Region Enhanced Neural Q-learning in Partially Observable Markov Decision Processes

Thijs Kooi
Department of Artificial Intelligence
University of Groningen
T.Kooi@ai.rug.nl

Bachelor Thesis

Abstract— **To get a robot to perform tasks autonomously, the robot has to plan its behavior and make decisions based on the input it receives. Unfortunately, contemporary robot sensors and actuators are subject to noise, rendering optimal decision making a stochastic process. To model this process, partially observable Markov decision processes (POMDPs) can be applied. In this paper we introduce the RENQ algorithm, a new POMDP algorithm that combines neural networks for estimating Q-values with the construction of a spatial pyramid over the state space. RENQ essentially uses region-based belief vectors together with state-based belief vectors, and these are used as inputs to the neural network trained with Q-learning. We compare RENQ to Qmdp and Perseus, two state-of-the-art algorithms for approximately solving model-based POMDPs. The results on three different maze navigation tasks indicate that RENQ outperforms Perseus on all problems and outperforms Qmdp if the problem becomes larger.**

I. Introduction

In robotics, a major goal is getting a robot to learn to perform a task autonomously. This task can involve getting a robot from a start to a goal position. A possible approach to this problem is to use *reinforcement learning* (RL) [24].
Reinforcement learning originated from early work in cybernetics, statistics, psychology and neuroscience, but lately has received a lot of attention from the Artificial Intelligence (AI) and machine learning disciplines [7]. It can be seen as a form of machine learning, but is different from other methods in the sense that the agent does not learn from correct input-output examples, provided by an external supervisor, but has to learn from feedback given by the environment. The feedback the agent receives is typically represented as a numerical value, where a positive reward is given for the display of a desired behavior and a negative reward for an undesired action. The robot's task is to develop a model of what action to take in a given state, thereby maximizing its long term reward.

Unfortunately the robot's actuators do not always act according to the instructions they have been given. When it has to move right it sometimes moves left or bumps into a wall and stays in the same place. This uncertainty in transition can be modelled using a *Markov Decision Process* (MDP). On top the uncertainty in the robot's actuators, there is noise in its sensors readings as well. This partial observability of the world can be captured in a generalization of an MDP, called a Partially Observable Markov Decision Process or POMDP.

Although robot navigation is one of the applications of POMDPs, the method is widely applicable to other problems. Givon and Grosfeld-Nir [5], for instance, used a POMDP for computing optimal termination times of TV shows. An application in an entirely different field is provided by Hoey et at. [6], who use the formalism to handle the uncertainty in observations from a monitor, assisting people with dementia washing their hands.

In this paper we will present RENQ, a novel approach for solving model-based POMDPs. RENQ uses a neural network in combination with Q-learning [25], [26], where the belief-state is given as input to the neural network. RENQ enhances the state-based belief vector input of the network by constructing a spatial pyramid over the state space [9], a method derived from machine vision. At every level of the pyramid, the average belief of a subset of the state space is computed and the enhanced belief state is presented to the neural network. We compare the RENQ algorithm to Qmdp [11], a method known to be fast in handling large state spaces and Perseus [22], an efficient state-of-the-art point-based value iteration algorithm. We will test the algorithms on three different maze navigation tasks and show how RENQ outperforms the other two methods.

Outline. This paper is divided into 6 sections. In section II, we will discuss the basic framework of Markov decision processes, followed by a brief description of value iteration and Q-learning, two techniques for solving MDPs and reinforcement learning problems. In section III the POMDP model will be presented along with two algorithms for handling POMDPs. Next we discuss RENQ, the new method based on Q-learning and neural networks, combined with a spatial pyramid approach. Section V will cover the experimental setup and results acquired with the three POMDP algorithms. A conclusion and discussion will be

presented in section VI.

II. Markov Decision Process

In this section, we will start by giving a formal definition of the MDP model, followed by an example and a description of value iteration. Next, we will sidestep from the MDP formalism and return to regular RL, where we discuss the Q-learning algorithm, which is part of the RENQ method, and two exploration policies.

A. Formal Description

An MDP is characterized by:

- a finite set of states $s \in \mathcal{S}$
- a finite set of actions $a \in \mathcal{A}$
- a transition function $T(s, a, s')$, specifying the probability of ending in state s' after taking action a in state s
- a reward function $R(s, a)$, providing the scalar reward the agent will receive for executing action a in state s

The MDP model assumes a full map of the environment is known to the agent and treats time and sets $\mathcal{S}$ and $\mathcal{A}$ as discrete. The Markov property, named after the Russian statistician Andrei Markov, entails the fact that the state of the environment and the reward the agent receives at time $t+1$ is completely determined by the state of the agent at time t and the action the agent takes. This is called a first order Markov process [19].

$$P(s_t, r_t | s_0, a_0, \ldots, s_{t-1}, a_{t-1}) = P(s_t, r_t | s_{t-1}, a_{t-1}) \quad (1)$$

Higher order Markov processes involve taking into account a longer history. Here, we will only concern ourselves with first order Markovian signals.

As mentioned before, the agent's task is to maximize its long term reward. Since a model of the environment is known to the agent, a simple action sequence could be learned. However, due to the stochastic nature of the state transitions, the agent might very well stray from its course somewhere during the execution of this sequence, resulting in a deviation from the goal state. Therefore a mapping is needed, describing what action to take for each individual state. We call such a mapping a *policy* and denote it as $\pi(s)$. An optimal policy π^*, will maximize the magnitude of the long term reward sequence. With this definition, we can translate the goal of the agent of maximizing the cumulative expected reward into finding an optimal policy.

The optimal policy depends on the nature of the task. We have to consider whether the task yields a *finite* or an *infinite* horizon for decision making. In a finite horizon problem the agent has a limited number of steps and has to achieve its goal within this limit. A policy in this scenario could be different from one in an infinite horizon task. After a few steps the agent might realize it is running out of time and take a different route. However, in most tasks this number of steps is rarely known, therefore we will only concern ourselves with infinite horizon tasks.

Example 1. Consider the small maze navigation task depicted in figure 1, where the goal of the robot is to get from start position S, to the goal position G, taking the least number of steps possible. The agent will be given a reward of 10 for reaching the goal and -1 for each step it has to take, ensuring it will try to get there as efficiently as possible. In every state the agent has 4 actions: $\mathcal{A} = \{\text{go left, go up, go right, go down}\}$

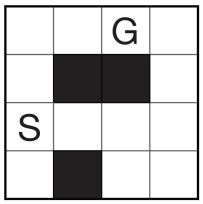

Fig. 1. A simple maze navigation task

Starting from $(1, 3)$, the least number of steps to get to the goal position is 4. The maximum cumulative reward the agent can expect to receive will be: $-1 + -1 + -1 + 10 = 7$. An optimal policy will urge the agent towards the execution of this action sequence.

In every action, there is a possibility the robot's actuators might fail to perform the given command and execute a random action from $\mathcal{A}$. The transition function models this probability.

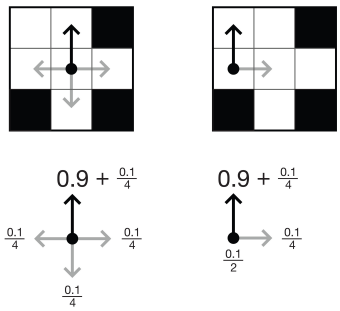

Fig. 2. Example of transition probabilities

If we define a 10% chance of a random action, the transition probabilities will look like those in figure 2. (Left): When intending to go up, the agent has a 90% chance of ending in the planned state, plus the chance of accidently performing the correct action and a 2.5% chance of ending in any of the remaining adjacent states. (Right): The robot is partially surrounded by walls and will remain in the same state when performing an action left or down by accident. Unsurprisingly, numerous other transition functions could be defined.

The task discussed here is an MDP in its simplest form.

Many other things could be added to a maze. For example, if we do not want the robot to damage itself, we could consider penalizing a collision. Also, extra rewards can be added. For instance, a battery charging station or the finding of some valuable object. Different tasks will result in different optimal policies. If we punish the robot for running into a wall, an optimal policy will steer the agent away from walls as much as possible, avoiding the probability of accidently hitting one and additional rewards in the maze will result in a policy trading off between collecting as many objects as possible and in the same time minimizing the number of actions.

In order to pick the action with the best outcome, the agent has to assign a certain value for being in a state, e.g., in the maze navigation example, the states adjacent to the goal state will be very valuable as they are most likely a gateway to the goal state, where the agent will receive a large reward. Farther away from the goal position, the states will become less and less valuable, as it is less likely to get a reward, since in every step there is uncertainty in transitions. A value function is needed to assign a value to each state.

B. Value Functions

The return R_t of a state is defined as the cumulative reward the agent can expect to receive after reaching the given state at time step t. Because of the transition noise, this is an estimate and can only be denoted as an expected value. Mathematically, R_t is written as the sum over all rewards the agent receives at each time step, weighted by a discount factor γ, where $0 \leq \gamma < 1$:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2}$$

Introducing a discount factor has two purposes: (1) it models the preference of the agent to immediate rewards as opposed to those received in the future, and (2) ensures the infinite sum is finite as long as $\gamma < 1$ and the reward sequence is bounded. When the discount factor is set close to 1, the agent will value future rewards greatly, whereas one close to 0 will make the agent focus on immediate rewards and value the future less.

With this knowledge in hand we can assign a value to every $s \in \mathcal{S}$. This value models the cumulative discounted reward the agent can expect to receive after reaching state s and is crucial in the selection of actions described earlier. The value depends on all future actions the agent will take and therefore on the policy. The value of state s under policy π is defined as the expected discounted cumulative reward and is given by:

$$V^{\pi}(s) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s] \tag{3}$$

In an optimal policy, the agent will select the best action for every state, thereby optimizing its long term cumulative reward. All optimal policies share the same optimal value function:

$$V^*(s) = \max_{\pi} V^{\pi}(s), \quad \forall s \in \mathcal{S} \tag{4}$$

In most situations it is desired to have knowledge of the value of an action in a certain state, we call this the Q-value, with $Q(s, a)$ providing the value of taking a in s, it is defined as:

$$Q^{\pi}(s, a) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a] \tag{5}$$

Assuming the value of all successor states s' is known to the agent, (5) can be rewritten as the reward the agent receives plus the discounted value of s', weighted by the probability of ending in s', after taking action a in s:

$$Q^{\pi}(s, a) = \sum_{s'} T(s, a, s')[R(s, a) + \gamma V^{\pi}(s')] \tag{6}$$

This formula is a form of the Bellman equation named after Richard Bellman, who introduced it in 1957 [3]. With this function, we can iteratively update the value of all states, until it reaches a convergence criterion, resulting in an optimal state-value function $V^*(s)$, from which we can derive an optimal state action-value function $Q^*(s, a)$. Knowing the value of all states, the agent can select the action with the highest utility in every state, which will lead to an optimal policy. Value iteration is an algorithm that uses this concept.

C. Value Iteration

Value iteration is a (truncated) dynamic programming algorithm for computing optimal value functions and provides an exact solution for solving MDPs. The main idea behind this method is to compute the value of all $s \in \mathcal{S}$ iteratively, and to truncate the algorithm as soon as the difference in value of a state between two iterations: $\Delta = \max_{s \in \mathcal{S}} |V_i(s) - V_{i-1}(s)|$ drops below a threshold, where Δ is typically referred to as the Bellman residual. To approximate the value of a state, value iteration uses the Bellman equation in (6) as an update rule. The outline of the method is given in algorithm 1.

Algorithm 1 Value iteration

Initialize V(s) arbitrarily
repeat
  $\Delta \leftarrow 0$
  for all $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, v - V(S))$
  end for
until $\Delta < \theta$
return a deterministic policy π, such that:
$\pi(s) = \arg\max_{a \in \mathcal{A}} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')]$

Formally, the algorithm would need an infinite number of sweeps through the state space to converge to an optimal value function, but the optimal value can be approximated

by aborting the algorithm if Δ is sufficiently small. A major drawback is that each iteration requires updating the value of every $s \in \mathcal{S}$, resulting in a computational complexity of $\mathcal{O}(|\mathcal{A}||\mathcal{S}|^2)$. This is time consuming for problems with a large state space.

Once the algorithm is finished, the agent can use the value of state-action pairs, to select the action with the best expected outcome:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s, a) \tag{7}$$

Reinforcement learning is commonly associated with MDPs. We can however, ignore the a-priori model of the environment and focus only on the state-action pairs. Q-learning is such a technique, working without beforehand defined models and can be used outside of the context of MDPs.

D. Q-Learning

The introduction of Q-learning by Watkins in 1989 [25], [26], signified a great leap forward in the progress of the field of reinforcement learning. It is different from value iteration in the sense that it does not require an a-priori model of the environment and can therefore also be used outside the context of MDPs. Q-learning is known as an *off policy* temporal difference (TD) method, meaning that one policy can be followed while updating another, as long as the two policies have state-action pairs in common.

The original algorithm works with a look-up table, storing the value of all state-action pairs encountered and updates these at it goes along. After each action taken, the agent directly evaluates the value of the action and uses this to update the current Q-value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta(s_t, a_t) \tag{8}$$

where α denotes the learning rate, to be decreased as the algorithm progresses, and $\delta(s_t, a_t)$ the TD-error, which is computed according to:

$$\delta(s_t, a_t) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{9}$$

It is known that for finite state and action spaces, Q-learning converges to the optimal $Q^*(s, a)$ as long as every state action-pair is visited infinitely often [26].

In section IIA, we briefly mentioned policies and discussed the deterministic variant, where there is a direct mapping between the state of the environment and the action to take. The agent always selects the action with the highest expected value. It can be said the agent acts *greedy* and always *exploits* the knowledge it has of the environment. Policies can however, be dichotomized into deterministic and stochastic cases. In some reinforcement learning tasks, we want the agent to *explore* the environment, making it register state-action pairs that would otherwise not be encountered. For instance, Q-learning needs to visit all state-action pairs repeatedly to get reliable estimates of their values and is therefore in need of an exploration method. A stochastic policy $\pi(s, a)$, describes the probability of taking action a in state s.

A simple way to provoke explorative behavior is the ϵ-*greedy* selection rule. Here, the agent occasionally selects a random action with probability ϵ, but will mostly behave in a greedy fashion. A drawback of the method is that, when not taking the optimal action, it selects remaining actions with equal probability. One might want to consider ranking actions, according to their expected value. This is modelled in the *softmax* action selection rule [24], also known as Boltzmann exploration. Softmax uses a Gibbs or Boltzmann distribution for acquiring the probability of an action.

$$\pi(s, a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{Q(s,a')}/\tau} \tag{10}$$

Where τ denotes the temperature parameter. A high temperature will cause all actions to be selected with nearly equal probability. Low temperatures will result in a greater difference in selection probability, between actions that differ in their value, and will result in a policy resembling ϵ-*greedy* if τ is set close to 0.

Consider again the robot trying to navigate itself from a start to a goal position. In all real world situations, there is noise in its sensor readings as well; the world is partially observable to the agent. On top of this, some sensor readings might seem similar, due to a similar looking environment or due to the distortion caused by the noise in its sensors. This phenomenon is known as *perceptual aliasing*. The uncertainty in observations can be incorporated into the MDP model. The acquired result is called a Partially Observable Markov Decision Process, or POMDP.

III. Partially Observable Markov Decision Process

A POMDP is a generalization of an MDP and models not only the stochasticity in transitions, but also in observations, rendering the state of the agent partially observable. The POMDP framework consists of the same set of states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, transition function $T(s, a, s')$ and reward function $R(s, a)$. On top of this, a POMDP defines a set of observations $z \in \mathcal{Z}$ and an observation function $O(s, a, z)$, providing the probability of observing z in state s, after executing action a (to avoid confusion, we choose $\mathcal{Z}$ instead of $\mathcal{O}$, commonly used to indicate an algorithms computational complexity). Similar to the MDP model all sets $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{Z}$ are assumed to be discrete, although work in continuous spaces has been done (e.g., [15]). In an MDP the agent acts according to what seems to be the best possible action for a given state, but since the agent is no longer certain of its location it has to estimate its position based on the input it receives and its

actions taken. A common approach to do this is by generating a *belief state* [2].

A. Belief states

A belief state $\vec{b}_t$ is a probability distribution over $\mathcal{S}$, to model the belief of the agent at time t. The set of all possible belief states is referred to as the belief space $\mathcal{B}$. The belief of state s at time t is denoted to as $b_t(s)$. Every time the agent takes an action, its belief state is updated. Given $O(s', a, z)$, the probability of observing z in successor state s' after action a and the transition probability $T(s, a, s')$, Bayes' theorem can be applied to update the belief of the agent:

$$b_a^z(s') = \eta O(s', a, z) \sum_{s \in \mathcal{S}} T(s, a, s') b(s) \tag{11}$$

Where η, is a normalizing constant. The belief state effectively sums up all of the agent's past actions and observations and is therefore a Markovian signal and a sufficient statistic to base its actions on.

Since the agent is no longer certain of its position, the expected reward for a belief state has to be weighted by the belief in all individual states:

$$R(\vec{b}, a) = \sum_{s \in \mathcal{S}} b(s) R(s, a) \tag{12}$$

Rewriting (3), the expected cumulative reward for a belief state $\vec{b}$ is defined as:

$$V(\vec{b}) = E[\sum_{k=0}^{\infty} \gamma^k R(\vec{b}, a)] \tag{13}$$

And the initial value function at $t = 0$ is then given by:

$$V_0(\vec{b}) = \max_a \sum_{s \in \mathcal{S}} R(s, a) b(s) \tag{14}$$

The value of a belief state under a policy π is computed according to:

$$V^\pi(\vec{b}) = \sum_{s \in \mathcal{S}} b(s) V^\pi(s) \tag{15}$$

The key observation here is that this knowledge is sufficient to transform the POMDP to a continuous state MDP, where belief space $\mathcal{B}$ represents the state space $\mathcal{S}$. In section IIIC we will further elaborate on this concept and show how value iteration can be applied in POMDPs.

Numerous algorithms have been developed for solving POMDPs [7], [11], [12], [13], most of these using some form of value iteration. Qmdp is one of these, applying value iteration in its most rudimentary form.

B. Qmdp

An easy method for solving POMDPs is to make use of the Q-values of the underlying MDP, thereby ignoring the observation model [11]. By treating the belief space as if it were the state space in an MDP, the value of taking action a in belief state $\vec{b}$ is given by

$$Q(\vec{b}, a) = \sum_{s \in \mathcal{S}} b(s) Q_{MDP}(s, a) \tag{16}$$

Where Q_{MDP} denotes the Q-value of the underlying MDP. With these values in hand, (7) can be rewritten to select the action with the highest expected value.

$$\pi(\vec{b}) = \arg\max_a [\sum_{s \in \mathcal{S}} b(s) Q_{MDP}(s, a)] \tag{17}$$

The Qmdp algorithm is easily implementable and can be very fast in problems with a large number of states. A disadvantage however, is that an agent following this policy does not take explorative actions. This can lead to the agent going back and forth between belief states, without achieving its goal, rendering the algorithm inexpedient for problems where repeated exploration of the environment is necesarry. A possible solution to battle this problem is to use the softmax action selection as discussed in section IID. Rewriting (10), the probability of taking action a in belief state $\vec{b}$ is computed according to:

$$\pi(\vec{b}, a) = \frac{e^{Q(\vec{b}, a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{Q(\vec{b}, a')/\tau}} \tag{18}$$

Although Qmdp can be very effective in large state spaces, its simplicity generally lacks the subtlety to efficiently solve complex problems. For a more exact solution we have to consider the observation model and adjust the value iteration algorithm to suit POMDPs. A short version of the methods involved will be provided in the next section.

C. Value iteration in POMDPs

Value iteration can also be applied to approximate solutions for POMDPs. Here, we will present a brief outline of the methods involved, as an introduction to the Perseus algorithm. For detailed descriptions we refer to Sondik [21] and Puterman [16].

Recall from section IIIA, that when acquiring the value function of a POMDP under a certain policy, the value of every state needs to be weighted by the agent's belief in the given state. For notational convenience (15) can be written as a dot product:

$$V^\pi(\vec{b}) = \vec{b} \cdot \alpha^\pi \tag{19}$$

Where $\alpha^\pi = \{V^\pi(s_1), V^\pi(s_2), \ldots, V^\pi(s_n)\}$. In section IIB, a policy was described as a function specifying which action to take in a given state. Working towards an optimal policy, the agent needs to select the best action at every time step t:

$$V_t(\vec{b}) = \max_{\alpha \in \Gamma_t} \vec{b} \cdot \alpha^\pi \tag{20}$$

with $\Gamma_t = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$. The state of the agent is a continuous functions of all individual beliefs in a state. Assigning the belief of a state to every axis, plotting the

belief state will result in an $|\mathcal{S}| - 1$-dimensional hyperspace (probabilities sum to 1, thus the belief in $|\mathcal{S}| - 1$ states is sufficient to determine the entire belief state). All belief points are contained in a belief simplex Δ. With every region of the belief space, an optimal action is associated, this is represented by one of the α-vectors. The value of every action is a linear function of the belief of a state.

Again, applying the concept of weighting probabilities, we can combine the functions defined so far into a general formula for an optimal value function:

$$V^*(\vec{b}) = \max_{a \in \mathcal{A}} [\sum_{s \in \mathcal{S}} R(s,a) b(s) + \gamma \sum_{z \in \mathcal{Z}} O(\vec{b}, a, z) V^*(b_a^z))] \tag{21}$$

where b_a^z is given by the belief function, defined in (10). For short, we can write: $V^* = HV^*$, with H defined as the Bellman backup operator.

Since for every region of the belief space, there is an α-vector optimizing the value, the optimal value function will be made out of a finite set of hyperplanes, building up the surface of the belief simplex. Sondik showed this function is *piecewise, linear and convex* for finite horizon POMDPs and is approximately PWLC for POMDPs with an infinite horizon. The piecewise property stems from the max operator and the linearity can be explained by the fact that the value of every action is a linear function of the belief state. The convexity of the value function has an intuitive explanation. In the corners of the function the entropy, the degree of uncertainty, is low, yielding a firm basis for decision making, whereas in the center of the simplex, where the entropy is high, there is much uncertainty in the state of the environment, meaning the outcome of a decision will be unclear.

Analogous to value iteration in MDPs, the value function is updated iteratively. In every iteration n, an $|\mathcal{A}|$ number of α-vectors will be added, one associated with each action and a new value function, made up of the surface of all optimizing vectors will be computed. Every stage can be seen as a backup of the previous value function. At every backup stage, the vector parameterizing the surface of the value function can be computed according to:

$$\text{backup}(\vec{b}) = \alpha_{n+1}^b = \arg \max_{\alpha \in \Gamma_{n+1}} \vec{b} \cdot \alpha_{n+1}^k \tag{22}$$

Where $\Gamma_{n+1} = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ and $k = |HV_n|$, i.e., the number of vectors in the current value function [22]. This notation will be useful for understanding the Perseus algorithm. Value iteration is computationally expensive in POMDPs, because at each iteration the value of every point in the entire belief space is updated. Recently developed methods, known as point-based algorithms, have started working with restricting value iteration to a subset of the belief space [8], [12], [14], [28]. Perseus is one of such algorithms.

D. Perseus

Perseus is an approximate point-based value iteration algorithm for solving POMDPs and was introduced by Spaan and Vlassis in 2005 [22], [23]. The algorithm starts by performing a random walk through the environment, thereby sampling a set $B = \{b_1, b_2, \ldots, b_n\}$ of reachable belief points. These points remain the same throughout the algorithm. This holds an advantage over other algorithms that work with the complete belief space in the sense that it only computes values for belief points that can actually be encountered by the agent.

The initial value function is set as a single vector, with all components set to $\frac{1}{1-\gamma} \min_{s,a} r(s,a)$, the minimal cumulative reward obtainable in the POMDP, guaranteed to be below V^*. Perseus introduces a new backup operator $\tilde{H}_{PERSEUS}$, and in every backup stage, tries to improve the value of all belief points, or at least makes sure that they do not decrease:

$$V_n(b) \leq V_{n+1}(b), \forall b \in B \tag{23}$$

It keeps track of the set of non-improved points $\tilde{B}$, and as long as $\tilde{B}$ is not empty, samples uniformly at random a belief point b and computes $\alpha = \text{backup}(b)$. If the vector improves the value of b, it is added to the value function of V_{n+1}, otherwise a copy of V_n will be inserted. In an ideal situation, an increase in value of a belief point $b \in B$ will increase the value of many other points in B. Given the shape of the value function, such a method can be very effective in approximating solutions. The backup stage is given in algorithm 2.

Algorithm 2 The backup stage $V_{n+1} = \tilde{H}_{PERSEUS}$

```
V_{n+1} ← ∅
B̃ ← B
repeat
    Sample a belief point b uniformly at random from B̃ and
    compute α = backup(b)
    if b · α ≥ V_n(b) then
        add α to V_{n+1}
    else
        add α' = arg max_{α∈Γ_n} b · α to V_{n+1}
    end if
    B̃ = {∀b ∈ B : V_{n+1}(b) < V_n(b)}
until B̃ = ∅
return V_{n+1}
```

This stage is performed iteratively, until some stopping criterion is met. This could be, analogous to regular value iteration, terminating the algorithm as soon as the maximum difference between two backup stages $\max_{b \in B}(V_{n+1}(b) - V_n(b))$, drops below a threshold.

The POMDP algorithms discussed so far, all make use of value iteration. We will now discuss some previous work on Q-learning in combination with reinforcement learning

and neural networks, followed by the introduction of a new approach, combining several of these techniques.

IV. Q-LEARNING IN POMDPS

As described in section IIE, the Q-learning algorithm updates each state-action pair after executing an action. However, because the belief space is used as a state space, the number of possible states encountered is infinite. Therefore, to work with a lookup table for each belief state-action pair becomes impossible and there is need for a prediction method, which generalizes between these pairs and associated Q-values. Neural Networks (NN) provide such a method and are known to be a powerful formalism in function approximation, gaining success in a wide variety of applications.

The use of a NN within the context of reinforcement learning has been demonstrated to be effective before [4], [10], [18], [27]. Schäfer and Udluft [20], for instance showed how an NN in combination with RL can be applied to solve the pole balancing problem. Riedmiller [17] argued that a weight change induced by a change in value of a certain part of the state space, can cause the value of other regions to change as well, and therefore undo previous training. He proposed a method of storing all previously encountered state-action pairs in memory and use these to perform offline training of the network. Unfortunately this is not convenient, when simulating Q-learning in POMDPs, and will result in longer learning times. We can however, use this insight in the choice of our neural network configuration for estimating the Q-values. The advantage of using neural networks in approximating Q-values is that only the weights of the network have to be stored in memory.

To use neural nets to predict Q-values, there are two possible approaches. Either one network is used, with $|\mathcal{A}|$ output units, or a single net is assigned to every $a \in \mathcal{A}$, as used by Lin [10]. The advantage of the latter approach is that, when trying to obtain the Q-value of a given action, one can easily address the responsible network. Also this will reduce the untraining of weights, caused by changes in the state space, as mentioned by Riedmiller. Therefore this method is used.

To each $a \in \mathcal{A}$, we assign an NN: Q_a^{NN}. The belief state at time t, $\vec{b}_t$, is fed to the network as an $|\mathcal{S}|$-dimensional input vector and a single output unit was used to predict the Q-value of a given belief state. The number of hidden units was left as a parameter. As opposed to normal neural network training, the networks in this case do not learn from correct examples, but from approximations. So as a target, the Q-value at the next time step was fed to the network and used to compute the error. To obtain this target, we can rewrite (9) and get:

$$Q(\vec{b}_t, a_t) = r_{t+1} + \gamma \max_{a \in \mathcal{A}} Q_a^{NN}(\vec{b}_{t+1}) \tag{24}$$

Instead of updating the Q-value as in (8), the backpropagation learning algorithm is used to update the weights of the networks, with learning rate α.

The method discussed so far, can be also used outside the context of a POMDP, because the agent does not need a model of the environment. Since we are working within the context of a POMDP and a model is known, we can exploit this knowledge and use it to provide the networks with more information.

A. RENQ: Region Enhanced Neural Q-learning

A technique for extracting information from images in object recognition is making use of a *spatial pyramid* [9], [1]. In this method, the image is divided into subsections and spatial features, e.g., a histogram, are computed for all sections. This approach is known to improve recognition performance greatly. RENQ uses this method in a novel way and applies the spatial pyramids to the state space of the POMDP. The approach works with several levels. At each level, the state space is divided into k regions of equal size. At level 1 the used region is equal to the original state space, where each state is a singleton region, thus $k = 1$. Level 2 decomposes $\mathcal{S}$ into 2×2 quadrants, making $k = 4$. Level 3 subsequently subdivides $\mathcal{S}$ in 3×3 regions with $k = 9$, level 4 in 4×4, etc.

For every k, $\varsigma_k \subseteq \mathcal{S}$ the average belief value $\bar{b}$ is computed:

$$\bar{b}(\varsigma_k) = \frac{1}{|\varsigma_k|} \sum_{s \in \varsigma_k} b(s) \tag{25}$$

The set of average beliefs at level L is $\bar{B}_L = \{\bar{b}_1, \bar{b}_2, \ldots, \bar{b}_k\}$, with $k = L^2$. The enhanced belief vector $\vec{b^+}$ is the union of all belief sets at every level:

$$\vec{b^+} = \bigcup_{L=1}^{i} \bar{B}_L \tag{26}$$

Subsequently, $\vec{b^+}$ is fed as an inputvector to the networks and Q-values are estimated. The computation of this additional information might seem redundant but, as we will show, this can actually be very effective.

Example 2. Consider again the 4 × 4 grid maze, used in example 1, with $|\mathcal{S}|$ = 12. The grid can be divided into 4 square regions of 2×2 (level 2). For each region we will compute the average belief according to (25). A depiction of the general idea is provided by figure 3.

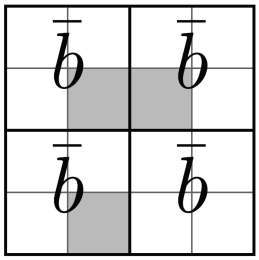

Fig. 3. Level 2 Spatial Pyramid applied to the belief state of a POMDP

The belief function represents a belief of the agent, being in a certain state. The knowledge we add will also provide the agent with an estimate of its approximate position. If the goal of the agent is to get to a goal state somewhere in the upper right corner of the state space and the agent has a fair degree of faith its position is somewhere within the boundaries of the lower left corner, it is very likely it will steer itself either north or east.

In this example, the state space was divided into 4 regions. For larger problems, one might consider breaking up the problem into more and/or larger sections, i.e., higher levels, and add the average belief of these regions to the input vectors of the networks as well. This is left as a parameter. Naturally, variations on this scheme can be developed, for instance for problems that are not captured in a square state space POMDP, the dimensions of spatial pyramid can be modified to suit the particular problem. In the following section we will provide details about the benchmark problems used to test the algorithms, the optimal parameter settings found and show the results comparing the RENQ algorithm to Qmdp and Perseus.

V. Experiments

To test the algorithms, we use 3 square maze navigation tasks of 6×6, 12×12 and 24×24, with $|\mathcal{S}| = 12$, $|\mathcal{S}| = 73$ and $|\mathcal{S}| = 344$, respectively. Every maze has one static goal position and every other unoccupied state can be the initial state. The objective of the agent is to reach the goal position as soon as possible and with every action can only reach adjacent states. In every maze, we use the set of actions: $\mathcal{A} = \{$go left, go up, go right, go down$\}$, with a 20% chance that the selected action is changed by a random action from $\mathcal{A}$. In every of the 4 directions the agent can either observe a wall or an empty field, making the cardinality of $|\mathcal{Z}| = 2^4 = 16$. We added 10% noise in the observation in each separate direction, meaning that an observation is correct with probability $0.9^4 = 66\%$. The agent receives a reward of 100 for reaching the goal position and is penalized by -0.1 for every other action. In all mazes the only opportunity for the agent to get a reward is reaching the goal state. We therefore chose to measure the number of steps to the goal position, since this does not depend on the size of the rewards.

We run the algorithms on an Intel Dual Core 2.33GHz, with 3.4 GB RAM. For Perseus, the Matlab implementation available on Spaan's website is used and the algorithm is run on Matlab 2009b for Linux. The RENQ and Qmdp algorithms use self written C++ implementations.

A. Small Maze

The maze is depicted in Figure 1, with G as the goal position. The starting state can be any other unoccupied state. The entire maze is surrounded by a 1 block wall (note that we did not draw the walls around the maze). For all algorithms, the discount factor is set to $\gamma = 0.7$.

RENQ. A simulation lasts for 100,000 steps. During an experiment, we perform 50 simulations. A run is finished if the goal is hit or if the agent performed 1000 actions during the run. The learning rate α of the neural network is set to 0.015. The neural network used 20 sigmoidal hidden units for each separate action network. We used Boltzmann exploration with τ = 1. We also used the same parameters to test the performance of BQNN and Q-learning with neural networks on an MDP.

Perseus. Following Spaan, we ran Perseus 10 times, each with a different random seed. For the small maze, we sampled a set $|B| = 200$ belief points. With every simulation, the algorithm performs 1000 episodes, starting from random positions. We let the algorithm run for 120 seconds, which proved to be enough for convergence. The average of the total 10.000 trajectories is computed along with the standard deviation.

Qmdp. We ran the algorithm 100 times, in each simulation letting the agent start at each different starting location. The average of all the $(12)(100) = 1200$ different episodes is computed, along with a success percentage, indicating how often the goal was found in an episode. As a stopping criterion for the value iteration part we use $\delta = 1E^{-6}$.
We also tested regular value iteration on an MDP, to compare with the Q-learning and neural networks method on an MDP. The results are shown in table I. For value iteration on the MDP (VI MDP) we did not compute standard deviations, since it always computes the same policies.

TABLE I
Results on the small maze.

Method	Final steps	Nr. Times Goal hit	% Success
RL+NN MDP	3.85 ± 0.04	25375 ± 533	100
BQNN	4.37 ± 0.09	22156 ± 967	100
RENQ LEVEL 2	4.37 ± 0.08	22172 ± 1001	100
RENQ LEVEL 3	4.36 ± 0.06	22307 ± 796	100
VI MDP	3.86		100
Qmdp	4.38 ± 0.68		100
Perseus	4.79 ± 0.03		100

Discussion. As can be seen in Table I, RENQ significantly outperforms Perseus at all levels, but performs the same as

Qmdp. Furthermore, it can be seen that RENQ at level 3 hits the goal more often during its learning process than BQNN, the method without region enhanced beliefs and therefore performs better. The results also show that solving this MDP with Value Iteration (VI MDP) results in a solution of 3.86 steps on average, whereas using Q-learning and a neural network as function approximator (RL+NN MDP) learns the same optimal policy. RENQ requires around 5 seconds of computational time for the 100,000 steps and is therefore faster than Perseus.

B. Middle-sized Maze

The 12x12 maze is depicted in figure 4. For all algorithms, the discount factor is set to $\gamma = 0.95$.

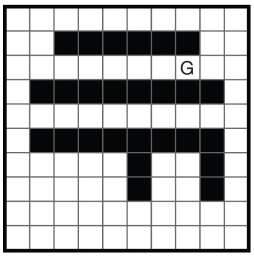

Fig. 4. The Middle-sized maze. G denotes the goal position.

RENQ. A simulation lasts for 200,000 steps. The learning rate α of the neural network is again set to 0.015. The neural network used 60 sigmoidal hidden units for each separate action network. We used Boltzmann exploration with τ = 1.

Perseus. Again we ran the algorithm 10 times, each with a different random seed and let the agent perform 1000 trajectories, each starting from a different random starting location. We sampled $|B| = 1000$ and let the value iteration stage run for 600 seconds. The average number of steps for the 10000 trajectories is computed along with a standard deviation.

Qmdp. We use 100 simulations, in each simulation consists of $|\mathcal{S}|$ episodes and the average of all the $100|\mathcal{S}| = 7300$ trajectories is computed. The results are shown in table II.

TABLE II
RESULTS FOR Q-LEARNING WITH NEURAL NETWORKS ON THE 12×12 MAZE.

Method	Final steps	Nr. Times Goal hit	% Success
RL+NN MDP	11.9 ± 0.3	14165 ± 436	100
BQNN	15.2 ± 0.4	10971 ± 208	100
RENQ LEVEL 2	15.1 ± 0.3	11526 ± 195	100
RENQ LEVEL 3	15.2 ± 0.3	11608 ± 200	100
RENQ LEVEL 4	15.2 ± 0.3	11606 ± 179	100
VI MDP	11.3		100
Qmdp	14.7 ± 0.6		100
Perseus	15.7 ± 0.1		100

Discussion. As can be seen in Table II, Qmdp performs the best for this maze, while RENQ significantly outperforms Perseus at all levels. At levels higher than 1, RENQ performs much better than when using only the state-based belief vector, since the goal is hit significantly more often, during training. We can also see that solving this MDP with Value Iteration (VI MDP) finds a solution of 11.3 steps on average and using Q-learning with a neural network as function approximator (RL+NN MDP) comes very close to this optimum. RENQ requires around 90 seconds of computational time for the 200,000 steps and is therefore a bit faster than Perseus.

C. Large Maze

For the large maze, depicted in Figure 5, we use a discount factor $\gamma = 0.99$.

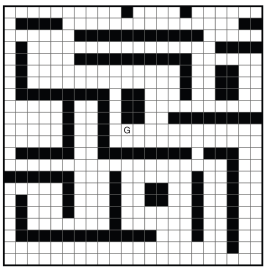

Fig. 5. The Large maze. G denotes the goal position.

RENQ. A simulation lasts 2,000,000 steps. The learning rate α of the neural network is set to 0.01. The neural network used 60 sigmoid hidden units for each separate action network. We used Boltzmann exploration with τ = 1.

Perseus. Continuing in the same fashion, we ran the algorithm 10 times, each with a different random seed. We sampled $|B| = 10000$ belief points and let the algorithm run for 2 hours. The average number of steps of 5 simulations is computed along with a standard deviation.

Qmdp. Again we use 100 simulations, in each simulation we let the algorithm start from all different non-goal positions. The average number of steps the total $100|\mathcal{S}| = 34400$ episodes is computed, along with a success percentage and a standard deviation. The results are shown in table III.

TABLE III
RESULTS FOR Q-LEARNING WITH NEURAL NETWORKS ON THE 24×24 MAZE.

Method	Final steps	Nr. Times Goal hit	% Success
RL+NN MDP	23.7 ± 2.4	51372 ± 5413	100
BQNN	397.8 ± 1438	27290 ± 7502	97
RENQ LEVEL 2	33.9 ± 0.5	37639 ± 1363	100
RENQ LEVEL 3	33.6 ± 0.5	38418 ± 1338	100
RENQ LEVEL 4	33.8 ± 0.4	38098 ± 1641	100
VI MDP	21.4		100
Qmdp	35.6 ± 1.0		99.3
Perseus	34.7 ± 0.3		100

Discussion. As can be seen in Table III, RENQ significantly outperforms Qmdp and Perseus with levels 2, 3, and 4. BQNN

fails to learn a good policy in 2 of the 50 simulations. RENQ at Levels 3 and 4 performs the best of all POMDP methods. We can also see that solving this MDP with Value Iteration (VI MDP) results in a solution of 21.4 steps on average and using Q-learning and a neural network as function approximator (RL+NN MDP) again comes very close to this optimum. RENQ requires around 1 hour of computational time for the 2,000,000 steps and is therefore a bit faster than Perseus for this problem. Note that for this largest problem, Qmdp may be a too simple algorithm and is outperformed by Perseus and RENQ. Unfortunately, we did not have time to run even larger problems to see whether the difference between RENQ and Qmdp becomes even larger, although we expect this to be the case.

VI. Conclusion

The partially observable Markov decision process (POMDP) framework provides a model for decision making under uncertainty, caused by for instance, noise in a robot's actuators and sensor readings. In this paper we have presented RENQ, a novel approach combining techniques from machine vision with Q-learning and neural networks to approximate an optimal solution for POMDPs. We have shown that RENQ outperforms Qmdp, a simple POMDP algorithm, and Perseus, a state-of-the-art algorithm when the maze problems become larger.

The benchmark problems all consist of maze navigation tasks, where state transitions are only defined for adjacent states. It would be interesting to see how RENQ can be used for problems where this is not the case. Ultimately, the goal is of course to work towards a method providing effective learning behavior in a real world situation. We would also like to study different hierarchical approaches to improve RENQ's learning speed, in future work.

ACKNOWLEDGEMENTS

As a final word I would like to express my gratitude towards Dr. Marco Wiering, for guiding me on this project, helping me with the programming parts and providing useful, constructive criticism on my writings and of course, all credit for the RENQ algorithm goes to Marco.

References

[1] A.Abdullah, R.C. Veltkamp, and M.A.Wiering. Spatial pyramids and two-layer stacking svm classifiers for image categorization: a comparative study. In *IJCNN'09: Proceedings of the 2009 international joint conference on Neural Networks*, pages 1130–1137. Institute of Electrical and Electronics Engineers Inc., The, 2009.

[2] K.J. Astrom. Optimal control of Markov decision processes with incomplete state estimation. *J. Math. Anal. Applic.*, 10:174–205, 1965.

[3] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] L.T. Dung, T. Komeda, and M. Takagi. Reinforcement learning for POMDPs using state classification. *Applied Artificial Intelligence*, 22(7&8):761–779, 2008.

[5] M. Givon and A. Grosfeld-Nir. Using partially observed markov processes to select optimal termination time of tv shows. *Omega, International Journal of Management science*, 36:477–485, 2006.

[6] J. Hoey, P. Poupart, A. von Bertoldi, T. Craig, C. Boutilier, and A. Mihailidis. Automated handwashing assistance for persons with dementia using video and a partially observable markov decision process. *Computer Vision and Image Understanding*, 2009.

[7] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[8] H. Kurniawati, D. Hsu, and W.S. Lee. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Proc. Robotics: Science and Systems*, 2008.

[9] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *In CVPR*, pages 2169–2178, 2006.

[10] L.J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning Journal*, 8(3/4), 1992. Special Issue on Reinforcement Learning.

[11] M.L. Littman, A.R. Cassandra, and L.P. Kaelbling. Learning policies for partially observable environments: scaling up. In *Proc. 12th International Conference on Machine Learning*, pages 362–370. Morgan Kaufmann, 1995.

[12] W.S. Lovejoy. A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[13] G.E. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, jan 1982.

[14] J. Pineau, G.J. Gordon, and S. Thrun. Anytime point-based approximations for large POMDPs. *Journal of Artificial Intelligence Research (JAIR)*, 27:335–380, 2006.

[15] J.M. Porta, N. Vlassis, M.T.J. Spaan, and P. Poupart. Point-based value iteration for continuous pomdps. *J. Mach. Learn. Res.*, 7:2329–2367, 2006.

[16] M.L. Puterman. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.

[17] M. Riedmiller. Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method. In João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors, *ECML*, volume 3720 of *Lecture Notes in Computer Science*, pages 317–328. Springer, 2005.

[18] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, October 04 1994.

[19] S.J. Russel and P. Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[20] A. M. Schäfer and S. Udluft. Solving partially observable reinforcement learning problems with recurrent neural networks, 2005.

[21] E. J. Sondik. *The Optimal Control of Partially Observable Markov Decision Processes*. PhD thesis, Stanford University, Stanford, California, 1971.

[22] M.T.J. Spaan. *Approximate planning under uncertainty in partially observable domains*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 2006.

[23] M.T.J. Spaan and N. Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.

[24] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[25] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England, 1989.

[26] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning Journal*, 8(3/4), May 1992. Special Issue on Reinforcement Learning.

[27] M.A. Wiering and H. van Hasselt. Ensemble algorithms in reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 38(4):930–936, 2008.

[28] W. Zhang and N.L. Zhang. Restricted value iteration: Theory and algorithms. *Journal of Artificial Intelligence Research (JAIR)*, 23:123–165, 2005.