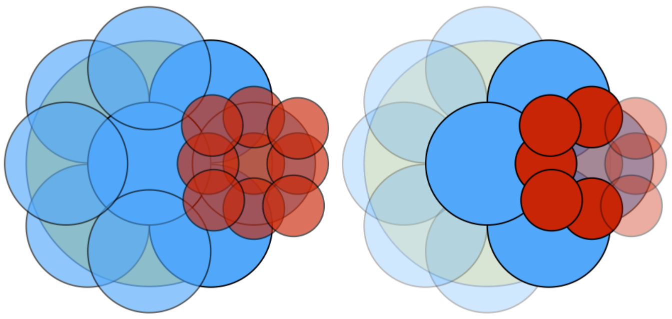

Figure B.2: A covering of a blue ball (red balls on the left), and the domains of the neighboring blue balls (on the right).

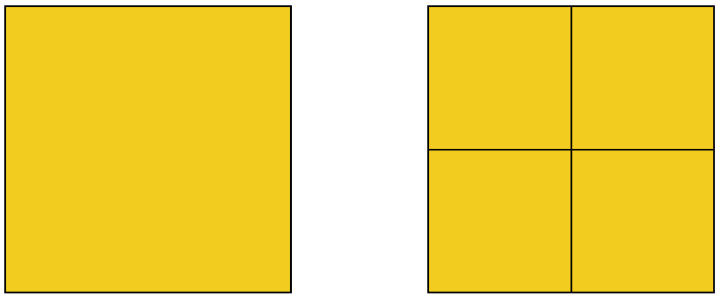

Figure B.3: The initial ball (on the left), and a covering of it (on the right).