CARLETON UNIVERSITY

SCHOOL OF
MATHEMATICS AND STATISTICS

HONOURS PROJECT

TITLE: Model-Based and Model Free Dynamic Programming

AUTHOR: Lorenzo Papa

SUPERVISOR: Dr. Minyi Huang

DATE: April 28th, 2024

Contents

1 Introduction

Dynamic programming is a term used to describe the theory and algorithms that solve multi-stage decision problems. The second chapter of this project will analyse how dynamic programming is used to minimize the cost of finite horizon problems by finding optimal controls at each stage. These problems will be stochastic in nature so we will deal with expected costs/rewards over a probability distribution of successor states. The third chapter will discuss tabular methods that compute optimal policies to infinite horizon problems. The fourth chapter will touch on the 'model-free' reinforcement learning algorithm Q-learning. Q-learning finds approximately optimal policies but does not require the knowledge of the environment's probability distribution. Instead, it learns optimal policies through experience.

2 Dynamic Programming in Finite Horizon

2.1 The Basic Problem and Notation

Consider the stationary discrete-time dynamic system

$$x_{k+1} = f_k(x_k, \mu_k, w_k), \quad k = 0, 1, ..., N-1$$

where,

k indexes time.

$x_k \in X$ is the state of the system at time k.

$u_k = \mu_k(x_k) \in U$ is the control applied at time k.

w_k is a random parameter with probability distribution $\mathcal{P}_k(w_k|x_k, u_k)$.

N indicates *finite horizon* and the number of times control is applied.

This process has a *finite horizon* because it reaches a state that terminates the entire process in N time steps, after which no controls can be applied. Given a state X_k, we want to find a sequence of controls $\pi = \{\mu_0, \mu_1, ..., \mu_{N-1}\}$ where, $\mu_k : X \to U$, $\mu(x_k) \in U(x_k)$, for all $x_k \in X$, $k = 0, 1, ..., N-1$, that minimizes the cost function

$$J_\pi(x_0) = E\{g_N(X_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\}$$

where, the expectation is with respect to the distribution of w_k. $g_k(x_k, \mu_k(x_k), w_k)$ is the cost function at stage k determined by the state, control and random parameters at time k. Furthermore, $g_N(X_N)$ is a given final cost of the $N-1^{st}$ stage. We write:

$$J^*(x_0) = \min_{\pi \in \Pi} J_\pi(x_0)$$

where, $J^*(x_0)$ is the *optimal cost-to-go function* of the N-stage problem starting at state x_0.

The process is most easily understood by viewing it as an agent moving through the state space at each discrete time step. If the agent starts at x_0 and observes the current state, it applies the control dictated by current policy. Following this, the system moves the agent to a successor state and a cost is incurred depending only on x_k, u_k and w_k. Here is how the process could be shown as a flow chart.

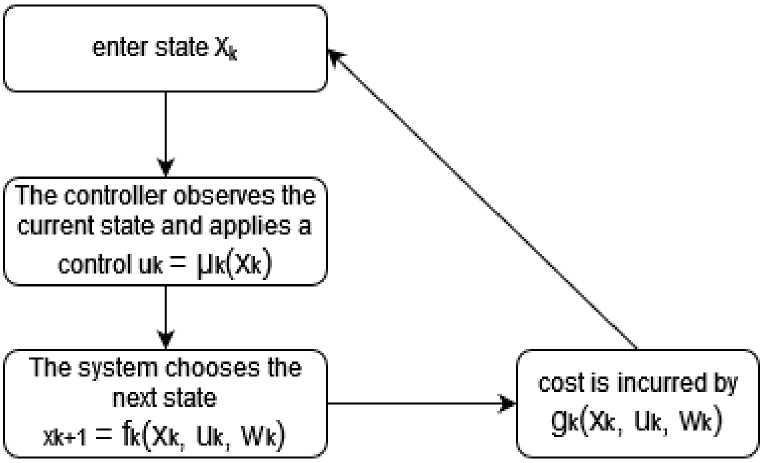

enter state Xk
The controller observes the current state and applies a control uk = μk(Xk)
The system chooses the next state xk+1 = fk(Xk, Uk, Wk)
cost is incurred by gk(Xk, Uk, Wk)

Figure 1: flow chart of one transition in an N-stage decision problem

If we start in X_0, how can we minimize the expected cost of the process until it stops? To solve a finite horizon decision process, we obtain an optimal policy $\pi^* = \{\mu_0^*(x_0), \mu_1^*(x_1), ..., \mu_{N-1}^*(x_{N-1})\}$ by using the dynamic programming algorithm.

2.2 Principle of Optimality and The Dynamic Programming Algorithm

The dynamic programming algorithm finds $J^*(X_k)$ and μ_k^* for every possible state that the system could be in at stage k. The algorithm rests on the principle of Optimality which is the idea that the optimal cost-to-go at the starting state can be found by solving increasingly longer sub problems. The principle of optimality suggests that an optimal policy can be constructed by solving a sub-problem starting from stage $N-1$, followed by solving a sub-problem starting from $N-2$, and so on. The dynamic programming algorithm is as follows,

Dynamic Programming Algorithm. *For every initial state x_0, the optimal cost $J^*(X_k)$ of the basic problem is equal to $J_0(x_0)$, given by the last step of the following algorithm, which proceeds backward in time from period N - 1 to period 0:*

$$J_N(X_N) = g_N(X_N)$$

$$J_k(x_k) = \min_{u_k \in \mu_k(x_k)} E\{g_k(x_k, \mu_k(x_k), w_k) + J_{k+1}(f(_k(x_k, u_k, w_k)\}$$

k = 0,1,...,N - 1

If $u_k = \mu_k(x_k)$ minimizes the right side of $J_k(x_k)$ for each x_k and k, the policy $\pi = \{\mu_0, ..., \mu_{N-1}\}$ is optimal.

[Ber17]

2.3 Dynamic Programming Example

Consider the system

$$x_{k+1} = x_k + u_k + w_k \qquad k = 0, 1, 2, 3$$

In stage 0 the system starts at $x_0 = 1$ with cost function

$$J_\pi(x_0) = E[\sum_{k=0}^{2}(x_k^2 + u_k^2)] + g_3(x_3)$$

where, $g_3(x_3)$ is the terminal cost. Let $g_3(x_3) \equiv 0$. The control constraint set $U_k(x_k)$ is $\{u|0 \leq x_k + u \leq 2 : u \in \mathbb{Z}\}$ for all x_k and k. The random parameter w_k has the following probability distribution:

$$P(w_k = 1) = \frac{1}{4}, \qquad P(w_k = -1) = \frac{1}{2}, \qquad P(w_k = 0) = \frac{1}{4}$$

except if $x_k + u_k$ is equal to 0 or 2, in which case $w_k = 0$ with probability 1.

The DP equation for each stage will be

$$J_k = \min_{-x_k \leq u_k \leq 2 - x_k} E\left[(x_k^2 + u_k^2) + J_{k+1}(x_k + u_k + w_k)\right]$$

recall that $0 \leq x_k + u \leq 2$ and $w_k = 0$ with probability 1 at $x_k + u_k = 0$ and 2. Since we have that $x_0 = 1$, x_k can only range between the values 0 and 2 in all stages. The final cost is given so that we can use the dynamic programming algorithm for stage 2.

stage 2

Since $J_3(x_3) = 0$, the DP equation here is

$$J_2(x_2) = \min_{-x_2 \leq u_2 \leq 2 - x_2} \left[(x_2^2 + u_2^2) + 0\right]$$

This implies that minimizing control in this stage is $u_2^* = 0$ and the optimal cost is $J_2^*(x_2) = x_2^2 \; \forall x_2$.

stage 1

The DP equation in stage 1 is.

$$J_1(x_1) = \min_{-x_1 \leq u_k \leq 2-x_1} E\Big[(x_1^2 + u_1^2) + J_2(x_1 + u_1 + w_1)\Big]$$

Now we proceed to find the optimal control for each possible value of x_1.

$x_1 = 0$

$$J_1(0) = 0^2 + u_1^2 + E[J_2(0 + u_1 + w_1)]$$

$$u_1 = 0 : \mathrm{J}_1(0) = 0^2 + (0)^2 + J_2(0) = 0$$

$$u_1 = 1 : \mathrm{J}_1(0) = 0^2 + (1)^2 + 0.25J_2(2) + 0.5J_2(0) + 0.25J_2(1) = \frac{9}{4}$$

$$u_1 = 2 : \mathrm{J}_1(0) = 0^2 + (2)^2 + J_2(2) = 8$$

$$\Longrightarrow \mu^*(x_1 = 0) = 0 \quad \textit{and} \quad J_1^*(0) = 0$$

$x_1 = 1$

$$J_1(1) = 1^2 + u_1^2 + E[J_2(1 + u_1 + w_1)]$$

$$u_1 = -1 : \mathrm{J}_1(1) = 1^2 + (-1)^2 + J_2(0) = 2$$

$$u_1 = 0 : \mathrm{J}_1(1) = 1^2 + (0)^2 + 0.25J_2(2) + 0.5J_2(0) + 0.25J_2(1) = \frac{9}{4}$$

$$u_1 = 1 : \mathrm{J}_1(1) = 1^2 + 1^2 + J_2(2) = 6$$

$$\Longrightarrow \mu^*(x_1 = 1) = -1 \quad \textit{and} \quad J_1^*(1) = 2$$

$x_1 = 2$

$$J_1(2) = 2^2 + u_1^2 + E[J_2(2 + u_1 + w_1)]$$

$$u_1 = -2 : \mathrm{J}_1(2) = 2^2 + (-2)^2 + J_2(0) = 8$$

$$u_1 = -1 : \mathrm{J}_1(2) = 2^2 + (-1)^2 + 0.25J_2(2) + 0.5J_2(0) + 0.25J_2(1) = \frac{25}{4}$$

$$u_1 = 0 : \mathrm{J}_1(2) = 2^2 + 0^2 + J_2(2) = 8$$

$$\Longrightarrow \mu^*(x_1 = 2) = -1 \quad \textit{and} \quad J_1^*(2) = 6.25$$

stage 0

In stage 0 we have the initial condition that $x_0 = 1$. So we compute the cost of every control with $x_0 = 1$

$$J_0(1) = 1^2 + u_0^2 + E[J_1(1 + u_0 + w_0)]$$

$$u_0 = -1 : \mathrm{J}_0(1) = 1^2 + (-1)^2 + J_1(0) = 2$$

$$u_0 = 0 : \mathrm{J}_0(1) = 1^2 + 0^2 + 0.25J_1(2) + 0.5J_1(0) + 0.25J_1(1) = 4.5625$$

$$u_0 = 1 : \mathrm{J}_0(1) = 1^2 + 1^2 + J_1(2) = 8.25$$

$$\implies \mu^*(x_0 = 1) = -1 \quad \mathit{and} \quad J_0^*(1) = 2$$

finally here are all the optimal controls computed in stages 0,1 and 2

x_2	u_2^*	$J_2^*(x_2)$
0	0	0
1	0	1
2	0	4

x_1	u_1^*	$J_1^*(x_1)$
0	0	0
1	-1	2
2	-1	6.25

x_0	u_0^*	$J_0^*(x_0)$
1	-1	2

3 Dynamic Programming in Infinite Horizon

3.1 Preliminaries

In this chapter we assume an underlying Markov decision process that models the dynamic system previously mentioned. In most reinforcement learning literature, the goal is to gain the highest total reward by maximizing a value function denoted as V_π . These are in direct comparison to minimizing costs using cost functions. In this chapter we make the switch in notation from optimal cost J^* to optimal value V^* to emphasize our new goal here.

Definition 3.1 (Discrete Time Markov Chain)**.** *Let $\{X_n, n = 0, 1, 2, ...\}$ be a stochastic process that takes on a finite number of possible values indexed by the set $S = \{0, 1, 2, ..., n\}$. If $X_n = i \in S$, then the process is said to be in state i at time n. In state i, there is a fixed probability of transitioning to state j, P_{ij}, where, the markov property holds such that*

$$P\{X_{n+1} = j | X_n = i_n, X_{n-1} = i_{n-1}, ..., X_0 = i_0\} = P\{X_{n+1} = j | X_n = i_n\}$$

The transition probability can be understood as the conditional distribution of any future state being only dependent on the present state X_n.

Definition 3.2 (Markov Decision Process)**.** *A Markov Decision Process can be defined as a 4-tuple $\{\boldsymbol{S},\boldsymbol{A},\boldsymbol{P},\boldsymbol{R}\}$. $\boldsymbol{S}$ is a finite state space. $\boldsymbol{A}$ is a finite action space from which the agent chooses an action from when in state $s \in \boldsymbol{S}$. $\boldsymbol{P} = P\{S_{t+1} = j | S_t = i, a_t = a\} = P_{ij}(a)$ is the probability of transiting to state j given that action a is selected in state i at time t. $\boldsymbol{R} = R_a(i, j)$ is the immediate reward gained from transiting from state i to j due to action a.*

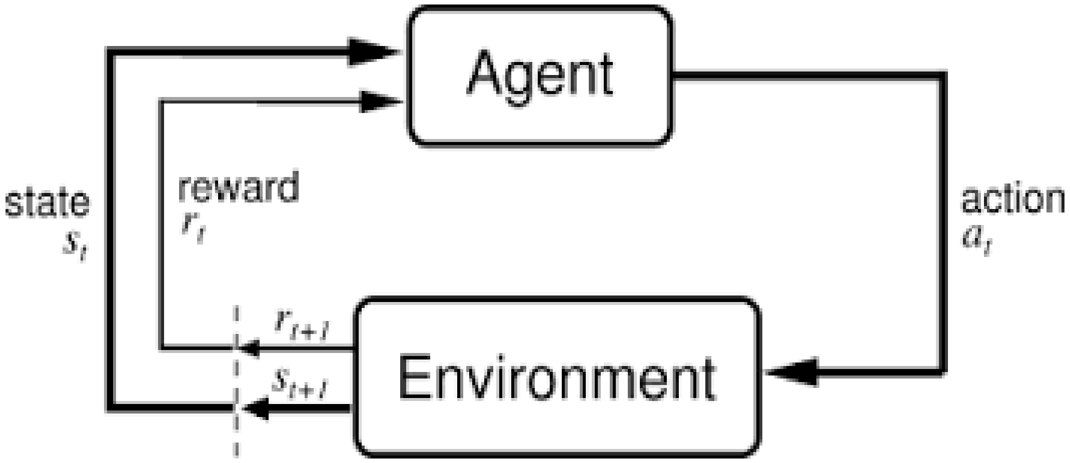


Figure 2: The Agent and Environment interface

For this project, solutions to MDP's are **stationary and deterministic** policies denoted as π, where $\pi : S \to A$. To have a sense of how optimal π is, it must satisfy some criterion of optimality. Stationary and deterministic mean the policy does not change with time. If a policy dictates that an action be selected at state i, then no matter how long the system operates between visits to state i, action a will be chosen with probability 1.

3.2 Discounted Total Reward

The criterion for optimality this project will use is discounted total reward. [SB20]. Let,

$$G_t = \sum_{k=t}^{\infty} \gamma^k r(s_k, a_k)$$

be the total discounted reward of a process with infinite time horizon starting at time t. $0 \leq \gamma < 1$, is a discount that is applied more often to rewards that are farther into the future. The intuition behind discounting is that rewards closer to the present are more valuable than rewards received later on. This formulation of total reward gives an important recurrent relationship between current and future rewards. Letting t be our starting time where γ^0 discount is applied, we have,

$$\begin{aligned} G_t &= r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \gamma^3 r(s_{t+3}, a_{t+3}) + ... \\ &= r(s_t, a_t) + \gamma(r(s_{t+1}, a_{t+1}) + \gamma r(s_{t+2}, a_{t+2}) + \gamma^2 r(s_{t+3}, a_{t+3}) + ...) \\ &= r(s_t, a_t) + \gamma G_{t+1} \end{aligned} \tag{1}$$

The **value function** predicts the effect of a policy on total reward. We define the value function of a state as:

$$V_\pi(s) = E_\pi[\sum_{k=t}^{\infty} \gamma^k r(s_k, a_k) | s_t = s]$$

for all $s \in S$.

This equation is interpreted as the expected return when starting in s and following policy π thereafter. Using (1), we can show an alternate characterization of the value function.

$$\begin{aligned} V_\pi(s) &= E_\pi[\sum_{k=t}^{\infty} \gamma^k r(s_k, a_k) | s_t = s] \\ &= E_\pi[r(s_t, a_t) + \gamma G_{t+1} | s_t = s] \\ &= \sum_{s'} p(s'|s, \pi(s))[r(s, \pi(s)) + \gamma E_\pi[G_{t+1} | s_{t+1} = s']] \\ &= \sum_{s'} p(s'|s, a)[r(s, a) + \gamma V_\pi(s')] \end{aligned} \tag{2}$$

This characterization shows that under policy π the value of s is the expectation of the immediate reward and the discounted value function of the successor state s'. Solving an MDP with infinite horizon means finding a policy that returns

$$V^*(s) = \max_{\pi} V_{\pi}(s)$$

for each $s \in S$.

3.3 Value Iteration

Value iteration is the DP algorithm turned into an update rule. Let $\mathcal{T}$ be a mapping in value space such that

$$(\mathcal{T}V)(s) = \max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V(s')]$$

Value iteration is successive application of this mapping to an initial guess V_0 to get increasingly accurate approximations of V^*. More specifically,

$$(\mathcal{T}^{k+1}V)(s) = \max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma(\mathcal{T}^k V)(s')]$$

We will see that VI converges to a fixed point ie. $\lim_{k \to \infty} \mathcal{T}^k V_0(s) = V^*(s)$. As a consequence

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V^*(s')]$$

holds. This is known as **Bellman's Equation**. It asserts that the optimal value function is a fixed point of the mapping $\mathcal{T}$. If Bellman's equation holds for each state s, action a is the optimal policy at state s

3.4 Solving an MDP via value iteration

Consider a problem with states 0,1 and 2 possible actions having rewards $R(s,a)$,

$$R(0,1) = 1 \qquad R(0,2) = 2 \qquad R(1,1) = 0 \qquad R(1,2) = 0$$

and transition probabilities

$$\begin{bmatrix} P_{00}(1) & P_{01}(1) \\ P_{10}(1) & P_{11}(1) \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ 2/3 & 1/3 \end{bmatrix}$$

$$\begin{bmatrix} P_{00}(2) & P_{01}(2) \\ P_{10}(2) & P_{11}(2) \end{bmatrix} = \begin{bmatrix} 1/4 & 3/4 \\ 1/3 & 2/3 \end{bmatrix}$$

Let $\gamma = 1/2$. With initial guess $V_0 \equiv 0$, use value iteration to approximate V^* by V_3. Let $\pi^k = \{\mu(0), \mu(1)\}$ denote the policy derived from the k^{th} iteration of the value iteration algorithm where, $\mu(i)$ is the control applied to state i.

<u>k = 1</u>

$V_1(0) = \max\{R(0,1) + \gamma(P_{00}(1) \cdot 0 + P_{01}(1) \cdot 0), R(0,2) + \gamma(P_{00}(2) \cdot 0 + P_{01}(2) \cdot 0)\}$

$= \max\{R(0,1), R(0,2)\}$

$= \max\{1, 2\} = 2$

$V_1(1) = \max\{R(1,1), R(1,2)\} = \max\{0, 0\} = 0$

$\pi^1 = \{2, 1\}$ or $\{2, 2\}$

<u>k = 2</u>

$$V_2(0) = \max\{R(0,1)+\gamma(P_{00}(1)\cdot V_1(0)+P_{01}(1)\cdot V_1(1)), R(0,2)+\gamma(P_{00}(2)\cdot V_1(0)+P_{01}(2)\cdot V_1(1))\}$$

$= \max\{1 + \frac{1}{2}(\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 0), 2 + \frac{1}{2}(\frac{1}{4} \cdot 2 + \frac{3}{4} \cdot 0)\} = \max\{\frac{3}{2}, \frac{9}{4}\} = \frac{9}{4}$

$$V_2(1) = \max\{R(1,1)+\gamma(P_{11}(1)\cdot V_1(1)+P_{10}(1)\cdot V_1(0)), R(1,2)+\gamma(P_{11}(2)\cdot V_1(1)+P_{10}(2)\cdot V_1(0))\}$$

$= \max\{0 + \frac{1}{2}(\frac{1}{3} \cdot 0 + \frac{2}{3} \cdot 2), 0 + \frac{1}{2}(\frac{2}{3} \cdot 0 + \frac{1}{3} \cdot 2)\} = \max\{\frac{2}{3}, \frac{1}{3}\} = \frac{2}{3}$

$\pi^2 = \{2, 1\}$

<u>k = 3</u>

$$V_3(0) = \max\{R(0,1)+\gamma(P_{00}(1)\cdot V_2(0)+P_{01}(1)\cdot V_2(1))), R(0,2)+\gamma(P_{00}(2)\cdot V_2(0)+P_{01}(2)\cdot V_2(1))\}$$

$= \max\{1 + \frac{1}{2}(\frac{1}{2} \cdot \frac{9}{4} + \frac{1}{2} \cdot \frac{2}{3}), 2 + \frac{1}{2}(\frac{1}{4} \cdot \frac{9}{4} + \frac{3}{4} \cdot \frac{2}{3})\} = \max\{\frac{83}{48}, \frac{81}{32}\} = \frac{81}{32}$

$$V_3(1) = \max\{R(1,1)+\gamma(P_{11}(1)\cdot V_2(1)+P_{10}(1)\cdot V_2(0))), R(1,2)+\gamma(P_{11}(2)\cdot V_2(1)+P_{10}(2)\cdot V_2(0))\}$$

$= \max\{0 + \frac{1}{2}(\frac{1}{3} \cdot \frac{2}{3} + \frac{2}{3} \cdot \frac{9}{4}), 0 + \frac{1}{2}(\frac{2}{3} \cdot \frac{2}{3} + \frac{1}{3} \cdot \frac{9}{4})\} = \max\{\frac{31}{36}, \frac{43}{72}\} = \frac{31}{36}$

$\pi^3 = \{2, 1\}$

Here, the final policy we get dictates that action 2 is taken in state 0 and action 1 is taken in state 1.

Solving even these simple MDP's can be quite tedious by hand. Thankfully, using Matlab's MDP toolbox to solve this problem can simplify the process. Here is code in Matlab to obtain an optimal policy for the above MDP.

```
clear all;

transition_matrix = [1/2 1/2 ; 2/3 1/3];
transition_matrix(:,:,2) = [1/4 3/4 ; 1/3 2/3];

Reward_matrix = [1 2 ; 0 0];

mdp_value_iteration(transition_matrix,Reward_matrix, .5,0.01 )
```

then we get output

```
    ans =


     2
     1
```

which we also obtained by hand.

3.5 Policy iteration

Policy iteration is an alternative to value iteration. This algorithm starts with an initial guess of π^0 and generates iteratively a sequence of improving policies. One iteration of policy iteration is two steps **1.** Policy Evaluation **2.** Policy Improvement.

Policy evaluation makes use of equation (2), our alternate characterization of the value function. Recall that for every state s under policy π, their value functions satisfy

$$V_\pi(s) = \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V_\pi(s')]$$

furthermore, the values $V_\pi(1), ..., V_\pi(n)$ are the unique solution of this equation. The idea of policy evaluation rests on solving a system of $|S|$ linear equations with $|S|$ unknowns. In matrix notation, given a stationary policy π^k, we compute the corresponding value functions from the linear system of equations

$$(I - \gamma P_{\pi^k})V_{\pi^k} = R_{\pi^k}$$

The i^{th} row of P_{π^k} is the the conditional probability distribution $P_{s,s'}(a) = P(S_{t+1} = s'|S_t = s, A_t = a)$ of transiting to state the next state s' from state $s = i$ under π^k. Similarly, the i^{th} row of R_{π^k} is the immediate reward gained from taking action a dictated by policy π^k in state $s = i$.

When V_{π^k} are obtained from solving the system of linear equations, a sweep of the action space is done to find actions that would perform better than the current policy. This is the policy improvement step. The policy improvement step is given by the equation

$$\pi^{k+1}(s) = \arg\max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V_{\pi^k}(s')] \tag{3}$$

for each s.

3.6 Policy Iteration Example

Suppose we have 3 states $\{A, B, C\}$ and two possible actions available at each state $\{a_1, a_2\}$. These are the probability transition matrices as functions of the action taken.

$$P_{ij}(a_1) = \begin{bmatrix} 0.4 & 0.2 & 0.4 \\ 0.5 & 0.2 & 0.3 \\ 0.1 & 0.2 & 0.7 \end{bmatrix} P_{ij}(a_2) = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.6 & 0.3 & 0.1 \\ 0.25 & 0.25 & 0.50 \end{bmatrix}$$

similarly let,

$$R_{ij}(a_1) = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \; R_{ij}(a_2) = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

Be the reward matrices. First we start with a guess of π^0. Let $\pi^0 = \{a_1, a_2, a_1\}$ and let discount $=$ 0.9

step 1

Create the system of equations and obtain the value functions under π^0:

$$P_{\pi^0} = \begin{bmatrix} 0.4 & 0.2 & 0.4 \\ 0.6 & 0.3 & 0.1 \\ 0.1 & 0.2 & 0.7 \end{bmatrix} \qquad R_{\pi^0} = \begin{bmatrix} 1 \\ 4 \\ 5 \end{bmatrix}$$

Then solve $(I - \gamma P_{\pi^0})V_{\pi^0} = R_{\pi^0}$ in Matlab

```
syms x y z
eqn1 = 0.64*x - 0.18*y - 0.36*z == 1
eqn2 = -0.54*x + 0.73*y - 0.09*z == 4
eqn3 = -0.09*x -0.18*y + 0.37*z  == 5
[A,B] = equationsToMatrix([eqn1, eqn2, eqn3], [x, y, z])

values= linsolve(A,B)
```

which gives output:

```
    values =

217450/6643
  32650/949
253850/6643
```

or $V(A) = 32.73$ $V(B) = 34.40$ $V(C) = 38.21$

step 2

Once V_{π^0} is computed, an improved policy can be obtained according to the policy improvement step:

$$\pi^1(s) = \arg\max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V_{\pi^0}(s')]$$

for all s. A Matlab implementation of the improvement step can be found in the Appendix. Applying this function to the starting policy

```
 clear all;
e = 0.01;
TPM = [0.4 0.2 0.4; 0.5 0.2 0.3; 0.1 ,0.2, 0.7 ];
TPM(:,:,2) = [0.1 0.4 0.5; 0.6 0.3 0.1; 0.25 0.25 0.50];
TRM = [1 2; 3 4 ; 5 6];
Pol = [1,2,1];
Values = [32.73 34.40 38.21];
[oldpolicy , policy] = policy_improvement(TPM,TRM,Pol,transpose(Values),0.9)
```

improves it to:

```
oldpolicy =

     1     2     1

policy =

     2     2     2
```

The result is the new policy $\{a_2, a_2, a_2\}$. Evaluating this policy in the same way as in step 1 yields

$V(A) = 39.5605$

$V(B) = 40.0983$

$V(C) = 43.4880$

If we use the policy improvement function again on $\{a_2, a_2, a_2\}$, the policy is unchanged, and it has converged.

3.7 Optimistic Policy Evaluation

While solving a system of $|S|$ linear equations with $|S|$ unknowns is possible, this computation is inefficient. Instead, iterative techniques are preferred. To do this, an initial guess for each state's value is made. The vector of value functions is then successively approximated with the update rule

$$V_{k+1}(s) = \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V_k(s')] \tag{4}$$

Let's see an example of an MDP and how one can evaluate a policy for it with optimistic PI. Suppose we have 3 states $\{A, B, C\}$ and two possible actions available at each state $\{a_1, a_2\}$. These are the probability transition matrices depending on the action taken.

$$P_{ij}(a_1) = \begin{bmatrix} 0.4 & 0.2 & 0.4 \\ 0.5 & 0.2 & 0.3 \\ 0.1 & 0.2 & 0.7 \end{bmatrix} P_{ij}(a_2) = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.6 & 0.3 & 0.1 \\ 0.25 & 0.25 & 0.50 \end{bmatrix}$$

similarly let,

$$R_{ij}(a_1) = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} R_{ij}(a_2) = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

be the reward matrices.

Evaluate the policy $\pi^0 = \{a_1, a_2, a_1\}$ with discount $\gamma = 0.9$. Each policy has its unique transition probability matrix and reward matrix. For π^0, these are

$$P_{ij}(\pi^0) = \begin{bmatrix} 0.4 & 0.2 & 0.4 \\ 0.6 & 0.3 & 0.1 \\ 0.1 & 0.2 & 0.7 \end{bmatrix} R_{ij}(\pi^0) = \begin{bmatrix} 1 \\ 4 \\ 5 \end{bmatrix}$$

The update rule (4) in this instance is

$$V_{k+1} = 0.9 \begin{bmatrix} 0.4 & 0.2 & 0.4 \\ 0.6 & 0.3 & 0.1 \\ 0.1 & 0.2 & 0.7 \end{bmatrix} V_k + \begin{bmatrix} 1 \\ 4 \\ 5 \end{bmatrix}$$

The idea is to start with an initial guess for the first value vector and apply the update until $|V_k - V_{k+1}| < \epsilon$ for some small $\epsilon > 0$. The values for this scenario are:

$V(A) = 32.6869$

$V(B) = 34.3579$

$V(C) = 38.1664$

with 63 total iterations

$\epsilon = 0.01$

$|V_k - V_{k+1}| = 0.0090$

which is very close to what we got with regular policy iteration. Code provided at the end. Note that if only one iteration of optimistic policy evaluation is applied followed by a policy improvement step, this process is equivalent to value iteration [Ber18] (pg. 104).

3.8 Convergence analysis of Value Iteration

This section illustrates the role of contraction mappings in the convergence of value iteration. As seen before, value iteration is useful for approximating optimal deterministic policies. However, what guarantees can it provide in terms of convergence? It turns out that with bounded and discounted reward structure, VI will convergence to optimal values independent of the initial guess V_0 due to its **contractive property**. The value iteration algorithm is simply the update rule:

$$V_{k+1}(s) = max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V_k(s')] \tag{5}$$

Some literature will call this mapping the **Bellman Optimality Operator**.

Definition 3.3 (Bellman Optimality Operator)**.** *The Bellman Optimality Operator is a mapping $\mathcal{T}$ such that*

$$(\mathcal{T}V)(s) = max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V_\pi(s')]$$

Value iteration can then be written in operator notation as,

$$V_{k+1} = \mathcal{T}V_k$$

To build up to the convergence of value iteration, we require one more definition and two theorems

Definition 3.4 (Contraction Mapping). *Given a real vector space Y with a norm $||\cdot||$, a function $F\colon Y \mapsto Y$ is said to be a* contraction mapping *if for some $\rho \in (0,1)$, we have*

$$||Fy - Fz|| \leq \rho||y - z||, \qquad \forall y, z \in Y$$

For the next theorem, define $B(X)$ to be the set of all value functions $V : X \to \mathbb{R}$ bounded over the domain of the state space X.

Theorem 3.1 (Contraction Mapping Fixed-Point Theorem [Ber18] (pg.48)). *if $\mathcal{T} : B(X) \to B(X)$ is a contraction mapping with modulus $\gamma \in (0,1)$, then there exists a unique $V^* \in B(X)$ such that*

$$V^* = \mathcal{T}V^*$$

.

Finally we must prove the following theorem

Theorem 3.2 ([Gos08](pg. 360)). *$\mathcal{T}$ is a contraction mapping under sup norm $||\cdot||_\infty$. ie. there exists $\rho \in (0,1)$ such that*

$$||\mathcal{T}U - \mathcal{T}V|| \leq \rho||U - V||$$

Proof. Let the state space $\mathcal{S} = \mathcal{S}_1 \bigcup \mathcal{S}_2$ be defined as follows.
case 1 Assume that for all $s \in \mathcal{S}_1$ $(\mathcal{T}V)(s) \geq (\mathcal{T}U)(s)$
case 2 Assume that for all $s \in \mathcal{S}_2$ $(\mathcal{T}U)(s) \geq (\mathcal{T}V)(s)$

Case 1 Given value functions U, V, assume $\mathcal{T}V(s) \geq \mathcal{T}U(s)$ $\forall s \in S$. If a, b are actions that satisfy:

$$(\mathcal{T}V)(s) = max_a \sum_{s'} p(s'|s,a)[r_a + \gamma V_\pi(s')]$$

$$(\mathcal{T}U)(s) = max_b \sum_{s'} p(s'|s,b)[r_b + \gamma U_\pi(s')]$$

we have that

$$\mathcal{T}U(s) \geq \sum_{s'} p(s'|s,a)[r_a + \gamma U_\pi(s')]$$

since a does not satisfy the Bellman operator for $\mathcal{T}U$. Using all of the information provided:

$$\begin{aligned}
0 &\leq \mathcal{T}V(s) - \mathcal{T}U(s) \\
&\leq \sum_{s'} p(s'|s,a)[r_a + \gamma V_\pi(s')] - \sum_{s'} p(s'|s,a)[r_a + \gamma U_\pi(s')] \\
&\leq \gamma \sum_{s'} p(s'|s,a)[V_\pi(s') - U_\pi(s')] \\
&\leq \gamma \sum_{s'} p(s'|s,a) \max_{s'} |V_\pi(s') - U_\pi(s')| \\
&\leq \gamma \max_{s'} |V_\pi(s') - U_\pi(s')| \\
&\leq \gamma ||V - U||
\end{aligned}$$

thus for all $s \in \mathcal{S}_1$ $\mathcal{T}V(s) - \mathcal{T}U(s) \leq \gamma ||V - U||$

Case 2 similarly for all $s \in \mathcal{S}_2$ $\mathcal{T}U(s) - \mathcal{T}V(s) \leq \gamma ||V - U||$.

since the LHS for both inequalities is positive, it follows that $|\mathcal{T}V(s) - \mathcal{T}U(s)| \leq \gamma ||V - U||$ for any choice of s, including s that maximizes $|\mathcal{T}V(s) - \mathcal{T}U(s)|$. Therefore, $\max_s |\mathcal{T}V(s) - \mathcal{T}U(s)| \leq \gamma ||V - U||$ or $||\mathcal{T}V(s) - \mathcal{T}U(s)|| \leq \gamma ||V - U||$

□

We can now prove the convergence of value iteration

Theorem 3.3 ([Gan])**.** *Value iteration converges to V^* ie.*

$$\lim_{k \to \infty} \mathcal{T}^k V_0 = V^*$$

Proof. Note that V^* is a fixed point of $\mathcal{T}$ and $\mathcal{T}$ is a contraction mapping. So we have

$||\mathcal{T}^k V_0 - V^*|| = ||V_k - V^*|| = ||\mathcal{T}V_{k-1} - \mathcal{T}V^*|| \leq \gamma ||V_{k-1} - V^*|| \leq ... \leq \gamma^k ||V_0 - V^*||$

Letting $k \to \infty$ we have $\gamma^k ||V_k - V^*||_\infty \to 0$. Therefore,

$$\lim_{k \to \infty} \mathcal{T}^k V_0 = V^*$$

□

4 Model Free Reinforcement Learning

4.1 Introduction

The previous section dealt with solving Markov Decision Problems where the probability transitions and reward structures were known. This is called having a model of the environment. When the model is unknown, optimal policies cannot be obtained in a value or policy iteration fashion. When we want to find optimal actions without knowing the environment's model, **Q-Learning** algorithms can help explore the value of taking action a in state s. An environment we can apply Q-learning to is the grid-world example:

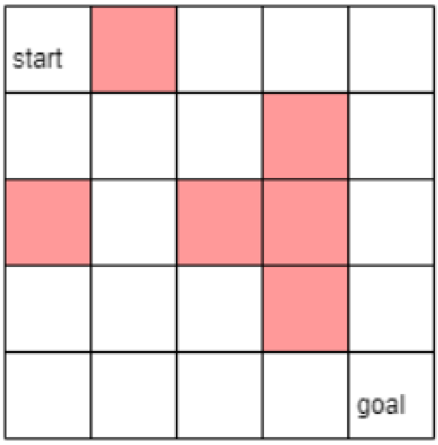


Figure 3: Grid World environment for an agent to learn optimal paths to the goal

The goal is to train an agent to learn the optimal path from the top left corner of this grid to the bottom right. It is tasked with finding the most optimal sequence of actions from the set $\mathcal{A} = \{left, right, up, down\}$ while avoiding pits displayed as red boxes. There is no inherent numerical reward tied to being in one state versus another, only desired (non-pit) and non-desired (pit) states. Additionally, there is no sense in having transition probabilities in this context since the environment does not evolve randomly on its own. We will see how implementations of Q-learning solve these issues an learns the value of state action pairs using the **Q-factor**.

4.2 The Q-factor

A Q-factor is similar to the value function $V(S)$ but instead provides the value of a state-action pair (s, a). The number of Q-factors is just the number of state action pairs there are. If there are three states and two actions, our Q-factors are

$$Q = \{Q(1,1), Q(1,2), Q(2,1), Q(1,2), Q(3,1), Q(3,2)\}$$

We define the Q-factor differently than the value function given earlier. The **Q-factor**, $Q(s, a)$ is defined in [Gos14] as,

$$Q(s,a) = \sum_{s'} p(s'|s,a)[r(s,a,s') + \max_{b \in A(s')} \gamma Q(s',b)] \tag{6}$$

recall the **bellman optimality equation** for the value function presented in section 3.3.

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V^*(s')]$$

It is important to note that

$$V^*(s) = \max_{a \in A(s)} Q(s,a) = Q^*(s,a)$$

so we have that the bellman optimality equation for the Q-factor is

$$Q^*(s,a) = \sum_{s'} \max_{b \in A(s')} p(s'|s,a)[r(s,a,s') + \gamma Q^*(s',b)]$$

[Gos14] (pg. 205)

As with the value function, we can turn the bellman optimality equation into an mapping for the value iteration method.

4.3 Q-factor value iteration

Before we tackle the grid world problem, we start with a simpler and more familiar model: the MDP. Just like value iteration with value functions, we can also perform Q-factor value iteration [Gos14] (pg. 206). Note that we are still in the model-based domain of dynamic programming. Here is the Q-factor value iteration algorithm. It is assumed that reward depends also on the successor state s'.

Step 1 Set k=1. Specify $\epsilon > 0$, and set for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ $Q^0(s,a) = 0$

Step 2 For each $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, compute

$$Q^{k+1}(s,a) \leftarrow \sum_{j=1}^{|\mathcal{S}|} p(s,a,s') \left[r(s,a,s') + \gamma \max_{b \in A(j)} Q^k(s',b) \right]$$

Step 3 Calculate for each $s \in \mathcal{S}$

$$V^{k+1}(s) = \max_{a \in \mathcal{A}(s)} Q^{k+1}(s,a) \quad and \quad V^k(s) = \max_{a \in \mathcal{A}(s)} Q^k(s,a)$$

Then if $||(\vec{V}^{k+1} - \vec{V}^k)|| < \epsilon$, go to step 4. Otherwise increase k by 1, and return to step 2.

Step 4 For each $s \in \mathcal{S}$, choose $\pi(s) \in \arg\max_{b \in \mathcal{A}(s)} Q(s, b)$, where $\vec{\pi}$ denotes the ϵ-optimal policy, and stop.

We give an example. We have three states and three actions with the following transition probabilities and reward structures.

$$P_{ij}(a_1) = \begin{bmatrix} 0.6 & 0.3 & 0.1 \\ 0.3 & 0.3 & 0.3 \\ 0 & 0 & 0 \end{bmatrix} P_{ij}(a_2) = \begin{bmatrix} 0.5 & 0.5 & 0.0 \\ 0.5 & 0.1 & 0.4 \\ 0.8 & 0.1 & 0.1 \end{bmatrix} P_{ij}(a_3) = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ .8 & 0.1 & 0.1 \end{bmatrix}$$

$$R_{ij}(a_1) = \begin{bmatrix} 1 & 9 & 9 \\ 11 & 2 & 7 \\ 1 & 2 & 3 \end{bmatrix} R_{ij}(a_2) = \begin{bmatrix} 8 & 5 & 7 \\ 3 & 6 & 1 \\ 1 & 1 & 1 \end{bmatrix} R_{ij}(a_3) = \begin{bmatrix} 9 & 8 & 4 \\ 7 & 20 & 1 \\ 1 & 9 & 5 \end{bmatrix}$$

Let our discount be $\gamma = 0.8$.

Performing Q-factor value iteration gives:

	action	1	2	3
state	-	-	-	-
1	-	30.4705	33.2115	0
2	-	28.8459	27.5618	33.5686
3	-	0	27.2133	28.4133

To obtain the optimal policy, take the row that maximizes each column. Here we have, $\pi^* = \{2, 3, 3\}$. Check this against values obtained by policy iteration.

```
Value_Iteration = mdp_value_iteration(TPM,TRM, 0.8 ,0.01,1000) %value iteration
[policy_iteration_value,policy_iteration] = mdp_policy_iteration(TPM,TRM,0.8) %PI
```

```
policy_iteration_value =
    33.2143
    33.5714
    28.4161
policy_iteration =
     2
     3
     3
```

We can also see how well Q-factor VI performs by seeing how quickly $||(\vec{V}^{k+1} - \vec{V}^{k})||$ decreases and how close the values of Q-factor VI's optimal policy converges to PI.

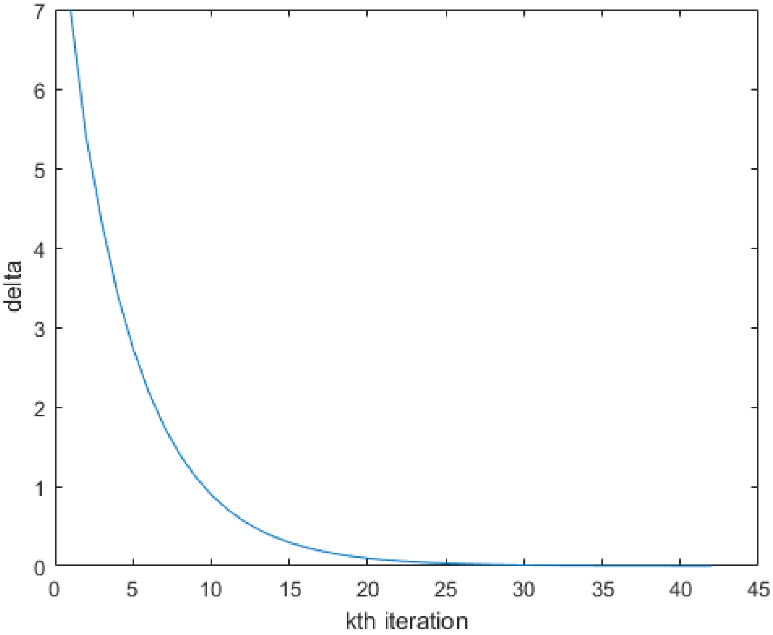


Figure 4: maximum distance between $\vec{Q^k}$ and $\vec{Q^{k+1}}$

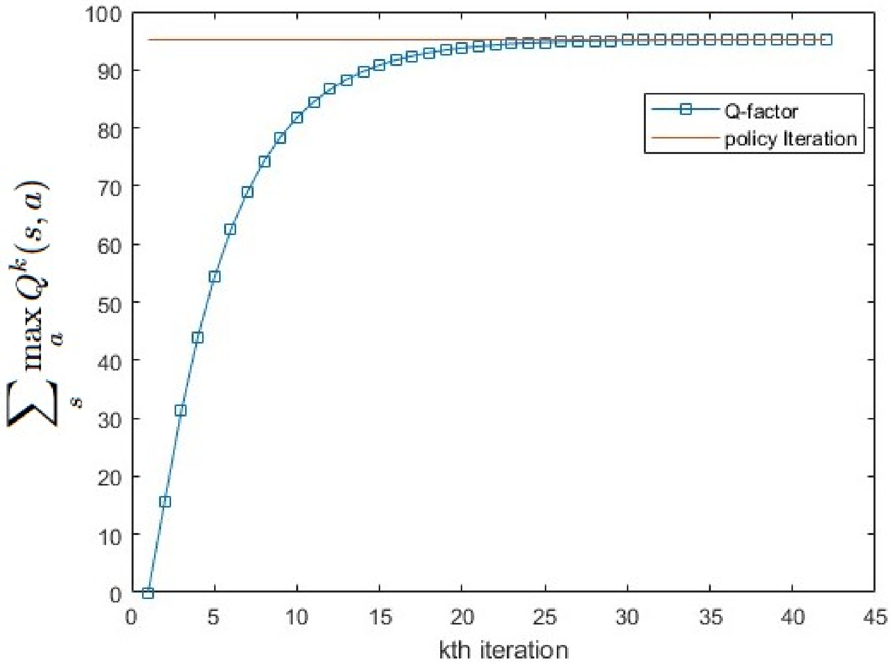


Figure 5: $\sum_s \max_a Q^k(s,a)$ over time k plotted against $\sum_s V^*(s)$ generated by policy iteration

4.4 Robbins-Monro Algorithm

The Robbins-Monro Algorithm as presented in [Gos14] (pg. 207) is our first step to model-free dynamic programming. It is a way to estimate a mean of a random variable from its samples. We will derive the Robbins-Monro algorithm from the simple averaging process and the Law of Large Numbers that for a random variable X

$$E(X) = \lim_{k \to \infty} \frac{\sum_{i=1}^{k} x_i}{k}$$

with probability 1. Let us denote X having k samples of x_i as X^k. So

$$X^k = \frac{\sum_{i=1}^{k} x_i}{k}$$

Now,

$$\begin{aligned}
X^{k+1} &= \frac{\sum_{i=1}^{k+1} x_i}{k+1} \\
&= \frac{\sum_{i=1}^{k} x_i + x_{k+1}}{k+1} \\
&= \frac{X^k k + x_{k+1}}{k+1} \\
&= \frac{X^k k + X^k - X^k + x_{k+1}}{k+1} \\
&= \frac{X^k (k+1) - X^k + x_{k+1}}{k+1} \\
&= \frac{X^k (k+1)}{k+1} - \frac{X^k}{k+1} + \frac{x_{k+1}}{k+1} \\
&= X^k - \frac{X^k}{k+1} + \frac{x_{k+1}}{k+1} \\
&= (1 - \alpha^{k+1}) X^k + \alpha^{k+1} x_{k+1} && (\text{if } \alpha^{k+1} = \tfrac{1}{k+1})
\end{aligned}$$

When $\alpha^{k+1} = \frac{1}{k+1}$, the Robbins-Monro algorithm is equivalent to simple averaging. However, we can use other values for α with the restriction that $0 < \alpha < 1$. α is called *step size* or *learning rate* of the algorithm.

4.5 Robbins-Monro, Q-factors, and Q-learning

To see the role of the Robbins-Monro algorithm in model-free dynamic programming, first remember that the Q-factor is an expectation of a random variable. The random variable is the value of the pair (s,a) when the MDP randomly transitions to a particular state s'. From a **simulation** perspective, we can get samples of the successor state s' using its distribution and estimate the expected value of Q instead of using value iteration. In this sense, the $k+1^{th}$ Q-factor value can be seen as the estimate with $k+1$ samples of the expected value of the pair (s,a). We can now embed the Q-factor samples into the Robbins-Monro algorithm like this [Gos14] (pg.209):

$$Q^{k+1}(s,a) \leftarrow (1-\alpha^{k+1})Q^k(s,a) + \alpha^{k+1}\left[r(s,a,s') + \gamma \max_{b \in A(s')} Q^k(s',b)\right] \tag{7}$$

This method of computing Q-factors does not use any sort of probability distribution! The underlying assumption is that the Q-factor can be approximated by sampling and simulation. In particular, an infinitely long sequence of state-control pairs $\{s_t, a_t\}$ is generated according to some probabilistic mechanism. Given the pair (s_t, a_t), a state s' is sampled according to the distribution $p_{s_t s'}(a_t)$. Then the Q-factor of (s_t, a_t) is updated using equation (7) with a step size $\alpha \in (0,1]$, while all other Q-factors are left unchanged [Ber18](pg.496). What we have derived is called the **Q-learning algorithm**. It is important to note that from the works of Dayan and Watkins [WD92] that there are specific requirements for α so that Q-learning converges. The step size α should satisfy

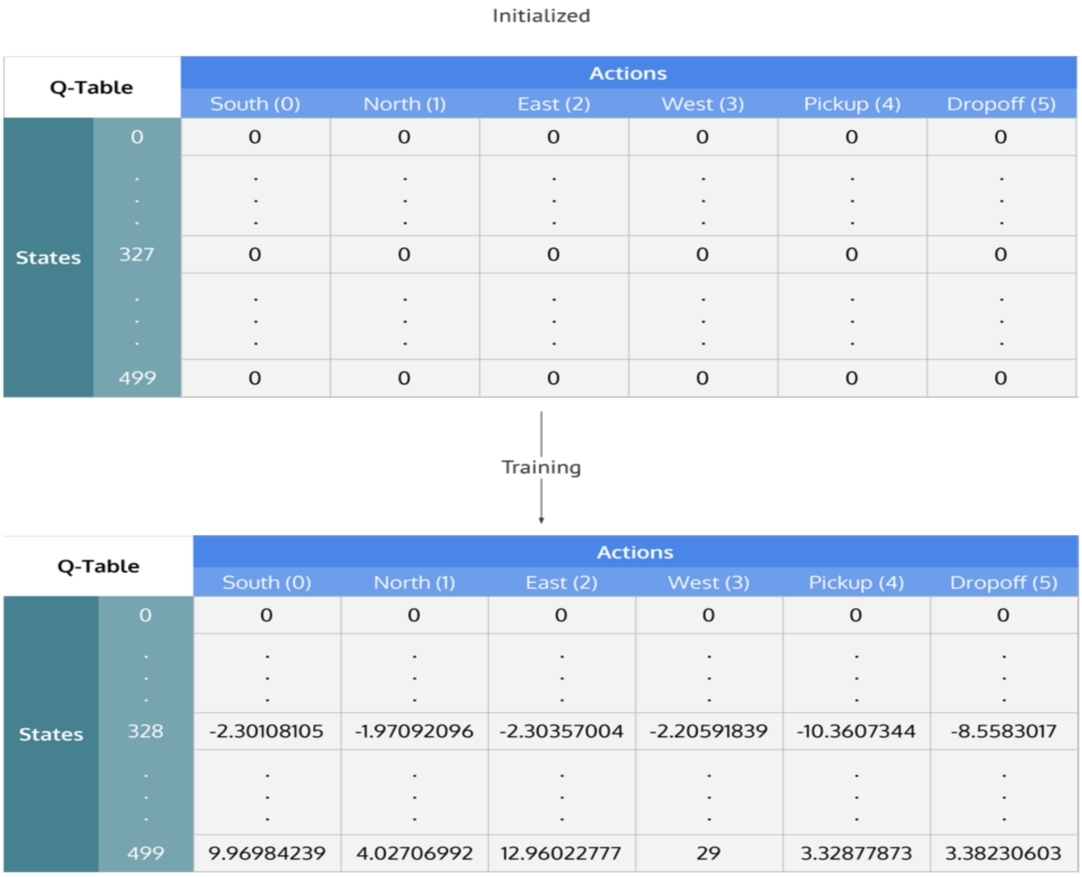
$$\alpha_k \geq 0, \forall k, \qquad \sum_{k=0}^{\infty} \alpha_k = \infty \qquad \sum_{k=0}^{\infty} \alpha_k^2 \leq \infty \qquad \alpha_k \to 0$$

for Q-learning to converge. Here is the pseudocode of the Q-learning algorithm applied to the MDP problem

```
Algorithm 1: MDP_Qlearning
Input: Transition Matrix, Reward Matrix, discount $\gamma$, Max_iter
Output: Qtable
$k \leftarrow 1$;
$S_t \leftarrow s_0$;
$Qtable \leftarrow$ initializeQtable;
while $k \leq Max\_iter$ do
    $A_t \leftarrow a_t \;\; w.p \;\; \frac{1}{\mathcal{A}(S_t)}$;
    $S_{t+1} \leftarrow s' \;\; w.p \;\; P_{s,s'} = P\{S_{t+1} = s' | S_t = s, A_t = a_t\}$;
    $R_{t+1} \leftarrow R(S_t, A_t, S_{t+1})$;
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_k \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$;
    $S_t \leftarrow S_{t+1}$;
    $k \leftarrow k + 1$;
end
```

4.6 MDP Q-learning

In this project's implementation of the Q-learning algorithm, Q-factors are stored in a look-up table called the Q-table. It is a two dimensional array where each column is an action, and each row is a state. We initialize each cell to be zero. Through the training process, each cell gets updated one at a time using (7).



Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 6: The initialized Q-table and post-trained table (taken from Wikipedia)

Imagine that the decision making agent is at its starting position s_0. The Q-learning algorithm starts with sampling an action $a_0 \in A(s_0)$. Then we simulate the next state s' using the conditional distribution $P_{ss'}(a_0) = P\{S_{t+1} = s' | S_t = s_o, a_t = a_0\}$ specified by the MDP transition probabilities. The cell that represents (s_0, a_0) is updated using equation (7) and the agent moves to s'. The Q-learning algorithm continues like this for a large maximum number of iterations. We apply the MDP Q-learning algorithm to the example in section 4.3.

recall that we obtained these values and $\pi^* = \{2, 3, 3\}$ by applying Q-factor value iteration

Q-factor VI	action	1	2	3
state	-	-	-	-
1	-	30.4705	33.2115	0
2	-	28.8459	27.5618	33.5686
3	-	0	27.2133	28.4133

The step size plays an important role in estimating Q-factors with the Q-learning algorithm. $\alpha = 1/k$ is the simplest of step sizes that satisfies the basic requirements specified in the section 4.5. We test three different step sizes found in [Gos08] with a maximum number of transitions = 10000.

Using $\alpha = 1/k$ rule we get

$1/k$	action	1	2	3
state	-	-	-	-
1	-	9.8227	12.3790	0
2	-	6.7516	4.7905	8.4003
3	-	0	2.6362	3.7817

The values here aren't very accurate but still produce the correct policy. The issue is that while $1/k$ satisfies the requirements for Q-learning step sizes, it decays too quickly to zero for each Q-factor to be updated many times. An alternative to $1/k$ is $A/(B + k)$, where A and B are **tuneable hyperparameters**. the AB step size rule performs well only if the scalar hyperparameters are well chosen. In the works of [Gos14] selecting $A = 150$ and $B = 300$ consistently performs well on discounted reward MDP problems. Using the $\alpha = (150/(300 + k))$ rule we get

$A/(B+k)$	action	1	2	3
state	-	-	-	-
1	-	29.9420	33.0152	0
2	-	31.7369	27.6329	33.4186
3	-	0	27.2275	28.3947

A much better approximation to Q-factor VI.

The last step size that we simulate is the $\alpha = \log(k+1)/k$ rule. The advantage of this rule is that there are no hyperparameters to select while also not decaying too quickly to zero. Using this rule we get

$log(k+1)/k$	action	1	2	3
state	-	-	-	-
1	-	24.7730	27.4903	0
2	-	23.4185	18.8718	26.6003
3	-	0	17.2918	18.7735

Clearly not as good of an approximation as the AB rule. However, if we increase the number of transitions allowed,

$log(k+1)/k$	action	1	2	3
state	-	-	-	-
1	-	29.4423	32.2393	0
2	-	30.4115	25.8818	32.3820
3	-	0	25.2219	26.4356

$log(k+1)/k$ gets closer to the AB rule values. Here is a graph of the sum of the optimal policy's value after k steps of the algorithm.

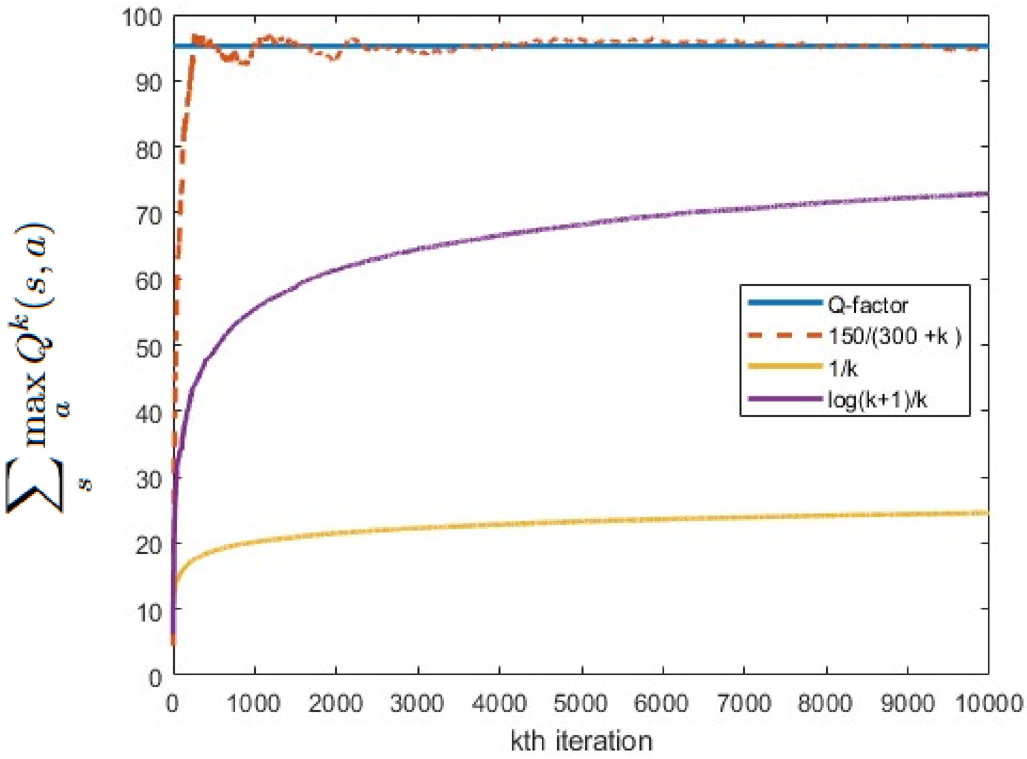


Figure 7: performances of different step sizes

We can see that the $150/(300+k)$ slightly oscillates around the true value of the optimal policy, $log(k+1)/k$ approaches very slowly, and $1/k$ even slower. While $1/k$ might decay too quickly to zero to be a practical step size, $log(k+1)/k$ might perform better if we drastically increase the number of state transitions. Letting the maximum number of transitions = 1000000.

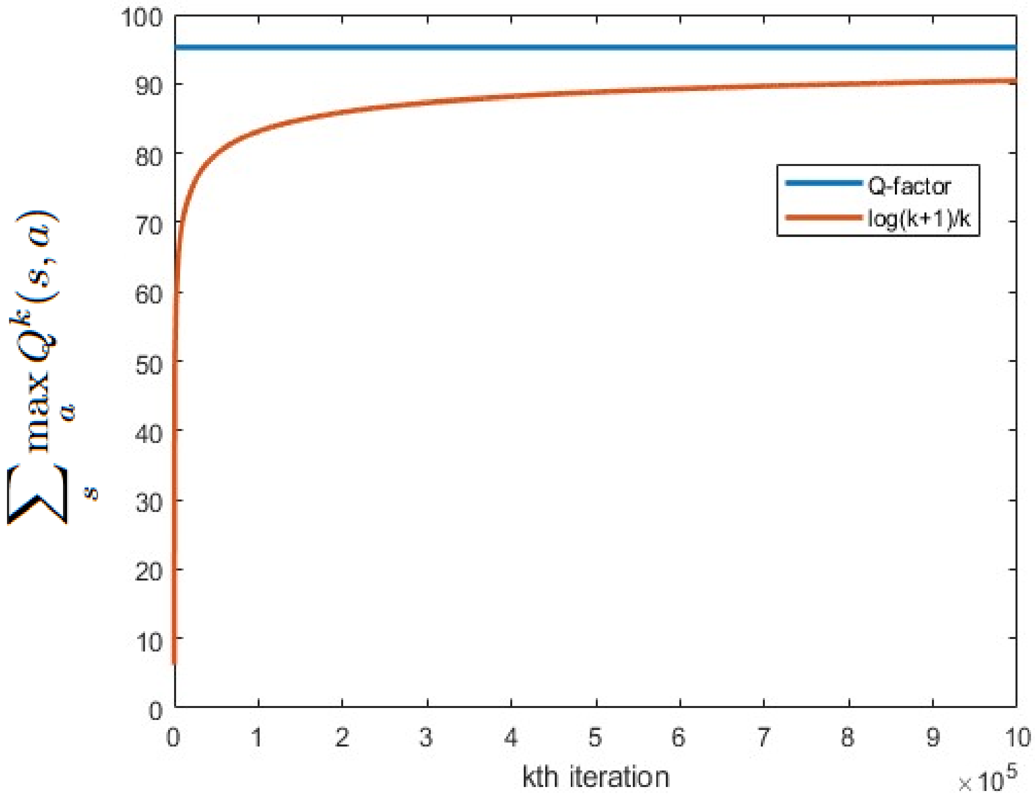
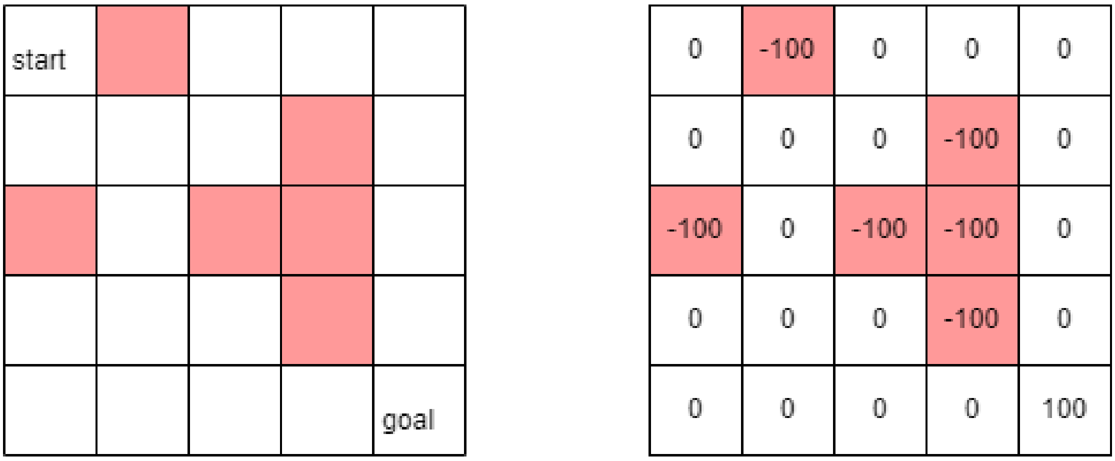


Figure 8: performance of $log(k+1)/k$ with 1 million transitions

Although the $log(k+1)/k$ approaches the true values of the optimal policy in the limit, the number of iterations it requires to reach a good approximation makes it impractical to use.

4.7 Grid World Q-learning

Here we tackle the problem laid out in the introduction. This problem is a modified version of the cliff walking problem found in [SB20] (pg. 132). Recall the environment described in section 4.1. The problem was to find an optimal sequence of actions from $\mathcal{A} = \{left, right, up, down\}$ to get from the starting point to the goal in a grid world environment. The issues with this problem were that the reward structures were completely unknown and that a probability transition model did not exist. The first can be solved by mapping qualitative values to numeric rewards.



start	🟥			
			🟥	
🟥		🟥	🟥	
			🟥	
				goal

0	-100	0	0	0
0	0	0	-100	0
-100	0	-100	-100	0
0	0	0	-100	0
0	0	0	0	100

Figure 9: Grid World environment for an agent to learn optimal paths to the goal

Here, the reward of falling in a pit is -100, reaching the goal is 100 and landing on a safe state that is not the goal is 0.

Unlike the MDP problem, we cannot access a probability transition model to simulate the next state. Here we rely on a randomized policy called the **ϵ-greedy** policy to choose an action (and state since the state-action pair determines the next state with probability 1). The ϵ-greedy policy specifies a parameter ϵ so that with probability $1 - \epsilon$ the agent while in state s, chooses a with the greatest Q-factor, and with probability ϵ an action is chosen randomly with probability $\frac{1}{|\mathcal{A}|}$.

When the agent falls into a pit, we assume that it can't take any actions and the process has ended. So here we have a finite horizon problem. In reinforcement learning, one sample path that ends in a terminal state is called an episode. At the end of the episode, we return agent back to the starting position, keep the Q-factors learned from the last episode and continue training the agent.

Here is how the Q-learning algorithm would be applied to the Grid World problem.

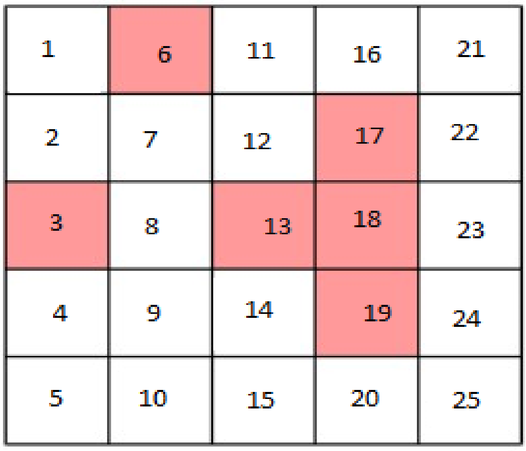
```
Algorithm 2: GridWorld_Qlearning
Input: Transition Matrix, Reward Matrix, discount $\gamma$, Max_iter
Output: Qtable
$k \leftarrow 1$;
$S_t \leftarrow S_0$;
$Qtable \leftarrow$ initializeQtable;
while $k \leq Max\_iter$ do
    sample $A_t = \begin{cases} A_t = \arg\max_{A_t} Q(S_t, A_t) & w.p \;\; 1-\epsilon \\ randomly \;\; a_t \in \mathcal{A}(S_t) & w.p \;\; \epsilon \end{cases}$
    $S_{t+1} \leftarrow$ getNextstate;
    $R_{t+1} \leftarrow$ getReward;
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_k \Big[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$;
    $S_t \leftarrow S_{t+1}$;
    $k \leftarrow k + 1$;
end
```

First we initialize the environment and Qtable

```
Gridworld = [0 -100 0 0 0;0 0 0 -100 0; -100 0 -100 -100 0;0 0 0 -100 0;0 0 0 0 100];
Env =


     0  -100     0     0     0
     0     0     0  -100     0
  -100     0  -100  -100     0
     0     0     0  -100     0
     0     0     0     0   100


terminals =


     0     1     0     0     0
     0     0     0     1     0
     1     0     1     1     0
     0     0     0     1     0
     0     0     0     0     1




Q =


     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

Here the columns are organized into the actions $\{up, down, left, right\}$

The Qtable initializes all Q-factors to zero and indexes each state from 1 to 25 along its rows according to this scheme:



1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

Figure 10: grid indexing scheme

running `Qlearning` with $\epsilon = 0.9$ and 500000 episodes returns

```
 1          0   47.8297         0 -100.0000
 2    43.0467 -100.0000         0   53.1441
 3          0         0         0         0
 4  -100.0000   65.6100         0   65.6100
 5    59.0490         0         0   72.9000
 6          0         0         0         0
 7  -100.0000   59.0490   47.8297   47.8297
 8    53.1441   65.6100 -100.0000 -100.0000
 9    59.0490   72.9000   59.0490   72.9000
10    65.6100         0   65.6100   81.0000
11          0   47.8297 -100.0000   59.0475
12    53.1427 -100.0000   53.1441 -100.0000
13          0         0         0         0
14  -100.0000   81.0000   65.6100 -100.0000
15    72.9000         0   72.9000   90.0000
16          0 -100.0000   53.1426   65.6083
17          0         0         0         0
18          0         0         0         0
19          0         0         0         0
20  -100.0000         0   81.0000  100.0000
21          0   72.8982   59.0474         0
22    65.6079   80.9983 -100.0000         0
23    72.8971   89.9985 -100.0000         0
24    80.9663   99.9989  -99.9884         0
25          0         0         0         0
```

The policy given by this real valued Qtable maps each state to the action with the greatest Q-factor. For example, in state 7 the best action to take would be *down*. To check that we have an *optimal policy*, we should get an optimal path by following each action starting from S_0. Indeed we do!

down	terminal	right	right	down
right	down	left	terminal	down
terminal	down	terminal	terminal	down
down/right	down\right	down	terminal	down
right	right	right	right	goal

More visually,

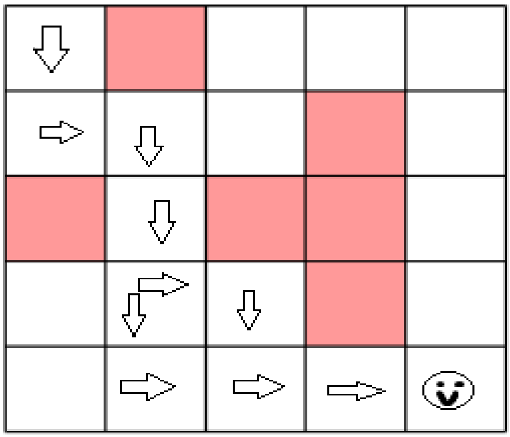


Figure 11: optimal pathing from start to finish

It is interesting to note that the optimal path is more valuable than the path that goes right in state 7. This is because there are more discounts applied in that path due to it being longer. Also interesting to note is that in state 9, there is no difference in taking *right* or *down*. It can be verified in the post-trained Q-table that the values for these actions are the same and maximal for row 9.

4.8 Exploration vs Exploitation

The previous section had Q-learning choose random actions at a high rate of ϵ. The intuition behind choosing random actions often, is to explore and update every state-action pair many times. Exploration allows that the best actions are found in each state and overall yields better estimates for all Q-factors. However, it leads to sub-optimal actions being explored very often (even when the optimal policy is known). As a result, we may converge to good estimates very slowly with a high rate of exploration. We can perhaps fine tune the exploration rate if we wish to train on lower episode numbers but this may result in forever exploiting a sub-optimal policy. To illustrate this, we can run the `Qlearning` function with exploration rates of 0.9, 0.25 and 0.6

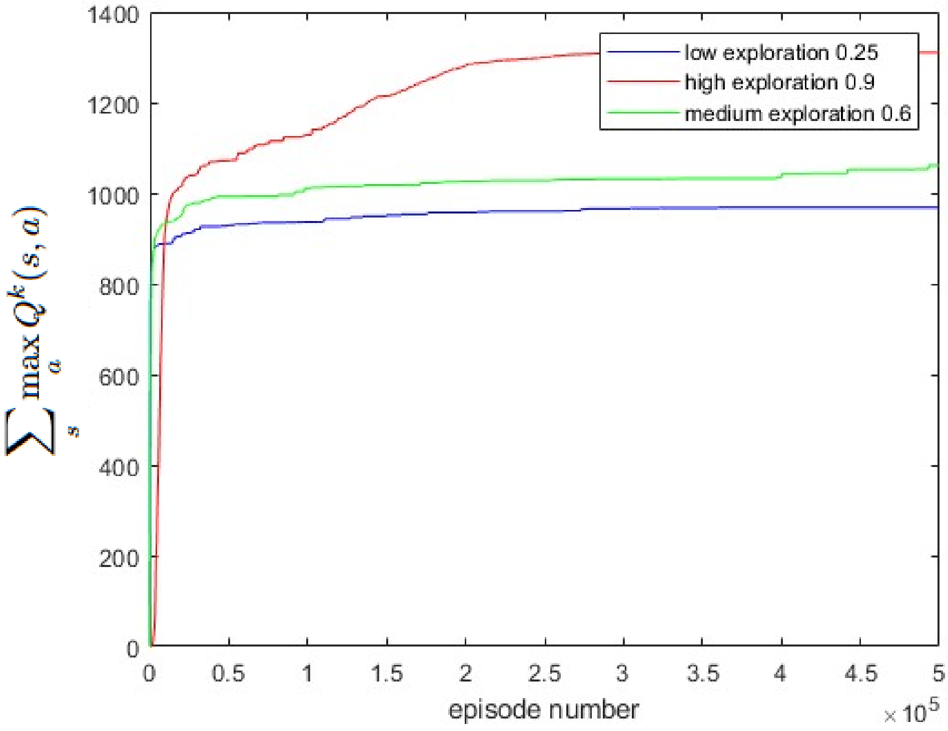


Figure 12: the effect of exploration rates on reward over time

Here we see that agents with medium and low rates of exploration find actions with high numeric value quicker than the agent with high exploration. However in the long run, lower rates of exploration result in exploiting sub-optimal policies and high rates of exploration continues to improve its policy.

5 Summary

Hopefully this project introduced model free dynamic programming from the ground up. In the first section we covered the basic problem of finding optimal actions in the stochastic finite horizon using the dynamic programming algorithm. Next, we saw value iteration, policy iteration and optimistic policy iteration to solve Markov decision processes over an infinite horizon. As well, the role of contraction mappings in the convergence of value iteration. Finally, we built upon these two chapters, ideas of model free dynamic programming where we chose to sample the expected value of a state-action pair free from using a probability model. We showed Q-factor VI, the Robbins-Monro Algorithm, Q-learning derived from it and a Grid World application.

The power of model free dynamic programming was shown when a probability distribution did not exist and we were able to use a randomized policy to solve the grid world problem.

6 Appendix

Optimistic Policy Evaluation and policy improvement

```
    function [TRM, TPM,V,policy,iter,delta] = policy_evaluation(P,R,e,policy,discount)
    iter = 0;
    delta = e + 0.01;
    V = zeros(size(P,1),1);   %initialize value vector
    n = size(P,1); %number of states in the problem
    TPM = zeros(n,n); %initialize the policy TPM to be filled according to the current policy
    TRM = zeros(size(R,1),1); %same with the policy
    for i= 1:n
    TPM(i,:) = P(i,:,policy(i)); %making the policy transition probability matrix
    TRM(i,1) = R(i,policy(i));
    end

    %this is the evaluation part

    while delta > e

        old_V = V;
        V = TRM + TPM*(discount*V);

        delta = max(0,norm(old_V - V));
        iter = iter + 1;

    end



end
function[old_policy, policy] = policy_improvement(P,R,policy,value_vector,discount)
old_policy = policy;
for i = 1:size(value_vector,1)
    for j = 1:size(R,2)
        actionj_value = R(i,j) + P(i,:,j)*(discount*value_vector);
        if actionj_value > value_vector(i)
            policy(i) = j;
        end
    end
end
end
```

Q-factor VI and MDP Qlearning

```
    %My own custom MDP
%rows in the probability tranistion matrix with only zeros denote
%that action in the state represented by that row is unavailable

TRM = [1 9 9; 11 2 7; 1 2 3];
TRM(:,:,2) = [8 5 7; 3 6 1; 1 1 1];
TRM(:,:,3) = [9 8 4; 7 20 1; 1 9 5];

TPM = [.6 .3 .1 ; .3 .3 .3; 0 0 0 ];
TPM(:,:,2) = [.5 .5  0 ; 0.5 .1 .4 ; .8 .1 .1];
TPM(:,:,3) = [0 0 0; 1 0 0; .8 .1 .1];

MATLAB_VI2 = mdp_value_iteration(TPM,TRM, 0.8 ,0.01,1000)
[policy_value,policy] = mdp_policy_iteration(TPM,TRM,0.8)
[Qf ,delta,rewards] = Qfactor_VI(TPM,TRM,0.8,0.01);
transpose(Qf)
[MDPQalpha1,alpha_type1,type1_rewards] = MDP_Qlearning(TPM,TRM,0.8,10000,1,150,300);
[MDPQalpha2,alpha_type2,type2_rewards] = MDP_Qlearning(TPM,TRM,0.8,10000,2,0,0);
[MDPQalpha3,alpha_type3,type3_rewards] = MDP_Qlearning(TPM,TRM,0.8,10000,3,0,0);

base_policy(:,1:size(type3_rewards,2)) = sum(max(Qf(:,:)));
plot(base_policy, ’-’,’LineWidth’,2)
hold on;
plot(type1_rewards, ’--’,’LineWidth’,2)
hold on;
plot(type2_rewards, ’-’,’LineWidth’,2)
hold on;
plot(type3_rewards, ’-’,’LineWidth’,2)
legend(’Q-factor’, ’150/(300 +k )’,’1/k’,’log(k+1)/k’)

xlabel(’kth iteration’)
ylabel(’sum of kth optimal policy values’)
alpha1 = transpose(MDPQalpha1)
alpha2 = transpose(MDPQalpha2)
alpha3 = transpose(MDPQalpha3)








function[Qk_factor,delta_list,reward_list,k] = Qfactor_VI(transition_matrix,
    reward_matrix, ...
    discount,epsilon)
    delta = inf;
    Qk_factor = zeros(size(reward_matrix,3),size(transition_matrix,1));
    k = 0;
    Q_factor_next = Qk_factor;
    delta_list = [];
    reward_list = [];
    while delta > (epsilon*(1-discount))/2*discount
    for a=1:size(reward_matrix,3)
        for s=1:size(transition_matrix,1)

            factor_reward = reward_matrix(s,:,a);
            Q_max = max(Qk_factor(:,:));
            update = factor_reward + discount*Q_max;
      Q_factor_next(a,s) = update(1,:)*transpose(transition_matrix(s,:,a));



        end
    end
   difference_matrix = abs(Qk_factor - Q_factor_next);
   delta = max(difference_matrix,[],'all');
   delta_list = [delta_list delta];
   sum_reward = sum(max(Qk_factor(:,:)));
   reward_list = [reward_list sum_reward];
   Qk_factor = Q_factor_next;
   k = k+1;

    end


end





function[Qtable,alpha_values,reward_list] = MDP_Qlearning(transition_matrix,
    reward_matrix,discount, ...
    max_iter,alpha_type,hyperparam_A,hyperparam_B)
    k=1;
    Qtable = initializeQtable(transition_matrix,reward_matrix);
    alpha_values = zeros(1,max_iter);
    currentstate = 1;
    reward_list = zeros(1,max_iter -1);
    while k<max_iter
        stateaction = getAction(currentstate,reward_matrix,transition_matrix);
        nextState = getNextstate(currentstate,stateaction,transition_matrix);
        reward = getReward(reward_matrix,stateaction,nextState,currentstate);
        alpha = getAlpha(k, alpha_type,hyperparam_A,hyperparam_B);
        Qtable = updateQtable(Qtable,currentstate,nextState,stateaction,reward,discount,alpha);
        currentstate = nextState;
        alpha_values(k) = alpha;
        sum_reward = sum(max(Qtable(:,:)));
        reward_list(k) = sum_reward;
        k = k + 1;

    end

end



%-----------------------------------------------------------------------%
%initialize the Qtable, all values are set to 0
%----------------------------------------------------------------------%
function[Qtable] = initializeQtable(transition_matrix,reward_matrix)
    Qtable = zeros(size(reward_matrix,3),size(transition_matrix,1));

end

%---------------------------------------------------------------%
%function that samples an action from the currentstate W.P 1/|A|
%where A is the admissible action set of the currentstate
%---------------------------------------------------------------%

function[action] = getAction(currentstate,reward_matrix,transition_matrix)
    actionspace = size(reward_matrix,3);
    %actionset = linspace(1,actionspace,actionspace);
    onehot = zeros(1,actionspace);

    for i=1:actionspace
        if sum(transition_matrix(currentstate,:,i)) ~= 0
          onehot(1,i) = 1;
        end
    end
    admissible_actionset = find(onehot == 1);

    action = datasample(admissible_actionset,1);


end


%----------------------------------------------------------------------%
%---gets next state sampled randomly from the distribution definied by
%---P(i,a) the current state and the action sampled previously
%----------------------------------------------------------------------%

function[nextState] = getNextstate(currentstate,action,transition_matrix)
  currentstate_distribution = transition_matrix(currentstate,:,action);
  stateset = linspace(1,size(transition_matrix,1),size(transition_matrix,1));
  nextState = datasample(stateset,1,2,'Weights',currentstate_distribution);


end


%-------------------------------------------------------------------%
%---gets the reward for input in the next function
%---doesn' really need to be a function but useful for outputing reward
%---in the training function
%-------------------------------------------------------------------%

function[reward] = getReward(Reward_matrix,action,nextstate,currentstate)
    reward = Reward_matrix(currentstate,nextstate,action);

end

%--------------------------------------------------------------------%
%----returns a step size as a function of the current step number
%--- type defines the kind of step size to be used
%----------------------------------------------------------------------%
function[alpha] = getAlpha(stepnumber,type,A,B)
if type == 1
    alpha = A/(B + stepnumber);
elseif type == 2
    alpha = 1/stepnumber;
elseif type == 3
    alpha = log(stepnumber + 1)/stepnumber;


end
```

```
end

%-----------------------------------------------------------------%
%----updates the Qtable
%-----------------------------------------------------------------%


function[Qtable] = updateQtable(Qtable,currentstate,nextstate,action, ...
    reward,discount_rate,stepsize)

    Qtable(action,currentstate) =  Qtable(action,currentstate) + stepsize*(reward + discount_rate*max(Qtable(:,nextstate)) - Qtable(action,currentstate));


end
```

Grid World Q-learning

```
    clear all;
%[1,2,3,4] = [up, down, left, right]
%current state in a nxn matrix is indexed like this

% 1 4 7
% 2 5 8
% 3 6 9

%threeGrid = [0 -100 0 ; 0 0 0;-100 -100 100];
rng(100)


%Gridworld =    [0 -100 0 0 0; 0 -100 0 -100 0;0 -100 0 -100 0 ;  0 0 0 -100 0; 0 0 0 0 100];

Gridworld = [0 -100 0 0 0; 0 0 0 -100 0; -100 0 -100 -100 0 ; 0 0 0 -100 0; 0 0 0 0 100];

[Q,Env,terminals] = initialize_QtableandEnv(4,Gridworld);
Q = transpose(Q)
Env
terminals
[q_highexp,rewards_highexp] = Qlearning(Gridworld,0.9,0.9,500000,0.1);
[q_lowexp,rewards_lowexp] = Qlearning(Gridworld,0.25,0.9,500000,0.1);
[q_midexp,rewards_midexp] = Qlearning(Gridworld,0.6,0.9,500000,0.1);

transpose(q_highexp)

```

```
plot(rewards_lowexp,'blue')
hold on;
%figure;
plot(rewards_highexp,'red')
hold on;
plot(rewards_midexp,'green')
legend('low exploration 0.25','high exploration 0.9','medium exploration 0.6')
xlabel('episode number');
ylabel('sum of episode opimtal policy')

%q_midexp;
%q_lowexp;
%q_highexp;

%legend('low exploration', 'high exploration','medium exploration');


function[Q,episode_rewards] = Qlearning(Environment,exploration_rate,discount_rate,
    maximum_episodes,learning_rate)
    episode_rewards = zeros(1,maximum_episodes);
    [Q,Env,terminals] = initialize_QtableandEnv(4,Environment);
    for i=1:maximum_episodes
        terminal_check = false;
        currentstate = 1;
        while terminal_check == false
            stateaction = getAction(currentstate,Q,exploration_rate);
            newState = getNewState(stateaction,currentstate,Env);
            terminal_check = isTerminal(newState,terminals);
            Q = updateQtable(Q,stateaction,currentstate,newState,discount_rate,
    learning_rate,terminal_check);
            currentstate = newState;
        end
    v = linspace(1,size(Q,2),size(Q,2));
    current_episode_reward = sum(max(Q(:,v)));
    episode_rewards(i) = current_episode_reward;


    end



end


function [Qtable,Env,terminals]= initialize_QtableandEnv(actionspace ...
    ,Env)

statespace = size(Env,1);
Qtable = zeros(actionspace,statespace^2);
terminals = zeros(statespace);
    for i = 1:statespace^2
        if Env(i) ~= 0
            terminals(i) = 1;
        end
    end
end


function [action,actionset] = getAction(currentstate, Qtable, epsilon)
   %this function needs to check the type of state (Edge of map,corner,center)
   %we are in and randomly
   %select an admissible action.
   %according to an epsilon greedy policy

   statespace = sqrt(size(Qtable,2));
   action = 0;

  if mod(currentstate,statespace) == 1 %check if on first row

    if currentstate == 1
         actionset = [2 4];
    elseif currentstate ==  1 + statespace*(statespace - 1)
        actionset = [2 3];
    else
        actionset = [2 3 4];
    end

  elseif currentstate <= statespace %check if on left most column
      if currentstate == 1
         actionset = [2 4];
      elseif currentstate == statespace
        actionset = [1 4];
      else
         actionset = [1 2 4];
      end
  elseif mod(currentstate,statespace) == 0 %check if on bottom row
      if currentstate == statespace
        actionset = [1 4];
       elseif currentstate == statespace^2
        actionset = [1 3];
      else
        actionset = [1 3 4];
      end
  elseif currentstate>= 1 + statespace*(statespace - 1) %check if on right col
      if currentstate == 1 + statespace*(statespace - 1)
        actionset = [2 3];
      elseif currentstate == statespace^2
          actionset = [1 3];
      else
          actionset = [1 2 3];
      end
  else
      actionset = [1 2 3 4];
  end

  r = rand(1);
  max_selection_set = zeros(1,4);
  if r < epsilon
      action = datasample(actionset,1);
  else
      for i =1:4
          if ismember(i,actionset)
             max_selection_set(i) = Qtable(i,currentstate);
          else
              max_selection_set(i) = -inf;

          end
      end
      Y = find(max_selection_set == max(max_selection_set));
      action = datasample(Y,1);

  end
end


function[newState] = getNewState(action,currentState,Env)

    statespace = size(Env,1);
    if action == 1
        newState = currentState - 1; %down action
    elseif action == 2
        newState = currentState + 1; %up action
    elseif action == 3
        newState = currentState - statespace; %left action
    else
```

```
        newState = currentState + statespace; %right action
    end
end



function[terminal] = isTerminal(Nextstate,terminalstates)
    if terminalstates(Nextstate) == 1
        terminal = true;
    else
        terminal = false;
    end
end



function[Qtable] = updateQtable(Qtable,action,currentstate,nextstate,discount_rate,learning_rate,terminal_next)
    if terminal_next == true
        if nextstate == size(Qtable,2)
            R = 100;
        else
            R = -100;
        end
    else
       R = 0;
    end
    Qtable(action,currentstate) = Qtable(action,currentstate) + learning_rate*(R + discount_rate*max(Qtable(:,nextstate)) - Qtable(action,currentstate));

end
```

References

[Ros83] Sheldon M. Ross. *Introduction to Stochastic Dynamic Programming.* Probability and Mathematical Statistics: A series of Monographs and Textbooks. Academic Press, 1983. ISBN: 9780125984201. DOI: `https://doi.org/10.1016/C2013-0-11415-8`.

[WD92] Christopher Watkins and Peter Dayan. “Technical Note: Q-Learning”. In: *Machine Learning* 8 (1992), pp. 279–292. DOI: `https://doi.org/10.1023/A:1022676722315`.

[Tsi94] John N. Tsitsiklis. “Asynchronous Stochastic Approximation and Q-Learning”. In: *Machine Learning* 16 (1994), pp. 185–202. DOI: `https://doi.org/10.1023/A:1022689125041`.

[Gos08] Abhijit Gosavi. “On step sizes, stochastic shortest paths, and survival probabilities in Reinforcement Learning”. In: *2008 Winter Simulation Conference.* 2008, pp. 525–531. DOI: `10.1109/WSC.2008.4736109`.

[Gos14] Abhijit Gosavi. *Simulation Based Optimization.* Springer, 2014. ISBN: 9781489974907. DOI: `https://doi.org/10.1007/978-1-4899-7491-4`.

[Ber17] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control Volume I (Fourth Edition).* Athena Books, 2017. ISBN: 9781886529434.

[Ber18] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Volume II, Approximate Dynamic Programming (Fourth Edition).* Athena Books, 2018. ISBN: 9781886529441.

[Ros19] Sheldon M. Ross. *Introduction to Probability Models (Twelfth Edition).* Academic Press, 2019, pp. 193–291. ISBN: 9780128143469. DOI: `https://doi.org/10.1016/C2017-0-01324-1`.

[SB20] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (2nd Edition).* MIT Press, 2020. ISBN: 9780262352703.

[Gan] Tanmay Gangwani. *Lecture 16: Value Iteration, Policy Iteration and Policy Gradient.* URL: `https://yuanz.web.illinois.edu/teaching/IE498fa19/lec_16.pdf`.