Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Daniel Lee
December 7, 2016

Linear Quadratic Regulation using Reinforcement Learning

Project supervisor: Dr Bing Chu
Second examiner: Dr Mohammed El-Hajjar

A progress report submitted for the award of
MEng Electronic Engineering with Artificial Intelligence

Abstract

The purpose of this project is to show the application of dynamic programming methods from *Reinforcement Learning*, based on *Bellman equation*, into discrete-time *Linear Quadratic Regulation (LQR)* control theory systems. The progress that has been achieved so far includes the background reading of the core topics, numerical derivations of the Bellman equation, simulations on *Greedy* and ϵ-*Greedy* algorithms and mathematical analysis of *policy evaluation*, *policy iteration* and *policy improvement* in dynamic systems. Moreover, the *Bellman Optimality Equation* for LQR systems was derived by solving the resulting *Algebraic Riccati Equation*. FINISH LAST

Contents

Project description

FINISH LAST

1 Background Research

With the aim 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 [**DUMMY:1**]. With respect to Linear Quadratic Regulation, the research was based on the article '*Reinforcement Learning and Feedback Control*' by Lewis, Vrabie and Vamvoudakis [2].

1.1 Reinforcement Learning

The concept of reinforcement learning describes the iterative computational process that an agent executes by interacting with the given environment. It provides the agent with mathematical algorithms to maximize the long-term return by taking optimal decisions from a set of states and actions described by the environment.

1.1.1 Components

The reinforcement learning system can be described with "four main elements: a policy, a reward function, a value function and a model of the environment" [3].

$$Q_t(A_t^*) = max_a Q_t(a) \tag{1}$$

The action-value of an action can be computed from the following equation (2):

$$Q_t(a) = \frac{R_1 + R_2 + ... + R_{K_a}}{K_a} \tag{2}$$

The equation states that the action-value of an action at a particular time-step is determined by the total reward accumulated, divided by the number times that it was selected in the past. In this simulation, the probability distribution that was used for the immediate actions' rewards was a Gaussian distribution $\mathcal{N}(0,1)$. Due to the nature of this distribution, if an action was selected infinite times, its action-value would eventually converge to its *actual* value, q_*.

1.1.2 Simulation

To test this algorithm, the *actual* values for each of the actions were set by selecting numbers from a $\mathcal{N}(0,1)$ distribution. Following this, the action-values were initialized by adding Gaussian noise of the form $\mathcal{N}(0,1)$. To test the performance of the greedy algorithm, the simulation was first run for 300 steps from a bandit problem with 5 possible actions and averaged over 300 bandits. In Figure (1), we can see the average return from simulating an 5-bandit problem with a range of 5 actions over 300 steps.

It can be observed that the greedy method performs around 0.2 worse than the optimal solution in average. A better way of analyzing the performance of the greedy algorithm can be computed by observing the percentage of the times that the agent chose the optimal action. This is shown in Figure (2).

It can be observed above that the agent converges to the optimal action in around 70 percent of the bandit tasks. This simulation shows that although being very simple, the greedy algorithm shows an overall positive performance.

1.2 ϵ-Greedy

1.2.1 Theory

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 textitactual value, this will allow the agent to have a more precise information of its options. This algorithm is called ϵ-Greedy.

1.2.2 Simulation

The first simulation was created by using an ϵ of 0.1. The 5-bandit problem was run for 600 steps and averaged over 300 tasks. The result was then compared to the optimal return. This can be seen in the figure below.

From the graph above, we can see that opposed to the greedy algorithm, the average reward gets closer to the optimal average reward as the number of steps increases. In theory, as the number of steps goes to infinity, the average reward would converge to the optimal reward.

Following this, a plot showing the percentage in which the agent converged to choose the optimal value is shown below.

From analyzing the graph above, we can see that as we reach step 600, the optimal action percentage reaches around 90 percent. 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 real value.

For the last e-greedy simulation, an epsilon value of 0.01 was used. The first simulation compares the average return of this particular e-greedy algorithm against the optimal return.

From this graph, we can see that the average return increases steadily. If the number of steps is increased, this version of the e-greedy algorithm will outperform the other ones and eventually converge to an accuracy of optimal action to 99.9 percent.

A more helpful graph is plotted above and show how how the optimal action percentage is steadily increased. Although is not as fast as using an epsilon 0.1, again, in the long run this configuration will outperform the previous configuration.

2 Dynamic Programming

2.1 Policy Evaluation

The idea behind policy evaluation is to find the state-value for all the possible states using an arbitrary policy π. To compute the state-value we use one of Bellman's equations:

$$v_{k+1}(s) = \mathbf{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \qquad (3)$$

This equation computes the estimated state-value for the following time step using information from the current state. This method is called full back up, because it uses the information in the values of all the available states to estimate a state-value.

2.2 Policy Iteration

In order to converge to the v_π of each state we can iterate over the episodic system many times. This concept is called *policy iteration*.

2.2.1 Gridworld example

Policy iteration can be applied in the gridworld scenario.