Final Year Project Report

Full Unit - Interim Report

Resourceful Robots

Cougar Tasker

A report submitted in part fulfilment of the degree of

MSci (Hons) in Computer Science (Artificial Intelligence)

Supervisor: Dr. Anand Subramoney

Department of Computer Science
Royal Holloway, University of London

November 30, 2023

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

Student Name: Cougar Tasker

Date of Submission:

Signature:

Table of Contents

Abstract

To complete many objectives robotic agents need to adapt and learn to new environments, therefore providing supervised training is not always feasible. This report investigates Reinforcement Learning (RL) techniques as a solution for this class of problem. Reinforcement Learning is a machine-learning paradigm where the learning component receives external judgment like supervised learning. However, in reinforcement learning this judgment is provided after the agent's decision. This distinction makes RL agents suitable for our problem.

For this report, our agents will be challenged with a resource-gathering objective. Resource gathering is widely applicable to many real-world autonomous robots such as smart vacuum cleaners. for consistency, the agents will operate in a simulated environment, such as a grid world for tabular approaches or an environment from the gymnasium library to evaluate deep reinforcement algorithms.

needs more

Chapter 1: **Introduction**

1.1 The Challenge

1.2 Objectives

1.3 Motivation

1.4 Literature Review

Chapter 2: **Software Engineering**

2.1 Workflow

2.2 System Design

2.3 Functionality and Usage

Chapter 3: **Timeline**

Chapter 4: **Fundamental Concepts**

4.1 Markov Decision Processes

Markov Decision Processes (MDP) provide a mathematical formalisation of decision-making problems. Markov Decision Processes provide the foundation for reinforcement learning (RL). This is because MDPs distil the fundamental parts of decision-making, allowing RL techniques built upon MDPs to generalise to learning in the real world and across different domains such as finance and robotics.

As a formal mathematical framework, MDPs allow us to derive and prove statements about our RL methods built upon them. An important example of this is that we can prove that Q-learning (an RL technique explained in chapter 4.4) will converge to the true Q-values as long as each Action-State pair is visited infinitely often. [1]. Furthermore, MDPs allow us to reason about problems with uncertainty allowing RL agents to account for randomness in their environment.

The standardisation of decision-making problems as MDPs allows for a uniform definition of optimality with the value functions. MDPs give a basis for assessing the performance of RL algorithms, facilitating like-for-like comparisons for different RL approaches.

4.2 Markov Property

The Markov property is that the future state of a Markov system only depends on the current state of the system. In other words, if we have a system that follows the Markov property, then the history preceding the current configuration of the system will not influence the following state.

To put the Markov property formally S_t represents the state at some time t. S_t represents the outcome of some random variable. Then the Markov property would hold if and only if:

$$\Pr(S_{c+1} \mid S_c, S_{c-1}, \ldots, S_0) = \Pr(S_{c+1} \mid S_c) \tag{4.1}$$

This definition demonstrates how the Markov property can hold in non-deterministic, stochastic processes. It also shows that predictions that are only based on the current state are just as good as those that record the history in a Markov process. The sequence of events in this definition, S_t, is called a Markov Chain[2].

4.2.1 Extending Markov Chains

Markov Decision Processes extend Markov Chains in two important ways. Firstly MDPs introduce decision-making through actions. Each state in an MDP has a set of available actions in that state. In each state, an action is required to transition to the next state; this action with the current state can affect what the following state will be. Secondly, MDPs introduce a reward value. The reward is determined from the current state and action; it is produced simultaneously with the following state.

A formal definition of a Markov Decision Process is a tuple $(\mathcal{S}, \mathcal{A}_s, p)$ where:

- $\mathcal{S}$ defines the set of all states
- $\mathcal{A}_s$ defines the set of available actions in state s
- p defines the relationship between states, actions and rewards: $p(s', r \mid s, a) \doteq \Pr(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$[3]
  - $s, s' \in \mathcal{S}$, $a \in \mathcal{A}_s$ and $r \in \mathbb{R}$
  - $p : \mathcal{S} \times \mathbb{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$

The function p is an integral part of this definition; it fully describes how the system will evolve. We call this function the dynamics of the MDP. What this definition does not describe is how actions are chosen. This decision-making is done by an entity called an agent. For our purposes, the agent will have complete visibility as to the current state of the MDP. However, like most real-world situations, our agent will not have any a priori knowledge of the dynamics.

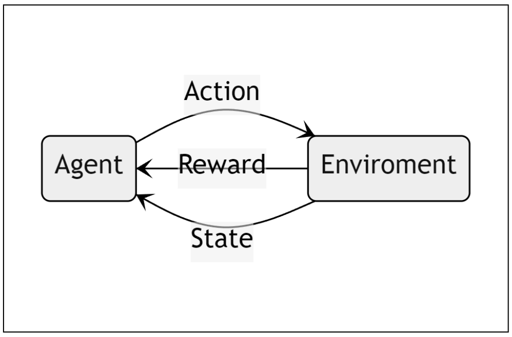

Figure 4.1: The agent-environment interface

The agent comprises the entire decision-making entity in an MDP; anything unknown or not wholly under the agent's control is called the agent's environment. In the context of reinforcement learning, the environment is essentially the dynamics of the MDP. Figure 4.1 demonstrates how the agent and environments affect each other in an MDP.

For learning agents, we wish to improve the agent's behaviour over time. For this purpose, we introduce a policy π. This policy defines the action chosen by an agent under a particular state. The policy can be represented with a lookup table like in Q-learning4.4 or a more complex process such as deep Q-learning. A policy like this is not hard-coded, allowing the agent to update the policy based on the information the agent learns from the environment.

4.3 Policy and Value functions

After each action, a reward is received. It follows that the goal of an agent should be to choose the actions to maximise these reward signals received. Following the Markov principle and the definition of an MDP, this reward only depends on the current state and the action chosen. Consequently, being in some states and performing some actions are more valuable to the agent than other states and actions. We can define value functions:

- $v(s)$ function determines the value of being in a given state

- $q(s, a)$ function determines the value of being in a given state and performing a specific action

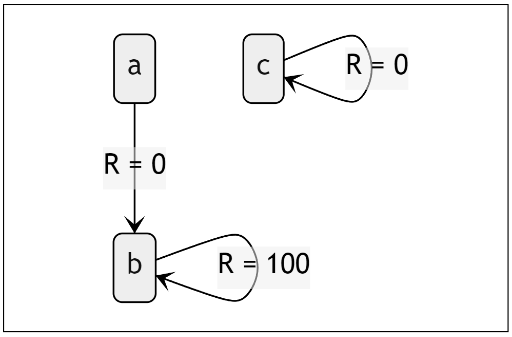

Figure 4.2: An example of transitive value of states $v(b) > v(a) > v(c)$

Intuitively, the value of being in a state is more than only its immediate reward that might be found from performing actions in that state. It is also related to the potential future reward that might be achieved in the reachable subsequent states. This can be demonstrated with two states a, and b, where there is a large reward at b and the only way to reach b is through a then being at a is also valuable regardless of the reward available at a. However, a can be considered less valuable than b, because it always requires more steps to achieve the reward from a than at b. To account for preferring more immediate rewards, the value function should also discount future value with a parameter γ.

These value functions go hand in hand with an agent's policy; a good policy maximises being in valuable states and performing valuable actions. On the other side of the coin, the value is determined by the subsequent states and rewards, which are in part determined by the actions the policy selects. Basing the policy on the value function gives the value function's definition impact over that agent's decision-making, in particular, the discount rate (γ). With high discount rates $\gamma \approx 1$ the agent can be far-sighted and ignore short-term high-reward actions available to it and take longer to learn. With low discount rates $\gamma \approx 0$, the agent can be short-sighted, ignoring the potential long-term benefits of certain actions.

While the policy is informally described at the end of chapter 4.2.1, a formal definition of a policy (π) is the probability distribution of an agent picking a given action in a given state:

$$\pi(a \mid s) \doteq \Pr(A_t = a | S_t = s) \tag{4.2}$$

Where $s \in \mathcal{S}$ and $a \in \mathcal{A}_s$. This definition shows how the policy can be stochastic. A stochastic policy can be beneficial in many ways, such as breaking ties where multiple actions are equally good and choosing between when to explore more or seek rewards.

4.3.1 optimal policy/value function via the Bellman equation

With a known policy and dynamics, the future state can be wholly determined, allowing for a complete mathematical definition of the value functions under a given policy that describes our above intuitions. for the state-value function (v) and action-value function (q) under a policy π we have the formulas:

$$q_\pi(s,a) \doteq \sum_{s',r} p(s',r \mid s,a)[r + \gamma v_\pi(s')] \tag{4.3}$$

$$v_\pi(s) \doteq \sum_{a} \pi(a \mid s) q_\pi(s,a) \tag{4.4}$$

These value functions are defined recursively in terms of each other; these definitions can be unrolled to only be in terms of themselves. The unrolled form of the state-value function is known as the Bellman equation. These equations are named Richard Bellman, who, in the process of developing the field of dynamic programming, created them[4].

These functions demonstrate the intertwined relationships between the policy chosen and the value of that state if there is a particularly valuable action a^* such that $q_\pi(s,a^*)$ is far better than for all other actions. A policy $\pi(a^* \mid s) = 0$ would hamper the potential value of s. Therefore, the value function can be used to compare how well different policies perform. If for policy π_a there does not exist another policy π_b such that π_b has a better value $v_{\pi_b}(s) > v_{\pi_a}(s)$ for all states $s \in \mathcal{S}$ then we can consider this policy π_a an optimal policy. There may be many optimal policies; however, we often do not need to distinguish them, so we often denote any optimal policy with π_*. This is because all optimal policies share the same state-value, and by definition action-value, function, we denote this v_* and q_*. The optimal value function v_* is known as Bellman optimality equation. These optimal equations can be written formally as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \tag{4.5}$$

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{4.6}$$

4.3.2 Finding optimal policies by iteration

Although optimal policies exist, finding them is another matter. The policy search space is potentially infinite so an intelligent method is required. An optimal policy can be extracted from an optimal value function and the dynamics of the MDP; the optimal policy would only select actions that result in the highest value. Finding the optimal value function with the optimal policy is straightforward, but this is a catch-22. The optimal value function must be self-consistent with the Bellman equations. One approach to solving these equations is iteration, for each step moving slightly closer to the optimal solution from an initial guess.

Policy iteration

In policy iteration, we improve a policy over time until it is optimal. updating a policy like this is only possible because of the policy improvement theorem. This theorem considers if we have a policy π_{old}. We are at some state s, π_{old} will pick the action a under this state $\pi_{\text{old}}(a \mid s) = 1$ what happens if we consider some other action a' but then continue to follow the original policy. Because we continue to follow the original policy, we can use the existing value function $v_{\pi_{\text{old}}}(S_{t+1})$ for the subsequent states. We call this slightly adjusted policy π_{new}. by applying the bellman equations and the existing value function then we can recalculate the value at s of π_{new} if this new value is better than the original policy then we know that π_{new} must be as good if not better for all states $s \in \mathcal{S}$ than π_{old} thus π_{new} would be a better policy.

$$\sum_a \pi_{\text{new}}(a \mid s) q_{\pi_{\text{old}}}(s,a) \geq v_{\pi_{\text{old}}}(s) \Rightarrow \pi_{\text{new}} \geq \pi_{\text{old}} \tag{4.7}$$

For some policy π you can apply this policy improvement theorem for every state and action in the MDP. This approach of comparing all actions over all states is called policy improvement. This policy improvement step can be applied iteratively until the policy stops improving. If the policy does not improve over this policy improvement step, then all of the actions are optimal, and this policy is optimal

Although this policy improvement sounds computationally expensive, each state can be considered simultaneously and with a shared base policy; in each iteration, the state-value function is the same; caching this removes redundant calculations. Calculating the value function is improved by using an iterative approach and utilising the previous value function as a launching point.

Value iteration

Policy iteration is a practical approach, for a finite MDP is guaranteed to finish in finite time. In practice, policy improvement does better and only takes a few iterations. However, in each iteration, multiple full sweeps of the state space are required. The idea of value iteration is to improve the policy within the value iteration step. This value iteration approach only requires one iteration.

The Bellman equation4.4 can be used as an update rule to compute the value function iteratively. A table of values is maintained for each state, initialised randomly. The value of each state can be updated based on the immediate reward and the current estimates of the subsequent states; this is guaranteed to reduce the error at that state because the γ discount rate discounts the error at the subsequent state. This process is called bootstrapping; the smaller γ, the quicker the error rate will decrease, and the faster the process will converge. When the inconsistency at each state is suitable, the process will stop.

This standard approach uses the traditional value Bellman equation4.4 to find the value for a given policy. In value iteration, the policy is one that exclusively picks the action that has the maximum value. This policy is optimal for the optimal value function; when the value iteration converges, it must be optimal because of these conditions. This augmented update rule can be defined as:

$$q_{\max}(s,a) \doteq \sum_{s',r} p(s',r \mid s,a)[r + \gamma v_{\max}(s')] \tag{from 4.3}$$

$$v_{\max}(s) \leftarrow \max_a q_{\max}(s,a) \tag{4.8}$$

$$\therefore v_{\max}(s) \leftarrow \max_a \sum_{s',r} p(s',r \mid s,a)[r + \gamma v_{\max}(s')] \tag{4.9}$$

As this new update rule involves no explicit policy, a final step is required in value iteration to extract an explicit policy. In this step, the action that leads to the best value for each step, according to the q_* function.q_*, can be derived from the v_* with the dynamics p function.

4.4 Q-learning

Q-learning is like value and policy iteration; all search for an optimal value function. However, Q-learning operates under extra constraints. The value and policy iteration extensively use the MDP's dynamics (p). In most real-world problems, the dynamics are unknown or too complex to be represented accurately. Iteration approaches can be adapted using samples to work without p. Samples are captured from the environment when the agent performs actions, which can be chosen randomly or by following another policy. Monti-Carlo is another technique that uses samples to emulate the value functions more directly, but these approaches are inherently offline. While offline methods have many advantages, their shortcomings, such as the inability to adapt to changing environments, make them unsuitable for many applications.

4.4.1 Temporal difference

Temporal difference (TD) algorithms are another class of RL algorithms. TD is an online process that improves the policy as new data becomes available. The goal of TD is to minimise the δ parameter that represents the difference (error) between the observed and predicted rewards. This difference δ is used to update the model like the iterative approaches we bootstrap our model over time. The magnitude of each update is controlled with the learning rate parameter, α. α helps avoid overfitting to samples since observations made in the real world may be noisy and change over time. The learning rate and discount rate can affect the convergence rate for TD; however, these are distinct variables and have different purposes. When α is low, the process will take longer to converge; however, if α is too large, the process may diverge

Reinforcement learning algorithms typically can learn in two fashions: on-policy and off-policy. On-policy algorithms learn while the agent uses the policy being improved; an example is SARSA. Off-policy algorithms typically learn the value functions while the agent follows a different "behaviour" policy. While on-policy techniques can start exploiting their knowledge for reward quickly, they can get stuck in local minima. Off-policy techniques can provide more control over the exploration and exploitation. Q-learning is an off-policy TD reinforcement learning algorithm implemented in this project.

4.4.2 Definition

As the name suggests, Q-learning learns the optimal action-value function q to find the optimal policy. TD techniques must learn the q function directly. The v function requires knowledge of the dynamics to derive the optimal policy; however, with the q alone, the optimal policy can be determined. For this purpose, Q-learning needs to maintain a table entry for each action in each state so these entries can be updated after each observed action. We will represent this estimate of q with Q. There are five parts to each transition:

- S_{t-1} the previous state before the transition
- A_{t-1} the action that was performed
- R_t the reward received
- S_t the new and now current state

Q-learning uses these observations to update its estimates with this formula:

$$Q(S_{t-1}, A_{t-1}) \leftarrow Q(S_{t-1}, A_{t-1}) + \alpha[R_t + \gamma \max_a(Q(S_t, a)) - Q(S_{t-1}, A_{t-1})] \tag{4.10}$$

This formula can be thought of as interpolating the old estimate at the old state $Q(S_{t-1}, A_{t-1})$ with this new observed Q-value $R_t + \gamma \max_a(Q(S_t, a))$. When $\alpha = 1$, this formula replaces the existing value with the new observed Q-value. When $\alpha = 0$, the observation is ignored like it never happened. The formula $R_t + \gamma \max_a(Q(S_t, a))$ calculates the new observed Q-value based upon the same principle as value iteration; the implicit policy is to pick the best possible action.

4.4.3 Implementation conditions

Q-learning is guaranteed to eventually converge the Q estimates to the optimal q values q_* provided the behaviour function visits all state-action pairs infinitely often. In practice, these q-values do not need to be perfect to derive an optimal policy. However, it can still take many visits to converge enough. Many observations may be necessary to build up a picture of the probability distribution and isolate noise. However, another reason for repeated observations is that the behaviour policy moves the agent forward in time, but the Q-learning table updates the last state. This conflicts as information propagates in the opposite direction of the updates that spread it. For example, suppose a sequence of n states-actions have no reward but lead to some large reward at the end. as the behaviour completes these actions. In that case, only the last action will be updated to reflect the potential value, and every time the sequence is repeated, some of the value will propagate back one step. Many Q-learning implementations like ours will replay recent observations in reverse order to improve this performance. This is called the action-replay process (ARP). Replaying observations can be particularly effective when getting new observations is costly or slow, allowing for quicker convergence.

In 4.10, we can see how α controls the influence of each observation. But how do we tune this hyper-parameter? One option is to treat it like other hyperparameters where possible and use previous experimentation to find a practical value. However, one of the main reasons to use RL is that it does not require prior knowledge, so this is not always suitable. A fixed learning rate may not also be suitable. Some observations may be more important than others; It has been observed that a decaying learning rate has been more effective. It is believed that a decaying learning rate allows for learning algorithms to avoid local minima at the beginning with the large learning rate and then settle on a global minima as the learning rate decays[5]. The paper “Learning Rates for Q-learning”[6] derives how polynomial learning rates such as $\alpha = 1/t^{\omega}$ converge much better than linear ($\alpha = 1/t$) rates.

Picking the behavioural policy is important; it must balance exploring and gaining rewards (exploitation). For some policies, the ϵ parameter determines the ratio of exploration and exploitation, a high ϵ would result in more exploration. There are two common behaviour policies for this:

- ϵ-greedy: this policy randomly picks between (A) selecting the best action based on the current Q-table or (B) selecting another action. A or B is random with the ratio determined by ϵ
- ϵ-soft: this policy assigns probabilities to all actions based upon their q-values, biased towards the higher Q-values by ϵ. Then, random actions are chosen according to this probability distribution.

Both ϵ-greedy and ϵ-soft policies utilise the current Q value estimates, which can lead to bias. Incorrect over-optimistic and over-pessimistic estimates can lead to a poor distribution of observations, compounding these effects. One approach to limit bias is called double Q-learning. This is where two Q-learning tables are kept, and actions are chosen based upon alternating tables this helps average out the bias and improves the accuracy of estimates.

Bibliography

[1] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[2] S. P. Meyn and R. L. Tweedie, *Markov chains and stochastic stability.* Springer Science & Business Media, 2012.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[4] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[5] K. You, M. Long, J. Wang, and M. I. Jordan, "How does learning rate decay help modern neural networks?" *arXiv preprint arXiv:1908.01878*, 2019.

[6] E. Even-Dar, Y. Mansour, and P. Bartlett, "Learning rates for q-learning." *Journal of machine learning Research*, vol. 5, no. 1, 2003.