LOGICALLY-CONSTRAINED NEURAL FITTED Q-ITERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper proposes a method for efficient training of Q-function for continuous-state Markov Decision Processes (MDP), such that the traces of the resulting policies satisfy a Linear Temporal Logic (LTL) property. LTL, a modal logic, can express a wide range of time-dependent logical properties including safety and liveness. We convert the LTL property into a limit deterministic Büchi automaton with which a synchronized product MDP is constructed. The control policy is then synthesised by a reinforcement learning algorithm assuming that no prior knowledge is available from the MDP. The proposed method is evaluated in a numerical study to test the quality of the generated control policy and is compared against conventional methods for policy synthesis such as MDP abstraction (Voronoi quantizer) and approximate dynamic programming (fitted value iteration).

1 INTRODUCTION

Markov Decision Processes (MDPs) are extensively used as a family of stochastic processes in automatic control, computer science, economics, etc. to model sequential decision-making problems. Reinforcement Learning (RL) is a machine learning algorithm that is widely used to train an agent to interact with an MDP when the stochastic behaviour of the MDP is initially unknown. However, conventional RL is mostly focused on problems in which MDP states and actions are finite. Nonetheless, many interesting real-world tasks, require actions to be taken in response to high-dimensional or real-valued sensory inputs (Doya, 2000). For example, consider the problem of drone control in which the drone state is represented as its Euclidean position $(x, y, z) \in \mathbb{R}^3$.

Apart from state space discretisation and then running vanilla RL on the abstracted MDP, an alternative solution is to use an approximation function which is achieved via regression over the set of samples. At a given state, this function is able to estimate the value of the expected reward. Therefore, in continuous-state RL, this approximation replaces conventional RL state-action-reward look-up table which is used in finite-state MDPs. A number of methods are available to approximate the expected reward, e.g. CMACs (Sutton, 1996), kernel-based modelling (Ormoneit & Sen, 2002), tree-based regression (Ernst et al., 2005), basis functions (Busoniu et al., 2010), etc. Among these methods, neural networks offer great promise in reward modelling due to their ability to approximate any non-linear function (Hornik, 1991). There exist numerous successful applications of neural networks in RL for infinite or large-state space MDPs, e.g. Deep Q-networks (Mnih et al., 2015), TD-Gammon (Tesauro, 1995), Asynchronous Deep RL (Mnih et al., 2016), Neural Fitted Q-iteration (Riedmiller, 2005), CACLA (Van Hasselt & Wiering, 2007).

In this paper, we propose to employ feedforward networks (multi-layer perceptrons) to synthesise a control policy for *infinite-state* MDPs such that the generated traces satisfy a Linear Temporal Logic (LTL) property. LTL allows to specify complex mission tasks in a rich time-dependent formal language. By employing LTL we are able to express complex high-level control objectives that are hard to express and achieve for other methods from vanilla RL (Sutton & Barto, 1998; Smith et al., 2011) to more recent developments such as Policy Sketching (Andreas et al., 2017). Examples include liveness and cyclic properties, where the agent is required to make progress while concurrently executing components, to take turns in critical sections or to execute a sequence of tasks periodically. The purpose of this work is to show that the proposed architecture efficiently performs and is compatible with RL algorithms that are core of recent developments in the community.

Unfortunately, in the domain of continuous-state MDPs, to the best of our knowledge, no research has been done to enable RL to generate policies according to full LTL properties. On the other hand, the

problem of control synthesis in *finite-state* MDPs for temporal logic has been considered in a number of works. In (Wolff et al., 2012), the property of interest is an LTL property, which is converted to a Deterministic Rabin Automaton (DRA). A modified Dynamic Programming (DP) algorithm is then proposed to maximise the worst-case probability of satisfying the specification over all transition probabilities. Notice that in this work the MDP must be known a priori. (Fu & Topcu, 2014) and (Brázdil et al., 2014) assume that the given MDP has unknown transition probabilities and build a Probably Approximately Correct MDP (PAC MDP), which is producted with the logical property after conversion to DRA. The goal is to calculate the value function for each state such that the value is within an error bound of the actual state value where the value is the probability of satisfying the given LTL property. The PAC MDP is generated via an RL-like algorithm and standard value iteration is applied to calculate the values of states.

Moving away from full LTL logic, scLTL is proposed for mission specification, with which a linear programming solver is used to find optimal policies. The concept of shielding is employed in (Alshiekh et al., 2017) to synthesise a reactive system that ensures that the agent stays safe during and after learning. However, unlike our focus on full LTL expressivity, (Alshiekh et al., 2017) adopted the safety fragment of LTL as the specification language. This approach is closely related to teacher-guided RL (Thomaz & Breazeal, 2008), since a shield can be considered as a teacher, which provides safe actions only if absolutely necessary. The generated policy always needs the shield to be online, as the shield maps every unsafe action to a safe action. Almost all other approaches in safe RL either rely on ergodicity of the underlying MDP, e.g. (Moldovan & Abbeel, 2012), which guarantees that any state is reachable from any other state, or they rely on initial or partial knowledge about the MDP, e.g. (Song et al., 2012) and (Lope & Martin, 2009).

2 BACKGROUND

2.1 PROBLEM FRAMEWORK

Definition 2.1 (Continuous-state Space MDP) *The tuple $\boldsymbol{M} = (\mathcal{S}, \mathcal{A}, s_0, P, \mathcal{AP}, L)$ is an MDP over a set of states $\mathcal{S} = \mathbb{R}^n$, where $\mathcal{A}$ is a finite set of actions, s_0 is the initial state and $P : \mathcal{B}(\mathbb{R}^n) \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a Borel-measurable transition kernel which assigns to any state and any action a probability measure on the Borel space $(\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n))$ (Durrett, 1999). $\mathcal{AP}$ is a finite set of atomic propositions and a labelling function $L : \mathcal{S} \to 2^{\mathcal{AP}}$ assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \subseteq 2^{\mathcal{AP}}$ Durrett (1999).* ⌟

A finite-state MDP is a special case of continuous-state space MDP in which $|\mathcal{S}| < \infty$ and $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function. The transition function P induces a matrix which is usually known as transition probability matrix in the literature.

Theorem 2.1 *In any MDP $\boldsymbol{M}$ with bounded reward function and finite action space, if there exists an optimal policy, then that policy is stationary and deterministic. (Puterman, 2014)(Cavazos-Cadena et al., 2000).* ⌟

An MDP **M** is said to be solved if the agent discovers an optimal policy $Pol^* : \mathcal{S} \to \mathcal{A}$ to maximize the expected reward. From Definitions A.3 and A.4 in Appendix, it means that the agent has to take actions that return the highest expected reward. Note that the reward function for us as the designer is known in the sense that we know over which state (or under what circumstances) the agent will receive a given reward. The reward function specifies what the agent needs to achieve but not how to achieve it. Thus, the objective is that the agent itself comes up with an optimal policy. In the supplementary materials in Section A.2, we present fundamentals of approaches introduced in this paper for solving infinite-state MDPs.

3 LINEAR TEMPORAL LOGIC PROPERTIES

In order to specify a set of desirable constraints (i.e. properties) over the agent policy we employ Linear Temporal Logic (LTL) (Pnueli, 1977). An LTL formula can express a wide range of properties, such as safety and persistence. LTL formulas over a given set of atomic propositions $\mathcal{AP}$ are

syntactically defined as

$$\varphi ::= true \mid \alpha \in \mathcal{AP} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi \cup \varphi. \tag{1}$$

We define the semantics of LTL formula next, as interpreted over MDPs. Given a path ρ, the i-th state of ρ is denoted by $\rho[i]$ where $\rho[i] = s_i$. Furthermore, the i-th suffix of ρ is $\rho[i..]$ where $\rho[i..] = s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} s_{i+2} \xrightarrow{a_{i+2}} s_{i+3} \xrightarrow{a_{i+3}} \dots$.

Definition 3.1 (LTL Semantics) *For an LTL formula φ and for a path ρ, the satisfaction relation $\rho \models \varphi$ is defined as*

$$\begin{array}{ll} \rho \models \alpha \in \mathcal{AP} \iff \alpha \in L(\rho[0]) & \rho \models \bigcirc\varphi \iff \rho[1..] \models \varphi \\ \rho \models \varphi_1 \wedge \varphi_2 \iff \rho \models \varphi_1 \wedge \rho \models \varphi_2 & \rho \models \varphi_1 \cup \varphi_2 \iff \exists j \in \mathbb{N} \cup \{0\} \text{ s.t.} \\ \rho \models \neg\varphi \iff \rho \not\models \varphi & \rho[j..] \models \varphi_2 \;\&\; \forall i,\ 0 \leq i < j,\ \rho[i..] \models \varphi_1 \end{array}$$

Using the until operator we are able to define two temporal modalities: (1) eventually, $\Diamond\varphi = true \cup \varphi$; and (2) always, $\Box\varphi = \neg\Diamond\neg\varphi$. LTL extends propositional logic with the temporal modalities until $\cup$, eventually $\Diamond$, and always $\Box$. For example, in a robot control problem, statements such as "eventually get to this point" or "always stay safe" are expressible by these modalities and can be combined via logical connectives and nesting to provide general and complex task specifications. Any LTL task specification φ over $\mathcal{AP}$ expresses the following set of words: $Words(\varphi) = \{\sigma \in (2^{\mathcal{AP}})^\omega \text{ s.t. } \sigma \models \varphi\}$.

Definition 3.2 (Policy Satisfaction) *We say that a stationary deterministic policy Pol satisfies an LTL formula φ if:*

$$\mathbb{P}[L(s_0)L(s_1)L(s_2)... \in Words(\varphi)] > 0,$$

where every transition $s_i \to s_{i+1},\ i = 0, 1, ...$ is constructed by taking action $Pol(s_i)$ at state s_i. ⌟

For an LTL formula φ, an alternative method to express the set of associated words, i.e., $Words(\varphi)$, is to employ an automaton. Limit Deterministic Büchi Automatons (LDBA) (Sickert et al., 2016) are one of the most succinct and simplest automatons for that purpose (Sickert & Křetínskỳ, 2016). We need to first define a Generalized Büchi Automaton (GBA) and then we formally introduce an LDBA.

Definition 3.3 (Generalized Büchi Automaton) *A GBA $\mathbf{N} = (\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ is a structure where $\mathcal{Q}$ is a finite set of states, $q_0 \subseteq \mathcal{Q}$ is the set of initial states, $\Sigma = 2^{\mathcal{AP}}$ is a finite alphabet, $\mathcal{F} = \{F_1, ..., F_f\}$ is the set of accepting conditions where $F_j \subset \mathcal{Q}, 1 \leq j \leq f$, and $\Delta : \mathcal{Q} \times \Sigma \to 2^{\mathcal{Q}}$ is a transition relation.* ⌟

Let Σ^ω be the set of all infinite words over Σ. An infinite word $w \in \Sigma^\omega$ is accepted by a GBA $\mathbf{N}$ if there exists an infinite run $\theta \in \mathcal{Q}^\omega$ starting from q_0 where $\theta[i+1] \in \Delta(\theta[i], \omega[i]),\ i \geq 0$ and for each $F_j \in \mathcal{F}$

$$inf(\theta) \cap F_j \neq \emptyset, \tag{2}$$

where $inf(\theta)$ is the set of states that are visited infinitely often in the sequence θ.

Definition 3.4 (LDBA) *A GBA $\mathbf{N} = (\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ is limit deterministic if $\mathcal{Q}$ can be partitioned into two disjoint sets $\mathcal{Q} = \mathcal{Q}_N \cup \mathcal{Q}_D$, such that (Sickert et al., 2016):*

- *$\Delta(q, \alpha) \subseteq \mathcal{Q}_D$ and $|\Delta(q, \alpha)| = 1$ for every state $q \in \mathcal{Q}_D$ and for every corresponding $\alpha \in \Sigma$,*
- *for every $F_j \in \mathcal{F}$, $F_j \subset \mathcal{Q}_D$.* ⌟

An LDBA is a GBA that has two partitions: initial ($\mathcal{Q}_N$) and accepting ($\mathcal{Q}_D$). The accepting part includes all the accepting states and has deterministic transitions.

4 Logically-Constrained Neural Fitted Q-iteration

In this section, we propose an algorithm based on Neural Fitted Q-iteration (NFQ) that is able to synthesize a policy that satisfies a temporal logic property. We call this algorithm Logically-Constrained NFQ (LCNFQ). We relate the notion of MDP and automaton by synchronizing them to create a new structure that is first of all compatible with RL and second that embraces the logical property.

Definition 4.1 (Product MDP) *Given an MDP* $\boldsymbol{M}(\mathcal{S}, \mathcal{A}, s_0, P, \mathcal{AP}, L)$ *and an LDBA* $\boldsymbol{N}(\mathcal{Q}, q_0, \Sigma, \mathcal{F}, \Delta)$ *with* $\Sigma = 2^{\mathcal{AP}}$*, the product MDP is defined as* $(\boldsymbol{M} \otimes \boldsymbol{N}) = \boldsymbol{M_N}(\mathcal{S}^\otimes, \mathcal{A}, s_0^\otimes, P^\otimes, \mathcal{AP}^\otimes, L^\otimes, \mathcal{F}^\otimes)$*, where* $\mathcal{S}^\otimes = \mathcal{S} \times \mathcal{Q}$*,* $s_0^\otimes = (s_0, q_0)$*,* $\mathcal{AP}^\otimes = \mathcal{Q}$*,* $L^\otimes = \mathcal{S} \times \mathcal{Q} \to 2^\mathcal{Q}$ *such that* $L^\otimes(s, q) = q$ *and* $\mathcal{F}^\otimes \subseteq \mathcal{S}^\otimes$ *is the set of accepting states such that for each* $s^\otimes = (s, q) \in \mathcal{F}^\otimes$*,* $q \in \mathcal{F}$*. The intuition behind the transition kernel* $P^\otimes$ *is that given the current state* (s_i, q_i) *and action* a *the new state is* (s_j, q_j) *where* $s_j \sim P(\cdot|s_i, a)$ *and* $q_j \in \Delta(q_i, L(s_j))$*.* ⌟

By constructing the product MDP we add an extra dimension to the state space of the original MDP. The role of the added dimension is to track the automaton state and, hence, to synchronize the current state of the MDP with the state of the automaton and thus to evaluate the satisfaction of the associated LTL property.

Definition 4.2 (Absorbing Set) *We define the set* $A \in \mathcal{B}(\mathcal{S}^\otimes)$ *to be an absorbing set if* $P^\otimes(A|s^\otimes, a) = 1$ *for all* $s^\otimes \in A$ *and for all* $a \in \mathcal{A}$*. An absorbing set is called accepting if it includes* $\mathcal{F}^\otimes$*. We denote the set of all accepting absorbing sets by* $\mathbb{A}$*.* ⌟

Note that the defined notion of absorbing set in continuous-state MDPs is equivalent to the notion of maximum end components in finite-state MDPs. In another word, once a trace ends up in an absorbing set (or a maximum end component) it can never escape from it (Tkachev et al., 2017).

The product MDP encompasses transition relations of the original MDP and the structure of the Büchi automaton, thus it inherits characteristics of both. Therefore, a proper reward function can lead the RL agent to find a policy that is optimal and that respects both the original MDP and the LTL property φ. In this paper, we propose an on-the-fly random variable reward function that observes the current state $s^\otimes$, the action a and observes the subsequent state $s^{\otimes\prime}$ and gives the agent a scalar value according to the following rule:

$$R(s^\otimes, a) = \begin{cases} r_p & \text{if } s^\otimes \notin \mathbb{A},\ s^{\otimes\prime} \in \mathbb{A}, \\ r_n & \text{otherwise,} \end{cases} \tag{3}$$

where $r_p = M + y \times m \times rand(s^\otimes)$ is a positive reward and $r_n = y \times m \times rand(s^\otimes)$ is a neutral reward where $y \in \{0, 1\}$ is a constant, $0 < m \ll M$, and $rand : \mathcal{S}^\otimes \to (0, 1)$ is a function that generates a random number in $(0, 1)$ for each state $s^\otimes$ each time R is being evaluated. The role of function $rand$ is to break the symmetry in LCNFQ neural nets. Note that parameter y essentially acts as a switch to bypass the effect of the $rand$ function on R. As we will see later, this switch is only active for LCNFQ.

In LCNFQ, the temporal logic property is initially specified as a high-level LTL formula φ. The LTL formula is then converted to an LDBA $\mathbf{N}$ to form a product MDP $\mathbf{M_N}$ (see Definition 4.1). In order to use the experience replay technique we let the agent explore the MDP and reinitialize it when a positive reward is received or when no positive reward is received after th iterations. The parameter th is set manually according to the MDP such that allows the agent to explore the MDP and also to prevent the sample set to explode in size. All episode traces, i.e. experiences, are stored in the form of $(s^\otimes, a, s^{\otimes\prime}, R(s^\otimes, a), q)$. Here $s^\otimes = (s, q)$ is the current state in the product MDP, a is the chosen action, $s^{\otimes\prime} = (s', q')$ is the resulting state, and $R(s^\otimes, a)$ is the reward. The set of past experiences is called the sample set $\mathcal{E}$.

Once the exploration phase is finished and the sample set is created, we move forward to the learning phase. In the learning phase, we employ n separate multi-layer perceptrons with just one hidden layer where $n = |\mathcal{Q}|$ and $\mathcal{Q}$ is the finite cardinality of the automaton $\mathbf{N}$. Each neural net is associated with a state in the LDBA and together the neural nets approximate the Q-function in the product MDP. For each automaton state $q_i \in \mathcal{Q}$ the associated neural net is called $B_{q_i} : \mathcal{S}^\otimes \times \mathcal{A} \to \mathbb{R}$. Once the agent is at state $s^\otimes = (s, q_i)$ the neural net B_{q_i} is used for the local Q-function approximation. The set of neural nets acts as a global hybrid Q-function approximator $Q : \mathcal{S}^\otimes \times \mathcal{A} \to \mathbb{R}$. Note that the neural nets are not fully decoupled. For example, assume that by taking action a in state $s^\otimes = (s, q_i)$ the agent is moved to state $s^{\otimes\prime} = (s', q_j)$ where $q_i \neq q_j$. According to (13) the weights of B_{q_i} are updated such that $B_{q_i}(s^\otimes, a)$ has minimum possible error to $R(s^\otimes, a) + \gamma \max_{a'} B_{q_j}(s^{\otimes\prime}, a')$. Therefore, the value of $B_{q_j}(s^{\otimes\prime}, a')$ affects $B_{q_i}(s^\otimes, a)$.

Let $q_i \in \mathcal{Q}$ be a state in the LDBA. Then define $\mathcal{E}_{q_i} := \{(\cdot, \cdot, \cdot, \cdot, x) \in \mathcal{E} | x = q_i\}$ as the set of experiences within $\mathcal{E}$ that is associated with state q_i, i.e., $\mathcal{E}_{q_i}$ is the projection of $\mathcal{E}$ onto q_i. Once the

Algorithm 1: LCNFQ

input : MDP $\mathbf{M}$, a set of transition samples $\mathcal{E}$
output : Approximated Q-function

initialize all neural nets B_{q_i} with (s_0, q_i, a) as the input and r_n as the output where $a \in \mathcal{A}$ is a random action
repeat
  for $q_i = |\mathcal{Q}|$ ***to*** 1 **do**
    $\mathcal{P}_{q_i} = \{(input_l, target_l),\ l = 1, ..., |\mathcal{E}_{q_i}|)\}$
      $input_l = ({s_l}^{\otimes}, a_l)$
        $target_l = R({s_l}^{\otimes}, a_l) + \gamma \max_{a'} Q({s_l}^{\otimes'}, a')$
      where $({s_l}^{\otimes}, a_l, {s_l}^{\otimes'}, R({s_l}^{\otimes}, a_l), q_i) \in \mathcal{E}_{q_i}$
    $B_{q_i} \leftarrow$ Rprop($\mathcal{P}_{q_i}$)
  end
until *end of trial*

experience set $\mathcal{E}$ is gathered, each neural net B_{q_i} is trained by its associated experience set $\mathcal{E}_{q_i}$. At each iteration a pattern set $\mathcal{P}_{q_i}$ is generated based on $\mathcal{E}_{q_i}$:

$$\mathcal{P}_{q_i} = \{(input_l, target_l), l = 1, ..., |\mathcal{E}_{q_i}|)\},$$

where $input_l = ({s_l}^{\otimes}, a_l)$ and $target_l = R({s_l}^{\otimes}, a_l) + \gamma \max_{a'} Q({s_l}^{\otimes'}, a')$ such that $({s_l}^{\otimes}, a_l, {s_l}^{\otimes'}, R({s_l}^{\otimes}, a_l), q_i) \in \mathcal{E}_{q_i}$. The pattern set is used to train the neural net B_{q_i}. We use Rprop (Riedmiller & Braun, 1993) to update the weights in each neural net, as it is known to be a fast and efficient method for batch learning (Riedmiller, 2005). In each cycle of LCNFQ (Algorithm 1), the training schedule starts from networks that are associated with accepting states of the automaton and goes backward until it reaches the networks that are associated to the initial states. In this way we allow the Q-value to back-propagate through the networks. LCNFQ stops when the generated policy satisfies the LTL property and stops improving for long enough.

Remark 4.1 *We tried different embeddings such as one hot encoding (Harris & Harris, 2010) and integer encoding in order to approximate the global Q-function with a single feedforward net. However, we observed poor performance since these encoding allows the network to assume an ordinal relationship between automaton states. Therefore, we turned to the final solution of employing n separate neural nets that work together in a hybrid manner to approximate the global Q-function.* ⌟

Recall that the reward function (3) only returns a positive value when the agent has a transition to an accepting state in the product MDP. Therefore, if accepting states are reachable, by following this reward function the agent is able to come up with a policy ${Pol}^{\otimes^*}$ that leads to the accepting states. This means that the trace of read labels over $\mathcal{S}$ (see Definition 4.1) results in an automaton state to be accepting. Therefore, the trace over the original MDP is a trace that satisfies the given logical property. Recall that the optimal policy has the highest expected reward comparing to other policies. Consequently, the optimal policy has the highest expected probability of reaching to the accepting set, i.e. satisfying the LTL property.

The next section studies state space discretization as the most popular alternative approach to solving infinite-state MDPs.

5 VORONOI QUANTIZER

Inspired by (Lee & Lau, 2004), we propose a version of Voronoi quantizer that is able to discretize the state space of the product MDP ${\mathcal{S}}^{\otimes}$. In the beginning, $\mathcal{C}$ is initialized to consist of just one c_1, which corresponds to the initial state. This means that the agent views the entire state space as a homogeneous region when no apriori knowledge is available. Subsequently, when the agent explores, the Euclidean distance between each newly visited state and its nearest neighbor is calculated. If this distance is greater than a threshold value Δ called "minimum resolution", or if the new state $s^{\otimes}$ has a never-visited automaton state then the newly visited state is appended to $\mathcal{C}$. Therefore, as the agent continues to explore, the size of $\mathcal{C}$ would increase until the relevant parts of the state space are

Algorithm 2: Episodic VQ

input : MDP $\mathbf{M}$, minimum resolution Δ
output : Approximated Q-function Q

initialize $Q(c_1, a) = 0,\ \forall a \in \mathcal{A}$
repeat
  initialize $c_1 =$ initial state
  set $c = c_1$
  $\alpha = \arg\max_{a \in \mathcal{A}} Q(c, a)$
  repeat
    execute action α and observe the next state (s', q)
    if $\mathcal{C}_q$ *is empty* **then**
      append $c_{new} = (s', q)$ to $\mathcal{C}_q$
      initialize $Q(c_{new}, a) = 0,\ \forall a \in \mathcal{A}$
    else
      determine the nearest neighbor c_{new} within $\mathcal{C}_q$
      if $c_{new} = c$ **then**
        if $||c - (s', q)||_2 > \Delta$ **then**
          append $c_{new} = (s', q)$ to $\mathcal{C}_q$
          initialize $Q(c_{new}, a) = 0,\ \forall a \in \mathcal{A}$
        end
      else
        $Q(c, \alpha) = (1 - \mu)Q(c, \alpha) + \mu[R(c, \alpha) + \gamma \max_{a'}(Q(c_{new}, a'))]$
      end
    end
    $c = c_{new}$
  until *end of trial*
until *end of trial*

partitioned. In our algorithm, the set $\mathcal{C}$ has n disjoint subsets where $n = |\mathcal{Q}|$ and $\mathcal{Q}$ is the finite set of states of the automaton. Each subset $\mathcal{C}^{q_j},\ j = 1, ..., n$ contains the centroids of those Voronoi cells that have the form of $c_i^{q_j} = (\cdot, q_j)$, i.e. $\bigcup_i^m c_i^{q_j} = \mathcal{C}^{q_j}$ and $\mathcal{C} = \bigcup_{j=1}^n \mathcal{C}^{q_j}$. Therefore, a Voronoi cell

$$\{(s, q_j) \in \mathcal{S}^{\otimes}, ||(s, q_j) - c_i^{q_j}||_2 \leq ||(s, q_j) - c_{i'}^{q_j}||_2\}$$

is defined by the nearest neighbor rule for any $i' \neq i$. The VQ algorithm is presented in Algorithm 2. The proposed algorithm consist of several resets at which the agent is forced to re-localize to its initial state s_0. Each reset is called an episode, as such in the rest of the paper we call this algorithm episodic VQ.

6 Fitted Value Iteration

In this section we propose a modified version of FVI that can handle the product MDP. The global value function $v : \mathcal{S}^{\otimes} \to \mathbb{R}$, or more specifically $v : \mathcal{S} \times \mathcal{Q} \to \mathbb{R}$, consists of n number of sub-value functions where $n = |\mathcal{Q}|$. For each $q_j \in \mathcal{Q}$, the sub-value function $v^{q_j} : \mathcal{S} \to \mathbb{R}$ returns the value the states of the form (s, q_j). As we will see shortly, in a same manner as LCNFQ, the sub-value functions are not decoupled.

Let $P^{\otimes}(dy|s^{\otimes}, a)$ be the distribution over $\mathcal{S}^{\otimes}$ for the successive state given that the current state is $s^{\otimes}$ and the current action is a. For each state (s, q_j), the Bellman update over each sub-value function v^{q_j} is defined as:

$$Tv^{q_j}(s) = \sup_{a \in \mathcal{A}} \{ \int v(y) P^{\otimes}(dy|(s, q_j), a) \}, \tag{4}$$

where T is the Bellman operator (Hernández-Lerma & Lasserre, 2012). The update in (4) is a special case of general Bellman update as it does not have a running reward and the (terminal) reward is embedded via value function initialization. The value function is initialized according to the following rule:

$$v(s^{\otimes}) = \begin{cases} r_p & \text{if } s^{\otimes} \in \mathbb{A}, \\ r_n & \text{otherwise.} \end{cases} \tag{5}$$

Algorithm 3: FVI

input : MDP $\mathbf{M}$, a set of samples $\{s_i^{\otimes}\}_{i=1}^{k} = \{(s_i, q_j)\}_{i=1}^{k}$ for each $q_j \in \mathcal{Q}$, Monte Carlo sampling number Z, smoothing parameter h

output : approximated value function Lv

initialize Lv

sample $\mathcal{Y}_a^Z(s_i, q_j),\ \forall q_j \in \mathcal{Q},\ \forall i = 1, ..., k\ ,\ \forall a \in \mathcal{A}$

repeat

  for $j = |\mathcal{Q}|$ ***to*** 1 **do**

    $\forall q_j \in \mathcal{Q},\ \forall i = 1, ..., k\ ,\ \forall a \in \mathcal{A}$ calculate $I_a((s_i, q_j)) = 1/Z \sum_{y \in \mathcal{Y}_a^Z(s_i, q_j)} Lv(y)$ using (6)

    for each state (s_i, q_j), update $v^{q_j}(s_i) = \sup_{a \in \mathcal{A}}\{I_a((s_i, q_j))\}$ in (6)

  end

until *end of trial*

where r_p and r_n are defined in (3). The main hurdle in executing the Bellman operator in continuous state MDPs, as in (4), is that no analytical representation of the value function v and also sub-value functions $v^{q_j},\ q_j \in \mathcal{Q}$ is available. Therefore, we employ an approximation method by introducing the operator L. The operator L constructs an approximation of the value function denoted by Lv and of each sub-value function v^{q_j} which we denote by Lv^{q_j}. For each $q_j \in \mathcal{Q}$ the approximation is based on a set of points $\{(s_i, q_j)\}_{i=1}^{k} \subset \mathcal{S}^{\otimes}$ which are called centers. For each q_j, the centers $i = 1, ..., k$ are distributed uniformly over $\mathcal{S}$ such that they uniformly cover $\mathcal{S}$.

We employ a kernel-based approximator for our FVI algorithm. Kernel-based approximators have attracted a lot of attention mostly because they perform very well in high-dimensional state spaces (Stachurski, 2008). One of these methods is the kernel averager in which for any state (s, q_j) the approximate value function is represented by

$$Lv(s, q_j) = Lv^{q_j}(s) = \frac{\sum_{i=1}^{k} K(s_i - s) v^{q_j}(s_i)}{\sum_{i=1}^{k} K(s_i - s)}, \tag{6}$$

where the kernel $K : \mathcal{S} \to \mathbb{R}$ is a radial basis function, such as $e^{-|s - s_i|/h}$, and h is smoothing parameter. Each kernel has a center s_i and the value of it decays to zero as s diverges from s_i. This means that for each $q_j \in \mathcal{Q}$ the approximation operator L is a convex combination of the values of the centers $\{s_i\}_{i=1}^{k}$ with larger weight given to those values $v^{q_j}(s_i)$ for which s_i is close to s. Note that the smoothing parameter h controls the weight assigned to more distant values (see Section A.3).

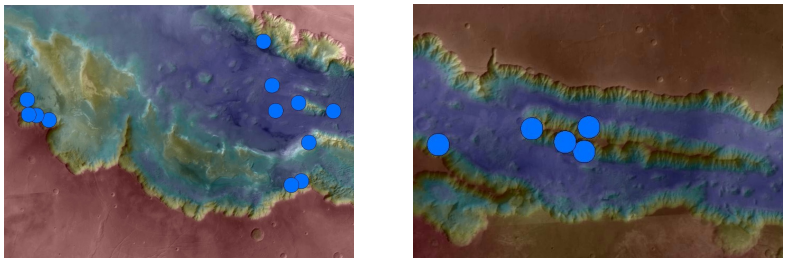
In order to approximate the integral in Bellman update (4) we use a Monte Carlo sampling technique (Shonkwiler & Mendivil, 2009). For each center (s_i, q_j) and for each action a, we sample the next state $y_a^z(s_i, q_j)$ for $z = 1, ..., Z$ times and append it to set of Z subsequent states $\mathcal{Y}_a^Z(s_i, q_j)$. We then replace the integral with

$$I_a(s_i, q_j) = \frac{1}{Z} \sum_{z=1}^{Z} Lv(y_a^z(s_i, q_j)). \tag{7}$$

The approximate value function Lv is initialized according to (5). In each cycle of FVI, the approximate Bellman update is first performed over the sub-value functions that are associated with accepting states of the automaton, i.e. those that have initial value of r_p, and then goes backward until it reaches the sub-value functions that are associated to the initial states. In this manner, we allow the state values to back-propagate through the transitions that connects the sub-value function via (7). Once we have the approximated value function, we can generate the optimal policy by following the maximum value (Algorithm 3).

7 Experimental Results

We describe a mission planning architecture for an autonomous Mars-rover that uses LCNFQ to follow a mission on Mars. The scenario of interest is that we start with an image from the surface of Mars and then we add the desired labels from $2^{\mathcal{AP}}$, e.g. safe or unsafe, to the image. We assume that we know the highest possible disturbance caused by different factors (such as sand storms) on the



(a) Melas Chasma (b) Coprates Chasma

Figure 1: Melas Chasma and Coprates Chasma, in the central and eastern portions of Valles Marineris. Map color spectrum represents elevation, where red is high and blue is low. (Image courtesy of NASA, JPL, Caltech and University of Arizona.)

rover motion. This assumption can be set to be very conservative given the fact that there might be some unforeseen factors that we did not take into account.

The next step is to express the desired mission in LTL format and run LCNFQ on the labeled image before sending the rover to Mars. We would like the rover to satisfy the given LTL property with the highest probability possible starting from any random initial state (as we can not predict the landing location exactly). Once LCNFQ is trained we use the network to guide the rover on the Mars surface.

We compare LCNFQ with Voronoi quantizer and FVI and we show that LCNFQ outperforms these methods.

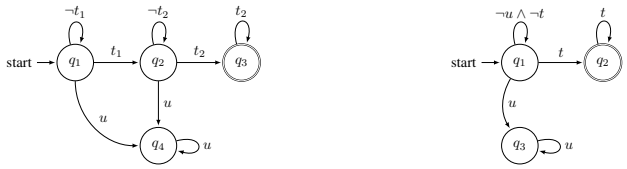
7.1 MDP Structure

In this numerical experiment the area of interest on Mars is Coprates quadrangle, which is named after the Coprates River in ancient Persia (see Section A.4). There exist a significant number of signs of water, with ancient river valleys and networks of stream channels showing up as sinuous and meandering ridges and lakes. We consider two parts of Valles Marineris, a canyon system in Coprates quadrangle (Fig. 1). The blue dots, provided by NASA, indicate locations of recurring slope lineae (RSL) in the canyon network. RSL are seasonal dark streaks regarded as the strongest evidence for the possibility of liquid water on the surface of Mars. RSL extend downslope during a warm season and then disappear in the colder part of the Martian year (McEwen et al., 2014). The two areas mapped in Fig. 1, Melas Chasma and Coprates Chasma, have the highest density of known RSL.

For each case, let the entire area be our MDP state space $\mathcal{S}$, where the rover location is a single state $s \in \mathcal{S}$. At each state $s \in \mathcal{S}$, the rover has a set of actions $\mathcal{A} = \{left, right, up, down, stay\}$ by which it is able to move to other states: at each state $s \in \mathcal{S}$, when the rover takes an action $a \in \{left, right, up, down\}$ it is moved to another state (e.g., s') towards the direction of the action with a range of movement that is randomly drawn from $(0, D]$ unless the rover hits the boundary of the area which forces the rover to remain on the boundary. In the case when the rover chooses action $a = stay$ it is again moved to a random place within a circle centered at its current state and with radius $d \ll D$. Again, d captures disturbances on the surface of Mars and can be tuned accordingly.

With $\mathcal{S}$ and $\mathcal{A}$ defined we are only left with the labelling function $L : \mathcal{S} \to 2^{\mathcal{AP}}$ which assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \subseteq 2^{\mathcal{AP}}$. With the labelling function, we are able to divide the area into different regions and define a logical property over the traces that the agent generates. In this particular experiment, we divide areas into three main regions: neutral, unsafe and target. The target label goes on RSL (blue dots), the unsafe label lays on the parts with very high elevation (red coloured) and the rest is neutral. In this example we assume that the labels do not overlap each other.

Note that when the rover is deployed to its real mission, the precise landing location is not known. Therefore, we should take into account the randomness of the initial state s_0. The dimensions of the area of interest in Fig. 1.a are 456.98×322.58 km and in Fig. 1.b are 323.47×215.05 km. The



(a) LDBA expressing (8)

(b) LDBA expressing (9)

Figure 2: Generated LDBAs

Table 1: Simulation results

Melas Chasma					
Algorithm	**Sample Complexity**	$U^{Pol^\star}(s_0)$	**Success Rate**†	**Training Time*****(s)**	**Iteration Num.**
LCNFQ	**7168** samples	**0.0203**	**99%**	95.64	**40**
VQ ($\Delta = 0.4$)	27886 samples	0.0015	**99%**	1732.35	2195
VQ ($\Delta = 1.2$)	7996 samples	0.0104	97%	273.049	913
VQ ($\Delta = 2$)	-	0	0%	-	-
FVI	40000 samples	0.0133	98%	**4.12**	80
Coprates Chasma					
Algorithm	**Sample Complexity**	$U^{Pol^\star}(s_0)$	**Success Rate**†	**Training Time*****(s)**	**Iteration Num.**
LCNFQ	**2680 samples**	**0.1094**	**98%**	166.13	**40**
VQ ($\Delta = 0.4$)	8040 samples	0.0082	**98%**	3666.18	3870
VQ ($\Delta = 1.2$)	3140 samples	0.0562	96%	931.33	2778
VQ ($\Delta = 2$)	-	0	0%	-	-
FVI	25000 samples	0.0717	97%	**2.16**	80

† Testing the trained agent (for 100 trials) * Average for 10 trainings

diameter of each RSL is 19.12 km. Other parameters in this numerical example have been set as $D = 2$ km, $d = 0.02$ km, the reward function parameter $y = 1$ for LCNFQ and $y = 0$ for VQ and FVI, $M = 1$, $m = 0.05$ and $\mathcal{AP} = \{$neutral, unsafe, target_1, target_2$\}$.

7.2 SPECIFICATIONS

The first control objective in this numerical example is expressed by the following LTL formula over Melas Chasma (Fig. 1.a):

$$\Diamond(t_1 \wedge \Diamond t_2) \wedge \Box(t_2 \rightarrow \Box t_2) \wedge \Box(u \rightarrow \Box u), \tag{8}$$

where n stands for "neutral", t_1 stands for "target 1", t_2 stands for "target 2" and u stands for "unsafe". Target 1 are the RSL (blue bots) on the right with a lower risk of the rover going to unsafe region and the target 2 label goes on the left RSL that are a bit riskier to explore. Conforming to (8) the rover has to visit the target 1 (any of the right dots) at least once and then proceed to the target 2 (left dots) while avoiding unsafe areas. Note that according to $\Box(u \rightarrow \Box u)$ in (8) the agent is able to go to unsafe area u (by climbing up the slope) but it is not able to come back due to the risk of falling. With (8) we can build the associated Büchi automaton as in Fig. 2.a.

The second formula focuses more on safety and we are going to employ it in exploring Coprates Chasma (Fig. 1.b) where a critical unsafe slope exists in the middle of this region.

$$\Diamond t \wedge \Box(t \rightarrow \Box t) \wedge \Box(u \rightarrow \Box u) \tag{9}$$

In (9), t refers to "target", i.e. RSL in the map, and u stands for "unsafe". According to this LTL formula, the agent has to eventually reach the target ($\Diamond t$) and stays there ($\Box(t \rightarrow \Box t)$). However, if the agent hits the unsafe area it can never comes back and remains there forever ($\Box(u \rightarrow \Box u)$). With (9) we can build the associated Büchi automaton as in Fig. 2.b. Having the Büchi automaton for each formula, we are able to use Definition 4.1 to build product MDPs and run LCNFQ on both.

7.3 SIMULATION RESULTS

This section presents the simulation results. All simulations are carried on a machine with a 3.2GHz Core i5 processor and 8GB of RAM, running Windows 7. LCNFQ has four feedforward neural networks for (8) and three feedforward neural networks for (9), each associated with an automaton state in Fig. 2.a and Fig. 2.b. We assume that the rover lands on a random safe place and has to find its way to satisfy the given property in the face of uncertainty. The learning discount factor γ is also set to be equal to 0.9.

Fig. 4 in Section A.5 gives the results of learning for LTL formulas (8) and (9). At each state $s^{\otimes}$, the robot picks an action that yields highest $Q(s^{\otimes}, \cdot)$ and by doing so the robot is able to generate a control policy $Pol^{\otimes^*}$ over the state space $\mathcal{S}^{\otimes}$. The control policy $Pol^{\otimes^*}$ induces a policy Pol^* over the state space $\mathcal{S}$ and its performance is shown in Fig. 4.

Next, we investigate the episodic VQ algorithm as an alternative solution to LCNFQ. Three different resolutions ($\Delta = 0.4,\ 1.2,\ 2$ km) are used to see the effect of the resolution on the quality of the generated policy. The results are presented in Table 1, where VQ with $\Delta = 2$ km fails to find a satisfying policy in both regions, due to the coarseness of the resulted discretisation. A coarse partitioning result in the RL not to be able to efficiently back-propagate the reward or the agent to be stuck in some random-action loop as sometimes the agent's current cell is large enough that all actions have the same value. In Table 1, training time is the empirical time that is taken to train the algorithm and travel distance is the distance that agent traverses from initial state to final state. We show the generated policy for $\Delta = 1.2$ km in Fig. 5 in Section A.5. Additionally, Fig. 7 in Section A.6 depicts the resulted Voronoi discretisation after implementing the VQ algorithm. Note that with VQ only those parts of the state space that are relevant to satisfying the property are accurately partitioned.

Finally, we present the results of FVI method in Fig 6 in Section A.5 for the LTL formulas (8) and (9). The FVI smoothing parameter is $h = 0.18$ and the sampling time is $Z = 25$ for both regions where both are empirically adjusted to have the minimum possible value for FVI to generate satisfying policies. The number of basis points also is set to be 100, so the sample complexity of FVI is $100 \times Z \times |\mathcal{A}| \times (|\mathcal{Q}| - 1)$. We do not sample the states in the product automaton that are associated to the accepting state of the automaton since when we reach the accepting state the property is satisfied and there is no need for further exploration. Hence, the last term is $(|\mathcal{Q}| - 1)$. However, if the property of interest produces an automaton that has multiple accepting states, then we need to sample those states as well. Note that in Table 1, in terms of timing, FVI outperforms the other methods. However, we have to remember that FVI is an approximate DP algorithm, which inherently needs an approximation of the transition probabilities. Therefore, as we have seen in Section 6 in (7), for the set of basis points we need to sample the subsequent states. This reduces FVI applicability as it might not be possible in practice.

Additionally, both FVI and episodic VQ need careful hyper-parameter tuning to generate a satisfying policy, i.e., h and Z for FVI and Δ for VQ. The big merit of LCNFQ is that it does not need any external intervention. Further, as in Table 1, LCNFQ succeeds to efficiently generate a better policy compared to FVI and VQ. LCNFQ has less sample complexity while at the same time produces policies that are more reliable and also has better expected reward, i.e. higher probability of satisfying the given property.

8 CONCLUSION

This paper proposes LCNFQ, a method to train Q-function in a continuous-state MDP such that the resulting traces satisfy a logical property. The proposed algorithm uses hybrid modes to automatically switch between neural nets when it is necessary. LCNFQ is successfully tested in a numerical example to verify its performance.

REFERENCES

Mohammed Alshiekh, Roderick Bloem, Ruediger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *arXiv preprint arXiv:1708.08611*, 2017.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *ICML*, volume 70, pp. 166–175, 2017.

Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Vojtěch Forejt, Jan Křetínskỳ, Marta Kwiatkowska, David Parker, and Mateusz Ujma. Verification of Markov decision processes using learning algorithms. In *ATVA*, pp. 98–114. Springer, 2014.

Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*, volume 39. CRC press, 2010.

Rolando Cavazos-Cadena, Eugene A Feinberg, and Raúl Montes-De-Oca. A note on the existence of optimal policies in total reward dynamic programs with compact action sets. *Mathematics of Operations Research*, 25(4):657–666, 2000.

Kenji Doya. Reinforcement learning in continuous time and space. *Neural computation*, 12(1): 219–245, 2000.

Richard Durrett. *Essentials of stochastic processes*, volume 1. Springer, 1999.

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *JMLR*, 6(Apr):503–556, 2005.

Jie Fu and Ufuk Topcu. Probably approximately correct MDP learning and control with temporal logic constraints. In *Robotics: Science and Systems X*, 2014.

Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning*, pp. 261–268. Elsevier, 1995.

R. Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, 1984.

David Harris and Sarah Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

Onésimo Hernández-Lerma and Jean B Lasserre. *Further topics on discrete-time Markov control processes*, volume 42. Springer Science & Business Media, 2012.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2): 251–257, 1991.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Ivan SK Lee and Henry YK Lau. Adaptive state space partitioning for reinforcement learning. *Engineering applications of artificial intelligence*, 17(6):577–588, 2004.

Long-H Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3/4):69–97, 1992.

Javier Lope and Jose Martin. Learning autonomous helicopter flight with evolutionary reinforcement learning. In *International Conference on Computer Aided Systems Theory*, pp. 75–82. Springer, 2009.

Alfred S McEwen, Colin M Dundas, Sarah S Mattson, Anthony D Toigo, Lujendra Ojha, James J Wray, Matthew Chojnacki, Shane Byrne, Scott L Murchie, and Nicolas Thomas. Recurring slope lineae in equatorial regions of Mars. *Nature Geoscience*, 7(1):53, 2014.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, pp. 1928–1937, 2016.

Teodor Mihai Moldovan and Pieter Abbeel. Safe exploration in Markov decision processes. *arXiv preprint arXiv:1205.4810*, 2012.

Dirk Ormoneit and Śaunak Sen. Kernel-based reinforcement learning. *Machine learning*, 49(2): 161–178, 2002.

Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science*, pp. 46–57. IEEE, 1977.

Martin L Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Martin Riedmiller. Concepts and facilities of a neural reinforcement learning control architecture for technical process control. *Neural computing & applications*, 8(4):323–338, 1999.

Martin Riedmiller. Neural fitted Q iteration-first experiences with a data efficient neural reinforcement learning method. In *ECML*, volume 3720, pp. 317–328. Springer, 2005.

Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Neural networks*, pp. 586–591. IEEE, 1993.

Ronald W Shonkwiler and Franklin Mendivil. *Explorations in Monte Carlo Methods*. Springer Science & Business Media, 2009.

Salomon Sickert and Jan Křetínskỳ. MoChiBA: Probabilistic LTL model checking using limit-deterministic Büchi automata. In *ATVA*, pp. 130–137. Springer, 2016.

Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínskỳ. Limit-deterministic Büchi automata for linear temporal logic. In *CAV*, pp. 312–332. Springer, 2016.

Stephen L Smith, Jana Tumová, Calin Belta, and Daniela Rus. Optimal path planning for surveillance with temporal-logic constraints. *The International Journal of Robotics Research*, 30(14):1695–1708, 2011.

Yong Song, Yi-bin Li, Cai-hong Li, and Gui-fang Zhang. An efficient initialization approach of Q-learning for mobile robots. *International Journal of Control, Automation and Systems*, 10(1): 166–172, 2012.

John Stachurski. Continuous state dynamic programming via nonexpansive approximation. *Computational Economics*, 31(2):141–160, 2008.

Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *NIPS*, pp. 1038–1044, 1996.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Gerald Tesauro. TD-Gammon: A self-teaching Backgammon program. In *Applications of Neural Networks*, pp. 267–285. Springer, 1995.

Andrea L Thomaz and Cynthia Breazeal. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artificial Intelligence*, 172(6-7):716–737, 2008.

Ilya Tkachev, Alexandru Mereacre, Joost-Pieter Katoen, and Alessandro Abate. Quantitative model-checking of controlled discrete-time Markov processes. *Information and Computation*, 253:1–35, 2017.

Hado Van Hasselt and Marco A Wiering. Reinforcement learning in continuous action spaces. In *ADPRL*, pp. 272–279. IEEE, 2007.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Eric M Wolff, Ufuk Topcu, and Richard M Murray. Robust control of uncertain Markov decision processes with temporal logic specifications. In *CDC*, pp. 3372–3379. IEEE, 2012.

A SUPPLEMENTARY MATERIALS

A.1 PRELIMINARIES

Definition A.1 (Path) *In a continuous-state MDP **M**, an infinite path ρ starting at s_0 is a sequence of states $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ...$ such that every transition $s_i \xrightarrow{a_i} s_{i+1}$ is possible in **M**, i.e. s_{i+1} belongs to the smallest Borel set B such that $P(B|s_i, a_i) = 1$ (or in a discrete MDP, $P(s_{i+1}|s_i, a_i) > 0$). We might also denote ρ as $s_0..$ to emphasize that ρ starts from s_0.* ⌟

Definition A.2 (Stationary Policy) *A stationary (randomized) policy $Pol : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a mapping from each state $s \in \mathcal{S}$, and action $a \in \mathcal{A}$ to the probability of taking action a in state s. A deterministic policy is a degenerate case of a randomized policy which outputs a single action at a given state, that is $\forall s \in \mathcal{S},\ \exists a \in \mathcal{A},\ Pol(s, a) = 1$.* ⌟

In an MDP **M**, we define a function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_0^+$ that denotes the immediate scalar bounded reward received by the agent from the environment after performing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.

Definition A.3 (Expected (Infinite-Horizon) Discounted Reward) *For a policy Pol on an MDP **M**, the expected discounted reward is defined as (Sutton & Barto, 1998):*

$$U^{Pol}(s) = \mathbb{E}^{Pol}[\sum_{n=0}^{\infty} \gamma^n\ R(s_n, Pol(s_n))|s_0 = s], \tag{10}$$

where $\mathbb{E}^{Pol}[\cdot]$ denotes the expected value given that the agent follows policy Pol, $\gamma \in [0, 1)$ is a discount factor and $s_0, ..., s_n$ is the sequence of states generated by policy Pol up to time step n. ⌟

Definition A.4 (Optimal Policy) *Optimal policy Pol^* is defined as follows:*

$$Pol^*(s) = \arg\sup_{Pol \in \mathcal{D}} U^{Pol}(s),$$

where $\mathcal{D}$ is the set of all stationary deterministic policies over the state space $\mathcal{S}$. ⌟

A.2 POLICY SYNTHESIS

The simplest way to solve an infinite-state MDP with RL is to discretise the state space and then to use the conventional methods in RL to find the optimal policy (Stachurski, 2008). Although this method can work well for many problems, the resulting discrete MDP is often inaccurate and may not capture the full dynamics of the original MDP. One might argue that by increasing the number of discrete states the latter problem can be resolved. However, the more states we have the more expensive and time-consuming our computations will be. Thus, MDP discretisation has to always deal with the trade off between accuracy and the curse of dimensionality.

A.2.1 CLASSICAL Q-LEARNING

Let the MDP **M** be a finite-state MDP. Q-learning (QL), a sub-class of RL algorithms, is extensively used to find the optimal policy for a given finite-state MDP (Sutton & Barto, 1998). For each state $s \in \mathcal{S}$ and for any available action $a \in \mathcal{A}$, QL assigns a quantitative value $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which is initialized with an arbitrary and finite value for all state-action pairs. As the agent starts learning and receiving rewards, the Q-function is updated by the following rule when the agent takes action a at state s:

$$Q(s,a) \leftarrow Q(s,a) + \mu[R(s,a) + \gamma \max_{a' \in \mathcal{A}}(Q(s',a')) - Q(s,a)], \tag{11}$$

where $Q(s,a)$ is the Q-value corresponding to state-action (s,a), $0 < \mu \leq 1$ is called learning rate or step size, $R(s,a)$ is the reward obtained for performing action a in state s, γ is the discount factor, and s' is the state obtained after performing action a. Q-function for the rest of the state-action pairs remains unchanged.

Under mild assumptions, for finite-state and finite-action spaces QL converges to a unique limit, as long as every state action pair is visited infinitely often (Watkins & Dayan, 1992). Once QL converges, the optimal policy $Pol^* : \mathcal{S} \to \mathcal{A}$ can be generated by selecting the action that yields the highest Q, i.e.,

$$Pol^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q(s,a),$$

where Pol^* is the same optimal policy that can be generated via DP with Bellman operation. This means that when QL converges, we have

$$Q(s,a) = R(s,a) + \mathbb{E}^{Pol^*}[\sum_{n=1}^{\infty} \gamma^n R(s_n, Pol^*(s_n)) | s_1 = s'], \tag{12}$$

where $s' \in B$ is the agent new state after choosing action a at s such that $P(B|s,a) = 1$.

A.2.2 NEURAL FITTED Q-ITERATION

Recall the QL update rule (11), in which the agent stores the Q-values for all possible state-action pairs. In the case when the MDP has a continuous state space it is not possible to directly use standard QL since it is practically infeasible to store $Q(s,a)$ for every $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Thus, we have to turn to function approximators in order to approximate the Q-values of different state-action pairs of the Q-function. Neural Fitted Q-iteration (NFQ) (Riedmiller, 2005) is an algorithm that employs neural networks (Hornik et al., 1989) to approximate the Q-function, due to the ability of neural networks to generalize and exploit the set of samples. NFQ, is the core behind Google famous algorithm Deep Reinforcement Learning (Mnih et al., 2015).

The update rule in (11) can be directly implemented in NFQ. In order to do so, a loss function has to be introduced that measures the error between the current Q-value and the new value that has to be assigned to the current Q-value, namely

$$L = (Q(s,a) - (R(s,a) + \gamma \max_{a'} Q(s',a')))^2. \tag{13}$$

Over this error, common gradient descent techniques can be applied to adjust the weights of the neural network, so that the error is minimized.

In classical QL, the Q-function is updated whenever a state-action pair is visited. In the continuous state-space case, we may update the approximation in the same way, i.e., update the neural net weights once a new state-action pair is visited. However, in practice, a large number of trainings might need

to be carried out until an optimal or near optimal policy is found. This is due to the uncontrollable changes occurring in the Q-function approximation caused by unpredictable changes in the network weights when the weights are adjusted for one certain state-action pair (Riedmiller, 1999). More specifically, if at each iteration we only introduce a single sample point the training algorithm tries to adjust the weights of the neural network such that the loss function becomes minimum for that specific sample point. This might result in some changes in the network weights such that the error between the network output and the previous output of sample points becomes large and failure to approximate the Q-function correctly. Therefore, we have to make sure that when we update the weights of the neural network, we explicitly introduce previous samples as well: this technique is called "experience replay" (Lin, 1992) and detailed later.

The core idea underlying NFQ is to store all previous experiences and then reuse this data every time the neural Q-function is updated. NFQ can be seen as a batch learning method in which there exists a training set that is repeatedly used to train the agent. In this sense NFQ is an offline algorithm as experience gathering and learning happens separately.

We would like to emphasize that neural-net-based algorithms exploit the positive effects of generalization in approximation while at the same time avoid the negative effects of disturbing previously learned experiences when the network properly learns (Riedmiller, 2005). The positive effect of generalization is that the learning algorithm requires less experience and the learning process is highly data efficient.

A.2.3 VORONOI QUANTIZER

As stated earlier, many existing RL algorithms, e.g. QL, assume a finite state space, which means that they are not directly applicable to continuous state-space MDPs. Therefore, if classical RL is employed to solve an infinite-state MDP, the state space has to be discretized first and then the new discrete version of the problem has to be tackled. The discretization can be done manually over the state space. However, one of the most appealing features of RL is its autonomy. In other words, RL is able to achieve its goal, defined by the reward function, with minimum supervision from a human. Therefore, the state space discretization should be performed as part of the learning task, instead of being fixed at the start of the learning process.

Nearest neighbor vector quantization is a method for discretizing the state space into a set of disjoint regions (Gray, 1984). The Voronoi Quantizer (VQ) (Lee & Lau, 2004), a nearest neighbor quantizer, maps the state space $\mathcal{S}$ onto a finite set of disjoint regions called Voronoi cells. The set of centroids of these cells is denoted by $\mathcal{C} = \{c_i\}_{i=1}^{m},\ c_i \in \mathcal{S}$, where m is the number of the cells. Therefore, designing a nearest neighbor vector quantizer boils down to coming up with the set $\mathcal{C}$. With $\mathcal{C}$, we are able to use QL and find an approximation of the optimal policy for a continuous-state space MDP. The details of how the set of centroids $\mathcal{C}$ is generated as part of the learning task in discussed in the body of the paper.

A.2.4 FITTED VALUE ITERATION

Finally, this section introduces Fitted Value Iteration (FVI) for continuous-state numerical dynamic programming using a function approximator (Gordon, 1995). In standard value iteration the goal is to find a mapping (called value function) from the state space to $\mathbb{R}$ such that it can lead the agent to find the optimal policy. The value function in our setup is (10) when Pol is the optimal policy, i.e. U^{Pol^*}. In continuous state spaces, no analytical representation of the value function is in general available. Thus, an approximation can be obtained numerically through approximate value iteration, which involves approximately iterating the Bellman operator T on some initial value function (Stachurski, 2008). FVI is explored more in the paper.

A.3 KERNEL AVERAGER

It has been proven that FVI is stable and converging when the approximation operator is non-expansive (Gordon, 1995). The operator L is said to be non-expansive if:

$$\sup_{s\in\mathcal{S}} |Lv_{t+1}^{q_j}(s) - Lv_t^{q_j}(s)| \leq \sup_{s\in\mathcal{S}} |v_{t+1}^{q_j}(s) - v_t^{q_j}(s)|,$$

where $Lv_t^{q_j}(s)$ is the approximated value function at (s, q_j) at iteration t of the algorithm. The kernel averager is a non-expansive approximator (Stachurski, 2008).

A.4 COPRATES QUADRANGLE

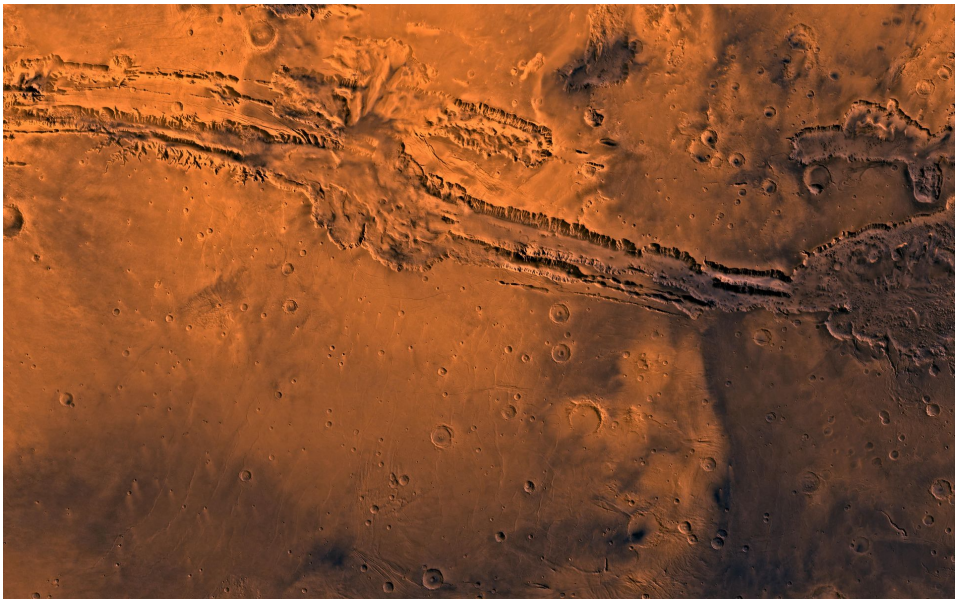
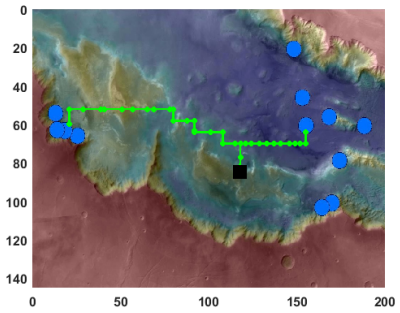
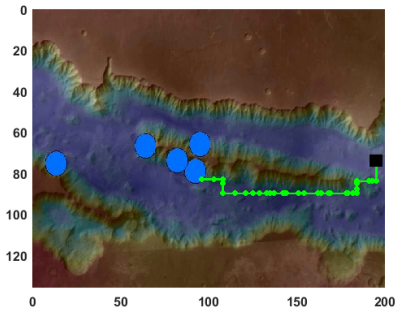


Figure 3: Coprates quadrangle (Image courtesy of NASA, JPL and USGS.)

A.5 GENERATED POLICIES

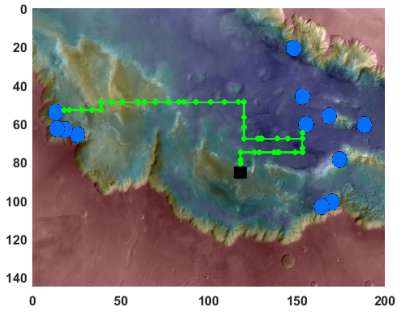


(a) The generated path in Melas Chasma when the landing location, i.e. the black rectangle, is $(118,\ 85)$

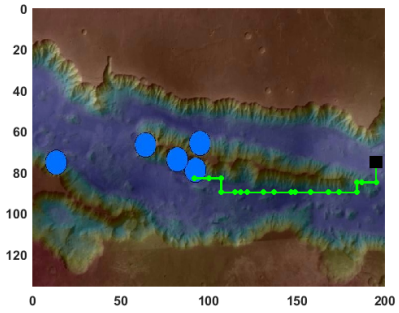


(b) The generated path in Coprates Chasma when the landing location, i.e. the the black rectangle, is $(194,\ 74)$

Figure 4: Results of learning with LCNFQ

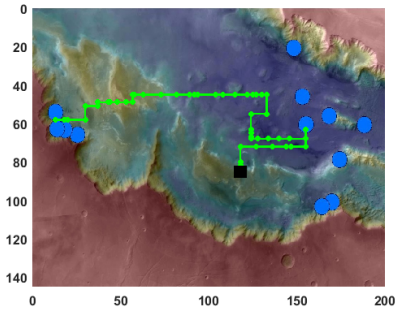


(a) The generated path in Melas Chasma when the landing location, i.e. the black rectangle, is $(118,\ 85)$

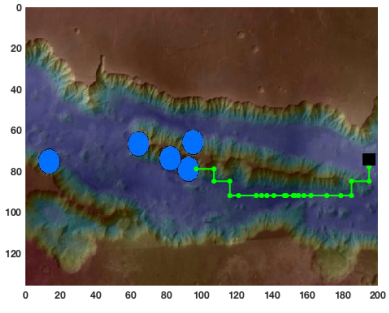


(b) The generated path in Coprates Chasma when the landing location, i.e. the black rectangle, is $(194,\ 74)$

Figure 5: Results of learning with episodic VQ



(a) The generated path in Melas Chasma when the landing location, i.e. the black rectangle, is $(118,\ 85)$



(b) The generated path in Coprates Chasma when the landing location, i.e. the black rectangle, is $(194,\ 74)$

Figure 6: Results of learning with FVI

A.6 GENERATED VORONOI CELLS

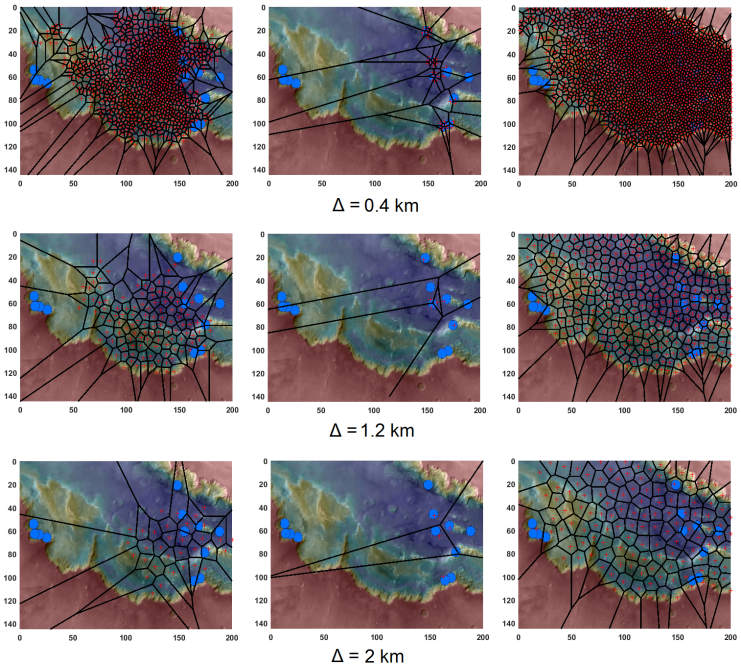


Figure 7: VQ generated cells in Melas Chasma for different resolutions