Single-partition adaptive Q-learning: algorithm and applications

João Pedro Estácio Gaspar Gonçalves de Araújo

joao.p.araujo@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal

July 2020

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

1. Introduction

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. However, until recently, it was not known whether model-free methods could be proved to be sample-efficient. This changed when Jin et al. (2018) 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. (2016). Later, Song and Sun (2019) extended that algorithm to continuous state-action spaces. Their algorithm, *net-based Q-learning* (NBQL), requires discretizing the state-action space with a network of fixed step size. This creates a trade-off between memory requirements and algorithm performance. To address this trade-off, Sinclair et al. (2019) 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. All approaches mentioned so far consider time-variant value functions and learn time-variant policies. 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.

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.

2. Background

For background in RL, the reader is referred to the classic book by Sutton and Barto (2018) (or the notes by Szepesvári (2010)). This section recalls some basic concepts of RL and metric spaces, relevant for the subsequent presentation.

2.1. Notation

Although they work essentially on the same problems, the RL and control communities use different notations. For consistency with previous RL work, the RL notation is used in Sections 2 (Background), 3 (Single-partition adaptive Q-learning), and 4.1 and 4.2 (oil discovery and ambulance routing problems). Since its main audience is the control community, Sections 4.3.2 and 4.4.2 (`Pendulum` and `CartPole` problems) use the control notation. Table 1 lists the symbols used to denote the problem variables in the following sections, along with their usual control counterparts.

2.2. 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*, and $r : \mathcal{S} \times \mathcal{A} \to R \subseteq \mathbb{R}$ is the *reward function*. The transition kernel $\mathbb{P}$ 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 $\mathbb{P}$). The reward function r assigns a reward (or a cost) to each state-action pair.

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.

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. If the policy is independent of the time step, it is said to be time-invariant.

2.3. Q-learning

One possible approach for learning a good (eventually optimal) policy is *Q-learning*. For each time step h, a value is associated with each state. This value is encoded by the value function, defined as

$$V_h^\pi(x) := \mathbb{E}\left[\sum_{i=h}^{H} r(x_i, \pi(x_i)) \;\middle|\; x_h = x\right]. \quad (1)$$

This function only takes states into account. The *Q function*,

$$Q_h^\pi(x, a) := r(x, a) + \mathbb{E}\Big[V_h^\pi(x') \;\Big|\; x,\ a\Big], \quad (2)$$

also considers actions. It allows the agent to rank all possible actions at state x according to the corresponding values of Q at time step h, and then pick an action. Notice that the expectation in Equation 2 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* (Sutton and Barto, 2018). These functions satisfy the so-called *Bellman equation*

$$Q_h^\star(x, a) = r(x, a) + \mathbb{E}[V_h^\star(x') \mid x, a]. \quad (3)$$

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 3, according to

$$\mathbf{Q}_h(x, a) \leftarrow (1 - \alpha)\mathbf{Q}_h(x, a) + \\ + \alpha(r(x, a) + \mathbf{V}_h(x')), \quad (4)$$

where $\alpha \in [0, 1]$ is the *learning rate*. Actions are chosen greedily according to the `argmax` policy

$$\pi(x) = \texttt{argmax}_a \mathbf{Q}_h(x, a). \quad (5)$$

RL Sections 2, 3, 4.1, and 4.2	Control Sections 4.3 and 4.4	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: Correspondence between the notation used in the following sections. "NC" (No Correspondence) means that the respective symbols only appear in Sections 2, 3, 4.1, and 4.2 (RL).

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. 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.

Another way to deal with local optima is to use *upper confidence bounds* (UCB). Algorithms that use UCB add an extra term $b(x, a)$ to the update rule

$$\mathbf{Q}_h(x, a) \leftarrow (1 - \alpha)\mathbf{Q}_h(x, a) + \\ + \alpha(r(x, a) + \mathbf{V}_h(x') + b(x, a)), \quad (6)$$

which models the uncertainty of the Q function estimate.

2.4. 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 a set of properties (identity of indiscernibles, symmetry, triangle inequality, and positivity).

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 $\text{diam}(B) = \sup_{x,y \in B} \mathcal{D}(x, y)$. The diameter of the entire space is denoted $d_{max} = \text{diam}(X)$.

Definition 2.1 (Sinclair et al. (2019))**.** *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 (Sinclair et al. (2019))**.** *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 $X \subseteq \cup_{x \in \mathcal{P}} B(x, r)$.*

3. Single-partition adaptive Q-learning

The proposed SPAQL algorithm builds upon AQL (Sinclair et al., 2019). 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 $\mathbf{V}$).

3.1. Auxiliary functions

Algorithm 1 requires four auxiliary functions, whose pseudocode is omitted for brevity. Function Rollout performs a full episode under the current policy, while recording the cumulative reward. Function Evaluate Agent runs several rollouts and averages the cumulative rewards, returning a measure of the agent's performance.

Function Boltzmann Sample implements Boltzmann exploration. It has as arguments a list

Algorithm 1 Single-partition adaptive Q-learning

procedure SINGLE-PARTITION ADAPTIVE Q-LEARNING($\mathcal{S}, \mathcal{A}, \mathcal{D}, H, K, N, \xi, \tau_{\min}, u, d$)
  Initialize partitions $\mathcal{P}$ and $\mathcal{P}'$ containing a single ball with radius d_{max} and $\mathbf{Q} = H$
  Initialize τ to $\tau_{\min}$
  Calculate agent performance using EVALUATE AGENT($\mathcal{P}, \mathcal{S}, \mathcal{A}, H, N$)
  for each episode $k = 1, \ldots, K$ **do**
    Receive initial state x_1^k
    for each step $h = 1, \ldots, H$ **do**
      Get a list $\mathbf{B}$ with all the balls $B \in \mathcal{P}'$ which contain x_h^k
      Sample the ball B_{sel} using BOLTZMANN SAMPLE($\mathbf{B}, \tau$)
      Select action $a_h^k = a$ for some $(x_h^k, a) \in \texttt{dom}\,(B_{sel})$
      Play action a_h^k, receive reward r_h^k, and transition to new state x_{h+1}^k
      Update Parameters: $v = n(B_{sel}) \leftarrow n(B_{sel}) + 1$

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

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

      if $n(B_{sel}) \geq \left(\frac{d_{max}}{r(B_{sel})}\right)^2$ **then** SPLIT BALL(B_{sel})
    Evaluate the agent using EVALUATE AGENT($\mathcal{P}', \mathcal{S}, \mathcal{A}, H, N$)
    if agent performance improved **then**
      Copy $\mathcal{P}'$ to $\mathcal{P}$ (keep the best agent)
      Reset τ to $\tau_{\min}$
      Decrease u using some function of d (for example, $u \leftarrow u^d$, assuming $d < 1$)
    else
      Increase τ using some function of u (for example, $\tau \leftarrow u\tau$)
      if more than two splits occurred **then**
        Copy $\mathcal{P}$ to $\mathcal{P}'$ (reset the agent)
        Reset τ to $\tau_{\min}$

$\mathbf{B}$ 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.

Finally, function SPLIT BALL outputs a $\frac{1}{2}r(B)$-packing of $\texttt{dom}\,(B_{sel})$ (the domain of a ball $\texttt{dom}\,(B_{sel})$ is defined in the next section).

3.2. Main algorithm

The algorithm (Algorithm 1) 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. If the number of arms increases twice without improvements in performance, the agent resets $\mathcal{P}'$ to P. This forces an increase in the number of arms to correspond to an improvement in performance, thus preventing over-partitioning of the state-action space. Initially, 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.

During each training iteration, 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. The policy is modified based on the evolution of agent performance. If the agent currently being trained achieved a better performance than previous agents, the value of τ is reset to a user-defined $\tau_{\min}$ (≈ 0), in order to ensure that the policy becomes greedy (`argmax`). If the agent performs worse than the best agent, the value of τ is thus increased to make the policy behave in a more exploratory way.

Updates to $\mathbf{Q}$ are done according to

$$\mathbf{Q}^{k+1}(B_{sel}) \leftarrow (1 - \alpha_v)\mathbf{Q}^k(B_{sel}) + \\ + \alpha_v\big(r_h^k + \mathbf{V}^k(x_{h+1}^k) + b_\xi(v)\big), \quad (7)$$

where r_h^k is the reward obtained during training iteration k on time step h, α_v is the learning rate,

$\mathbf{V}^k(x^k_{h+1})$ is the current estimate of the value of the future state, $b_\xi(v)$ is the bonus term related with the upper confidence bound (confidence radius), and v is the number of times ball B_{sel} has been **v**isited.

The learning rate is set as done by Sinclair et al. (2019), *i.e.*, according to

$$\alpha_v = \frac{H+1}{H+v}. \tag{8}$$

Before defining $\mathbf{V}^k(x^k_{h+1})$, it is necessary to recall the definition of domain of a ball and the definition of the set $\texttt{RELEVANT}(x)$ from the work of Sinclair et al. (2019). The *domain* of a ball B from a partition $\mathcal{P}$ is a subset of B which excludes all balls $B' \in \mathcal{P}$ of a strictly smaller radius. In other words, it is the set of all points $b \in B$ which are not contained inside any other ball of strictly smaller radius than $r(B)$. A ball B is said to be *relevant* for a point $x \in \mathcal{S}$ if $(x, a) \in \texttt{dom}\,(B)$, for some $a \in \mathcal{A}$. The set of all relevant balls for a state x is denoted by $\texttt{RELEVANT}(x)$. The definition of $\mathbf{V}^k(x^k_{h+1})$ uses the expression proposed by Sinclair et al. (2019),

$$\mathbf{V}^k(x^k_{h+1}) = \min(H, \max_{B \in \texttt{RELEVANT}(x^k_{h+1})} \mathbf{Q}^k(B)), \tag{9}$$

with the difference that it also holds for the final state, while Sinclair et al. (2019) set $\mathbf{V}^k(x^k_{H+1}) = 0$, for all x.

Finally, the term $b_\xi(v)$ is defined as

$$b_\xi(v) = \frac{\xi}{\sqrt{v}}, \tag{10}$$

where ξ is called the *scaling parameter* of the upper confidence bounds, and is defined by the user.

3.3. Single-partition adaptive Q-learning with terminal state

The main difference between SPAQL and SPAQL with terminal state (SPAQL-TS) is the definition of the value function,

$$\mathbf{V}^k(x^k_{h+1}) = \exp\left(-\left(\frac{\mathcal{D}(x^k_h, x_{ref})}{\lambda}\right)^2\right)\mathbf{V}^k_{\text{Eq. 9}}(x^k_{h+1}), \tag{11}$$

where $\lambda > 0$ is a parameter that controls the weight given to the error in tracking reference state x_{ref} (both are user-defined).

4. Results

The code used to run the experiments, along with the corresponding parameters, is available on GitHub.[1]

[1] `https://github.com/jaraujo98/SinglePartitionAdaptiveQLearning`

4.1. Oil discovery

4.1.1 Setup

In this problem, used as benchmark by Sinclair et al. (2019), an agent surveys a 1D map in search of hidden "oil deposits". The state and action spaces are the set of locations that the agent has access to ($\mathcal{S} = \mathcal{A} = [0, 1]$). 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. (2019) 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 $0.7 + \pi/60$).

4.1.2 Results

The scaling parameter ξ was tuned for both AQL and SPAQL agents. After tuning, the best performing agents of each algorithm were compared. The SPAQL agents match the AQL ones when using the quadratic survey function with $\lambda = 50$ (Figure 1). On the other instances of the problem, the relatively small difference in cumulative rewards at the end of training is compensated by the lower number of arms used by SPAQL policies. Figure 1 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.

4.2. Ambulance routing

4.2.1 Setup

This problem, also used by Sinclair et al. (2019), 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. (2019) 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

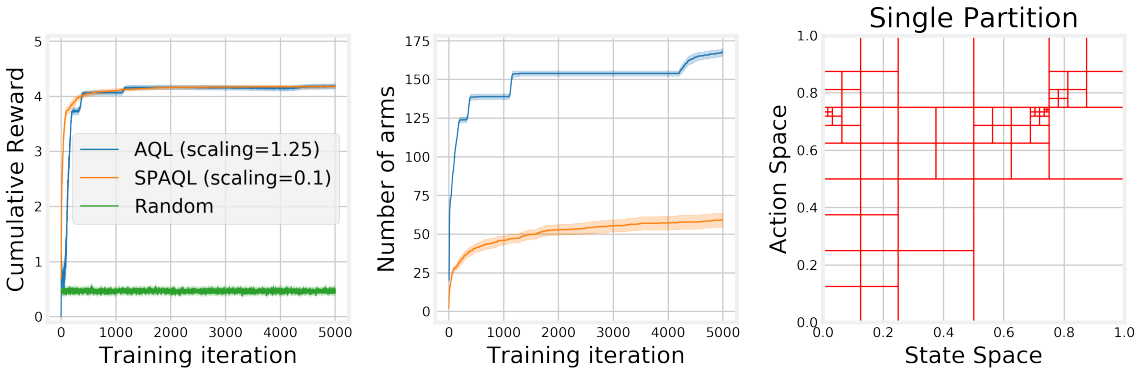

Figure 1: Average cumulative rewards, number of arms, and best SPAQL agent partition for the agents trained in the oil problem with reward function $r(x, a) = \max\{0, 1 - 50(x - c)^2 - |x - a|\}$. Shaded areas around the solid lines represent the 95% confidence interval.

the trade-offs between the cost of relocation and the cost of traveling to serve the request.

The experimental setups considered are

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

The optimal policy depends on the value of c. Sinclair et al. (2019) suggest two heuristics for both extreme cases ($c \in \{0, 1\}$). The "No Movement" heuristic is optimal when $c = 1$. The "Mean" heuristic is optimal when $c = 0$.

4.2.2 Results

After tuning ξ, the best agents were compared. SPAQL agents perform better than AQL ones in almost all instances of the problem, 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.

Figure 2 shows the cumulative rewards for an ambulance problem with a uniform arrival distribution and $c = 1$ (only relocation is penalized). The optimal policy is the "No Movement" one, which corresponds to the line $a = x$. This line appears finely partitioned, as would be expected. The SPAQL agents reach higher cumulative rewards earlier in training, and with fewer arms.

4.3. `Pendulum`

4.3.1 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 (Brockman et al., 2016). This 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 (Sinclair et al., 2019) and Section 3, but it is clear that Sinclair et al. (2019) have the observation space in mind when they mention state 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 (Brockman et al., 2016)

$$J = \sum_{t=1}^{H} \text{normalize}(\theta_t)^2 + 0.1\dot{\theta}_t^2 + 0.001(u_t^2), \quad (12)$$

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 original problem (with rewards ranging from approximately -16 to 0) and on a problem with rewards scaled to be in the interval $[0, 1]$.

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

$$\mathcal{S} \times \mathcal{A} = ([-1, 1] \times [-1, 1] \times [-8, 8]) \times [-2, 2]. \quad (13)$$

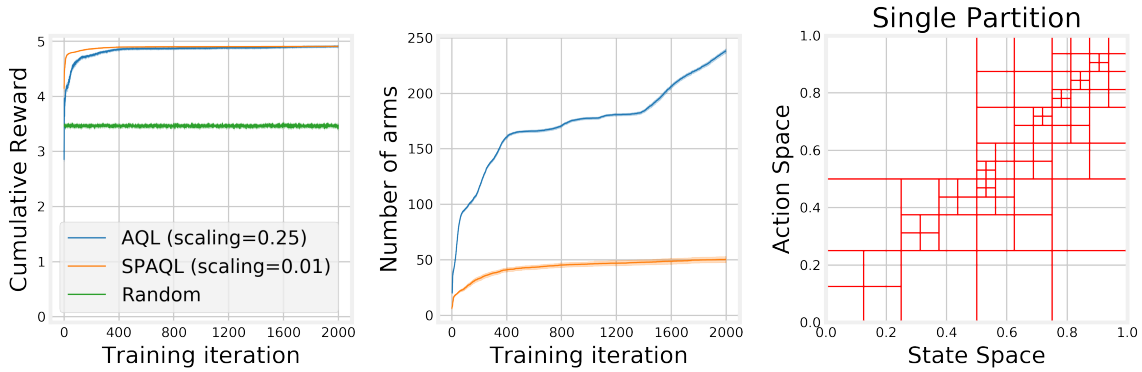

Figure 2: Average cumulative rewards, number of arms, and best SPAQL agent partition for the agents trained in the ambulance problem with uniform arrival distribution and reward function $r(x, a) = \max\{0, 1 - |x - a|\}$. Shaded areas around the solid lines represent the 95% confidence interval.

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

$$\mathcal{D}((x,u),(x',u')) = \max\Big\{|c_\theta - c_{\theta'}|, |s_\theta - s_{\theta'}|, \frac{|\dot\theta - \dot\theta'|}{8}, |u - u'|\Big\}. \quad (14)$$

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 $[\theta_{ref}, \dot\theta_{ref}]^T = [0, 0]^T$.

4.3.2 Results

The experiments to tune the scaling parameter ξ lead to the conclusion that reward scaling does not impact learning significantly. The training curves and number of arms for the agents trained in the `Pendulum` problem without reward scaling are shown in Figure 3. The average cumulative reward at the end of training for each algorithm is recorded in Table 2. 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 (Colas et al., 2018) 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.

4.4. `CartPole`

4.4.1 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 (Barto et al., 1983). 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 $\mathbf{x} = [x, \dot x, \theta, \dot\theta]^T$. 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 (Brockman et al., 2016): 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 (Brockman et al., 2016). 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 state-action space of this problem (as observed by the agent) is (Brockman et al., 2016)

$$\mathcal{S} \times \mathcal{A} = \left([-4.8, 4.8] \times \mathbb{R} \times \left[-\frac{24\pi}{180}, \frac{24\pi}{180}\right] \times \mathbb{R}\right) \times \{0, 1\}. \quad (15)$$

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

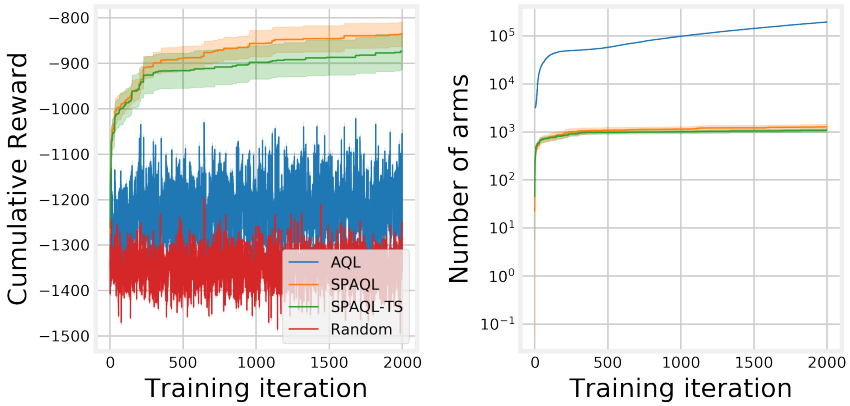

Figure 3: Average cumulative rewards and number of arms for the agents trained in the `Pendulum` system without reward scaling. Shaded areas around the solid lines represent the 95% confidence interval.

maximum value). Cart and pole velocities are allowed to assume any real value. To bound them, the sigmoid function (shifted and rescaled) is used

$$\phi_m(y) = \frac{2}{1+\exp(-2\,m\,y)} - 1. \tag{16}$$

Parameter m controls the slope at the origin, and can be used to include some domain knowledge. For the linear and angular velocity it was set to $m_{\dot{x}} = 1/240$ and $m_{\dot{\theta}} = 1/21$, respectively.

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

$$\begin{aligned}\mathcal{D}((\mathbf{x},u),(\mathbf{x}',u')) &= \\ &= \max\Big\{\frac{|x-x'|}{4.8}, |\phi_{m_{\dot{x}}}(\dot{x}) - \phi_{m_{\dot{x}}}(\dot{x}')|, \\ \frac{|\theta-\theta'|}{24\pi/180}, |\phi_{m_{\dot{\theta}}}(\dot{\theta}) - \phi_{m_{\dot{\theta}}}(\dot{\theta}')|, 2\times\mathbb{1}_{[u\neq u']}\Big\},\end{aligned} \tag{17}$$

For SPAQL-TS, the reference state $\mathbf{x}_{ref}$ is set to $[0,0,0,0]^T$.

4.4.2 Results

The training curves for the three algorithms (after choosing the best value of ξ) are shown in Figure 4. The average cumulative rewards are recorded in Table 3. 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 that SPAQL-TS is better than SPAQL at 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 ones (1.29×10^3 and 5.58×10^2, respectively). The AQL agents finish training with 25 times more arms (3.30×10^4).

4.5. Comparing AQL, SPAQL, and TRPO

`Pendulum` (avg. cum. reward)	
AQL	-1287.31 ± 5.79
SPAQL	-835.99 ± 26.91
SPAQL-TS	-873.40 ± 39.94
Random	-1340.43 ± 6.24
TRPO	$\mathbf{-176.76 \pm 15.79}$

Table 2: Average cumulative rewards ($\pm$95% confidence interval) for the different agents at the end of training in the `Pendulum` system. The best performance is shown in bold.

Trust region policy optimization (TRPO) is an RL algorithm that is known to perform well in control problems (Schulman et al., 2015; Duan et al., 2016). 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

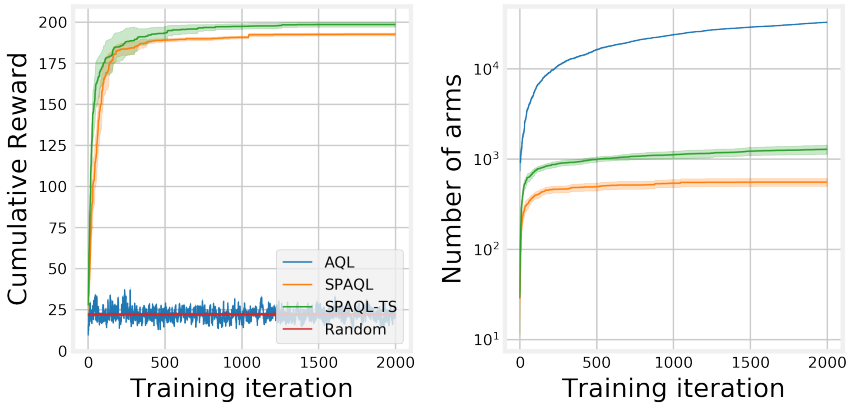

Figure 4: Average cumulative rewards and number of arms for the agents trained in the `CartPole` system. Shaded areas around the solid lines represent the 95% confidence interval.

`CartPole` (avg. cum. reward)	
AQL	22.40 ± 0.60
SPAQL	192.57 ± 0.94
SPAQL-TS	$\mathbf{198.53 \pm 1.55}$
Random	22.15 ± 0.64
TRPO	197.27 ± 3.17

Table 3: Average cumulative rewards (±95% confidence interval) for the different agents at the end of training in the `CartPole` system. The best performance is shown in bold.

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` (The garage contributors, 2019) 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. Figures 5 and 6 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 5, Table 2) 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 6, Table 3), 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).

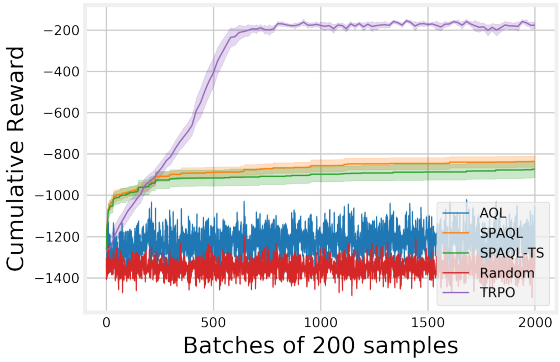

Figure 5: Average cumulative rewards and number of arms for the agents trained in the `Pendulum` system without reward scaling. Shaded areas around the solid lines represent the 95% confidence interval.

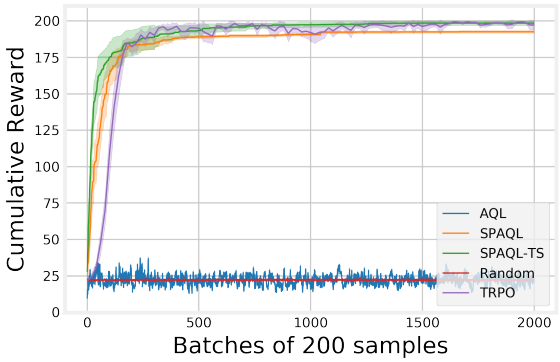

Figure 6: Average cumulative rewards and number of arms for the agents trained in the `CartPole` system. Shaded areas around the solid lines represent the 95% confidence interval.

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, and SPAQL with terminal state (SPAQL-TS), an improved version of SPAQL that borrows concepts from control theory. In order to balance exploration and exploitation, SPAQL uses Boltzmann exploration with a cyclic temperature schedule in addition to upper confidence bounds (UCB).

Experiments show that, with very little parameter tuning, SPAQL performs satisfactorily in the oil discovery and ambulance routing problems, resulting in partitions with a lower number of arms, and requiring fewer training iterations than AQL to converge.

In the `Pendulum` and the `CartPole` problems, both SPAQL and SPAQL-TS outperform AQL. 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.

There are several possible directions for further work. Both AQL and SPAQL can be further modified in several ways. For example, several episodes could be run during each training iteration, instead of only one. Furthermore, the parameters of SPAQL are highly conjugate with the episode length. It would be interesting to study automatic ways of setting u and d given H. Similarly to what was done in this paper for ξ, an empirical study regarding the effect of tuning parameters u and d on SPAQL performance could be done.

Another possible direction is to do a formal analysis to the complexity of SPAQL. This would allow a rigorous comparison in terms of sample efficiency to AQL or similar algorithms.

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.

Acknowledgements

The author 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 calculations presented in this thesis.

References

A. G. Barto, et al. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Syst. Man Cybern.*, 13(5):834–846, 1983.

G. Brockman, et al. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

C. Colas, et al. How many random seeds? Statistical power analysis in deep reinforcement learning experiments. *CoRR*, abs/1806.08295, 2018.

Y. Duan, et al. Benchmarking deep reinforcement learning for continuous control. In *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.

C. Jin, et al. Is Q-learning provably efficient? In *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.

I. Osband, et al. Generalization and exploration via randomized value functions. In *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.

J. Schulman, et al. Trust region policy optimization. In *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.

S. R. Sinclair, et al. Adaptive discretization for episodic reinforcement learning in metric spaces. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3): 55:1–55:44, 2019.

Z. Song and W. Sun. Efficient model-free reinforcement learning in metric spaces. *CoRR*, abs/1905.00475, 2019.

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

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

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