COMP3200: Individual Project
Reinforcement Learning

Daniel Lee
dl2g14@ecs.soton.ac.uk

November 28, 2016

Motivation

The purpose of this paper is to document every relevant work that I do during my background research.

1 Bandit Problems

In the classic bandit problem, the agent is given a limited set of actions to choose from at every time-step. Each action provides the agent with its respective stochastic reward from a particular distribution. The objective is to collect as much total reward, or *return*, from the limited amount of available time-steps. In this section, the popular greedy and ϵ-greedy algorithms will be implemented to tackle this task and the resulting performances will be compared.

1.1 Greedy

1.1.1 Theory

As mentioned above, the agent can choose an action from a limited range of options. The greedy algorithm leads the agent to choose the action that has the highest *action-value* at every single time-step. This can be numerically represented in this way:

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

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.

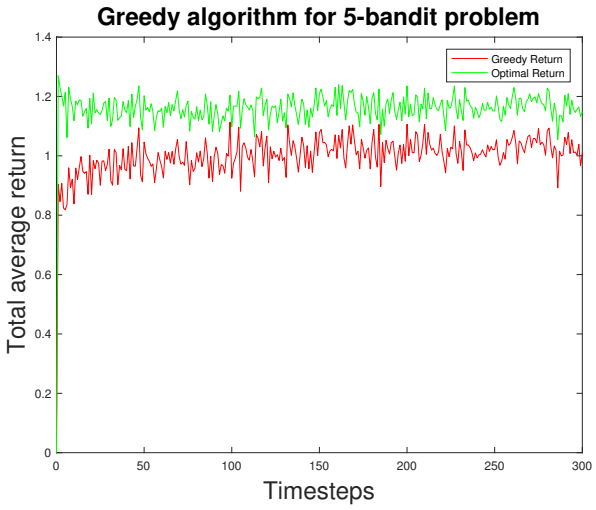

Figure 1: Greedy vs Optimal

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).

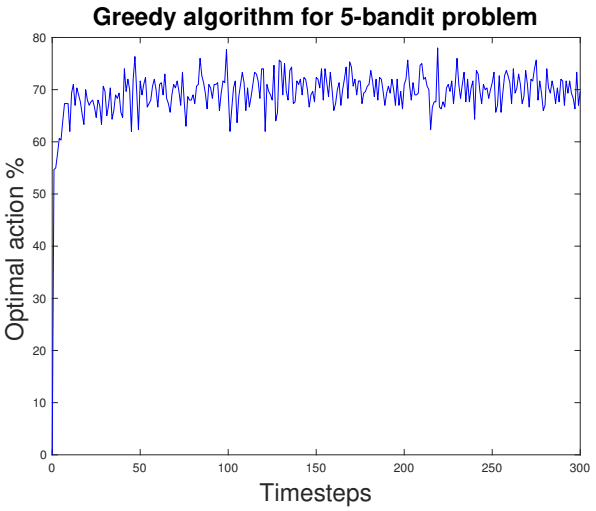

Figure 2: Optimal action for greedy algorithm

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.

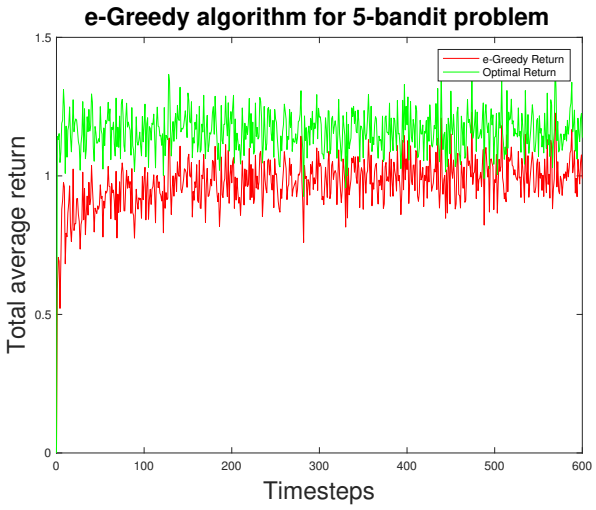

Figure 3: e-Greedy vs Optimal ($\epsilon = 0.1$)

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.

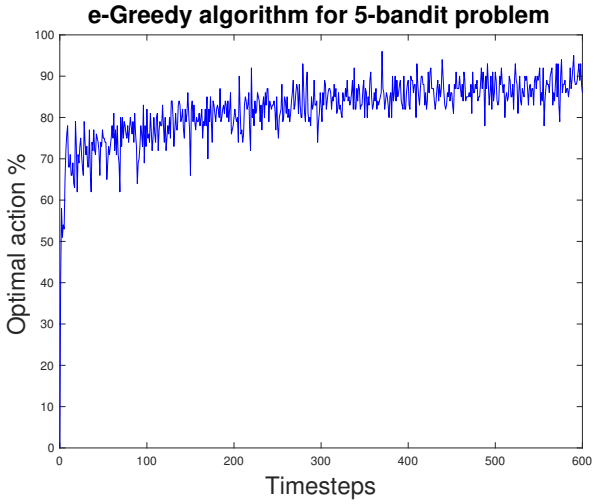

Figure 4: Optimal action for e-greedy algorithm ($\epsilon = 0.1$)

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.

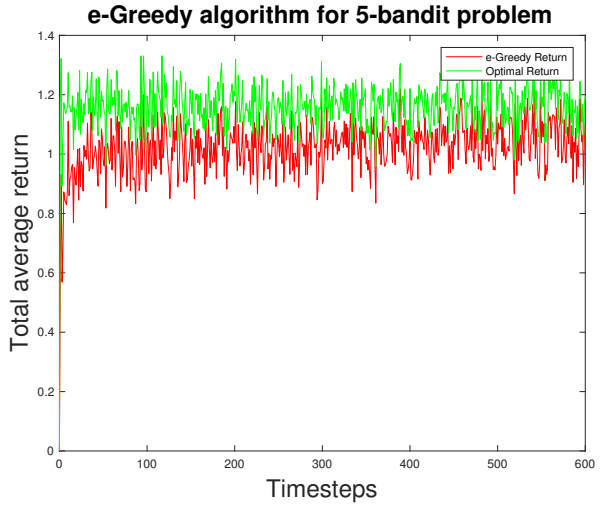

Figure 5: e-Greedy vs Optimal ($\epsilon = 0.01$)

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.

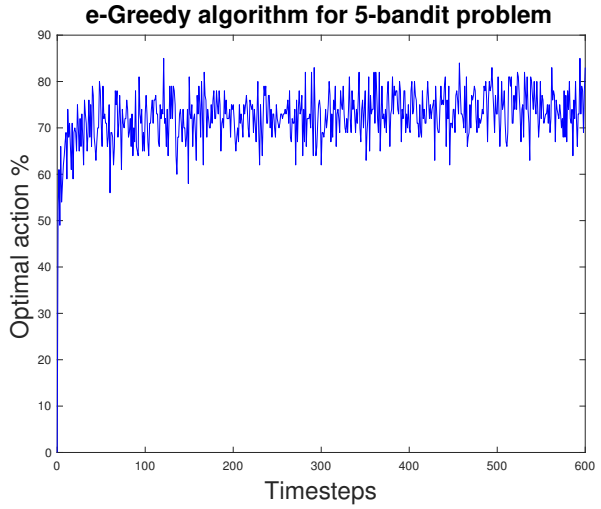

Figure 6: Optimal action for e-greedy algorithm ($\epsilon = 0.01$)

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 we can iterate over the episodic system many times. This concept is called *policy iteration*.

Appendices

Greedy function MATLAB code

```
function [total_average_return, optimal_action, optimal_return] = greedy(bandit_n, actions_n
    , timesteps, actual_distr, noise_distr)
% This function generates the total average return for n-bandit problems
% with the assigned number of bandits, number of timesteps, number of actions,
% actual values and noise.

% Algorithm Implementation Steps
% Step 1: Set actual action-values Q* from nrand(0,1)
% Step 2: Initialize action-values adding Gaussian Noise
% Step 3: Start Greedy decision process
% Step 4: Choose action with the maximum estimated action-value
% Step 5: Update accumulated reward
% Step 6: Update estimated action-value
% Step 7: Store total average return for each bandit
% Step 8: Check if it converged to the optimal value
% Step 9: Plot average total return against optimal average return


% CONSTANTS
% bandit_n = 20;
% actions_n = 5;
% timesteps = 1000;
% actual_distr = [0,1];
% noise_distr = [0,1];

% External variables
tot_avg_rew = zeros(1,bandit_n);
tot_opt_rew = zeros(1,bandit_n);
optimal_choice = 0;

for bandit=1:bandit_n

    % Clean action-values every loop
    actual_q = zeros(1,actions_n);
    estimate_q = zeros(1,actions_n);

    % 1. Generate actual values q*(A) and store them in q[] OK
    for i=1:actions_n
        actual_q(i) = normrnd(actual_distr(1),actual_distr(2));
    end

    % 2. Initiliaze action-values
    for i=1:actions_n
        gauss_noise = normrnd(noise_distr(1),noise_distr(2));
        estimate_q(i) = actual_q(i) + gauss_noise;
    end

    % 3. Greedy decision process
    % Clean values every loop
    total_reward = 0;
    acc_reward = zeros(1, actions_n);
    opt_reward = 0;
    action_counter = zeros(1,actions_n);

    for i=1:timesteps
        % 4. Choose option with max(estimate_q)
        [~, index] = max(estimate_q);
        [maxval,~] = max(actual_q);
        gauss_noise = normrnd(noise_distr(1),noise_distr(2));
        gauss_noise2 = normrnd(noise_distr(1),noise_distr(2));

        % Rewards update
        imm_reward = actual_q(index) + gauss_noise;
        total_reward = total_reward + imm_reward;
        opt_reward = opt_reward + maxval + gauss_noise2;

        % 5. Update accumulated reward
        acc_reward(index) = acc_reward(index) + imm_reward;

        % 6. Update estimated value
        action_counter(index) = action_counter(index) + 1;
```

```
        estimate_q(index) = acc_reward(index) / action_counter(index);
    end

    % 7. Store total return for each bandit
    tot_avg_rew(bandit) = total_reward / timesteps;
    tot_opt_rew(bandit) = opt_reward / timesteps;

    % 8. Check it it converged to the optimal value
    [~, optimal_index] = max(action_counter);
    [~, actual_index] = max(actual_q);
    if optimal_index == actual_index
        optimal_choice = optimal_choice + 1;
    end
end

% 9. Plot total average return against optimal average return
total_average_return = sum(tot_avg_rew) / bandit_n;
optimal_action = optimal_choice / bandit_n;
optimal_return = sum(tot_opt_rew) / bandit_n;
```

e-Greedy function MATLAB code

```
function [total_average_return, optimal_action, optimal_return] = egreedy(bandit_n,
    actions_n, timesteps, actual_distr, noise_distr, e)
% EGREEDY Returns the total average return, optimal return and optimal action percentage for
    the
% n-bandit problem with the given number of bandits, actions, timesteps, action-values and
    noise distributions
% and epsilon.
% EGREEDY(bandits, actions, timesteps, action_value_distribution[mean,
% variance], noise_distribution[mean,variance], epsilon)

% Algorithm Implementation Steps
% Step 1: Set actual action-values Q* from nrand(0,1)
% Step 2: Sort action-values in ascending or
% Step 3: Initialize action-values adding Gaussian Noise
% Step 4: Start e-Greedy decision process
% Step 5: Check if x is lower than epsilon
% Step 6: Choose action with the maximum estimated action-value
% Step 7: Update accumulated reward
% Step 8: Update estimated action-value
% Step 9: Store total average return for each bandit
% Step 10: Plot average total return against optimal average return

% DEBUG VARIABLES
% bandit_n = 20;
% actions_n = 5;
% timesteps = 1000;
% actual_distr = [0,1];
% noise_distr = [0,1];

% External variables
tot_avg_rew = zeros(1,bandit_n);
tot_opt_rew = zeros(1,bandit_n);
optimal_choice = 0;

for bandit=1:bandit_n

    % Clean action-values every loop
    actual_q = zeros(1,actions_n);
    estimate_q = zeros(1,actions_n);

    % 1. Generate actual values q*(A) and store them in q[] OK
    for i=1:actions_n
        actual_q(i) = normrnd(actual_distr(1),actual_distr(2));
    end

    % 2. Initiliaze action-values
    for i=1:actions_n
        gauss_noise = normrnd(noise_distr(1),noise_distr(2));
        estimate_q(i) = actual_q(i) + gauss_noise;
    end

    % 3. Greedy decision process
    % Clean values every loop
    total_reward = 0;
```

```
52      acc_reward = zeros(1, actions_n);
53      opt_reward = 0;
54      action_counter = zeros(1,actions_n);
55
56      for i=1:timesteps
57          % 4. Choose option with max(estimate_q)
58          [~, index] = max(estimate_q);
59
60          % 5. If we fall under epsilon, choose random action
61          x = rand;
62          if x < e
63              index = randi(actions_n);
64          end
65
66          [maxval,~] = max(actual_q);
67          gauss_noise = normrnd(noise_distr(1),noise_distr(2));
68          gauss_noise2 = normrnd(noise_distr(1),noise_distr(2));
69
70          % Rewards update
71          imm_reward = actual_q(index) + gauss_noise;
72          total_reward = total_reward + imm_reward;
73          opt_reward = opt_reward + maxval + gauss_noise2;
74
75          % 6. Update accumulated reward
76          acc_reward(index) = acc_reward(index) + imm_reward;
77
78          % 7. Update estimated value
79          action_counter(index) = action_counter(index) + 1;
80          estimate_q(index) = acc_reward(index) / action_counter(index);
81      end
82
83      % 8. Store total return for each bandit
84      tot_avg_rew(bandit) = total_reward / timesteps;
85      tot_opt_rew(bandit) = opt_reward / timesteps;
86
87      % 9. Check it it converged to the optimal value
88      [~, optimal_index] = max(action_counter);
89      [~, actual_index] = max(actual_q);
90      if optimal_index == actual_index
91          optimal_choice = optimal_choice + 1;
92      end
93  end
94
95  % 10. Plot total average return against optimal average return
96  total_average_return = sum(tot_avg_rew) / bandit_n;
97  optimal_action = optimal_choice / bandit_n;
98  optimal_return = sum(tot_opt_rew) / bandit_n;
```