Notes on Reinforcement Learning (DRAFT)

"Reinforcement Learning uses actions and rewards to directly or indirectly learn a system."

Paul F. Roysdon, Ph.D.

I. The Concept

(Note: This paper is incomplete, but it is posted as a starting point for the reader.)

Unlike other areas of machine learning, e.g. unsupervised or supervised learning, reinforcement learning (RL) uses a feedback loop of *actions* and *rewards* to learn a system; see Figure 1. The system is learnt by providing the RL algorithm a large reward for actions with good result, and small reward for actions with poor result.

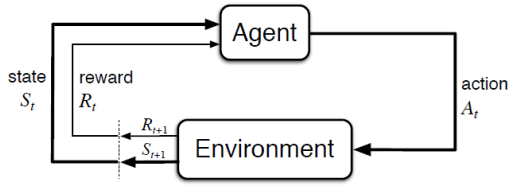

Fig. 1. Agent-environment interaction in a Markov decision process [1].

Consider we have a floor-plan for the house shown in Figure 2, and we want to know the fastest escape route for each room in the house. We can represent the paths in a *directed-graph*, as shown in Figure 3. Rooms with doors connecting them have bi-directional arrows. The exterior doors we assign the arrow a *reward* of 100, while all others receive 0 reward. An *action* is movement from one room, or *state*, to the next. A unique reward is given for each action. These assignments *encourage* the algorithm to *search* for the highest cumulative reward. Then, using RL, we can quickly *learn* the optimal paths for each room. While we can assign any value to each arrow – provided that the values are highest near the goal state – it is easiest to assign a large positive value for the goal state and zero elsewhere. We solve this example in the Section IV-A.1.

Of course RL has much broader applications than this trivial example. RL is capable of leaning human tasks and performing them at *superhuman* speed and accuracy, e.g. playing video games, flying an airplane, driving a car, etc. The concept of actions and rewards apply to all RL applications.

II. Advantages & Limitations

√ Fast, robust, easy to understand and implement.
X Not applicable to...

III. Digging Deeper

We begin by defining the nomenclature and standard equations of reinforcement learning (RL), then give examples

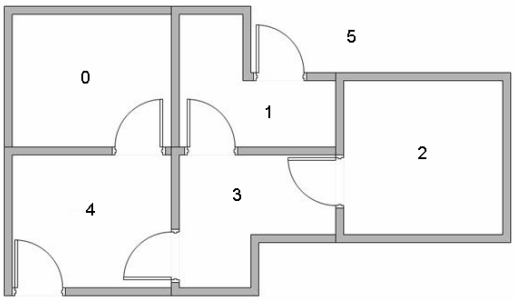

Fig. 2. House floor-plan example.

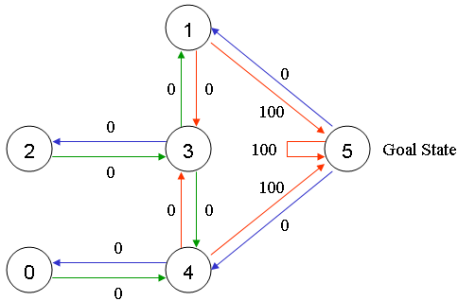

Fig. 3. Directed-graph representation of house floor-plan.

of common RL algorithms, e.g. Q-learning, polity gradient (PG) learning, deep Q-learning networks (DQN) and deep deterministic policy gradient (DDPG). For a complete review of RL, please refer to [1].

A. Action, Reward & State

An **action**, A_t is...

A **reward**, R_t is a scalar feedback signal at time step t that indicates the ability of the agent to maximize cumulative rewards.

Consequently the sequence of actions and rewards are important, and the common *i.i.d.* assumption does not hold.

The **history** is the sequence of observations O, actions A and rewards R for all observable variables up to time t,

$$H_t = O_1, A_1, R_1, \ldots, A_{t-1}, O_t, R_t.$$

The **state** is a function of the history, and contains the information used to determine the next action,

$$S_t = f(H_t).$$

The **environment state**, S_t^e is the environments private representation of the state, not visible to the agent, and determines the *reward*.

The **agent state**, S_t^a is the agent's internal representation, used to pick the next *action*, and can be any function of history,

$$S_t^a = f(H_t).$$

This is the information used by RL algorithms.

An **information state**, or *Markov state*, contains all useful information from its history. A state S_t is **Markov** *if and only if* the future is independent of the past, given the present. We define this mathematically as

$$p[S_{t+1}|S_t] = p[S_{t+1}|S_1, \ldots, S_t],$$

the probability of the future state, given the current state is equal to the probability of the future state, given the history of all prior states, including the current state. Here, the state is a *sufficient statistic* of the future, therefore once the state is known, the history can be discarded. To simplify notation, let

$$S_t = s \text{ and } S_{t+1} = s'.$$

When an agent directly observes the environment state, it is said to be *fully observable*: $O_t = S_t^a = S_t^e$. This is known as a **Markov Decision Process** (MDP).

An agent that *indirectly* observes the environment is said to be *partially observable*. This is known as a *partially observable Markov Decision Process* (POMDP).

B. Policy, Value Function, & Model

A **policy**, π, is the agent's behavior. It is a *map* from state to action that can be represented as either:

- Deterministic policy: $a = \pi(s)$
- Stochastic polity: $\pi(a|s) = p[A_t = a|S_t = s]$

A **value functions** is a prediction of future reward, used to evaluate the quality of the states. The value function is used to select between actions

$$\mathbf{v}_\pi(s) = \mathrm{E}\left\langle R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s \right\rangle$$

A **model** predicts what the environment will do next, where $\mathcal{P}$ *predicts the next state*, and $\mathcal{R}$ *predicts the next reward*:

$$\mathcal{P}_{ss'}^a = p[S_{t+1} = s'|S_t = s, A_t = a]$$

$$\mathcal{R}_s^a = \mathrm{E}\left\langle R_{t+1}|S_t = s, A_t = a \right\rangle$$

C. RL Categories & Solution Methods

RL can be used on both model-based, e.g. the known physics and balancing of an inverted pendulum, and model-free applications, e.g. images of something trying to balance an inverted pendulum. RL agents are categorized as follows:

- Value Based
  - No policy (implicit)
  - Value function
- Policy Based
  - Policy
  - No value function
- Actor-Critic
  - Policy
  - Value function
- Model Free
  - Policy and/or Value function
  - No model
- Model Based
  - Policy and/or Value function
  - Model

These are represented by a Venn diagram in Figure 4. We can further define RL into solution methods:

- Tabular Solution Methods
  - Multi-armed Bandits
  - Finite Markov Decision Processes
  - Dynamic Programming
  - Monte Carlo Methods
  - Temporal-Difference Learning
  - n-step Bootstrapping
  - Planning and Learning
- Approximate Solution Methods
  - On-policy Prediction with Approximation
  - On-policy Control with Approximation
  - Off-policy Methods with Approximation
  - Eligibility Traces
  - Policy Gradient Methods

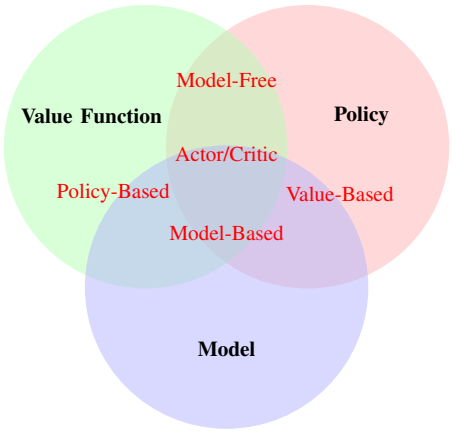

Fig. 4. RL agent taxonomy Venn diagram [1].

D. Types of Decision Making

In *reinforcement learning*, the environment is initially unknown, so the agent interacts with the environment to improve its policy.

In *planning*, a model of the environment is already is known, and the agent performs calculations with its model (without interaction) to improve its policy.

Improving the agent policy is achieved through either

- *Exploration* - find more information about the environment,
- *Exploitation* - exploit known information to maximize reward.

Computationally the agent can perform either

- *Prediction* - evaluate the future given a policy,
- *Control* - optimize the future by finding the best policy.

E. Markov Decision Process

For a Markov state s and a successor state s', the **state transition probability** is defined by

$$\mathcal{P}_{ss'} = p[S_{t+1} = s' | S_t = s].$$

The state transition matrix $\mathcal{P}$ defines the transition probabilities from all states s to all successor states s',

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}$$

where each row of the matrix sums to 1. The rows of $\mathcal{P}$ are the *states*, the columns of $\mathcal{P}$ the *actions*.

A **Markov process** (MP) is a memoryless random process, i.e. a set, or sequence, of random states $\{S_1, S_2, \ldots, S_t\}$ with the Markov property defined as a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$, where $\mathcal{S}$ is a finite set of states.

A **Markov reward process** (MRP) is a tuple that includes values $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{R}$ is a reward function

$$\mathcal{R}_s = \mathrm{E}\left[R_{t+1} | S_t = s \right],$$

and γ is a discount factor, $\gamma \in [0, 1]$.

The **return**, G_t is the total discounted reward from time t,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (1)$$

The discount γ is the present value of future rewards.

The **state value function** $\mathbf{v}(s)$ of an MRP is the expected return starting from state s,

$$\mathbf{v}(s) = \mathrm{E}\left[G_t | S_t = s \right]. \quad (2)$$

A **Markov decision process** (MDP) is a tuple that includes values $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{A}$ is a finite set of actions. We extend $\mathcal{P}$ as

$$\mathcal{P}^a_{ss'} = p[S_{t+1} = s' | S_t = s, A_t = a],$$

and $\mathcal{R}$ as

$$\mathcal{R}^a_s = \mathrm{E}\left[R_{t+1} | S_t = s, A_t = a \right].$$

A *policy*, π, is a distribution over actions, given the states,

$$\pi(a|s) = p[A_t = a | S_t = s].$$

A policy fully defines the behavior of an agent, and depends on the current state. Policies are *stationary* (time independent): $A_t \sim \pi(\cdot|S_t), \forall t > 0$.

F. Bellman Equation

From eqn. 2, we can derive the Bellman equation that we will later use for Q-learning. The Bellman equation for MRP is a value function that has two parts: immediate reward R_{t+1}, and discounted value of the successor state $\gamma \mathbf{v}(S_{t+1})$.

$$\begin{aligned} \mathbf{v}(s) &= \mathrm{E}\left[G_t | S_t = s \right] \\ &= \mathrm{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s \right] \\ &= \mathrm{E}\left[R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \ldots) | S_t = s \right] \\ &= \mathrm{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] \\ &= \mathrm{E}\left[R_{t+1} + \gamma \mathbf{v}(S_{t+1}) | S_t = s \right] \end{aligned}$$

The solution follows: first insert eqn. 1 into eqn. 2. Then factor γ. Next, notice the parenthesis is just G_{t+1} which is equal to $v(S_{t+1})$.

From $s \mapsto \mathbf{v}(s)$, let

$$\mathbf{v}(s) = \mathrm{E}\left[R_{t+1} + \gamma \mathbf{v}(S_{t+1}) | S_t = s \right]$$

then, given reward r, $s' \mapsto \mathbf{v}(s')$, let

$$\mathbf{v}(s) = \mathcal{R}_s = \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} \mathbf{v}(s'). \quad (3)$$

Using eqn. 3, we can express the Bellman equation in matrix form

$$\mathbf{v} = \mathcal{R} + \gamma \mathcal{P} \mathbf{v}, \quad (4)$$

where $\gamma \in \mathbb{R}$, $\mathbf{v} \in \mathbb{R}^{n \times 1}$, $\mathcal{R} \in \mathbb{R}^{n \times 1}$, and $\mathcal{P} \in \mathbb{R}^{n \times n}$, i.e.

$$\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}.$$

To solve for $\mathbf{v}$ in eqn. 4, we perform some algebra,

$$\begin{aligned} \mathbf{v} &= \mathcal{R} + \gamma \mathcal{P} \mathbf{v} \\ (\mathbf{I} - \gamma \mathcal{P}) \mathbf{v} &= \mathcal{R} \\ \mathbf{v} &= (\mathbf{I} - \gamma \mathcal{P})^{-1} \mathcal{R}. \end{aligned} \quad (5)$$

Eqn. 5 can be solved iteratively using: dynamic programming, Monte-Carlo, or Temporal-Difference learning.

G. Optimal Policy

There is always a deterministic optimal policy for any MDP that is found by maximizing over $\mathbf{q}^*(s, a)$:

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \underset{a \in A}{\arg\max}\ \mathbf{q}^*(s, a) \\ 0, & \text{otherwise} \end{cases}$$

The optimal value functions are recursively related by the Bellman optimality equation

$$\mathbf{v}^*(s) = \max_a\ \mathbf{q}^*(s, a).$$

Given a reward, r, then

$$\mathbf{q}^*(s, a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \mathbf{v}^*(s').$$

The Bellman optimality equation for $\mathbf{v}^*$ is

$$\mathbf{v}^*(s) = \max_a \ \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \mathbf{v}^*(s'),$$

and the Bellman optimality equation for $\mathbf{q}^*$ is

$$\mathbf{q}^*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} \ \mathbf{q}^*(s',a'). \tag{6}$$

The Bellman optimal equation, eqn. 6, is non-linear with no closed-form solution. Eqn. 6 can be solved by many iterative methods, e.g. value iteration, policy iteration, Q-learning, and SARSA.

IV. Algorithm Description

A. Q-Learning

The state-value function can be decomposed into reward plus discounted value of successor state, such that

$$\mathbf{v}_\pi(s) = \mathrm{E}_\pi \left\langle R_{t+1} + \gamma \mathbf{v}_\pi(S_{t+1}) | S_t = s \right\rangle .$$

The action-value function is defined as

$$\mathbf{Q}_\pi(s,a) = \mathrm{E}_\pi \left\langle R_{t+1} + \gamma \mathbf{Q}_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a \right\rangle .$$

Numerically,

```
Q(s,a) = R(s,a)+gamma*(max(Q(s_next,:)))
```

1) Example: Returning to the example in Figures 2 and 3. There are 6 states with the following connections, listed as [state; connected states]:

- $[0;\ 4]$
- $[1;\ 3, 5]$
- $[2;\ 3]$
- $[3;\ 2, 1, 4]$
- $[4;\ 0, 3, 5]$
- $[5;\ 1, 4, 5]$

This is represented mathematically as shown in eqn. 7, where the rows are *states* and the columns are *actions*; Each element of R has a value, wherein a connection, e.g. state 0 to 4 has value 0, a non-connection, e.g. state 0 to 1 has the value -1, and connections to the goal state, e.g. state 4 to 5 has the value 100.

$$R = \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \tag{7}$$

We can implement this in just 24 lines of MatLab code as shown below.

```
s = randi([1,max(size(R))]);  %Select a state randomly
Qn = zeros(400,1); %initialize the norm of Q
Qc = Q; %use this to "see" how many times an index is updated
for iter = 1:400 %Learn Q: run for 400 iterations
    a_set = [];  %Create an empty action-set
    %Build an actionset
    for a = 1:max(size(R))
        if R(s, a) ~= (-1)
            a_set(end+1) = a;
        end %Created set of all possible actions
    end
    a = a_set(randi(numel(a_set))); %Select Random action
    s_next = a;
    Q(s,a) = R(s,a)+gamma*(max(Q(s_next,:))); %Bellman equation
    Qc(s,a) = Qc(s,a)+1;
    s = s_next;
    Qn(i,1) = norm(Q);
end
Q = Q*100/max(Q(:)); %re-scale Q
s = s_start
while s ~= s_goal %Solve: iterate to find path start to finish
    [a_val, a] = max(Q(s,:));
    s = a
end
```

Running this example, we evaluate the optimization of Q, i.e. *learning* Q, by calculating $\|Q\|_2$ at each iteration. In Figure 5 we see that Q does not improve after 150 iterations. Evaluating Q, starting at state 2 with a goal of 5, the algorithm easily solves the states $2, 3, 1, 5$. If we look at eqn. 8 we see the number of times each index of Q_c, and therefore Q, was updated through random selection.

$$Q_c = \begin{bmatrix} 0 & 0 & 0 & 0 & 24 & 0 \\ 0 & 0 & 0 & 33 & 0 & 40 \\ 0 & 0 & 0 & 26 & 0 & 0 \\ 0 & 33 & 26 & 0 & 28 & 0 \\ 24 & 0 & 0 & 28 & 0 & 27 \\ 0 & 40 & 0 & 0 & 27 & 44 \end{bmatrix} \tag{8}$$

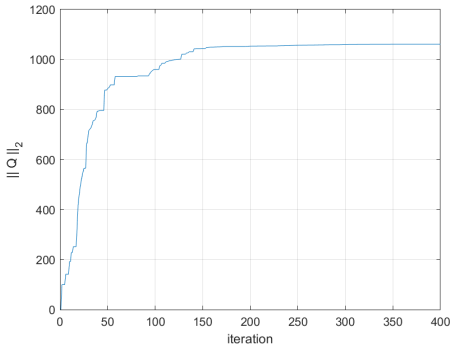

Fig. 5. $\|Q\|_2$ at each iteration.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.