Electronics and Computer Science

Faculty of Physical and Applied Sciences

University of Southampton

Daniel Lee

<Date>

Linear Quadratic Regulation using Reinforcement Learning

Project supervisor: Bing Chu

Second examiner: Mohammed El-Hajjar

A project report submitted for the award of

MEng Electronic Engineering with Artificial Intelligence

Abstract

The aim of this project is to demonstrate that the *optimal control* for a *Linear Quadratic Regulation* (LQR) problem can be obtained by the use of *Reinforcement Learning* methods, without prior knowledge of the system’s dynamics. For a better understanding of the core topics of reinforcement learning, simulations and demonstrations of core reinforcement learning topics, including *Bellman equations*, *greedy* and *-greedy* algorithms, *policy evaluation*, *policy iteration*, *policy improvement*, *temporal difference learning*, *exploration* and *exploitation,* *Markov Decision Process* and *Q-Learning*, were performed. Related to control systems, research on *optimal control theory*, *state-space modelling*, *cost function* and the *Algebraic Riccati Equation* was also made, as they were essential concepts for this project. Finally, the inverted pendulum problem, which consists of balancing an inverted pendulum attached to a cart that can only move sideways, was solved through the application of the *adaptive policy iteration* algorithm using Q-Learning.

Contents

Abstract ii

Contents iii

Acknowledgements iv

1 Introduction 1

1.1 Background Reading 1

1.1.1 Reinforcement Learning 1

1.1.2 System Elements 1

1.1.3 Markov Decision Process 2

1.1.4 Bellman Equations 2

1.1.5 Linear Quadratic Regulation 3

1.2 Previous Work 4

1.2.1 K-Arm Bandit Problem 5

1.2.2 Greedy Method 5

1.2.3 ε-Greedy Method 6

1.2.4 Optimal Policy 7

1.2.5 Grid World Problem 7

1.3 Adaptive controller using Q Learning 8

1.3.1 Q Function 8

1.3.2 Value function approximation 9

1.3.3 Algorithm implementation and procedure 9

2 Design Strategies 10

2.1 Power Plant 10

2.2 Inverted Pendulum 10

3 Conclusions 11

4 Test 12

References 13

Appendix A. Hardware Costs 14

Appendix B. Selected Listings 15

Acknowledgements

I would like to thank Dr Bing Chu for his guidance and support on this project.

# 

# Introduction

If an accurate description of the dynamics of a system is available, the solution for the optimal control, within the constraints of LQR, can be trivially obtained by solving the Algebraic Riccati equation. However, in many cases, an exact representation of a dynamic system’s model cannot be easily found. This is where reinforcement learning can fill in the gaps. A very powerful feature of reinforcement learning is that it can *learn* through experience. This means that by analysing several aspects of the behaviour of a system throughout many finite iterations, or *episodes*, a close estimate of the system dynamics can be computed, therefore allowing us to find the optimal policy.

## Background Reading

With the purpose of getting familiar with the core topics of this project, a great portion of the available time was invested in reading the appropriate literature. For the topic of reinforcement learning, the main source of reference was the book titled ‘Reinforcement Learning: An Introduction’ by Sutton and Barto [1]. With respect to Linear Quadratic Regulation, the research was mainly based on the article ‘Reinforcement Learning and Feedback Control’ by Lewis, Vrabie and Vamvoudakis [2].

### Reinforcement Learning

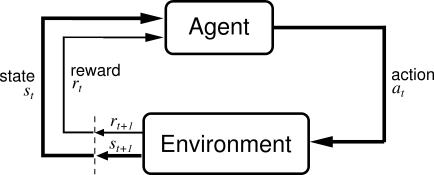
To begin with, it is important to define what reinforcement learning is. Reinforcement learning is ‘learning by interacting with an environment’ [3]. An agent can be considered to be using reinforcement learning if it adjusts the way it behaves at different states of the environment by considering the consequences of its past experiences. This can be compared to the concept of ‘trial and error’, which is the simplest natural learning behaviour presented in animals and humans alike. By assessing the consequences of the actions taken in the past, the agent will seek to select the action that provided it with the best outcome. By following an iterative process for this procedure, the agent’s knowledge on the consequences for the different actions will become increasingly more ‘clear’. This will consequently shape the agent’s behaviour in favour of selecting the most rewarding actions.

Figure 1: Agent-environment interaction (Sourced from: [1])

### System Elements

For a more formal description, the reinforcement learning system can be partitioned into three main elements: a policy, a reward and a value [4]. A policy is a rule that maps the actions to the states of the environment. In simpler words, it determines the way in which the agent should behave in different situations that occur during its interaction with the environment. When an agent selects an action from a state, it collects a numerical value called reward. A reward quantitatively expresses how positive the performance of the action taken at a state was. While a reward represents an immediate acquisition, the value of a state provides the information of the total expected reward, or return, that is expected to be collected by following the state. Therefore, it can be considered as a quantity that expresses how much the agent can benefit in the long-term.

A graphical representation of the agent-environment interaction, illustrating the elements described above, is shown in Figure 1.

### Markov Decision Process

A dynamic system for which ‘*the environment’s response at depends only on the state and action representations at time*  [1], is said to have the *Markov property*. In other words, a dynamic system has the Markov property if at any each state-action pair, , in the environment, equation (1) is equivalent to equation (2).

A reinforcement learning task that has the Markov property and which its environment has a finite number of states and actions is called a *Markov Decision Process*. In these systems, the agent’s interaction with the environment is governed by a set of equations called the *transition probabilities*, equations (3) and (4).

Equation (3) describes the probability that the agent will transition to a certain state from choosing action at state and equation (4) provides the estimated reward from transitioning to state by choosing action from state . These set of equations describe the dynamics of the system and consequently determine the agent’s behavior throughout its interaction with the environment.

### Bellman Equations

The *value* of a state or state-action pair represents how positively an action or state-action pair performed in the past. This information is essential for the agent to take the most appropriate decision. Values for each state, or *state-value*s , can be computed using equation (5), where is the *discounted return* for and is the policy in which the agent is acting on.

As illustrated in the equation, the value of a function is an estimate of the total discounted return from the current time, , to . If the discount factor is set closer to 1, it means that the agent becomes more farsighted. The same expression can be obtained for the state-action pair, or action-values . This is shown in equation (6).

By further expanding equation (5), the derivation for the stochastic Bellman equation for value functions can be obtained. This procedure is shown in equation (7).

The same can be performed on the action-value to derive the Bellman equation for the action-value, equation (8).

These two Bellman equations are the base equations necessary to approach a reinforcement learning problem through *on-line* *dynamic programming*.

### Linear Quadratic Regulation

Linear quadratic regulation defines a problem in which the system dynamics are defined by linear differential equations and the cost is represented by a quadratic function [?]. In this context, the aim is to minimize the cost generated by the system during its lifetime.

For a discrete-time LQR, the governing equations are defined by equations (9), (10), (11) and (12).

Equation (9) defines the state transition, (10) is the control policy, (11) the immediate reward and (12) represents the cost function, in this case, the infinite-horizon cost, since it is the total sum of the costs from the starting point to infinity. The matrices Q and R defined in (11) are arbitrary matrices that regulate the weight, or importance, of the cost that the states and inputs contribute to the system. These equations contain all the required information to describe a reinforcement learning problem, thus allowing us to build the bellman equations from them, as shown in equations (13) and (14).

Equations (13) and (14), are the value and the action-value functions, respectively, and these are equivalent to the standard bellman previously defined in equations (7) and (8). For quadratic value functions, equation (13) can be rewritten as (15).

As in any reinforcement learning problem, the objective is to find the optimal policy. In the LQR case, this is the policy that will minimize the cost function J. For the infinite horizon discrete-time LQR, this can be achieved by minimizing the *Hamiltonian function* [?], which represents the *temporal difference error* when transitioning from a state to another. By minimizing the Hamiltonian function, as in (16), the optimal control policy (17) can be obtained. Moreover, by substituting (17) into (15) and rearranging, the algebraic Riccati equation can be computed and solved to find the matrix P, required in the optimal control (17).

The resulting optimal control solution will minimize the cost of the system along its entire trajectory.

## Previous Work

In parallel to the background research, practical simulations were performed on the relevant topics with the aim to complement the theory.

### K-Arm Bandit Problem

A good example that illustrates the basic concepts of reinforcement learning in practice is the k-arm bandit problem. In this scenario, the aim of the agent is to maximize its return by pulling the desired arm, among the k number of arms, from the bandit, at each finite discrete step, or turn. To solve this problem, two simple methods can be implemented. These are the *greedy method* and the *ε-greedy* method.

### Greedy Method

The greedy method represents a policy where the agent always chooses the action that has the highest expected reward. In the case of the k-arm bandit problem, the expected reward of an action, or value , can be expressed with equation (19), where and represents the number of times that the actions was selected before the current time .

The equation shows that every time the agent selects the action, or arm, , its value is updated by averaging all the past collected rewards. This means that if an action was selected an infinite amount of times, , its value would eventually converge to its actual value . Therefore, at every turn, the agent will pull the arm that returned the most profit in past experiences. This can be represented by (20).

However, as in many real cases, the bandit problem is a stochastic system where the immediate reward has a probability distribution, or noise, that might negatively affect the agent’s shortsighted decisions. An agent could be ignoring the optimal action only based on few past experiences and never come back to it again. To further demonstrate this, a simulation of this example was performed by creating a MATLAB function called greedy, Appendix A. The greedy function’s inputs are, the number of bandits, the number of actions, the number of turns, and the Gaussian distribution parameters to set the actual action values and the Gaussian distribution parameters to set the noise for every immediate reward. For this simulation, the initial actual values for all were selected from a set of numbers from a Gaussian distribution . In addition to this, Gaussian noise was added to all immediate rewards. For a more accurate evaluation, the results for the 15-arm bandit problem were averaged over 1000 bandits. Figure 2 illustrates how often the agent converged to the optimal action over a 1000 steps.

By analyzing the graph, it can be seen that greedy algorithm led the agent to converge to the optimal action in around %35 of the cases. To improve the performance of the agent in the long term, it is possible to use a slight variation of this method called the ε-Greedy method.



Figure 2: Total Average Return Figure 3: Optimal Action

### ε-Greedy Method

From the previous subsection, the theory and implementation of the greedy algorithm were shown. In this section, a variation of the greedy algorithm called ε-greedy will be explored. One of the issues from the ε-greedy was that because it always forced the agent to choose the best action-value in every state, if a particular action-value was low from the start, the agent never chose it for the rest of the simulation, therefore, never being able to discover its actual value. To avoid this problem, a variant of this algorithm that allows the agent to explore from time to time can be used. In every state, the agent will randomly choose an action with equal probability in all of them. This means that because if an action is chosen infinite times it converges to its actual value, this will allow the agent to have a more precise information of its options. This algorithm is called ε-greedy. To simulate this method, a function was develop in MATLAB and can be found in Appendix B.

The first simulation was created by using an ε of 0.10. The 15-bandit problem was run for 1000 steps and averaged over 1000 bandits. The result was then compared to the total average return for the greedy method in Figure 2. It can be seen that although for the first 400 steps the greedy algorithm was able to collect a higher average return, the ε-greedy method for shows a better performance in the long term. In theory, for ε-greedy algorithms, as the number of steps goes to infinity, the average reward would converge to the optimal reward.

A plot showing the percentage in which the agent converged to choose the optimal value can be seen in Figure 3. The graph shows that as it reaches the last time step, the optimal action percentage reaches around 60 percent, which is almost as double as the performance for the greedy method. This is because we allow the agent to explore for 10 percent of the time. This means that although 10 percent of the time the action might not receive the maximum immediate reward, it will aid the agent to get a more precise estimate of the actual state-action values.

For the last ε-greedy simulation, an epsilon value of 0.05 was used. From Figure 3, it can be seen that although the optimal action percentage does not increase as steeply as the previous ε-greedy simulation, it still outperforms the greedy method. If the number of steps were to be increased, this version of the e-greedy algorithm would outperform the other ones and eventually converge to an accuracy of optimal action to 99.9 percent. Taking these simulations into account, depending on the requirements of the reinforcement learning problem, it is important to set a suitable balance between *exploration* and *exploitation*. If the immediate performance is important, it would be more effective to use a lower exploration factor, or in this case epsilon, and if the priority lies in the long term performance, a higher exploration factor would be more convenient.

### Optimal Policy

The *optimal policy* for a finite MDP can be obtained through two methods called *policy evaluation* and *policy iteration,* which are based on the Bellman equations for state and action-state values derived in equations (7) and (8). Policy evaluation is used to compute the deterministic state-values for all for an arbitrary policy , using the values obtained in the last episode, . This can be represented using equation (21).

By iterating this process for many times, , a state-value for all with an increasing accuracy can be computed. With the resulting information, an optimal policy for the dynamic system can be found. This process is called policy iteration.

### Grid World Problem

To show these methods in practice the ‘*grid world*’ problem from *Sutton and Barto’s* book [1] was reproduced solved by hand in Appendix C. The grid world problem can be classified as an *episodic* finite MDP, therefore there are a finite number of states and actions and does not require a discounting factor . For the grid shown in Figure 4, the 14 states are represented by each of the squares and the four possible actions that can be taken at each state are . The objective is to find the optimal policy for each state, that leads the agent to one of the terminal states, shaded in grey, in the minimum number of steps. For every step taken the immediate reward is -1 and if the agent takes an action that makes it go off the grid, it will return back to the last state. With the described system dynamics, policy iteration was applied for 3 times, starting with initial state-values set to 0. After three iterations, the optimal policy was retrieved using the equation for the optimal action-value in equation (22).

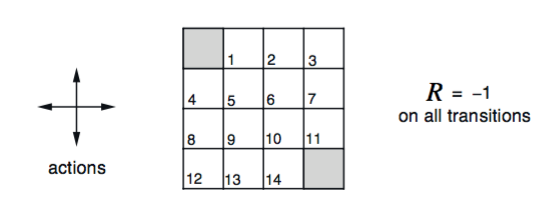


Figure 4: 4x4 Grid world Problem (*Sourced from:* [1])

## Adaptive controller using Q Learning

In contrast to all the problems analysed so far, such as the grid world problem analysed above, where all the environment’s characteristics where known in priori, there are systems in which this information is unavailable or simply unobtainable by mere observation. Aspects of the system such as state transitions, immediate rewards and number of states, may be unknown. In some other cases, not only the absence of information is a problem, but also, the scale of the system states. This is because if the number of states in a system were extremely large, e.g. >, generating the value function of all of these states would take an excessive amount of time and computational power.

The adaptive controller algorithm can overcome these issues because it does not require having knowledge of the model of a system to converge to the optimal policy. Furthermore, it is unaffected by the number of states in the system as it uses *value function approximation* [?] with the aid of a neural network.

### Q Function

The term Q Learning is derived from the action-value function, or *Q function*, defined in equation (6) and stores the information of the expected reward, or cost, in the case of LQR, of taking a particular action in a state. For a given policy, the Q Learning algorithm consists of applying policy evaluation and policy improvement in an online procedure, and updating the Q function of the states in the state space every discrete step.

In LQR, the Q function can be defined as in the following equation (21).

Rearranging into matrix form,

In parametric form, the Q function defined in (21) can be written as (23), where

Finally, by grouping the quadratic terms of the vector into , the expression for the Q function becomes (24), where the vector contains the variables in the matrix defined in (22) and the vector contains the state and input variable values

The main purpose of transforming the definition of the Q function in this form is that the unknown matrix in (22), which is dependant on the system’s dynamics, and , can be estimated by the adaptive controller algorithm without any prior knowledge of system’s model.

### Value function approximation

Considering that the system dynamics are not known, which implies that the matrix (22) can’t be directly computed, a method is required to estimate these parameters. This can be achieved by finding solutions for the variables in the vector, which are equivalent to the variables in the unknown matrix, but put into vector form, by applying *Recursive Least Squares* (RLS) [?] on equation (25), until convergence of the components in the vector.

RLS can be applied to estimate by the use of the following set of equations, where and ().

The vector can be seen as a simple single layer *neural network*, as it takes several inputs from , multiplies them by a weight and adds the resulting products to output a single output, and more importantly, it gets trained every RLS iteration.

Following this procedure will allow us to obtain the components of the vector that can be unpacked to obtain the matrix, which is needed to approximate the value function across the states in the system’s trajectory.

### Algorithm implementation and procedure

A step-by-step implementation can be achieved by following the steps below, Figure 5.

This algorithm makes use of a method named *one-step look ahead* *temporal difference* learning, since it requires the data from the current state and the following state, to perform RLS on the vector.

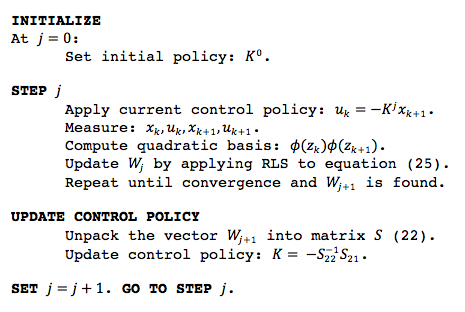


Figure 5: Adaptive controller algorithm using Q Learning (*Adapted from:* [2])

This iterative procedure finishes when the control policy is no longer being updated. This algorithm is equivalent to solving the algebraic Riccati equation, without prior knowledge of the system dynamics, in an online manner, and will eventually converge to the optimal policy .

# Design Strategies

In this section, the adaptive controller algorithm described in section 1.3 is applied to two different dynamic systems through software simulations. An analysis of the results and an overview of the performance using different parameters are also discussed and illustrated.

## Power Plant

12 pt text for main body

## Inverted Pendulum

# Conclusions

12 pt text for main body

# Test

References

[1] Book barto and Sutton

[2] Article

1. Hardware Costs

There weren’t any costs for this project as everything was done through available software.

1. Selected Listings

1 begin

2 c := c+1;

3 end