Fundamentals of Reinforcement Learning

Weidong Kuang

University of Texas Rio Grande Valley

August, 2021

Contents

Preface

Who should read this manuscript?

Anyone who wants to start a journey of reinforcement learning, including college students, professors, researchers, engineers.

What has been covered?

This manuscript covers all important fundamental aspects of reinforcement learning. In chapter2, Markov Decision Process is presented to model the interaction between the agent and the environment. Chapter 3 introduces a core technique – dynamic programming – to solve an MDP problem, based on a given model of the environment. In general, solving an MDP problem involves iterations of policy evaluation and policy improvement. In chapter 4, we discuss model-free methods: Monte Carlo, TD learning, SARSA, and Q-learning. A general discussion on value function approximation in reinforcement learning is given in chapter 5. As an important example, deep Q-learning is detailed by chapter 6. The policy gradient methods are addressed in chapter 7, concentrating on the basic concepts: policy gradient theorem, REINFORCE algorithm, Advantage Actor-Critic algorithm.

Prerequisites for reading this manuscript

- Probability and statistics
- Basic optimization: stochastic gradient descent
- Basic Python programming (PyTorch framework for neural networks)
- Basic supervised learning: regression, classification by neural networks

I hope you enjoy the reading. If you have any feedback or question on this manuscript, don't hesitate to contact me at weidong.kuang@utrgv.edu .

Acknowledgement: Sutton & Barto book, DeepMind lectures by David Silver and Stanford CS234 Winter 2019, various blogs from internet, Dr. Dong-Chul Kim at CS UTRGV.

Chapter 1 Introduction to Reinforcement Learning

(to be completed)

This chapter will introduce the basic definitions, such as machine learning, supervised learning, unsupervised learning, and reinforcement learning, and their recent development history with a focus on reinforcement learning. Some typical applications of RL will be reviewed.

Readers can go to Chapter 2 now.

Chapter 2 Markov Decision Processes

Learning Objectives

The purpose of this chapter is to introduce the notations that will be used in the subsequent chapters and the most essential facts about Markov Decision Processes (MDPs) we will need in the rest of the book. Upon completion of this chapter, you should be able to

- Be familiar with the reinforcement learning settings.
- Describe the dynamics of the environments in terms of Markov decision process (MDP)
- Understand the basic concepts: policy, state-value function and action-value function
- Apply Bellman equations to solve a problem associated with MDP

2.1 Reinforcement Learning Settings and Notations

A supervised learning agent make decisions based on a set of labelled examples that is called training set. For example, a classifier can tell a picture whether contains a dog or a cat if it has been trained by a set of pictures labelled as either dog or cat. In contrast, a reinforcement learning agent can learn what to do through its interactions with the environment, i.e., how to map situations to actions. The "situations" may include the states of the environment and the feedback (reward) from the environment. The "reward" indicates how good (favorite) the action is. In some reinforcement settings (e.g. chess game), the reward is received only at the end of episode (win or loss for the game) while the rewards come more frequently in other settings. The interaction between the agent and its environment can be typically illustrated by Fig.1 The notations and terms in Fig.1 are described as follows.

Agent: learner and decision maker.

Environment: everything outside the agent.

Timestep: at each timestep t, the agent and environment interacts, t =0, 1,2,3,…

At timestep t:

State: $S_t \in \mathcal{S}$

Action: $A_t \in \mathcal{A}(\mathcal{S})$

Reward: at timestep t+1, as the consequence of the action A_t, the agent receives a ***reward*** $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and the environment moves to a new state S_{t+1}.

History (or trajectory): an interaction sequence, $S_0, A_0, R_1, S_1, A_1, R_2, S_2, \ldots$

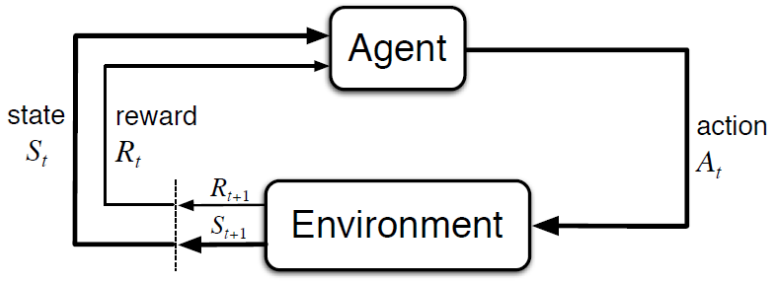


Fig.1 Agent-environment interaction

Dynamics of the environment: $p(s', r|s, a) \equiv Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$ is the conditional joint probability of the next state and reward, given the current state and action. In our context, we assume that the dynamics satisfy the Markov property.

Markov property:

- $p(s', r|s, a)$ completely characterizes the environment's dynamics.
- The state captures all relevant information from history.
- Once the state is known, the history may be thrown away.
- The state is a sufficient statistic of the past.

Return at t: the sum of the rewards received after timestep t (from t+1 to T, T is the final time step)

$$G_t \equiv R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots + \gamma^{T-(t+1)} R_T$$

The general goal of reinforcement learning is to maximize the overall return by selecting appropriate actions during the interaction.

Discount rate: $\gamma \in [0,1]$. A small discount rate gives less weights of future rewards towards the return. There are two extreme cases: 1) only immediate reward (R_{t+1}) is counted when $\gamma = 0$; 2) all rewards are equally counted when $\gamma = 1$.

2.2 Markov Decision Process

A countable Markov decision process (MDP) is defined a triplet $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P})$, where

$\mathcal{S}$: the countable non-empty set of states

$\mathcal{A}$: the countable non-empty set of actions

$\mathcal{P}$: the transition probability kernel assigns to each state-action pair a probability measure over $\mathcal{S} \times \mathcal{R}$. One element is denoted as $p(s', r|s, a)$.

If both $\mathcal{S}$ and $\mathcal{A}$ are finite, the MDP is called finite.

MDPs are a tool for modeling sequential decision-making problems where an agent interacts with an environment in a sequential fashion. This interaction can be described as follows: given an MDP $\mathcal{M}$, at the timestep t, the environment's state is S_t and the agent chooses an action A_t. This action will make a transition according to the dynamics of environment, $p(s', r|s, a)$. Then, at timestep t+1, the agent observes the next state S_{t+1} and reward R_{t+1}, chooses a new action A_{t+1}, and the process is repeated. The goal of the agent is to find a way of choosing the actions so as to maximize the expected total discounted reward, based on the observed history.

[**Example 1]** Dice game. The game rule is defined as:

For each round r =1, 2, …

- You choose STAY or QUIT.
- If QUIT, you get $10, and we end the game.
- If STAY, you get $4, and then I roll a 6-sided dice:
    - If the dice results in 1 or 2, we end the game.
    - Otherwise, continue to the next round.

This game can be described and treated as an MDP. At any time, the game is specified by two states: IN and END. When the game in the state IN, the agent can choose an action from the set $\mathcal{A} = \{STAY, QUIT\}$. If QUIT is selected, then the game will go to state END and the agent receive a reward \$10 with a probability of 1. If STAY is selected, the game will stay at IN state and the agent receive a reward \$4 with a probability of 2/3, or transition to state END and the agent receive a reward \$4 with a probability of 1/3. This MDP is illustrated by the transition diagram in Fig.2 (a). A blank circle represents a state while a solid bubble represents a probability splitting after an action a is taken. Along the path to the next state, the corresponding probability and reward are specified. For example, $p(IN, 4|IN, STAY) = 2/3$, $p(IN, 10|IN, QUIT) = 0$.

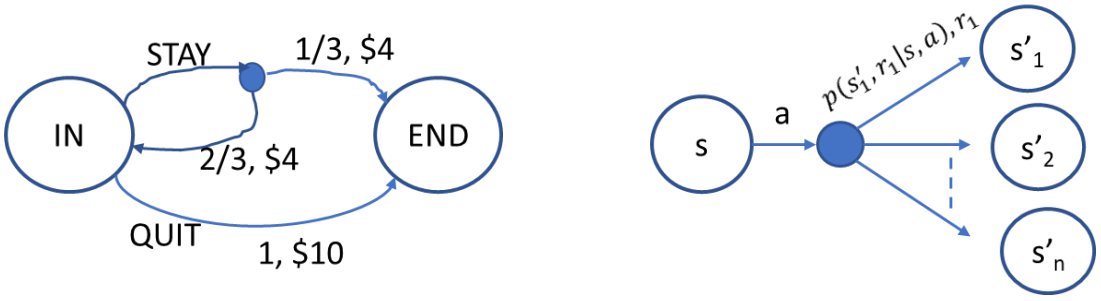


Fig.2 (a) an MDP (b) general diagram for MDP

An interesting question is: if you play the game, will you choose QUIT or STAY to get the maximal return in average? We will revisit this question later.

[Example 2] Volcano Crossing. As shown in Fig. 3, the land is divided into 3x4 grid cells: (x,y), x=1,2,3 and y=1,2,3,4. The volcano is located in the cells (1,3) and (2,3). There are two island cells (3,1) and (1,4). An agent travels on the land. The agent can move in four directions: east (E), south (S), west (W), and north (N). If the agent tries to move off the grid, it remains in the same cell. The number in each cell represents the reward when the agent moves to it. The rewards are zero for the cell without a number. For example, if the agent moves to any volcano cell (either (1,3) or (2,3)), it receives -50 reward. If the agent moves to the island cell (1,4), a reward of 20 is received. Although the agent can select the moving direction, the agent slips with a probability p when moving. If it slips, it will move randomly in one of the four directions. If it does not slip (with a probability of 1-p), it will go to the cell it intends for. Whenever the agent reaches any island or volcano cell, the travel ends.

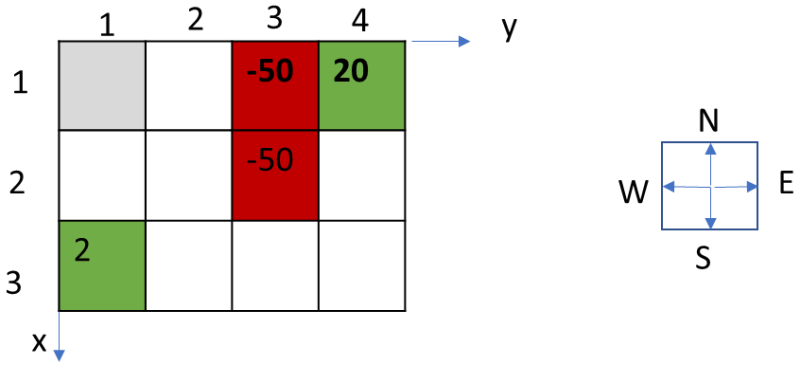


Fig. 3 Volcano crossing: an MDP

The question is: how should the agent select a moving direction given its current location so that it can get a maximal return for the travel in average sense? Without the loss of generality, we can assume the agent starts at cell (1,1) and discount rate $\gamma = 1$. A further question is: should the agent try to move to the island (3,1) for a reward of 2 or to move to the island (1,4) for a larger reward of 20 but with a bigger risk of slipping into volcano? We will revisit these questions later.

Now let's describe this volcano crossing in terms of MDP. In this example, the state is the location of the agent (x,y). Action set $\mathcal{A} = \{E, S, W, N\}$. The dynamics of the MDP is defined by $p(s', r|s, a) \equiv Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$. For example, by assuming slipping probability p=0.1, $p((1,1), 0|(1,1), E) = 0.1 \times \left(\frac{1}{4} + \frac{1}{4}\right) = 0.05$. This probability means that when the agent selects an action E at the state (1,1), the agent will stay at the same place (1,1) with a reward of 0 with a probability of 0.05. Note that even though it selects E, there is a chance for slipping to occur. When slipping occurs, it could move E,S,W,N with a probability of ¼ for each direction, and W and N will lead the agent to stay. A reader can verify $p((1,4), 20|(2,4), N) = 0.9 + 0.1 \times \frac{1}{4} = 0.925$.

2.3 Policies and value functions

To answer the questions mentioned in the examples of dice game and volcano crossing, we need to specify them as MDPs first, and then solve the problems associated with the MDPs. In this section, we will introduce two important concepts: policy and value function. Then Bellman equations will be introduced to calculate the value function based on the policy and the dynamics of the environment.

The behavior of the agent is described by a ***policy*** specifying a mapping from states to actions being selected by the agent. Thus, the policy is not the property of the environment, but the behavior of the agent. A deterministic stationary policy can be defined by a mapping π, which maps states to actions, $a = \pi(s)$. More generally, a stochastic stationary policy is defined by the probability distribution of selecting each possible action, denoted as $\pi(a|s)$. This implies that, at timestep t, $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. In general, a policy is probabilistic, instead of deterministic. Thus, a policy dictates how an agent should take an action given the current state, and it is independent of the property of environment in general. However, an optimal policy for a particular environment depends on the property of the environment.

Value functions measure how good it is for the agent to be in a given state or how good it is to perform a given action in a given state. Thus, there are two types of value functions: state-value function and action-value function. A value function is defined in terms of future expected return starting from a state or a state-action pair, by following a policy π.

1) State-value function is defined as the expectation value of future total return given a current state s and a policy π.

$$v_\pi(s) \equiv \mathbb{E}_\pi[G_t|S_t = s], \text{ for all } s \in \mathcal{S}$$

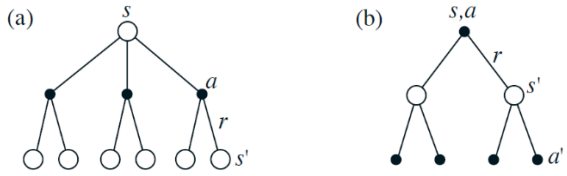
2) Action-value function is defined as the expectation value of future total return give a current state s and action, plus a policy π.

$$q_\pi(s, a) \equiv \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

3) Bellman equations (recursive relationship). With the aid of backup diagram, we can easily calculate the state-value function and the action-value function for a given policy π, in a recursive way:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s'))$$

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



Backup diagram

Please note that the equations are recursive, which means that $v_\pi(s)$ and $q_\pi(s,a)$ show up in both sides of the equations.

4) Bellman equation in matrix form, given a policy π,

$$\mathbf{v} = \mathbf{r} + \gamma \boldsymbol{P^\pi} \mathbf{v}$$

where

$$v_i \equiv v(s_i)$$
$$r_i = \mathbb{E}[R_{t+1}|S_t = s_i, A_t \sim \pi]$$

Solve for **v**, by

$\mathbf{v} = (\mathbf{I} - \gamma \mathbf{P^\pi})^{-1}\mathbf{r}$ **(for small problems)**

Dynamic programming

Monte Carlo evaluation

Temporal difference learning

Policy comparison (better or equal): $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$

[Example 3] Consider an MDP for the dice game described in the previous section Fig.2(a). Suppose the agent use a random policy (uniform) on selecting the action: STAY (S) or QUIT (Q), i.e.

$$\pi(a|s) = \frac{1}{2}, \qquad for\ a = STAY\ or\ QUIT, s = IN$$

The Bellman equations for state-value function can be written as

$$\begin{cases} v(END) = 0 \\ v(IN) = \frac{1}{2}\big(10 + v(END)\big) + \frac{1}{2}\left(\frac{1}{3}\big(4 + v(END)\big) + \frac{2}{3}\big(4 + v(IN)\big)\right) \end{cases}$$

By solving the equations, we get the state-value function for a random policy,

$$\begin{cases} v(END) = 0 \\ v(IN) = 10.5 \end{cases}$$

The result tells us that the player will get \$10.5 in average if selecting STAY or QUIT with an equal probability. This random policy is obviously better than just selecting QUIT deterministically that results in a \$10 return.

Similarly, if the agent adopts a policy of always selecting STAY, The Bellman equations for state-value function can be written as

$$\begin{cases} v(END) = 0 \\ v(IN) = 0\big(10 + v(END)\big) + 1\left(\frac{1}{3}\big(4 + v(END)\big) + \frac{2}{3}\big(4 + v(IN)\big)\right) \end{cases}$$

By solving the equations, we get the state-value function for a policy of always selecting STAY,

$$\begin{cases} v(END) = 0 \\ v(IN) = 12 \end{cases}$$

Since this policy (i.e. always selecting STAY) results in a larger state-value function, it is better than the random policy.

Now let's calculate the action-value function for a random policy by solving the corresponding Bellman equations. The Bellman equations can be written as

$$\begin{cases} q(IN, QUIT) = 10 \\ q(IN, STAY) = \frac{1}{3}(4+0) + \frac{2}{3}\left(4 + \frac{1}{2}q(IN, QUIT) + \frac{1}{2}q(IN, STAY)\right) \\ q(END, STAY) = 0 \\ q(END, QUIT) = 0 \end{cases}$$

By solving the equations, we have the action-value function for the random policy,

$$\begin{cases} q(IN, QUIT) = 10 \\ q(IN, STAY) = 11 \\ q(END, STAY) = 0 \\ q(END, QUIT) = 0 \end{cases}$$

This result implies that selecting STAY will get more return than selecting QUIT in average, given thereafter following random policy.

[Example 4] Recall the volcano crossing MDP, shown in Fig.3. The state is represented by the location of the agent, (x,y). Suppose that the agent selects the path toward the island with a reward of 20, indicated by the blue arrows in Fig.4(a). However, even if the agent selects one moving direction, it may slip with a probability of *p*, instead of moving in the selected direction. When a slip occurs, the agent will randomly move into its adjacent cell with equal probability (1/4), indicated by the red line arrows. Thus, the real policy $\pi(a|s)$ can be obtained based on the selected path and the location of the agent. For example, consider s=(1,1), we have the policy for this state (location)

$\pi\big(N|(1,1)\big) = p * \frac{1}{4}$ (slip to North)

$\pi\big(W|(1,1)\big) = p * \frac{1}{4}$ (slip to West)

$\pi\big(E|(1,1)\big) = p * \frac{1}{4}$ (slip to East)

$\pi\big(S|(1,1)\big) = 1 - p + p * \frac{1}{4}$ (intend for South and slip to South)

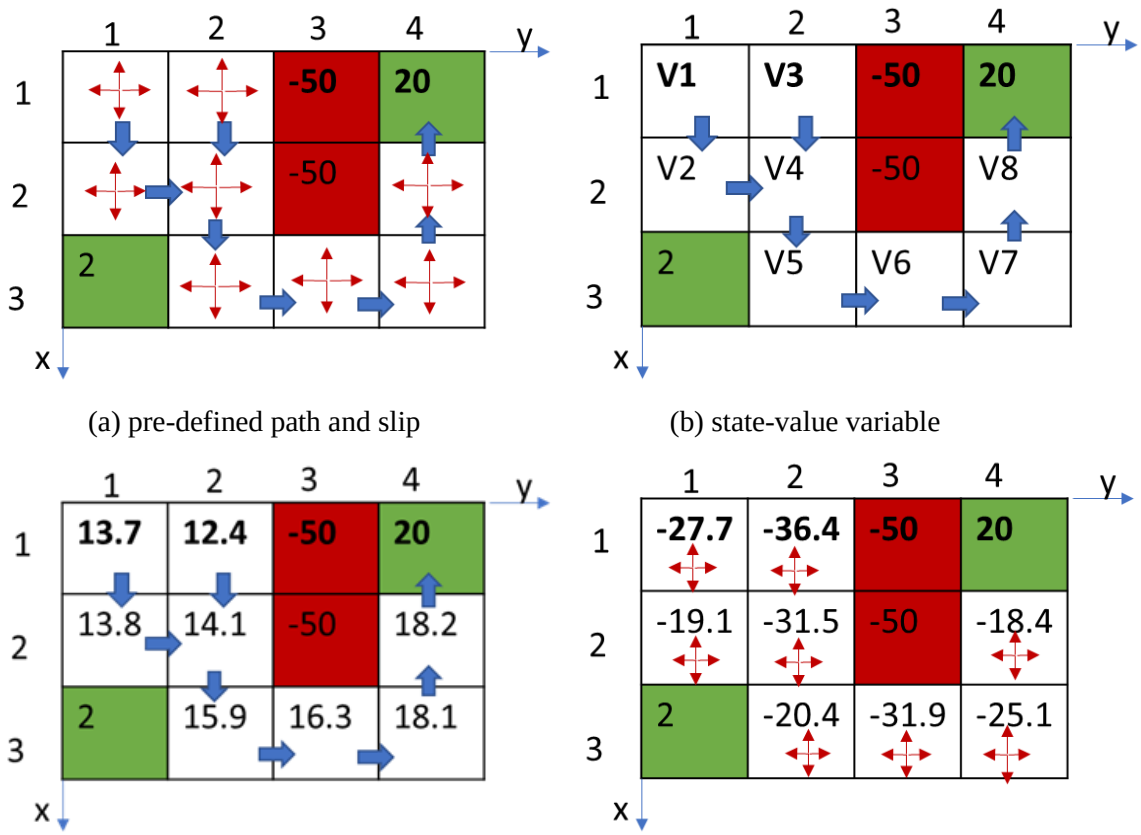
For the state (3,2), the policy is

$\pi\big(N|(1,1)\big) = p * \frac{1}{4}$ (slip to North)

$\pi\big(W|(1,1)\big) = p * \frac{1}{4}$ (slip to West)

$\pi\big(E|(1,1)\big) = 1 - p + p * \frac{1}{4}$ (intend for East and slip to East)

$\pi\big(S|(1,1)\big) = p * \frac{1}{4}$ (slip to South)



(a) pre-defined path and slip (b) state-value variable

(c) State-value function with p=0.1 (d) state values for random path (i.e. p=1)

Fig.4 Policy and value function

To find the state-value function, we first define the state values for the states as V1, V2, …., V8, in Fig.4(b), and then write one Bellman equation for each state. For instance, the Bellman equation for state (1,1) is

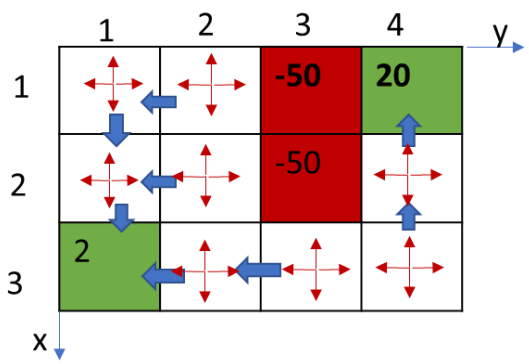
$$V1 = \underbrace{(1-p) \times V2}_{\text{Move to V2}} + \underbrace{p \times \frac{1}{4} \times (V1 + V1 + V2 + V3)}_{\text{slip}}$$

The first term in the right side of the equation corresponds to the intentional moving from V1 to V2. The second term accounts for slipping randomly to four directions. Recall that if the agent tries to move off the grid, it remains in the same cell. The Bellman equation for state (2,4) is

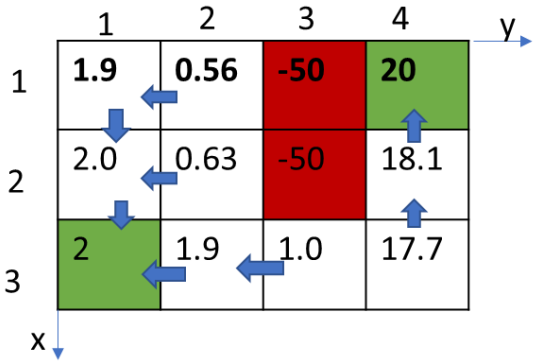
$$V8 = \underbrace{(1-p) \times 20}_{\text{Move to island}} + \underbrace{p \times \frac{1}{4} \times (20 - 50 + V7 + V8)}_{\text{slip}}$$

In total, we can have totally 8 equations for the 8 states and get the state-value function by solving these 8 linear equations. The solutions are shown in Fig.4(c) for p=0.1. Fig.(d) shows the state values for a policy of complete random move at all states (i.e., p=1).

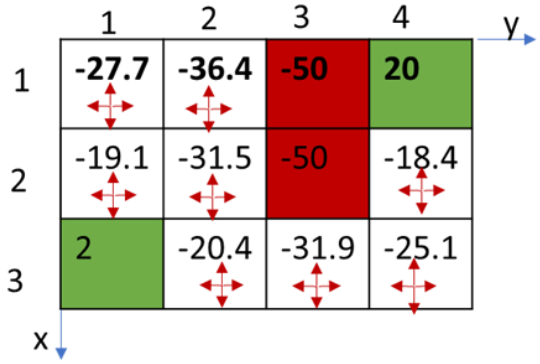
Now, consider a coward (timid) policy, shown in Fig.5(a). By following this policy, the agent tries to avoid slipping into volcano as possible as it can, regardless of a big reward 20 in the island. The resulting state values are shown in Fig.5(b). To compare the three policies (timid policy, random policy, and good policy), we just need to compare their state values. Obviously, the random policy is the worst one while the policy in Fig.5(d) is the best one since the state values in Fig.5(d) are more than those in Fig.5(b) and Fig.5(c).



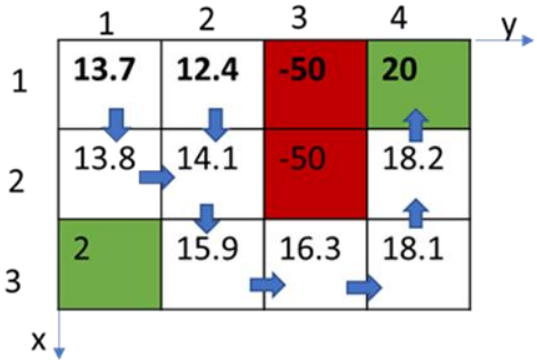
(a) A timid policy



(b) state values for the timid policy



(c) state values for the random policy



(d) state values for a good policy

Fig.5 Comparison for different policies

2.4 Optimal policies and optimal value functions

The state-value function depends on the policy adopted by the agent. As mentioned in the previous section, the policy π is said to be better (or equal) than π' (i.e., $\pi \geq \pi'$) if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$.

For an MDP, there is always at least one policy that is better than or equal to all other policies. This best policy is called an **optimal policy**, denoted as π_*, and the corresponding state-value function is the **optimal state-value function**, defined as

$$v_*(s) \equiv \max_\pi v_\pi(s)\,, for\ all\ s \in \mathcal{S}$$

Similarly, the **optimal action function** is defined as

$$q_*(s,a) \equiv \max_\pi q_\pi(s,a)\,, for\ all\ (s,a)\ pairs$$

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

How to find the optimal policy is a critical topic in reinforcement learning. Under the optimal policy, the Bellman equation is called **Bellman optimality equation**. It expresses the fact that the value of a state under an optimal policy must equal to the expected return for the best action from that state. The backup diagram in Fig.6 can help us write Bellman optimality equations.

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_*(s,a) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_*(s')) \end{aligned}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,|S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r|s, a)(r + \gamma \max_{a'} q_*(s', a')) \end{aligned}$$

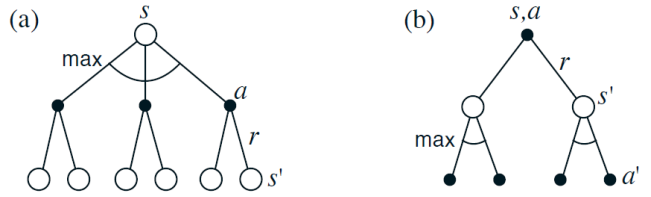


Fig.6 Backup diagram for optimal value functions: (a) v_* and (b) q_*.

[**Example 5**] Bellman optimality equation for the dice game.

State-value optimality equation is

$$v_*(IN) = \max_{STAY,QUIT} \left\{ \frac{1}{3}4 + \frac{2}{3}\big(4 + v_*(IN)\big), 10 \right\}$$

Note that this is a nonlinear equation. The solution to this equation is $v_*(IN) = 12$, corresponding to the policy of always STAY. This is consistent with Example 3. However, it is not feasible to directly solve the optimality equation for a large MDP problem in general. Instead, as we will see later, searching for approximate solution is more practical.

Optimality equation for action-value function is

$$q_*(IN, STAY) = \frac{1}{3}4 + \frac{2}{3}\left(4 + \max_{STAY,QUIT}\{q_*(IN, STAY), q_*(IN, QUIT)\}\right)$$
$$q_*(IN, QUIT) = 10$$

By solving the above equations, we obtain

$$q_*(IN, STAY) = 12$$

Finding an optimal policy: an optimal policy can be found by maximizing over $q_*(s,a)$,

$$\pi_*(a|s) = \begin{cases} 1 & if\ a = \operatorname{argmax}_a q_*(s,a) \\ 0 & otherwise \end{cases}$$

Note: there is always a deterministic optimal policy for any MDP

If we know $q_*(s,a)$, we immediately have the optimal policy
There can be multiple optimal policies
If multiple actions maximize $q_*(s,a)$, we can pick any of these.

Solving the Bellman optimality equation (non-linear), i.e. finding $q_*(s,a)$, is a fundamental topic in reinforcement learning, which will be addressed the subsequent chapters.

Using models: dynamic programming
- Value iteration
- Policy iteration

Using samples:
- Monte Carlo
- Q-learning
- Sarsa

2.5 Summary

In this chapter, we introduced a typical setting of reinforcement learning. An agent interacts with an environment over time. At each time step t, the agent receives a state S_t, and selects an action A_t from an action space $\mathcal{A}$, following a policy $\pi(a|s)$. The agent receives a scalar reward R_{t+1} and the environment transitions to the next state S_{t+1}. The goal of the agent is to learn (or search) an optimal policy by which the agent is expected to receive the maximal accumulated rewards at each state. We assume that the environment can be specified by a **Markov decision process**. A simple MPD can be visualized by a **state transition graph**.

Three basic concepts: **policy, state-value function and action-value function**, are defined, and will be used to describe a MDP problem. Deep understanding these concepts is essential. A value function is a prediction of the expected, accumulative, discounted, future rewards, measuring how good each state, or state-action pair is. **Bellman equation** is a tool for us to solve a task associated with MDPs. The **backup diagram** of Bellman equations is very helpful for us to develop and understand various concepts and algorithms.

Chapter 3 Dynamic Programming

The 1950s were not good years for mathematical research. I felt I had to shield the Air Force from the fact that I was really doing mathematics. What title, what name, could I choose? I was interested in planning, in decision making, in thinking. But planning is not good word for various reasons. I decided to use the word "programming." I wanted to get across the idea that this was dynamic, this was time-varying – I thought, let's kill two birds with one stone. Let's take a word that has precise meaning, namely dynamic, in the classical physical sense. It also is impossible to use the word, dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. it was something not even a Congressman could object to. So I used it as an umbrella for my activities.

-- Richard Bellman from Deep-mind lecture3 video

Learning Objectives

In this chapter, we solve an MDP problem through dynamic programming, instead of solving formal equations. Specifically, we will present how to compute the followings by dynamic programming:

- Policy evaluation, i.e., state-value function, given a particular policy.
- Policy improvement
- Policy iteration
- Optimal policy

Dynamic programming: refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. (Sutton & Barto). Dynamic programming for MDP assumes full knowledge of the transition and reward models of the MDP. Dynamic programming consists two alternating processes: 1) policy evaluation, which is to evaluate the value function for a given policy, and 2) policy improvement, which is to find a better value function and/or a better policy (eventually an optimal policy).

3.1 Policy evaluation

The purpose of policy evaluation is to compute the state-value function $v_\pi(s)$ for a given policy π. Instead of directly solving the state value equations (i.e., Bellman equations), we iteratively compute the state-value function using dynamic programming. Recall that the state-value function is defined as

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s'))$$

The idea behind dynamic programming is to turn this equality into an update operation. All the state values in the right side are old values while the left-hand side state value is to be updated. Specifically, we can initialize all values to zero, v_0=0, at step k=0. Then, we update the state-value for all states by iterating at step k, i.e.,

$$\text{for all s: } v_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | s, \pi]$$

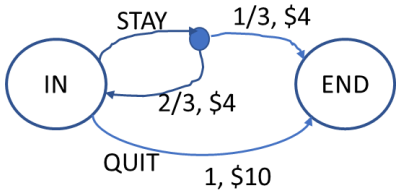
When $v_{k+1}(s) = v_k(s)$ for all s, we must have found v_π, and terminate the iteration.

Policy evaluation always converge? Yes, under appropriate condition (e.g. $\gamma < 1$). The policy evaluation can be implemented by either two arrays or one array (in-place), shown in Fig.7. The in-place way converges faster in general because the new state value (updated at iteration k) is used immediately for other state updates in the same iteration k, but converge rate depends on the order in which states are updated.

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Fig.7 Algorithm of policy evaluation by in-place update [from Sutton]

[Example 6] policy evaluation for the dice game with a random policy. Recall the dice game introduced in Example 1. The state transition diagram is re-drawn below. The value of state END is obviously zero, i.e. $V(END) = 0$. Assume that the probability of selecting STAY at state IN is 0.5.



We can evaluate the state-value function through the following iterative steps:

- k=0, $V_0(IN) = 0$
- k=1, $V_1(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_0(IN)\right)\right) = 7$
- k=2, $V_2(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_1(IN)\right)\right) = 9.33$
- k=3, $V_3(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_2(IN)\right)\right) = 10.11$
- k=4, $V_4(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_3(IN)\right)\right) = 10.37$
- k=5, $V_5(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_4(IN)\right)\right) = 10.46$
- k=5, $V_6(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_5(IN)\right)\right) = 10.49$
- k=6, $V_6(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_5(IN)\right)\right) = 10.50$
- k=7, $V_7(IN) = 0.5 \times \left(10 + V(END)\right) + 0.5\left(\frac{1}{3}\left(4 + V(END)\right) + \frac{2}{3}\left(4 + V_6(IN)\right)\right) = 10.50$

Since $V_7(IN) = V_6(IN) = 10.50$, the state value for IN is $v(IN) = 10.5$, which is the same as the result in Example 3.

[Example 7] consider a 4x4 gridworld with a random policy, shown in Fig.8. The agent moves on the grid randomly and receive a reward of -1 for each move. The episode ends at the up-left or bottom-right corners.

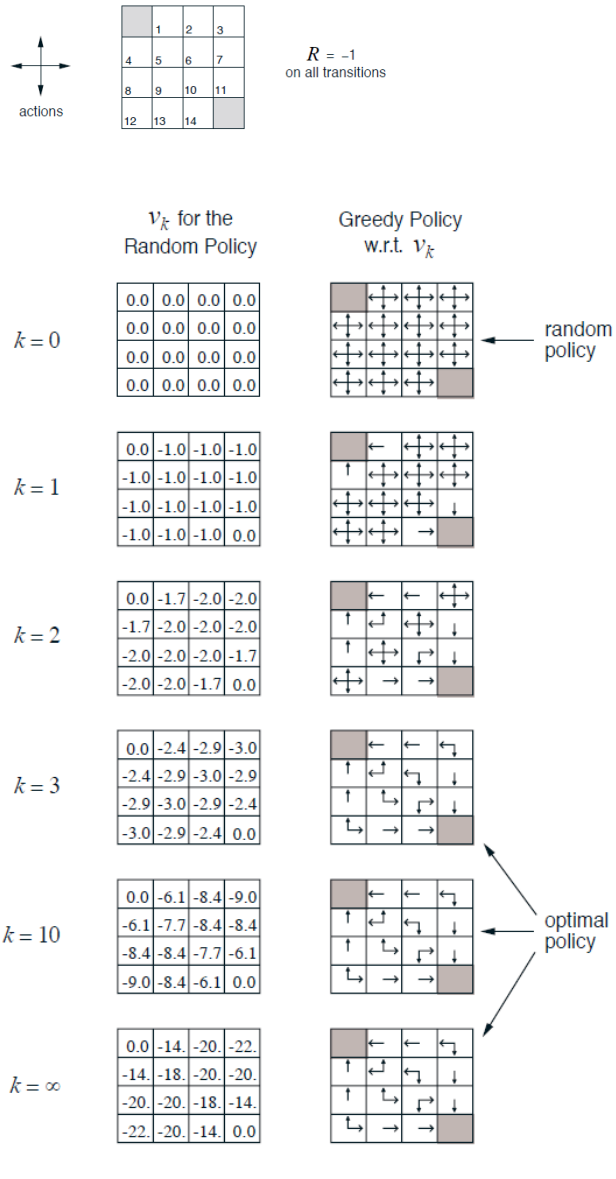


Fig.8 4x4 gridworld policy evaluation [from Sutton]

Step k =0, all V(s) are initialized to zero.

For a demonstration purpose, policy evaluation is implemented by two-arrays method (instead of in-place method). The left column shows the V(s) for k=1,2,3,10,∞. The right column shows the better policies (greedy policies) given the V(s) at the left side. The last three policies happen to be the optimal policy.

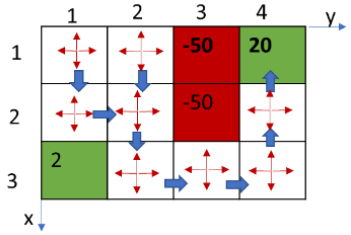
[Example 8] Policy evaluation for volcano crossing with a policy shown in Fig.9(a) at a slipping probability of 0.1. A Matlab file is developed to compute the iterative evaluations.

```
V=zeros(1,8)
for k=1:20
    temp(1)=0.9*V(2)+(V(1)+V(1)+V(2)+V(3))/40;
    temp(2)=0.9*V(4)+(V(1)+V(2)+V(4)+2)/40;
    temp(3)=0.9*V(4)+(V(1)+V(3)+V(4)-50)/40;
    temp(4)=0.9*V(5)+(V(2)+V(3)+V(5)-50)/40;
    temp(5)=0.9*V(6)+(V(4)+V(5)+V(6)+2)/40;
    temp(6)=0.9*V(7)+(V(5)+V(6)+V(7)-50)/40;
    temp(7)=0.9*V(8)+(V(6)+V(7)+V(7)+V(8))/40;
    temp(8)=0.9*20+(V(7)+V(8)+20-50)/40;
```

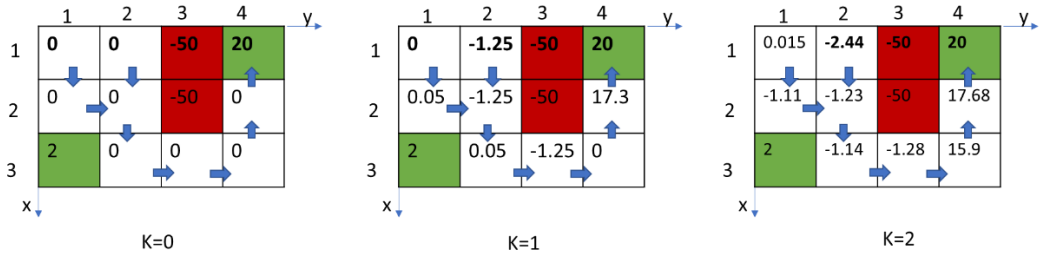
```
    V=temp;
    k %print k
    temp %print V
end
```

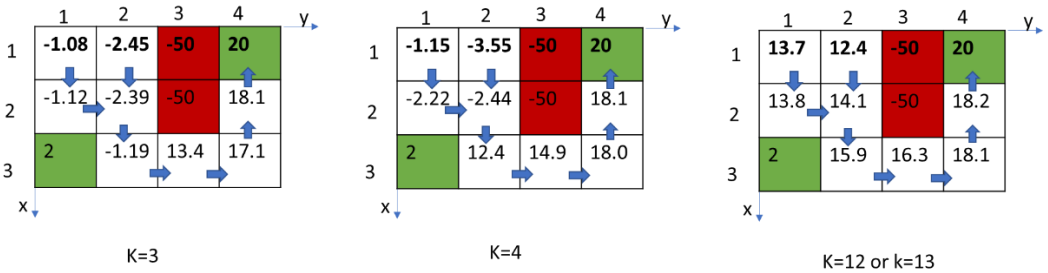
output:

V =0 0 0 0 0 0 0 0
k =1
temp =0 0.0500 -1.2500 -1.2500 0.0500 -1.2500 0 17.2500
k =2
temp =0.0150 -1.1050 -2.4375 -1.2338 -1.1362 -1.2800 15.9250 17.6812
k =3
temp =-1.0823 -1.1185 -2.4518 -2.3896 -1.1933 13.4202 17.1194 18.0902
k =4
temp =-1.1500 -2.2154 -3.5487 -2.4430 12.3741 14.8911 17.9249 18.1302
…
k =12
temp =13.6917 13.7629 12.4296 14.0914 15.8805 16.3043 18.1079 18.1566
k =13
temp =13.7260 13.7709 12.4375 14.0942 15.8808 16.3044 18.1079 18.1566



(a) volcano crossing policy





(b) policy evaluation using two-arrays approach: it converges at k=12. The result is the same as Fig.4(c).

Fig.9 Dynamic programming for policy evaluation

The following Matlab code implements the in-place evaluation procedure, which converges faster (at k=5) when each iteration starts at the cell next to the island of 20 (i.e. V(8)).

```
V=zeros(1,8)
```

```
for k=1:20
    V(8)=0.9*20+(V(7)+V(8)+20-50)/40;
    V(7)=0.9*V(8)+(V(6)+V(7)+V(7)+V(8))/40;
    V(6)=0.9*V(7)+(V(5)+V(6)+V(7)-50)/40;
    V(5)=0.9*V(6)+(V(4)+V(5)+V(6)+2)/40;
    V(4)=0.9*V(5)+(V(2)+V(3)+V(5)-50)/40;
    V(3)=0.9*V(4)+(V(1)+V(3)+V(4)-50)/40;
    V(2)=0.9*V(4)+(V(1)+V(2)+V(4)+2)/40;
    V(1)=0.9*V(2)+(V(1)+V(1)+V(2)+V(3))/40;
    k %print k
    V %print V
end
```

k=0: V =0 0 0 0 0 0 0 0
k =1: V =9.1148 9.6287 8.3287 10.3553 12.5463 13.5095 15.9563 17.2500
k =2: V =12.6883 12.9115 11.5790 13.3977 15.3500 15.9216 17.8597 18.0802
k =3: V =13.5423 13.6223 12.2890 13.9809 15.8039 16.2543 18.0784 18.1485
k =4: V =13.7072 13.7510 12.4177 14.0777 15.8702 16.2979 18.1043 18.1557
k =5: V =13.7357 13.7723 12.4389 14.0928 15.8795 16.3036 18.1074 18.1565
k =6: V =13.7403 13.7756 12.4422 14.0950 15.8808 16.3043 18.1078 18.1566

3.2 Policy improvement

If we have policy evaluation $v_\pi(s)\ for\ all\ s\epsilon\mathcal{S}$, we can find a new policy that is better than π by taking the action for the maximal state-action value:

$$\forall s: \pi_{new}(s) = \underset{a}{\operatorname{argmax}}\, q_\pi(s,a) = arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a]$$

Then, evaluate π_{new}. We can show that

$$v_{\pi_{new}}(s) \geq v_\pi(s) \text{ for all s}$$

In example 7 shown in Fig.8, for the random policy π, we evaluated the value function $v_\pi(s)$, shown by the table at the bottom k=∞. From this policy evaluation, we can get a new policy shown at the right side by police improvement, and the resulting new policy at the right bottom happens to be optimal in this case.

3.3 Policy iteration

In general, one step improvement does not necessarily get the optimal policy. Instead, a series of iterative steps are required for converging to the optimal policy and optimal value,

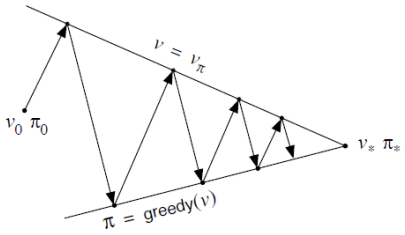
$$\pi_0 \to v_{\pi_0} \to \pi_1 \to v_{\pi_1} \to \cdots \to \pi_* \to v_*$$

Policy iteration alternates between policy evaluation and policy improvement, to generate a sequence of improved policies. In policy evaluation, the value function of the current policy π_i is estimated to obtain v_{π_i}. In policy improvement, the current value function is used to generate a better policy, e.g., by selecting actions greedily with respect to the value function v_{π_i}. This iteration process will converge to an optimal policy and value function.

3.4 Value iteration

One drawback of the above policy iteration is that each of its iteration involves policy evaluation which may itself require a long iterative computation. In fact, we can truncate the policy evaluation process (e.g. to only one sweep V0 to V1). In other words, we update the policy after each **value iteration (**not policy evaluation**)**. Thus, this algorithm is called **value iteration**, illustrated in Fig.10. For a special case, we take the Bellman optimality equation and modify into an update as below. This is equivalent to policy iteration, with k=1 step of policy evaluation between each two (greedy) policy improvement steps.

$$
\begin{aligned}
v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_k(s')\Big],
\end{aligned}
$$



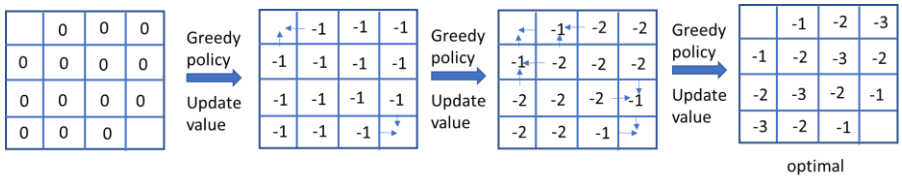
Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

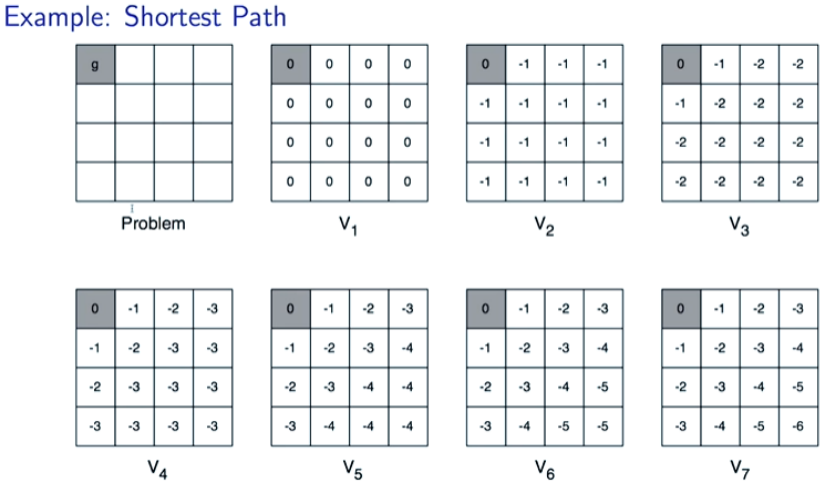
Output a deterministic policy, π, such that
  $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Fig.10 Find the optimal policy and optimal value function by value iteration

[Example 9] Find the shortest path to the destination. In Fig.11(a) the agent receives a reward of -1 for each move and stops at the up-left corner or the bottom right corner. It only takes three steps to find the optimal value function (and optimal policy) for the 4x4 gridworld task. In Fig.11 (b), the destination is located at the up-left corner. The last table in each case shows the optimal value function, and the corresponding optimal policy can be found by identifying the neighbor with the maximal value.



(a) Find the shortest path to the up-left or the bottom-right corner.



(b) Find the shortest path to the up-left corner.

Fig.11 Find the shortest path

[Exercise] Find the optimal path for the volcano crossing problem using Matlab program.

3.5 Summary

In this chapter, we introduced a technique, dynamic programming, to evaluate the state value function for an MDP, given a policy π. This state value evaluation process is called **policy evaluation**. In policy evaluation, we typically initialize the value function to zero for all states. Then the value of each state will be updated by one-step backup based on the currently available state values. There are two strategies for updating:

1) Two copies of value function are stored for old values and new values, respectively, during one iteration.
   For all s in S: $V_{new}(s) \leftarrow \mathbb{E}[R_{t+1} + \gamma V_{old}(S_{t+1}) | S_t = s, A_t \sim \pi]$
   $$V_{old} \leftarrow V_{new}$$
2) Only one copy of value function is stored during iteration, called in-place value iteration, where the newly updated state value will be used immediately for its neighbor state value update during the same iteration.
   For all s in S: $V(s) \leftarrow \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) | S_t = s, A_t \sim \pi]$

To search for **an optimal policy**, we can perform **policy iteration** through a series of **policy improvements**. After initializing the state value function, we update the function value based on one-step backup greedy policy, instead of a particular given policy. Similar to policy evaluation, there are two strategies for updating:

1) Two copies of value function are stored for old values and new values, respectively, during one iteration.
   For all s in S: $V_{new}(s) \leftarrow \max_a \mathbb{E}[R_{t+1} + \gamma V_{old}(S_{t+1}) | S_t = s, A_t = a]$
   $$V_{old} \leftarrow V_{new}$$
2) Only one copy of value function is stored during iteration, called in-place value iteration, where the newly updated state value will be used immediately for its neighbor state value update during the same iteration.
   For all s in S: $V(s) \leftarrow \max_a \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) | S_t = s, A_t = a]$

Please note that dynamic programming can be similarly applied to action-value functions, $q(s, a)$, which was not treated in this chapter but is summarized below,

---

Initial array Q(s,a) for all s and a
Repeat:
    $\Delta \leftarrow 0$
    For each s in S:
        For each a in A:
            $q(s, a) \leftarrow Q(s, a)$
            $Q(s, a) \leftarrow \sum_{s',r} P(s', r | s, a) \left(r + \gamma \max_{a'} Q(s', a') \right)$
            $\Delta \leftarrow \max(\Delta, |q(s, a) - Q(s, a)|)$
Until $\Delta \leq \theta$ (a small positive number)

Output a deterministic policy $\pi(s) = arg \max_a q(s, a)$

Chapter 4 Model-Free Prediction and Control

Prediction: Stanford CS234 lecture 3, David Silver's Lecture 4: Model-Free Prediction. SB Chapter/Sections: 5.1; 5.5; 6.1-6.3
Control: Stanford CS234 lecture 4, David Silver's Lecture 5. For additional reading please see SB Sections 5.2-5.4, 6.4, 6.5, 6.7
http://www.incompleteideas.net/book/ebook/ Sutton old book.

Learning Objective

In previous chapters, we assume that the model of MDP dynamics be available. In many of applications the MDP model is unknown but can be sampled, or the MDP model is known but it is computationally infeasible to use, except through sampling. In this case, the agent needs to learn the model from the sample interactions between the agent and the environment. In this chapter, we will address sample-based learning, which requires only experience – sample sequences of states, actins, and rewards from the interaction, without knowing the explicit model of the MDP dynamics. Thus, sample-based learning is model-free and typical reinforcement learning.
Upon the completion of this chapter, you should be able to

- Understand and implement MC policy evaluation: first-visit, and every-visit.
- Understand and implement TD learning for policy evaluation.
- Understand state-action function Q(s,a) evaluation for policy control.
- Compare and implement the following policy control (iteration) algorithms
  - ✓ MC method
  - ✓ TD(0), TD(λ)
  - ✓ TD control: SARSA, SARSA(λ)
  - ✓ Q-learning

4.1 Monte-Carlo Policy Evaluation

Let's consider Monte-Carlo (MC) policy evaluation. Its goal is to learn state value function v_π from **episodes** of experience under policy π: $S_1, A_1, R_2, \dots, S_k$. The return is the total discounted reward for an episode ending at time T>t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T$$

The value function is the expected return:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

MC policy evaluation use sample average return instead of expected return. Since a state may show up multiple times in one episode, there are two types of MC policy evaluation methods: 1) first-visit and 2) every-visit. For the first-visit one, the return for the state s is updated only for the state's first show up for an episode, i.e., thereafter show-ups in the episode will not result in the update of return. On the other hand, the every-visit one will update the return whenever the state shows up during an episode.

First-visit MC on policy evaluation

Initialize N(s) =0, G(s) =0, for all s ϵS

Loop (for each episode)

Sample episode i: $S_{i,1}, A_{i,1}, R_{i,2}, S_{i,2}, A_{i,2}, R_{i,3,\ldots,} A_{i,T_i-1} R_{i,T_i}, S_{i,T_i}$

Define: $G_{i,t} = R_{i,t+1} + \gamma R_{i,t+2} + \gamma^2 R_{i,t+3} + \cdots + \gamma^{T_i-t-1} R_{i,T_i}$ as the return from time step t to the end of episode i

For each state s visited in episode i

If s is visited at first time in episode i

Increment counter of total first visits: N(s)=N(s)+1

Increment total return G(s)=G(s)+$G_{i,t}$

Update estimate V^{π}(s)=G(s)/N(s)

Properties: It can be proved that the estimator V^{π}(s) is an unbiased estimator of true $\mathbb{E}_{\pi}[G_t|S_t = s]$. By law of large numbers, as N(s) →∞, V^{π}(s) →$\mathbb{E}_{\pi}[G_t|S_t = s]$

Every-visit MC on policy evaluation

Initialize N(s) =0, G(s) =0, for all s ϵS

Loop (for each episode)

Sample episode i: $S_{i,1}, A_{i,1}, R_{i,2}, S_{i,2}, A_{i,2}, R_{i,3,\ldots,} A_{i,T_i-1} R_{i,T_i}, S_{i,T_i}$

Define: $G_{i,t} = R_{i,t+1} + \gamma R_{i,t+2} + \gamma^2 R_{i,t+3} + \cdots + \gamma^{T_i-t-1} R_{i,T_i}$ as the return from time step t to the end of episode i

For each state s visited in episode i

For **every time t** that state s is visited in episode i

Increment counter of total visits : N(s)=N(s)+1

Increment total return G(s)=G(s)+$G_{i,t}$

Update estimate V^{π}(s)=G(s)/N(s)

Properties: the estimator V^{π}(s) is a **biased** estimator of true $\mathbb{E}_{\pi}[G_t|S_t = s]$. By law of large numbers, as N(s) →∞, V^{π}(s) →$\mathbb{E}_{\pi}[G_t|S_t = s]$

To evaluate the quality of policy estimation, we introduce two concepts: mean squared error (MSE) bias, variance, and consistency of estimation. In statistics, MSE is usually used to measure the accuracy of estimation. Let $\hat{V}$ be the estimate of V. The MSE, $\mathbb{E}\left[\left(\hat{V} - V\right)^2\right]$, can be decomposed into two components: bias and variance,

$$\mathbb{E}\left[\left(\hat{V} - V\right)^2\right] = \left(\mathbb{E}\left(\hat{V}\right) - V\right)^2 + \mathbb{E}\left[\left(\hat{V} - \mathbb{E}\left(\hat{V}\right)\right)^2\right] = bias^2 + variance$$

If bias =0 (.i.e. $\mathbb{E}\left(\hat{V}\right) = V$), the estimator is **unbiased**, otherwise **biased.**

Suppose we are interested in the value of some parameter V that describes a feature of a population. We draw a random sample from the population, X1, . . . , Xn, and have an estimator $\hat{V}$ which is a function of the sample: $\hat{V}=\hat{V}(X_1, X_2, \ldots, X_n)$. We would like $\hat{V}$ to get closer and closer to V as we draw larger and larger samples. An estimator $\hat{V}$ is said to be a **consistent** estimator of V if, for any positive ϵ,

$$\lim_{n\to\infty} P\left(\left|\hat{V} - V\right| \leq \epsilon\right) = 1$$

or, equivalently,

$$\lim_{n\to\infty} P\left(\left|\hat{V} - V\right| > \epsilon\right) = 0$$

We say that $\hat{V}$ **converges in probability** to V and we write $\hat{V} \xrightarrow{P} V$.
It can be shown that the first-visit MC is unbiased while the every-visit is biased, **but both are consistent**.

[**Example 1**] Mars rover: reward is R=[1, 0, 0, 0, 0, 0, +10] for any action, for when it is in states s1 to s7, respectively. $\pi(s) = A_1, \gamma = 1$, any action from s1 or s7 terminates episode.
If a trajectory $=\{S_3, A_1, 0, S_2, A_1, 0, S_2, A_1, 0, S_1, A_1, 1\ terminal\}$, find the first-visit MC estimate and the every-visit MC estimate of each state value V.

First-visit MC: V(s1)=1, V(s2)=1, V(s3)=1, V(s4)=V(s5)…=V(s7)=0
Every-visit MC: the same. V(s1)=1, V(s2)=1, V(s3)=1, V(s4)=V(s5)…=V(s7)=0
Note that if $\gamma \neq 1$,
First-visit MC: V(s1)=1, V(s2)=γ^2, V(s3)=γ^2, V(s4)=V(s5)…=V(s7)=0
Every-visit MC: the same. V(s1)=1, V(s2)=$0.5(\gamma + \gamma^2)$, V(s3)=γ^2, V(s4)=V(s5)…=V(s7)=0

[**Example 2**] Consider a two-state MDP, shown in Fig.1. The reward is 1 for moving to state s1 and 0 for moving to state s0. The probability of moving from s1 to s2 is p. s2 is the terminal state. Evaluate the state value function using model-based method (Bellman equation), first-visit MC method, and every-visit MC method. Compare first-visit and every-visit methods in terms of bias, variance and consistency.

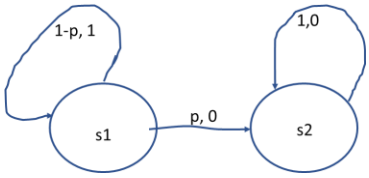


Fig.1 a two-state MDP

a) Model-based solution. With the knowledge of the model, we can use Bellman equations to calculate state value function. It is obvious that V(s2)=0. For V(s1), we have the Bellman equation

$$V(s_1) = p \cdot \left(0 + V(s_2)\right) + (1-p) \cdot \left(1 + V(s_1)\right)$$

By solving the equation, we get $V(s_1) = \frac{1}{p} - 1$.

If we use Monte Carlo method to evaluate the value function, the states in the episode is {s1, s1, s1, …, s1, s0}. It can be shown that the length of the state sequence in the episode, X, is a **geometric** random variable with an expectation 1/p.

b) Consider a single random episode. The return for the **first-visited** s1 is $\hat{V}(s_1) = X - 1$ because each state s1 receives a reward of 1. Since $\mathbb{E}\left[\hat{V}(s_1)\right] = \mathbb{E}[X-1] = \mathbb{E}[X] - 1 = \frac{1}{p} - 1 = V(s_1)$, this first-visit MC method is an unbiased estimator. The variance of $\hat{V}(s_1)$ is

$$\mathbb{E}\left[\left(\hat{V}(s_1) - \mathbb{E}\left[\hat{V}(s_1)\right]\right)^2\right] = \mathbb{E}\left[\left(X - \frac{1}{p}\right)^2\right] = \frac{1-p}{p^2}$$

c) Consider a single random episode. The return for **every-visited s1** is

$$\hat{V}(s_1) = \frac{1}{X-1}\sum_{i=1}^{X-1}(X-i) = \frac{X}{2}$$

Thus,

$$\mathbb{E}\left[\hat{V}(s_1)\right] = \mathbb{E}\left[\frac{X}{2}\right] = \frac{1}{2p} \neq V(s_1)$$

The every-visit MC method is a **biased** estimator. The variance of $\hat{V}(s_1)$ is

$$\mathbb{E}\left[\left(\hat{V}(s_1) - \mathbb{E}\left[\hat{V}(s_1)\right]\right)^2\right] = \mathbb{E}\left[\left(\frac{X}{2} - \frac{1}{2p}\right)^2\right] = \frac{1-p}{4p^2}$$

d) Comparison between first-visit and every-visit MC method. Based on the results in b) and c), the first-visit MC method is unbiased while the every-visit MC method is biased but with smaller variance.

e) Consistency: consider n episodes. For the first-visit MC method,

$$\hat{V}(s_1) = \frac{1}{n}(X_1 - 1 + X_2 - 1 + \cdots + X_n - 1) \to \mathbb{E}[X] - 1 = \frac{1}{p} - 1 = V(s_1)$$

Thus, it is consistent.

For the every-visit MC method,

$$\hat{V}(s_1) = \frac{\sum_{i=1}^{n}(1 + 2 + \cdots + X_i - 1)}{\sum_{i=1}^{n}(X_i - 1)} = \frac{1}{2}\frac{\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \frac{1}{n}\sum_{i=1}^{n} X_i}{\frac{1}{n}\sum_{i=1}^{n} X_i - 1} \to \frac{1}{2}\frac{\mathbb{E}[X^2] - \mathbb{E}[X]}{\mathbb{E}[X] - 1}$$

$$= \frac{1}{2}\frac{\sigma^2 + m^2 - m}{m - 1} = \frac{1}{p}$$

where $\sigma^2 = \frac{1-p}{p^2}$ and $m = \frac{1}{p}$ are variance and mean of X, respectively.

Question? $\hat{V}(s_1)$ not $\to$ $V(s_1) = \frac{1}{p} - 1$, not consistent?

Equivalently, we update the value function incrementally based on the previous value function and the returns in the current episode, instead of by computing the total return over all episodes. If we know the previous estimate $V_{old}(s)$, the total number of visit (including a new visit), N(s), and the return for the new visit, G_t, then the new value estimate can be represented as

$$V_{new}(s) = \frac{V_{old}(s) \cdot (N(s) - 1) + G_t}{N(s)} = V_{old}(s) + \frac{1}{N(s)}(G_t - V_{old}(s))$$

Incremental MC on policy evaluation

Initialize N(s) =0, G(s) =0, for all s ϵS

Loop (for each episode)

Sample episode i: $S_{i,1}, A_{i,1}, R_{i,2}, S_{i,2}, A_{i,2}, R_{i,3,\ldots,} A_{i,T_i-1} R_{i,T_i}, S_{i,T_i}$

Define: $G_{i,t} = R_{i,t+1} + \gamma R_{i,t+2} + \gamma^2 R_{i,t+3} + \cdots + \gamma^{T_i - t - 1} R_{i,T_i}$ as the return from time step t to the end of episode i

For each state s visited in episode i

For every-visit of state s at t (or for first-visit)

Increment counter of total visits: N(s)=N(s)+1

Update estimate based on old V(s) and current return $G_{i,t}$

$$v^{\pi}(s) \leftarrow v^{\pi}(s) + \frac{1}{N(s)}\left(G_{i,t} - v^{\pi}(s)\right)$$

In general, we can use a parameter α to replace $\frac{1}{N(s)}$ in the update equation. If $\alpha > \frac{1}{N(s)}$, put low weights on older data (or heavy weights on current data), thus being helpful for non-stationary domains.

Summary: MC updates the value estimate using a sample of the return to approximate an expectation, instead of using explicit dynamic model.

Limitations of MC: generally high variance estimator. Reducing variance requires a lot of data. Episode must end before the data can be used to update the value function. Thus, MC methods are applicable only to episodic tasks.

4.2 Temporal difference learning for estimating V

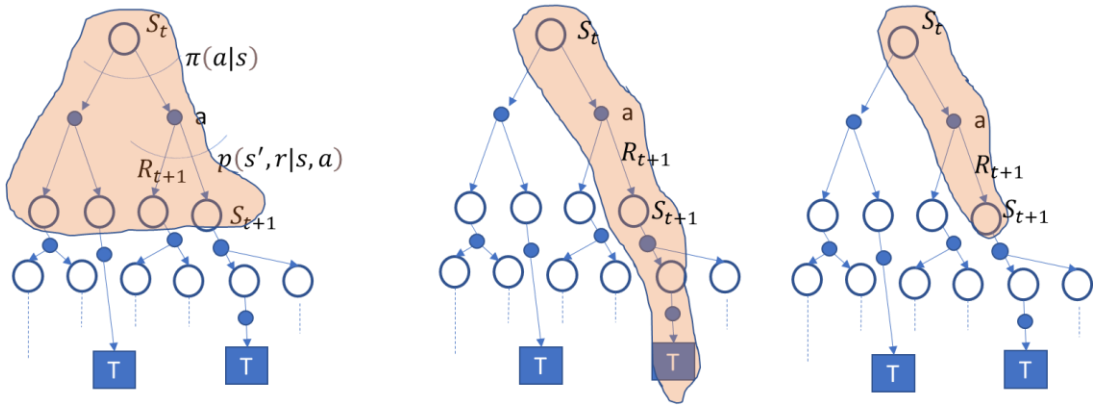
If one had to identify one idea as central or fundamental to reinforcement learning, it would undoubtedly be temporal difference (TD) learning. TD learning is a combination of MC ideas and dynamic programming ideas. -- Sutton and Barto 2017. TD learning learns value function V (s) directly from experience with TD error, with bootstrapping, in a model-free, online, and fully incremental way. The subsequent control methods, such as SARSA and Q-learning, are directly inspired by TD learning.

The goal of TD learning is the same as MC: to estimate $v^{\pi}(s)$ given episodes generated under policy π, but with a different approach. Define the state value function as

$$v^{\pi}(s) = \mathbb{E}_{\pi}[G_t|S_t = s], where\ G_t = R_{t+1} + \gamma R_{t+2} + \cdots$$

When the MDP model is available, i.e., the policy $\pi(a|s)$ and dynamics $p(s', r|s, a)$ are given, we can use Bellman equations or dynamic programming (discussed in the previous chapter) to evaluate the state value function $v^{\pi}(s)$, as shown in Fig.2(a)

$$V^{\pi}(S_t) \leftarrow \mathbb{E}_{\pi}[R_{t+1} + \gamma V^{\pi}(S_{t+1})] \qquad \text{(Dynamic programming)}$$



(a) Dynamic programming (b) Monte Carlo update (c) TD learning

Fig.2 comparison for dynamic programming, MC method, and TD learning

If the MDP model is not available, we can evaluate the state value function through the information on the received return during the experimental interaction with the environment, as shown in Fig.2(b). For instance, in every-visit MC, we evaluate the state value function by averaging the returns over every-visits of the state during all episodes, and this can be implemented in an incremental style

$$V^{\pi}(S_t) \leftarrow V^{\pi}(S_t) + \alpha\big(G_t - V^{\pi}(S_t)\big), \text{ where } \alpha = \frac{1}{N(S_t)} \qquad \text{(Monte Carlo update)}$$

where G_t is a sample of return following state St, N(St) is the total number of visits of state St till to time t.

Like MC, TD learning similarly updates the value estimate based on episode. But in TD, we estimate the return for state s at timestep t as the immediate reward plus discounted old value of the next state S_{t+1}, as shown in Fig.2(c), so that we can update the estimate without waiting for the entire episode.

$$V^{\pi}(S_t) \leftarrow V^{\pi}(S_t) + \alpha\big(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma V^{\pi}(S_{t+1})} - V^{\pi}(S_t)\big) \qquad \text{(TD update)}$$

We define TD target and TD error, respectively, as

TD target: $\boldsymbol{R_{t+1}} + \boldsymbol{\gamma V^{\pi}(S_{t+1})}$
TD error: $\boldsymbol{\delta_t} = \boldsymbol{R_{t+1}} + \boldsymbol{\gamma V^{\pi}(S_{t+1})} - V^{\pi}(S_t)$

The TD target is a new estimate of the value function based on the immediate reward and the old value of the subsequent state. The resulting TD error will be used to adjust the value of the current state. α is a learning rate. A closer inspection of the update equation reveals that the only value changed is the one associated with St, i.e., the state just visited. Further, the value of St is moved towards the "TD target". Since the target depends on the estimated value function, the algorithm uses **bootstrapping**. The term "**temporal difference**" in the name of the algorithm comes from that TD error, which is defined as the difference between values of states corresponding to successive time steps. In general, the TD target can be evaluated based on multiple-step return, though the scenario here is one-step return and called TD(0).

TD(0) learning algorithm

Input : the policy π to be evaluated, learning rate α
Output: value function V^{π}
Initialize $V^{\pi}(s) = 0 \; for \; all \; s$
for each episode **do**
  for each step t of episode, state S_t is not terminal **do**
    - Take action A_t by following policy π
    - Observe sample tuple $(S_t, A_t, R_{t+1,} S_{t+1})$
    - $V^{\pi}(S_t) \leftarrow V^{\pi}(S_t) + \alpha\big(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma V^{\pi}(S_{t+1})} - V^{\pi}(S_t)\big)$
    - $S_t \leftarrow S_{t+1}$
  end
end

[Example 3] Mars rover: R=[1, 0, 0, 0, 0, 0, +10] for any action, for s1 to s7, respectively. $\pi(s) = A_1, \gamma = 1$, any action from s1 and s7 terminates episode.

Trajectory ={$S_3, A_1, 0, S_2, A_1, 0, S_2, A_1, 0, S_1, A_1, 1, terminal$}
First-visit MC: V(s) = [1, 1, 1, 0, 0, 0, 0]
Every-visit MC: V(s)=[1, 1, 1, 0, 0, 0, 0]
TD with α=1: V(s)=[1, 0, 0, 0, 0, 0, 0]

The comparison between MC and TD, in terms of advantages and disadvantages, is summarized as the following table. (David Silver lecture 4).

	Monte Carlo method	**TD method**
Sequence completeness	Must wait until end of episode Can only learn from complete sequences Only works for episodic environments	Can learn online after every step Can learn from incomplete sequences Works in continuing environments
Variance and bias	High variance, zero bias (even with function approximation) Not very sensitive to initial value	Low variance, some bias Usually more efficient than MC TD(0) converges to True Vπ(s), but does not always converge with function approximation More sensitive to initial value
Markov property	Does not exploit Markov property MC converges to solution with minimum mean-squared error	Usually more efficient in Markov environments TD(0) converges to solution of max likelihood Markov model

Property comparison between DP, MC, TD

Properties	**DP**	**MC**	**TD(0)**
Sampling		√	√
Bootstrapping	√		√
Usable when no models of current domain		√	√
Handles continuing (non-episodic) domains	√		√
Handles non-Markovian domains		√	
Converges to true value in limit (Markov)	√	√	√
Unbiased estimate of value	Exact	√ (first-visit)	

4.3 TD(λ)

TD(0) can be generalized to n-step return update: let TD target look n steps into the future. The generalized TD methods are called TD(λ), where $\lambda \in [0,1]$ is a parameter that allows one to interpolate between the Monte-Carlo and TD(0) updates: λ = 0 gives TD(0) (hence the name of TD(0)), while λ = 1, i.e., TD(1) is equivalent to a Monte-Carlo method.

Consider the following n-step returns for n =1,2,…,

n=1 (TD) $G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$

n=2 (TD) $G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$

n=∞ (MC) $G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T$

In general, we define the n-step return as

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

The n-step TD learning update can be implemented by

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(n)} - V(S_t) \right)$$

A particular n-step TD method may not be always better than others. It is intuitive to average all n-step returns to generate an overall return. To combine all possible n-step returns (n =1,2...), a parameter λ is introduced to specify the weight for each return so that the overall return, TD(λ) return, is the weighted sum of all returns, defined by

$$G_t^{\lambda} = (1-\lambda)G_t^{(1)} + (1-\lambda)\lambda G_t^{(2)} + \cdots + (1-\lambda)\lambda^{n-1}G_t^{(n)} + \lambda^{T-t-1}G_t^{(\infty)}$$
$$= (1-\lambda)G_t^{(1)} + (1-\lambda)\lambda G_t^{(2)} + \cdots + (1-\lambda)\lambda^{n-1}G_t^{(n)} + \lambda^{n}G_t^{(\infty)}$$
$$= (1-\lambda)\sum_{i=1}^{n}\lambda^{i-1}G_t^{(i)} + \lambda^{n}G_t^{(\infty)}$$

where $n = T-t-1$, T is the length of the episode.

The weights for $G_t^{(1)}\ G_t^{(2)} \ldots G_t^{(n)}\ G_t^{(\infty)}$ are $(1-\lambda)$, $(1-\lambda)\lambda$, …, $(1-\lambda)\lambda^{n-1}$, λ^n, respectively. It is clear that the sum of all weights is equal to one. Fig.3 shows a case of n=3. Fig.4 illustrates the weight distribution over the n-step returns. TD(λ) is reduced to TD(0) when λ=0, and to MC methods when λ=1. From Fig.4, we can see that more weights are put on shorter step returns.

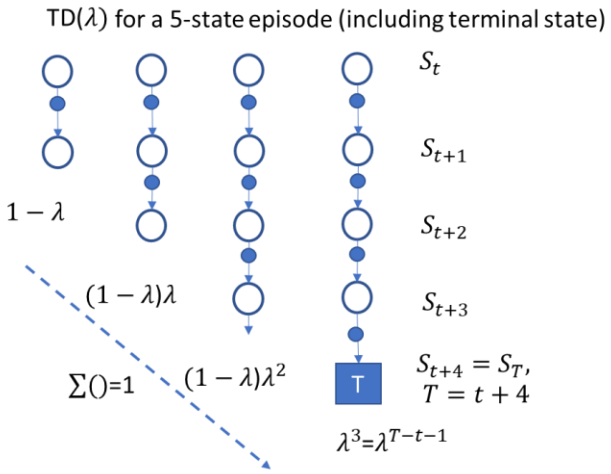


Fig.3 TD(λ) return for n=3

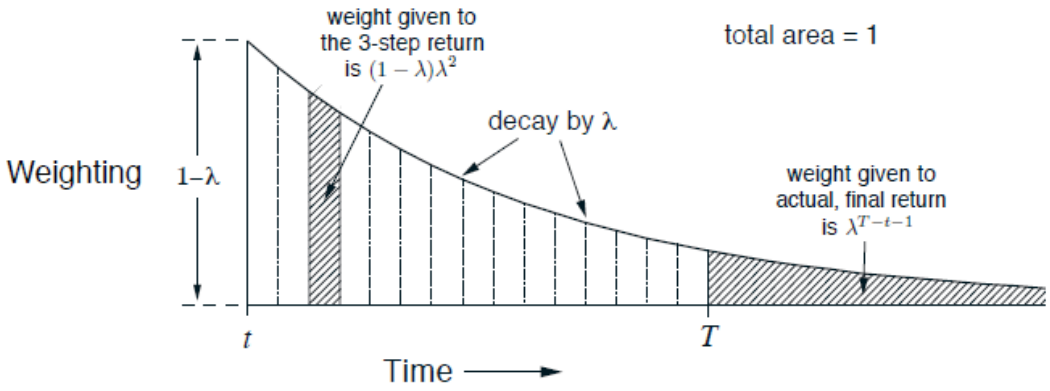


Fig.4 Weighting given in the λ-return to each of the n-step returns (from Sutton book)

We are now ready to define a learning algorithm based on the λ-return: the ***offline*** λ-return algorithm. As an offline algorithm, it makes no changes to the values of the states during the episode because G_t^λ is not available until the end of the episode. At the end of the episode, a whole sequence of offline updates are made for the states visited in the episode. The TD(λ) update is given by

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t^\lambda - V(S_t)\right)$$

Note that the update only applies to the state occurring on the step t, i.e., the values of other states are unchanged, and that this update is available at the end of the episode.

This approach is called the theoretical or ***forward view***, illustrated in Fig.5. Imagine that we ride the stream of states, looking forward from each state to determine its update. After update one state, we move on to the next and never look backward with the preceding state again.

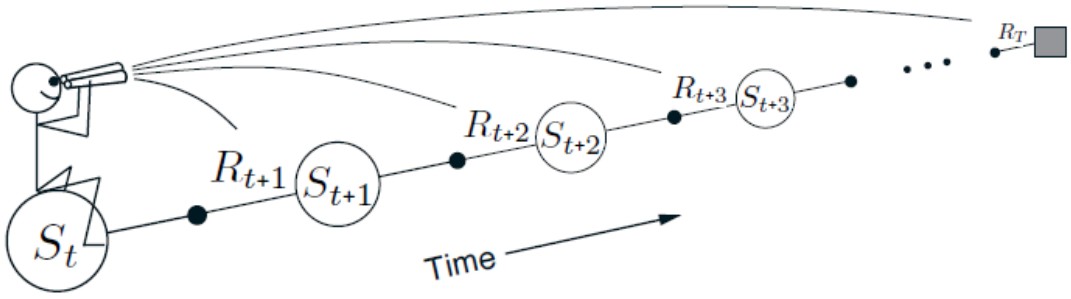


Fig.5 the forward view of TD(λ) algorithm (Sutton book)

The forward view of TD(λ) is not directly implementable online because it is non-causal, using at each step knowledge of what will happen many steps later. Now we introduce a mechanistic, or ***backward view*** of TD(λ) that is simple computationally and implementable either online or offline. As we will see later, the backward view provides a causal, incremental mechanism for approximating the forward view, and in the off-line case, for achieving it exactly.

In the backward view of TD(λ), there is a function associated with each state, its *eligibility trace*. The eligibility trace keeps the record of how "recently" the state has been visited. Specifically, the eligibility trace for state s at time t is denoted $E_t(s) \in \mathbb{R}^+$. On each step, the eligibility traces for all states decay by $\gamma\lambda$, and the eligibility trace for the state visited on the step is incremented by 1,

$$E_t(s) = \begin{cases} \gamma\lambda E_{t-1}(s) & if\ s \neq s_t \\ \gamma\lambda E_{t-1}(s) + 1 & if\ s = s_t \end{cases}$$

The eligibility trace is also called an accumulating trace because it accumulates each time the state is visited, then fades away gradually, illustrated in Fig.6.

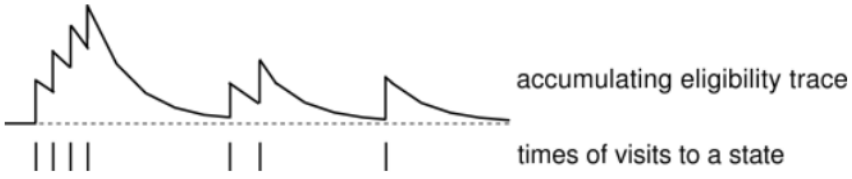


Fig.6 sketch of an eligibility trace

The traces indicate the degree to which each state is eligible for undergoing learning changes should a reinforcing event occur. The reinforcing events are the one-step TD errors,

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

In the backward view of TD(λ), the global TD error (applicable to all states) signal triggers proportional updates to all recently visited states

$$V(S) \leftarrow V(S) + \alpha \delta_t E_t(s) \qquad \text{for all s}$$

The backward view of TD(λ) is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to the state's eligibility trace at that time. Imagine riding along the stream of states, computing TD errors, and shouting them back to the previously visited states, as illustrated by Fig.7.

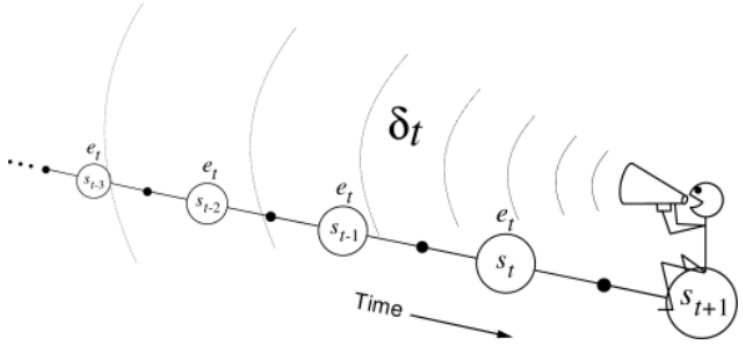


Fig.7 the backward view of TD(λ)

This update can be done on each step to form an online algorithm. A complete algorithm for online TD(λ) is given below,

Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in \mathcal{S}$
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r, and next state, s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    For all s:
      $V(s) \leftarrow V(s) + \alpha \delta e(s)$
      $e(s) \leftarrow \gamma \lambda e(s)$
    $s \leftarrow s'$
  until s is terminal

To better understand the backward view, consider the scenarios for various values of λ. When λ=0, all traces are zero at t except the trace of St. Thus the TD(λ) reduces to the one-step return algorithm, TD(0). For larger values of λ (but less than 1), the preceding states that are temporally farther from the current state change less because the earlier states are given less credit for the TD error. When λ=1, TD(1), then the credit given to earlier states falls only by γ per step. TD(1) is a incremental and online way of implementing

MC algorithms whereas MC methods are limited to episodic tasks and learn nothing until the episode is over. It has been shown that off-line TD(λ), as the backward view defined mechanistically above, achieves the same updates as the off-line λ-return algorithm (forward view). The details can be found at the early version of Sutton's book chapter 7 (http://www.incompleteideas.net/book/ebook/node76.html).

In summary, eligibility traces unify and generalize TD and Monte Carlo methods. In TD(λ), the λ refers to the use of an eligibility trace. When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has MC methods at one end (λ=1) and one-step TD methods at the other (λ=0). In between are intermediate methods that are often better than either extreme method. Eligibility traces also provide a way of implementing MC methods online and on continuing problems without episodes.

4.4 Model free control

All previous sections in this chapter addressed a problem of policy evaluation, i.e., evaluating the state value function given a policy. From this section on, we will describe how to search for an optimal policy, given an environment to be sampled. As we mentioned earlier, the goal of reinforcement learning is to learn to select actions to maximize the total expected future reward. This learning process is also called policy control. The purpose of control is to improve the policy and eventually reach the optimal policy. Let's recall policy iteration based on a known model. First, we initialize policy π (e.g. random policy), and then repeat the evaluation and improvement cycles:

- Policy evaluation: compute V^{π}
- Policy improvement: update π, as shown in Fig.8.

$$\pi'(s) = arg\max_a q^{\pi}(s,a) = arg\max_a \{R(s,a) + \gamma \sum_{s' \epsilon S} p(s'|s,a)V^{\pi}(s')\}$$

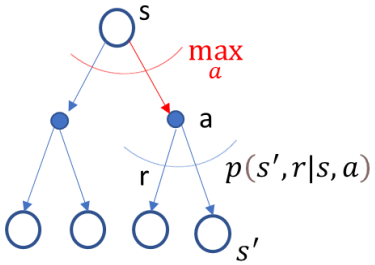


Fig.8 Model-based policy improvement

Now we want to do the above two steps by interacting with the environment, without the knowledge of the true dynamics and reward models. Without the explicit model of the dynamics, it is particularly useful to evaluate the state-action value, Q(s,a), rather than the state-value, V(s), for the policy improvement, because the policy improvement based on V(s) specified in the above equation requires the model $p(s'|s,a)$.

Model-free generalized policy improvement

If the model of MDP is available, we can evaluate Q(s,a) function using either Bellman equations or dynamic programming for a given policy. Then, based on the evaluated Q function, we can improve the policy by taking the greedy action, *a*, resulting in the maximal Q(s,a) value for the state, *s*. If the MDP model is unknown, as we will see later, we can evaluate the Q(s,a) for a given policy based on sampling

methods. If the Q function is available for a policy, we can update the policy for an improvement by greedy action selection.

However, a greedy policy is deterministic. If π is deterministic for a state s, we will not be able to compute Q(s,a) for any $a \neq \pi(s)$ using MC episodes since the deterministic policy never leads to those pairs (s,a) for $a \neq \pi(s)$. In order to update the Q(s,a) function in the entire (s,a) space during the policy iteration, we just need to simply replace the greedy policy in the policy update stage with the corresponding ***ϵ-greedy policy***.

We use ***ϵ-greedy policy*** to balance **exploration** and **exploitation**, that is, to make sure all (s,a) pairs to be covered and greed policy π to be almost followed except that taking non-maximal actions occasionally and randomly. In other words, we take actions for the highest return at most of time, but also explore some new actions occasionally. Let |A| be the number of actions available for state s, then an ϵ-greedy policy w.r.t. a state-action value $Q^{\pi}(s,a)$ is

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & if\ a = arg\max_a Q^{\pi}(s,a) \\ \frac{\epsilon}{|A|} & if\ a \neq arg\max_a Q^{\pi}(s,a) \end{cases}$$

[ϵ-greedy policy theorem] Let π_{i+1} be the ϵ-greedy policy w.r.t $Q^{\pi_i}(s,a)$, where π_i is the old ϵ-greedy policy. π_{i+1} is a monotonic improvement, i.e., $V^{\pi_{i+1}} \geq V^{\pi_i}$.

[proof]

$$V^{\pi_{i+1}}(s) = Q^{\pi_i}\big(s, \pi_{i+1}(s)\big) = \sum_{a \in A} \pi_{i+1}(a|s) Q^{\pi_i}(s,a) = \sum_a \frac{\epsilon}{|A|} Q^{\pi_i}(s,a) + (1-\epsilon) \max_a Q^{\pi_i}(s,a)$$

$$= \sum_a \frac{\epsilon}{|A|} Q^{\pi_i}(s,a) + (1-\epsilon) \max_a Q^{\pi_i}(s,a) \frac{1-\epsilon}{1-\epsilon}$$

$$= \sum_a \frac{\epsilon}{|A|} Q^{\pi_i}(s,a) + (1-\epsilon) \max_a Q^{\pi_i}(s,a) \frac{\sum_{a \in A}\left(\pi_i(a|s) - \frac{\epsilon}{|A|}\right)}{1-\epsilon}$$

$$\geq \sum_a \frac{\epsilon}{|A|} Q^{\pi_i}(s,a) + \sum_{a \in A}\left(\pi_i(a|s) - \frac{\epsilon}{|A|}\right) Q^{\pi_i}(s,a)$$

$$= V^{\pi_i}(s)$$

This theorem assumes that Q^{π_i} has been fully evaluated. However, this assumption is not required in practical policy iteration. Instead, as soon as the Q-value has been updated by one episode (MC methods) or one time step (SARSA, Q-learning), the e-greedy policy improvement can be applied, without waiting for a full evaluation of Q(s,a).

To guarantee the policy improvement process converges to the optimal policy, we define the condition: Greedy in the Limit of Infinite Exploration (**GLIE**):

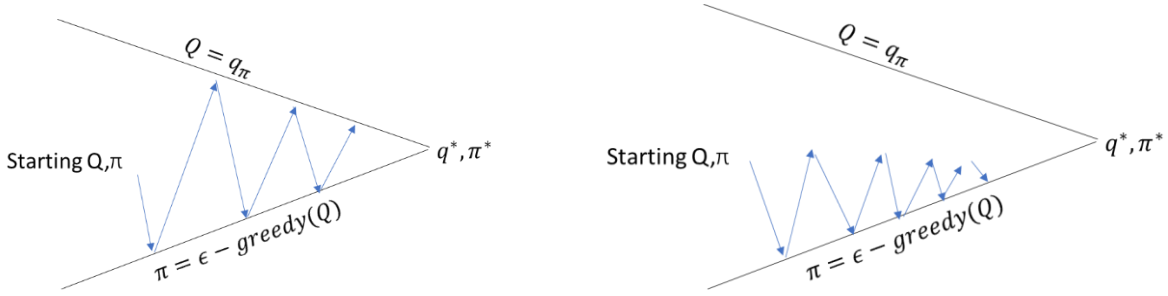
1) all state-action pairs are visited an infinite number of times
$$\lim_{i \to \infty} N_i(s,a) \to \infty$$
2) behavior policy converges to greedy policy
$$\lim_{i \to \infty} \pi_i(a|s) \to arg\max_a Q(s,a)$$

The ϵ-greedy is GLIE, where ϵ is reduced to 0 with the rate $\epsilon_i = \frac{1}{i}$.

As illustrated in Fig.9, this policy improvement process consists of iterations of two alternative steps:

$$\pi_0 \to Q_0 \to \pi_1 \to Q_1 \to \pi_2 \to \cdots$$

- Estimate $Q^{\pi_i}(s,a)$
- Update new policy by e-greedy with respect to $Q^{\pi_i}(s,a)$



(a) theoretical (b) practical: partial evaluation and more frequently update

Fig.9 Model-free generalized policy improvement. (a) full evaluation of q(s,a) before policy update, (b) partial evaluation of q(s,a) before policy update

4.4.1 MC methods for control

Estimation of Q-values

It is intuitive to evaluate Q function for a given policy, in a similar way of MC state function V(s) evaluation:

Initialize N(s,a)=0, G(s,a)=0, $Q^\pi(s,a) = 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$

Loop

- Using policy π to sample episode i: $S_{i,1}, A_{i,1}, R_{i,2}, S_{i,2}, A_{i,2}, R_{i,3,\ldots,} A_{i,T_i-1} R_{i,T_i}, S_{i,T_i}$
- $G_{i,t} = R_{i,t+1} + \gamma R_{i,t+2} + \gamma^2 R_{i,t+3} + \cdots + \gamma^{T_i - t - 1} R_{i,T_i}$
- For each state-action (s,a) visited in episode i

$$N(s,a) \leftarrow N(s,a) + 1$$
$$G(s,a) \leftarrow G(s,a) + G_{i,t}$$
$$Q^\pi(s,a) \leftarrow G(s,a)/N(s,a)$$

The problem of this algorithm for Q evaluation is that it cannot compute Q(s,a) for any action excluded by the policy, i.e. $a \neq \pi(s)$. To deal with this problem, we can start each episode with a random (s,a) pair and then follow the policy. But a better or more practical solution will be ϵ-greedy policy interleaved in the policy iteration, discussed below.

[Example 4] MC for on policy Q evaluation

Mars rover with new actions: reward is defined by r(-, a1)=[1, 0, 0, 0, 0, 0, +10], r(-, a2)=[0,0,0,0,0,0,+5] (note that – represent the sequence of states: s1, s2, …, s7), γ=1. Assume current greedy π(s)=a1, ∀s, ϵ=0.5. Sample trajectory from ϵ-greedy policy.

Trajectory ={s3, a1, 0, s2, a2, 0, s3, a1, 0, s2, a2, 0, s1, a1, 1, terminal}

First visit MC estimate of Q of each (s, a) pair?

Answer: Q(-, a1)=[1, 0, 1, 0, 0, 0, 0], Q(-, a2)=[0, 1, 0, 0, 0, 0, 0]

MC Online Control / On Policy Improvement

On-policy MC control follows a generalized policy iteration scheme. For policy evaluation, it uses MC policy evaluation for the action value. For policy improvement, it uses e-greedy policy improvement. Specifically, first, we initialize the Q function and visit counter for all (s,a) pairs. In the loop iterating over episodes, we sample an episode using ϵ-greedy policy with respect to current Q function, and then use MC method (first-visit or every visit) to evaluate (update) Q function based on the episode, finally the ϵ-greedy policy will be updated according to the updated Q function, as shown in Fig.10.

[**Theorem**] GLIE MC control converges to the optimal action-value function, $Q(s,a) \rightarrow q^*(s,a)$

Algorithm: GLIE Monte Carlo Control (from Stanford CS234 lecture video)

Initialize Q(s,a)=0, N(s,a) =0, for all (s,a), set ϵ=1, k=1
$\pi_k = \epsilon - greedy(Q)$ //create initial ϵ-greedy policy
loop
Sample k-th episode $(s_{k,1}, a_{k,1}, R_{k,2}, s_{k,2}, \dots, s_{k,T})$ given π_k
Compute return for each step t in this episode: $G_{k,t} = R_{k,t+1} + \gamma R_{k,t+2} + \cdots + \gamma^{T-1} R_{k,T}$
for t=1,…,T **do**
if first visit to (s,a) in episode k **then** // could do every visit
N(s,a) = N(s,a)+1
$Q(s_t, a_t) = Q(s_t, a_t) + \frac{1}{N(s,a)}\left(G_{k,t} - Q(s_t, a_t)\right)$
end if
end for
k=k+1, ϵ=1/k
$\pi_k = \epsilon - greedy(Q)$ //policy improvement
end loop

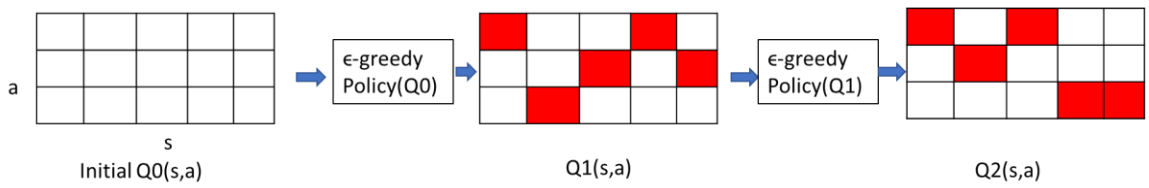


Fig.10 MC on-policy control (from Stanford CS234)

[Example 5] Mars rover with new actions: r(-, a1)=[1, 0, 0, 0, 0, 0, +10], r(-, a2)=[0,0,0,0,0,0,+5], γ=1. Assume current greedy π(s)=a1, ∀s, ϵ=0.5. Sample trajectory from ϵ-greedy policy.

Trajectory ={ s3, a1, 0, s2, a2, 0, s3, a1, 0, s2, a2, 0, s1, a1, 1, terminal}

1) First visit MC estimate of Q of each (s, a) pair?

Answer: Q(-, a1)=[1, 0, 1, 0, 0, 0, 0], Q(-, a2)=[0, 1, 0, 0, 0, 0, 0]

2) What is $\pi(s) = arg\max_a Q^{\epsilon-\pi}(s,a)$?

   Answer: [a1,a2, a1, a1/a2, a1/a2, a1/a2, a1/a2]

3) What is the new ϵ-greedy policy, if k=3, ϵ=1/k ?

   Answer: π(-, a1)=[1-1/3+1/6 (e.g.5/6), 1/6, 5/6, ½, ½, ½, ½]

   π(-, a2)=[1/6, 5/6, 1/6, ½, ½, ½, ½]

4.4.2 SARSA: On-policy Iteration with TD Method

In general, temporal difference (TD) learning has several advantages over Monte-Carlo methods, such as lower variance, online, and incomplete sequences. It is a natural idea to use TD instead of MC in our control loop: apply TD to estimate Q(S,A), and use ϵ-greedy policy improvement. We can update the action value and the policy every time-step.

As we discussed earlier, we can use TD methods to update the state (St) function value based on the old function values of state S_t and state S_{t+1} and immediate reward R_{t+1}: (St, At, Rt+1, St+1), with bootstrapping:

$$V^{\pi}(S_t) \leftarrow V^{\pi}(S_t) + \alpha\big(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma V^{\pi}(S_{t+1})} - V^{\pi}(S_t)\big)$$

We will similarly use TD methods to evaluate Q functions.

$$Q^{\pi}(S_t, A_t) \leftarrow Q^{\pi}(S_t, A_t) + \alpha\big(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma Q^{\pi}(S_{t+1}, A_{t+1})} - Q^{\pi}(S_t, A_t)\big)$$

By combining TD method for Q evaluation and ϵ-greedy policy for policy improvement, we can develop an algorithm for model-free policy iteration, called SARSA, since the update is based on the quintuple (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}). In SARSA algorithm, both actions A_t and A_{t+1} are selected using ϵ-greedy policy. SARSA is an **on-policy** algorithm because it evaluates the policy being followed. The general form of SARSA algorithm is described in Fig.11.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

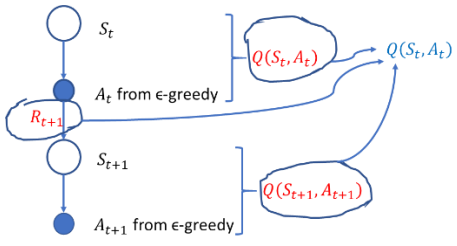


Fig. 11 SARSA algorithm: (a) algorithm (from Sutton book) (b) Backup diagram

Compared to MC method, SARSA updates the Q function and policy at every timestep based on a part of an episode, (St, At, R_{t+1}, S_{t+1}, A_{t+1}), while MC methods require an entire episode for one update.

Convergence properties of SARSA (theorem): SARSA for finite-state and finite-action MDPs converges to the optimal action-value, under the following conditions:

1) The policy sequence satisfies the condition of GLIE
2) The step-sizes α_t satisfy the Robbins-Munro sequence such that

   $\sum_{t=1}^{\infty} \alpha_t = \infty$ $\quad$ $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$

4.4.3 SARSA(λ)

Just like TD(λ) using all n-step TD returns for policy prediction, SARSA(λ) generalizes the one-step SARSA algorithm by combining all n-step returns and eligibility traces for policy control.

Consider the following n-step returns of action values for n =1,2,…,

n=1 (SARSA) $q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

n=2 $q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A_{t+2})$

n=∞ (MC) $q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T$

In general, we define the n-step Q-return as

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

The n-step SARSA updates $Q(S_t, A_t)$ towards the n-step Q-return

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^{(n)} - Q(S_t, A_t) \right)$$

The Q-return for SARSA(λ) is defined as the combination of all n-step Q-returns $q_t^{(n)}$ using weight $(1-\lambda)\lambda^{n-1}$, illustrated in Fig. 12,

$$q_t^{\lambda} = (1-\lambda) \sum_{i=1}^{n} \lambda^{i-1} q_t^{(i)} + \lambda^n q_t^{(\infty)}$$

where *n =T-t-1*, T is the length of the episode.

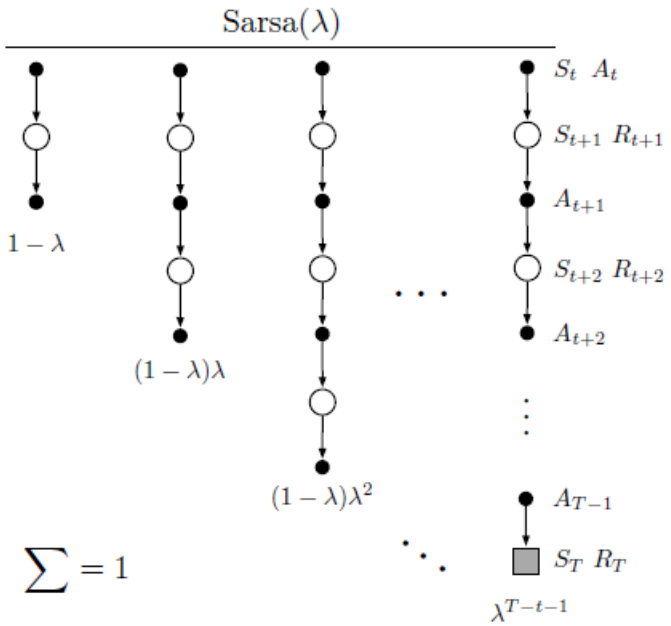


Fig.12 SARA(λ) backup diagram (Sutton book)

The update for the theoretical or ***forward view*** of SARSA(λ) is given

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(q_t^{\lambda} - Q(S_t, A_t)\right)$$

This forward view only updates the Q-value of the current (S_t, A_t) at one time step and requires the entire episode. Just like TD(λ), we use eligibility traces to implement the forward view through the backward view which is an online algorithm. The ***backward view*** will update all Q-values at each time step without waiting for the entire episode, and at the end of the episode the total accumulated updates are equivalent to the updates specified in forward view. The eligibility trace for (s,a) at time step t is defined as

$$E_t(s,a) = \begin{cases} \gamma\lambda E_{t-1}(s,a) + 1 & if\ s = s_t, a = a_t \\ \gamma\lambda E_{t-1}(s,a) & otherwise \end{cases}$$

Then Q(s,a) is updated for ***every*** state-action pair (s,a) at time step t,

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t E_t(s,a)$$

where δ_t is the current local TD-error at time step t,

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

SARSA(λ) Algorithm (David lecture 5)

Initialize Q(s,a) arbitrarily, for all s,a
Repeat (for each episode):
    E(s,a)=0, for all s, a
    Initialize S, A,
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g. e-greedy)
        $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
        $E(S, A) \leftarrow E(S, A) + 1$
        For all s, a:
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
            $E(s,a) \leftarrow \gamma\lambda E(s,a)$
        $S \leftarrow S'; A \leftarrow A'$
    until S is terminal

Compared to SARSA(0) (i.e. one-step SARSA), SARSA(λ) can substantially increase the efficiency of updating process. This is illustrated by the gridworld example in Fig.13. The first panel shows the trajectory taken by an agent in a single episode, ending at the goal with high reward, marked by *. We assume the values were all initialized to 0, and all rewards were zero except for a positive reward at the goal. The arrows in the other two panels show which action values were increased as a result of this episode by one-step SARSA and SARSA(λ) methods, respectively. The one-step SARSA strengthens only the last action of the sequence of actions that led to the high reward, whereas the trace method strengthens many actions of the sequence. The degree of strengthening (indicated by the size of the arrows) fading with steps from the reward.

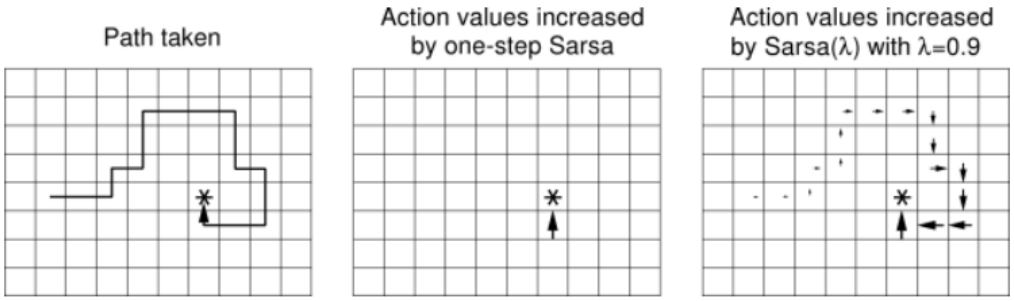


Fig. 13 Comparison of update efficiency between one-step SARSA and SARSA(λ)
(Sutton book old version Example 7.4)

4.4.4 Q-learning: off-policy

One of the most important breakthroughs in reinforcement learning was the development of an off-polcy TD control algorithm known as Q-learning (Watkins, 1989). In Q-learning, we use ϵ-greedy to generate (S_t, A_t, R_{t+1}, S_{t+1}), and then update Q(S_t, A_t) by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

GLIE is the condition for Q-learning to converge. In Q-learning, the Q-function is updated based on the four-tuple SARS: (S_t, A_t, R_{t+1}, S_{t+1}). Q-learning is an **off-policy** learning algorithm because it evaluates a policy (greedy policy) using experience gathered by following another policy (e-greedy). Q-learning updates are done regardless to the actual action chosen for next state, it just assumes that we are always choosing the argmax one. So if we are using epsilon greedy policy and chose a random action for the action on the next state, instead of the argmax action, Q-learning will still update its estimation according to the argmax action. SARSA, on the other hand, will use this random action to be next one to update Q(s,a).

[Example 6] Mars rover with new actions: r(-, a1)=[1, 0, 0, 0, 0, 0, +10], r(-, a2)=[0,0,0,0,0,0,+5], γ=1. Assume current greedy π(s)=a1, ∀s, ϵ=0.5. Sample trajectory from ϵ-greedy policy.

Trajectory ={ s3, a1, 0, s2, a2, 0, s3, a1, 0, s2, a2, 0, s1, a1, 1, terminal}

1) First visit MC estimate of Q of each (s, a) pair?

Answer: Q(-, a1)=[1, 0, 1, 0, 0, 0, 0], Q(-, a2)=[0, 1, 0, 0, 0, 0, 0]

2) What is $\pi(s) = arg\max_a Q^{\epsilon-\pi}(s,a)$? (greedy policy for the estimated Q in 1))

Answer: [a1,a2, a1, a1/a2, a1/a2, a1/a2, a1/a2]

3) What is the new ϵ-greedy policy, if k=3, ϵ=1/k ? (based on the estimated Q in 1))

   Answer: π(-, a1)=[1-1/3+1/6 (e.g.5/6), 1/6, 5/6, ½, ½, ½, ½]

   π(-, a2)=[1/6, 5/6, 1/6, ½, ½, ½, ½]

4) Q-learning updates? Initialize ϵ=1/k, k=1, and α=0.5, policy π: random with probability ϵ, else π=[1, 1, 1, 2, 1, 2, 1], first tuple: (s3, a1, 0, s2).
   Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

4.5 Example 7: Mountain Car by Q-learning

The task of mountain car is one of the most popular examples for reinforcement learning. Consider driving a car up a steep mountain road. However, the gravity is stronger than the car's engine even at full throttle, illustrated by Fig.14. The only solution is to move away from the goal and up the opposite slope to accumulate enough momentum, and then move towards the goal direction.

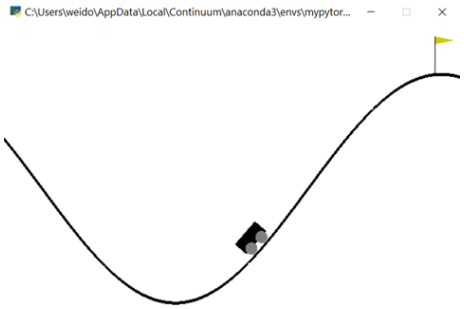


Fig.14 Mountain car

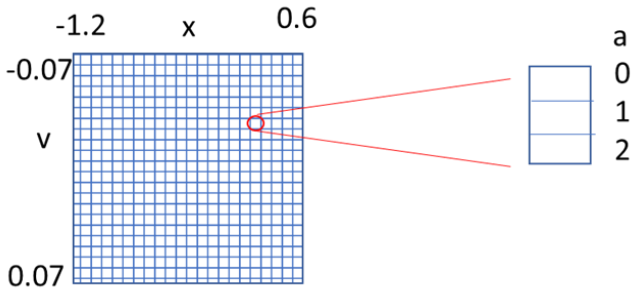


Fig.15 Q-table

The following description of the mountain car task is based on the environment defined in gym package.

Its **state space** (x,v) is 2-dimensional. The first dimension x represents the current horizontal position of the car. The flag is located x=0.5, and the bottom of the shape is located at x= - 0.5. Another dimension v is the velocity of the car (positive for right direction). The range of x is [-1.2,0.6]. The range of v is [-0.07, 0.07].

reward: every time step, -1, except 0 for the terminal state (reaching flag).

done: a sign that the game is over. The episode ends when either the car reaches the goal (flag), or a maximum number of timesteps has passed. By default, the episode will terminate after 200 steps. You can customize this with the _max_episode_steps attribute of the environment. Limiting the number of steps per episode has the immediate benefit of forcing the agent to reach the goal state in a fixed amount of time, which often results in a speedier trajectory by the agent (MountainCar-v0 further penalizes long trajectories through the reward signal). Also, the underlying learning algorithm may only perform policy updates after completion of the episode. If the agent will never reach the goal state under its current policy (i.e. the policy

is very bad, lacks much randomness, etc.), then terminating the episode after a fixed amount of time will ensure that the agent is able to perform a policy update and try a new policy on the next episode (alternatively, the learning algorithm could perform a policy update during the episode).

action: a ϵ [0,1,2], representing push left, no push, push right, respectively.

The agent issues an action based on the current state (location and velocity), and the environment will return a new state (new location and velocity) and reward. The goal of the agent is to speed up the car so that it reaches the flag.

To solve the mountain car problem by Q-tabular learning, we discrete the state space (x,v) into (20,20),for instance. Thus the discrete Q-table has three dimensions: x, v, a, denoted as Q(x,v,a) with a shape of (20,20,3), as shown in Fig.15.

We implement the RL for the mountain car in Python utilizing the environment package gym. The results are plotted in Fig.16, which shows the average, min, max of episode returns over the recent 100 episodes. The horizontal axis is the current number of episodes that have been used for training. 10000 episodes, the first 5000 episodes epsilon decayed, and the second 5000 episodes use a small constant epsilon. Note that the absolute value of the reward is the number of steps for an episode, which is equal to either the number of steps for the car to reach the goal or 200 when the car fails to reach the goal.

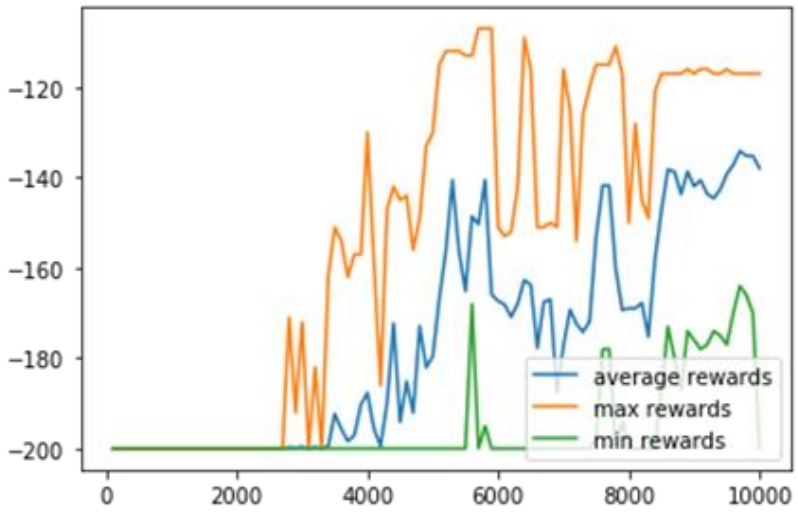


Fig.16 Rewards during the training process: LEARNING_RATE = 0.1,DISCOUNT = 0.95

4.6 Summary

In this chapter, we delt with the **policy evaluation** and **iteration** for the situation when we don't have access to true MDP models. Monte Carlo method and its variations (e.g. TD learning, Q-learning) have been discussed for this purpose.

MC policy evaluation is to estimate the state value function V(s) by the empirical average of the return for each state, given episodes generated under a policy. This method does not require Markov property. The estimated value converges to the true value under some assumptions. However, this estimator is **unbiased** but has a high variance, and thus requires a lot of data to keep an acceptable low variance. The episode must end before the data in the episode can be used for update. **TD policy evaluation**, a combination of

MC and dynamic programming methods, immediately updates estimate of V(s) after each tuple (s,a, r,s'), and thus can be used in episodic or non-episodic settings. TD estimator is **biased** but has a much lower variance than MC estimator. TD(λ) unifies one-step TD prediction, i.e., TD(0), with MC methods, i.e., TD(1), using eligibility traces and the decay parameter λ for prediction algorithms, and results in a family of methods with various values of λ.

For **policy iteration**, an **ϵ-greedy policy** is adopted to generate sample episodes. Two TD methods are used to update the Q(s,a) function: **SARSA** and **Q-learning**. SARSAR is an on-policy algorithm since the Q function is updated by exactly following the ϵ-greedy policy, while Q-learning is an off-policy algorithm because it updates the Q function for the greedy policy by following another policy (behavior policy) (e-greedy policy). Just like TD(λ) for prediction, SARSA(λ) is available for policy control by using eligibility traces for more efficient updates.

Chapter 5 Value Function Approximation

Resources: Stanford CS234 lecture 5, David Silver's Lecture 6.

Learning objectives

So far, we have been assuming we can represent the value function or state-action value function as a vector or matrix, which is called tabular representation. The value for each state or each state-action pair can be explicitly specified in a table. However, many real-world problems have enormous state and/or action spaces. For instance, Backgammon has 10^{20} states, computer Go has 10^{170} states, and helicopter has a continues state space. (David lecture 6)

In value function approximation (VFA), we apply supervised learning to learn a function that maps the state (or state-action) to its value, using episodes as learning examples. The number of weights in the function is usually much less than the number of states. The value of any unseen state (or state-action) can be generalized by the learned function. The key idea is to represent a (state or state-action) value function with a parameterized function instead of a table, as shown in Fig.1. Thus, we don't have to explicitly store or learn for every single state a dynamics or reward model, value, state-actin value or policy. Instead, we have a more compact representation that generalizes across states or state-action pairs. The benefits of generalization include reducing required memory, computation and experience (or data) for a reinforcement learning task.

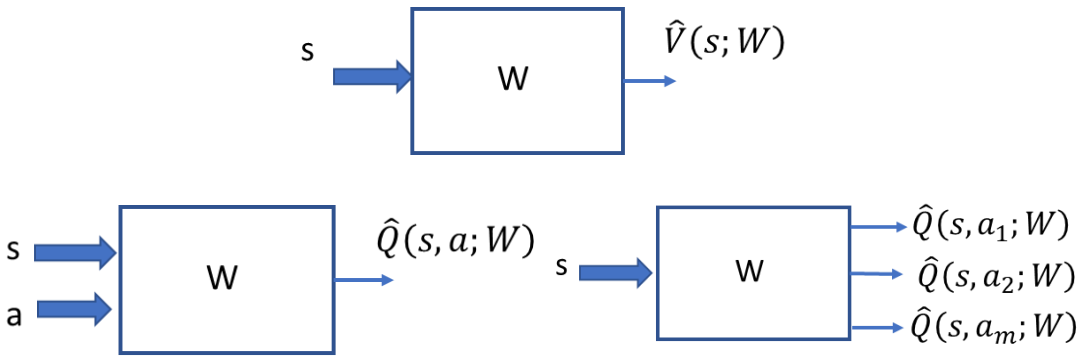


Fig.1Paramterized state and action functions

Theoretically, we can use any method for supervised learning from examples, including linear regression, neural networks, decision trees, nearest neighbors, Fourier/wavelet basis. However, not all function approximation methods are equally well suited for reinforcement learning. We will discuss linear combinations of features and deep neural networks for function approximation.

Upon the completion of this chapter, you should be able to:

- Implement MC and TD(0) on policy evaluation with linear value function approximation.
- Implement MC, SARSA, and Q-learning control algorithms.

- Be aware of what TD(0) and MC on policy evaluation with linear VFA are converging to and when this solution has 0 error and non-zero error.
- List the 3 issues that can cause instability and describe the problems qualitatively: function approximation, bootstrapping and off policy learning.

5.1 Revisit of gradient descent

Consider an optimization problem, the goal of which is to find parameter w that minimizes J(w). The function J(w) is usually a differentiable (error, or cost, or loss) function, based on a set of data examples. The gradient of J(w) is

$$\nabla_w J(w) = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_n}\right]^T$$

The parameter W can be updated as

$$w \leftarrow w - \frac{1}{2}\alpha \nabla_w J(w)$$

where α is the learning rate.

5.2 Value function approximation for policy evaluation

First assume that we could query any state s and an **oracle** would return the true value for $V^\pi(s)$. The policy π may be deterministic (i.e., a=π(s)) or stochastic (i.e., π(a|s)). The objective is to find the best approximate representation of V^π based on a parameterized function. We define the loss function between a true value function $V^\pi(s)$ and its approximation $\hat{V}(s;w)$

$$J(w) = \mathbb{E}_\pi\left[\left(V^\pi(s) - \hat{V}(s;w)\right)^2\right]$$

We can use gradient descent to find a local minimum.

$$w \leftarrow w - \frac{1}{2}\alpha \nabla J(w) = w + \alpha \mathbb{E}_\pi\left[\left(V^\pi(s) - \hat{V}(s;w)\right) \cdot \nabla_w \hat{V}(s;w)\right]$$

Stochastic gradient descent (SGD) method samples the loss function J(w) at each example of state, and adjusts the weight vector after each example by a small amount in the direction that would most reduce the error on that example.

$$w \leftarrow w + \alpha\left[V^\pi(s) - \hat{V}(s;w)\right] \nabla \hat{V}(s;w)$$

When we discussed model-free policy evaluation in the previous chapter, we maintained a lookup table to store estimates V(s) or Q(s,a), and updated these estimates after each episode (MC methods) or after each step (TD methods), by following a fixed policy. Now, we don't want to explicitly update the V(s) or Q(s,a) for all s and a. Instead, we try to find a good parameterized function $\hat{V}(s;w)$ or $\hat{Q}(s,a;w)$ to approximate the true value function. Thus, we will learn the parameter w, based on the episodes. In general, the resulting VFA does not get all states, or even all examples, exactly accurate. In addition, the VFA should generalize to all other states that have not been seen in examples.

However, in practice the true value of $V^\pi(s)$ is usually unknown. We will substitute the true function value with a quantity obtained from sample episodes. For instance, as we will see later, in MC policy evaluation,

we will use the return for state s G(s) in episodes to replace $V^{\pi}(s)$ in the W update equation while in TD(0) policy evaluation we use the one-step update $r + \gamma V(s')$ to replace $V^{\pi}(s)$.

5.3 Linear value function approximation for MC policy evaluation

Linear model methods assume that the value function can be represented as the inner product of weight vector w with state feature vector. The state feature vector can be considered as the coordinate of the state in the state space. Note that the model is the linear function of weights, not necessarily a linear function of s. A linear model function can be represented as

$$\hat{V}(s;w) = w^T X(s) = X(s)^T w$$

$$X(s) = \begin{pmatrix} x_1(s) \\ x_2(s) \\ \cdots \\ x_n(s) \end{pmatrix}$$

where X(s) is the feature vector to represent a state s, in other words, for each state s, there is a real-valued vector X(s), with the same dimension as w.

$$w \leftarrow w - \frac{1}{2}\alpha \nabla J(w) = w + \alpha\left[V^{\pi}(s) - \hat{V}(s;w)\right] \mathrm{X(s)}$$

The update amount of W can be summarized in the format,

$$update = step_{size} \times prediction_{error} \times feature_{value}$$

In practice, it is usually the case that we don't have an oracle to give us the exact true value function. In reinforcement learning, we substitute a target for $V^{\pi}(s)$. In Monte Carlo methods, we can use return G_t as the target, an unbiased but noisy sample of the true expected return $V^{\pi}(s_t)$. Therefore, we can implement MC value function approximation by doing supervised learning on a set of state-return pairs: (s_t, G_t), t=1,2,…T. The weights will be updated as

$$w \leftarrow w - \frac{1}{2}\alpha \nabla J(w) = w + \alpha[G_t - X(s_t)^T w]\, X(s_t)$$

Please note that G_t may be a very noisy estimate of true return.

MC linear value function approximation for policy evaluation algorithm:

Initialize w=0, k=1
Loop
    Sample k-th episode given π
    For t=1,2….L_k do
        If first visit to (s) in episode k then
            compute the return $G_t(s)$
            Update weights
                $w \leftarrow w + \alpha[G_t - X(s_t)^T w]\, X(s_t)$
        End if
    End for
    k=k+1
End loop

Note that MC every-visit could be done as well.

Convergence for linear VFA for policy evaluation

To compute the loss function J(w) defined in the previous section, we will introduce the stationary distribution over states. A Markov chain defined by an MDP with a particular policy will eventually converge to a probability distribution over states, d(s), which is called the stationary distribution over states of π. d(s) satisfies the following balance equation

$$d(s') = \sum_s \sum_a \pi(a|s)\, p(s'|s,a)\, d(s)$$

$$d(s) \geq 0, \qquad \sum_s d(s) = 1$$

The stationary distribution d(s) shows how likely the system stays in the state s. Now we define the **mean squared value error** of a linear value function approximation for a particular policy π relative to the true value as

$$MSVE(w) = \sum_{s \in S} d(s)\big(V^\pi(s) - \hat{V}^\pi(s;w)\big)^2$$

where

$$\hat{V}^\pi(s;w) = X(s)^T w$$

it has been shown [*] that Monte Carlo policy evaluation with VFA converges to the weights w_{MC} which has the minimum mean squared error possible:

$$MSVE(w_{MC}) = \min_w \sum_{s \in S} d(s)\big(V^\pi(s) - \hat{V}^\pi(s;w)\big)^2$$

[*] An Analysis of Temporal-Difference Learning with Function Approximation, John N. Tsitsiklis, and Benjamin Van Roy

Batch Monte Carlo value function approximation

In Monte Carlo methods, we can define the loss function based on a set of episodes from a policy π. Let $G(s_i)$ be an unbiased sample of the true expected return $V^\pi(s_i)$

$$MSVE(w) = \sum_{i=1}^{N} (G(s_i) - X(s_i)^T w)^2$$

N is the number of return samples. Note that the frequency of state s, specified by d(s), has been taken into consideration by the sum operator since the more $d(s_i)$, the more likely s_i will appear in the summation. To find the value of w for minimizing MSVE(w), we take the derivative and set to 0, and solve for w

$$w = (X^T X)^{-1} X^T G$$

where G is a vector of all N returns, and X is a matrix of the features of each of the N states $X(s_i)$. Note that Markov assumption is not required. This approach has a high computing cost, and thus is not feasible.

However, batch gradient descent algorithm can be applied to solve for the optimal W iteratively, detailed in the next chapter.

5.4 TD learning with linear value function approximation

Recall TD learning for policy evaluation with lookup table discussed in the previous chapter. We use bootstrapping and sampling to approximate V(s). Specifically, we updated V(s) after each transition (s, a, r, s'):

$$V(s) \leftarrow V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

where target is $r + \gamma V(s')$, a biased estimate of the true value V(s). The value of each state is specified by a separate table entry.

In value function approximation, the target is $r + \gamma \hat{V}(s'; w)$, a biased and approximated estimate of the true value V(s). TD learning with VFA involves three forms of approximation: function approximation, bootstrapping, and sampling, but it is still on-policy learning. We can reduce TD learning with VFA to supervised learning on a set of data pairs:

$$\langle s_1, r_2 + \gamma \hat{V}(s_2; w)\rangle, \quad \langle s_2, r_3 + \gamma \hat{V}(s_3;\, w)\rangle, \quad \ldots.$$

The loss function is defined as the mean square error

$$J(w) = \mathbb{E}_\pi\left[\left(V(s) - \hat{V}(s; w)\right)^2\right]$$

The goal is to find weights W to minimize the loss. In linear model, given the tuple (S_t,A_t,R_{t+1},S_{t+1}) from an episode, we update weights as

$$w \leftarrow= w + \alpha\big[R_{t+1} + \gamma \hat{V}(S_{t+1}; w) - \hat{V}(S_t; w)\big]\, \nabla \hat{V}(S_t; w)$$

$$w \leftarrow w + \alpha[R_{t+1} + \gamma X(S_{t+1})^T w - X(S_t)^T w]\, \mathrm{X}(S_t)$$

Question: from

$$J(w) = \mathbb{E}_\pi\left[\left(V(s) - \hat{V}(s; w)\right)^2\right] \approx \mathbb{E}_\pi\left[\left(r + \gamma X(s')^T w - \hat{V}(s; w)\right)^2\right]$$

why not? $w \leftarrow w + \alpha[r + \gamma X(s')^T w - X(s)^T w]\,\big(-\gamma X(s') + X(s)\big)$

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^n \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\boldsymbol{\theta}$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\big[R + \gamma\hat{v}(S',\boldsymbol{\theta}) - \hat{v}(S,\boldsymbol{\theta})\big]\nabla\hat{v}(S,\boldsymbol{\theta})$
    $S \leftarrow S'$
  until S' is terminal

[print from Sutton book, θ is the weight W in our text]

The property of convergence: MC policy evaluation with VFA converges to the weights W_{MC} which has the minimum mean squared error possible:

$$MSVE(w_{MC}) = \min_{w} \sum_{s \in S} d(s)\left(V^{\pi}(s) - \hat{V}^{\pi}(s; w)\right)^2$$

TD(0) policy evaluation with VFA converges to weights W_{TD} which is within a constant factor of the minimum mean squared error possible: (see Sutton book for a proof)

$$MSVE(w_{TD}) = \frac{1}{1-\gamma} \min_{w} \sum_{s \in S} d(s)\left(V^{\pi}(s) - \hat{V}^{\pi}(s; w)\right)^2$$

If the VFA is a tabular representation (one feature for each state), the MSVE for MC and TD is zero. (Why?)

Similarly, TD(λ) with VFA will have the update in forward view,

$$w \leftarrow w + \alpha\left[G_t^{\lambda} - \hat{V}(S_t; w)\right] \nabla \hat{V}(S_t; w)$$

This update can be efficiently implemented in a backward view using eligibility traces. The eligibility trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient, and then decays by a factor γλ:

$$z_{-1} = 0$$

$$z_t = \gamma \lambda z_{t-1} + \nabla \hat{V}(S_t, w_t)$$

The eligibility trace keeps track of which components of the weight vector have contributed, positively or negatively to recent state evaluations. In linear function approximation, the gradient is just the feature vector, and the eligibility trace vector is just a sum of past input vectors with fading effects. The TD error for state-value prediction is

$$\delta_t = R_{t+1} + \gamma \hat{V}(S_{t+1}; w_t) - \hat{V}(S_t; w_t)$$

Then the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace:

$$w_{t+1} = w_t + \alpha \delta_t z_t$$

Semi-gradient (bootstrapping) TD(λ) with VFA for estimating state-value function

---

Input: the policy π to be evaluated
Input: a differentiable function $\hat{V}(s; w)$, $\hat{V}(\text{terminal}; w) = 0$
Algorithm parameters: step size (or learning rate) α, trace decay rate λ
Initialize value-function weights w arbitrarily (e.g. w=0)
Loop for each episode:
    Initialize S
    z←0 (a d-dimensional vector, and d is the dimension of w)
    loop for each step of episode:
        choose A by following π
        take action A, observe R, S'
        update trace: $z \leftarrow \gamma \lambda z + \nabla \hat{V}(S, w)$

TD error: $\delta = R + \gamma\hat{V}(S'; w) - \hat{V}(S; w)$
update weights: $w \leftarrow w + \alpha\delta z$
$S \leftarrow S'$
until S' is terminal

If λ=0, the algorithm reduces to a simple TD(0) (one-step TD gradient descent) because z is always equal to the gradient.
Example for practice: random walk with 19 states. Example 7.1 and Figure 12.6 in Sutton book.

5.5 Control using Value Function Approximation

For policy control, we use function approximation to represent state-action values $\hat{Q}^{\pi}(s, a; w) \approx Q^{\pi}(s, a)$, and perform ϵ-greedy policy improvement. The idea of policy improvement is illustrated in Fig.2. At each time step, instead of evaluating the state-action function accurately, we just update the weight vector and update the policy by e-greedy immediately without waiting for the completed Q-function evaluation. This generally involves intersection of the three factors: function approximation, bootstrapping/sampling and off-policy learning.

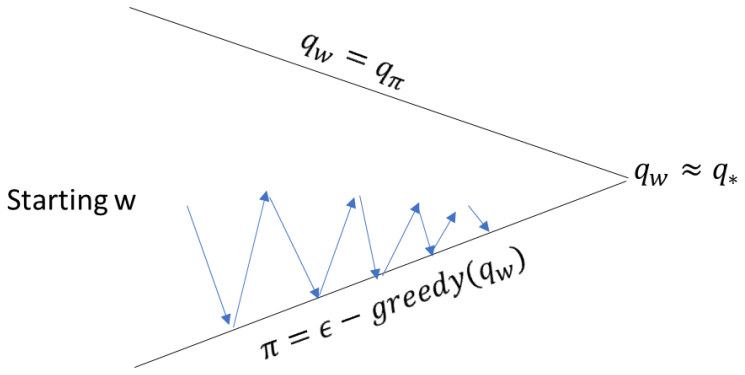


Fig.2 Iterative cycles for policy evaluation and improvement

5.5.1 Action-value function approximation with a true action-value function

The goal is to minimize the mean-squared error between the true action-value function $Q^{\pi}(s, a)$ and the approximate action-value function:

$$J(w) = \mathbb{E}_{\pi}\left[\left(Q^{\pi}(s, a) - \hat{Q}^{\pi}(s, a; w)\right)^2\right]$$

We can use stochastic gradient descent to find a local minimum, by sampling the gradient.

$$\nabla_w J(w) = -2\mathbb{E}\left[\left(Q^\pi(s,a) - \hat{Q}^\pi(s,a;w)\right) \nabla_w \hat{Q}^\pi(s,a;w)\right]$$

$$w \leftarrow w - \frac{1}{2}\alpha\nabla_w J(w) = w + \alpha\mathbb{E}\left[\left(Q^\pi(s,a) - \hat{Q}^\pi(s,a;w)\right) \nabla_w \hat{Q}^\pi(s,a;w)\right]$$

In the linear model, the state-action value function is represented by a weighted linear combination of features

$$\hat{Q}(s,a;w) = X(s,a)^T w = \sum_{j=1}^{n} X_j(s,a) w_j$$

where $X(s,a)$ are the features representing both the state and action

$$X(s,a) = \begin{bmatrix} X_1(s,a) \\ X_2(s,a) \\ \cdots \\ X_n(s,a) \end{bmatrix}$$

5.5.2 Action-value function approximation for policy control with sampling

Similar to policy evaluation, the true-action value function for a state is unknow and so is substituted by a target value. For policy control, the sample episodes are generated by following an ϵ-greedy policy with respect to the current Q function. In Monte Carlo methods, we use a return G_t as a substitute target

$$\Delta w = \alpha\left(G_t - \hat{Q}(s_t,a_t;w)\right) \nabla_w \hat{Q}(s_t,a_t;w)$$

In SARSA, we use a TD(0) target $R_{t+1} + \gamma\hat{Q}(S_{t+1},A_{t+1};w)$ which leverages the current function approximation value

$$\Delta w = \alpha\left(R_{t+1} + \gamma\hat{Q}(S_{t+1},A_{t+1};w) - \hat{Q}(S_t,A_t;w)\right) \nabla_w \hat{Q}(S_t,A_t;w)$$

In SARSA(λ), we use TD(λ) return as the target G_t^λ. The forward view update is given by

$$\Delta w = \alpha\left(G_t^\lambda - \hat{Q}(S_t,A_t;w)\right) \nabla_w \hat{Q}(S_t,A_t;w)$$

The equivalent backward view update (online implementation of SARSA(λ) is given by

$$\Delta w = \alpha\delta_t z_t$$

where the current TD error

$$\delta_t = R_{t+1} + \gamma\hat{V}(S_{t+1};w_t) - \hat{V}(S_t;w_t)$$

eligibility trace

$$z_{-1} = 0$$

$$z_t = \gamma\lambda z_{t-1} + \nabla\hat{V}(S_t,w_t)$$

In Q-learning, we use a TD target $R_{t+1} + \gamma \max_{a'} \hat{Q}(S_{t+1}, a'; w)$ which leverages the maximum of the current function approximation value

$$\Delta w = \alpha \left(R_{t+1} + \gamma \max_{a'} \hat{Q}(S_{t+1}, a'; w) - \hat{Q}(S_t, A_t; w) \right) \nabla_w \hat{Q}(S_t, A_t; w)$$

At each time step, after the weight has been updated, the e-greedy policy is applied to the current state-action value to generate the subsequent part (new action, new reward and new state) of the episode: new action, new reward and new state. The alternating process of weight update and policy improvement is repeated until the end of an episode.

SARSA(0) with function approximation for control

Input: a differentiable function $\hat{Q}(s, a; w)$
Parameters: discount γ, learning rate α
Initialize Q-function weight vector w arbitrarily (e.g. w=0)
Repeat (for each episode):
    Initialize S
    Choose A from S using the e-greedy policy w.r.t. $\hat{Q}(s, a; w)$
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using the e-greedy policy w.r.t. $\hat{Q}(s, a; w)$
        $w \leftarrow w + \alpha \left(R + \gamma \hat{Q}(S', A'; w) - \hat{Q}(S, A; w) \right) \nabla_w \hat{Q}(S, A; w)$
        $S \leftarrow S', A \leftarrow A'$
    Until S is terminal

Q-learning with function approximation for control

Input: a differentiable function $\hat{Q}(s, a; w)$
Parameters: discount γ, learning rate α
Initialize Q-function weight vector w arbitrarily (e.g. w=0)
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using the e-greedy policy w.r.t. $\hat{Q}(s, a; w)$
        Take action A, observe R, S'
        $w \leftarrow w + \alpha \left(R + \gamma \max_{a'} \hat{Q}(S', a'; w) - \hat{Q}(S, A; w) \right) \nabla_w \hat{Q}(S, A; w)$
        $S \leftarrow S'$
    Until S is terminal

Example for practice: mountain car. (example 10.1 in Sutton book)

Other possible examples: random walk, puddle world, cart and pole

5.5.3 Convergence of TD methods with VFA

TD with VFA is not following the gradient of an objective function (see Sutton's book chapter 11), instead gradient TD follows true gradient of projected Bellman error. Informally, updates involve doing an (approximate) Bellman backup followed by best trying to fit underlying value function to a particular feature representation. This is why TD can diverge when off-policy or using non-learning function approximation. Bellman operators are **contractions**, but value function approximation fitting can be an **expansion**.

Convergence of control methods with VFA

Algorithm	Tabular	Linear VFA	Nonlinear VFA
Monte-Carlo control	Yes	Yes	No
SARSA	Yes	Yes	No
Q-learning	Yes	No	No

5.6 Summary

In this chapter, value function approximation was introduced to policy prediction and control so that the reinforcement learning agents can generalize. This capability of generalization is required for large applications. We use a parameterized function, $\hat{V}(s;w), or\ \hat{Q}(s,a;w)$, to represent an approximation of state value function or state-action value function, respectively. The goal of learning process is to find the best value of the weight vector w to fit the true value function $v^{\pi}(s), or\ q^{\pi}(s,a)$ in a policy prediction task or to fit the optimal value function $v^{*}(s), or\ q^{*}(s,a)$ for a control task. The most popular methods to learn the parameter are stochastic gradient descent (SGD) and its variations.

For policy prediction (given a fixed policy), we update the weight vector by the amount that is proportional to the product of the TD(λ) error and the gradient of function at each time step. The TD(λ) error is defined as the difference between the sampled TD(λ) return and the estimate of the value function. Note that for the case of λ=1, it is a gradient Monte Carlo algorithm whereas for the cases of λ<1, the method is semi-gradient. Semi-gradient TD methods are not true gradient methods because the TD(λ) return includes the weight vector that is not taken into account in computing the gradient.

For a control task, after updating the weight vector at each time step, we improve the policy by following an e-greedy policy to continue the trajectory at the next time step.

Function approximation methods can be applied to tasks with arbitrarily large state spaces, for which there is not enough memory for state/state-action tables, or that these tables cannot be filled optimally even in the limit of infinite amounts of time and data. For example, an autonomous car camera will never capture the exact same image but will usually encounter images of scenarios it can generalize (e.g. "a single pedestrian walking on the zebra crossing"). For these tasks, an optimal solution cannot be obtained and generalization and finding a good approximated solution with limited compute resources is the goal. Function approximation in RL is related to Supervised Learning, but it also deals with some unique issues such as nonstationarity, bootstrapping, and delayed targets. Although in theory all supervised learning methods can be applied to RL tasks, in practice we will usually use a linear function approximator or artificial neural network (ANN).

Chapter 6 Deep Q-Learning

Resources: Stanford CS234 lecture 6, David Silver, and Ruslan.

Learning objectives

Many applications of reinforcement learning, such as self-driving cars, Atari, consumer marketing, healthcare, education, etc., have enormous states and/or action spaces. We may never encounter some states during the training process. This requires representations of models (e.g. Transition or Reward), state-action values (Q), state values (V), and policy π that can generalize across states and/or actions. In the previous chapter, we have demonstrated how a value function can be represented by a parameterized function (e.g. a linear model) instead of a table.

Linear value function approximators assume value function is a weighted combination of a set of features, where each feature is a function of the state. Linear VFA often works well given the right set of features. This implies that we need to carefully handle designing the feature set. An alternative is to use much richer function approximation class that is able to directly go from states without requiring an explicit specification of features.

In the previous chapter, the agent learns the VFA by updating the weights at each time step based on the current TD return. After updating, the TD return is thrown away, and a new TD return for the next time step is used for the next weight updating. Thus, this method does not make the maximal use of the sampled data to fit the function.

In this chapter, we will discuss how neural networks (NNs) can be used for VFA in reinforcement learning, and how the neural networks can be trained based on experience data. The reason we use "Deep Q-learning" as the title of this chapter is that the neural networks are usually implemented by deep neural networks (DNNs) for many applications. Of course, what we cover in this chapter is applicable to general neural networks.

Upon the completion of this chapter, you will be able to

- Understand how to apply deep neural networks to Q-learning.
- Understand two techniques to training DQN: 1) experience replay and 2) Fixed target network.
- Be familiar with other improvement for DQN: double-DQN, prioritized replay, and dueling DQN.
- Be familiar with applications of DQN in Atari games.
- Implement DQN in Python.

6.1 Basic Deep Q-Networks (DQNs)

We represent the state-action value function by a DNN, called Q-network with weights w: $\hat{Q}(s,a;w) \approx Q(s,a)$. To find the optimal w, we minimize the mean-square error by SGD

$$J(w) = \mathbb{E}_{\pi}\left[\left(Q^{\pi}(s,a) - \hat{Q}^{\pi}(s,a;w)\right)^2\right]$$

For Q-learning, we update the weights based on the tuple (s, a, r, s')

$$\Delta w = \alpha\left(r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w)\right) \nabla_w \hat{Q}(s, a; w)$$

There are two issues which may cause VFA to diverge: correlations between samples and non-stationary targets. For instance, in the episode: s,a,r,s',a',r',s''…, V(s) and V(s') are typically highly correlated because they are *close*. However, one of the fundamental requirements for SGD optimization is that the training data is independent and identically distributed (iid). The target, $r + \gamma \max_{a'} \hat{Q}(s', a'; w)$, depends on the weights. This can make our training very unstable. Deep Q-learning addresses both challenges by experience replay and fixed Q-targets (for a while).

6.1.1 Experience Replay

To help remove correlations, we store dataset from prior experience into a replay buffer. The tuples are gradually added to the buffer as we are interacting with the Environment. The simplest implementation is a buffer with a fixed size, with new data added to the end of the buffer so that it pushes the oldest experience out of it.

The act of sampling a small **batch** of tuples from the replay buffer for training the neural network is known as experience replay, as shown in Fig.1. In addition to reducing the correlations, experience replay allows us to learn more from individual tuples multiple times, and in general make better use of our experience. To perform experience replay, we randomly sample a batch of experience tuples from the dataset, compute the target values for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; w)$, and then use SGD to update the network weights. General methods of training neural network can be used for this training purpose.

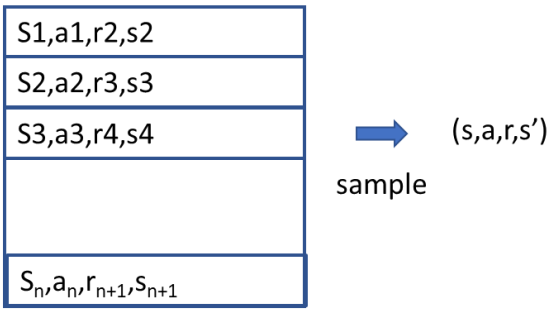


Fig.1 Experience replay

6.1.2 Fixed Q-Targets

Notice that the weights will get updated on each training iteration, which implies the weights involving computation of target is not stationary. To improve the stability, we reduce the noise of target by fixing the target weights used in the target calculation for multiple updates. Although the target is a function of weights

w, it serves an estimate of the true value function. Thus, it is ok to keep the target unchanged for multiple weight updates. Instead, fixing the target for a while can reduce the noise in the target.

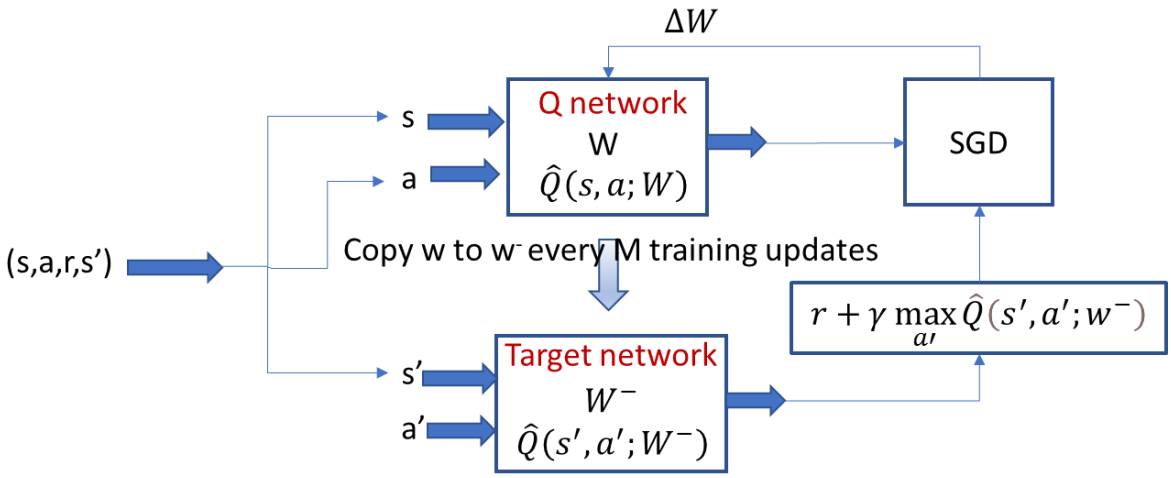


Fig.2 Q-learning network with target network.

We initially make a copy of the neural network $\hat{Q}(s,a;w)$, called target network, denoted as $\hat{Q}(s',a';w^-)$, and thereafter pass the value of w to w^- once a while (e.g. every M training updates). The purpose of this target network is to calculate the Q-learning target. The output of the target network is used to train the main network $\hat{Q}(s,a;w)$. During the training process, w is updated for each tuple sample while the weight in $\hat{Q}(s',a';w^-)$ is periodically but less frequently copied from the weight in the main network $\hat{Q}(s,a;w)$. Fig.2 illustrates a general DQN.

The DQN algorithm can be described as the repetitions of the following steps:

1) Take action a_t according to ϵ-greedy policy
2) Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
3) Sample random mini-batch of transitions (s,a,r,s') from $\mathcal{D}$
4) Compute Q-learning targets for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s',a';w^-)$
5) Optimize MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} \hat{Q}(s',a';w_i^-) - \hat{Q}(s,a;w_i)\right)^2\right]$$

using variant of stochastic gradient descent

The pseudo code for DQN algorithm can be described as follows.

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**
  `/* Sample phase`
  $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
  Choose an action a from state s using policy ϵ-greedy(Q)
  Agent takes action a, observe reward r, and next state s'
  Store transition $(s, a, r, s', done)$ in the experience replay memory D
  if *enough experiences in* D **then**
    `/* Learn phase`
    Sample a random *minibatch* of N transitions from D
    for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
      if $done_i$ **then**
        $y_i = r_i$
      else
        $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
      end
    end
    Calculate the loss $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
    Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
    Every C steps, copy weights from Q to $\hat{Q}$
  end
end

(From: https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c)

Further Improvements

Success in Atari has led to huge excitement in using deep neural networks to do value function approximation in RL. Some immediate improvements include ***double DQN*** (Deep Reinforcement Learning with Double Q-learning, Van Hasselt et al, AAAI 2016), ***prioritized replay*** (prioritized experience replay, Schaul et al, ICLR 2016), ***Dueling DQN*** (base paper ICML 2016) (Dueling Network Architectures for Deep Reinforcement Learning, Wang et al, ICML 2016).

6.2 Double DQN

In Q-learning, the maximum of the estimated state-action values can be a biased estimate of the true maximum. Recall the target of Q-learning

$$r + \gamma \max_{a'} Q(s', a')$$

The Q-learning would work perfectly if $Q(s', a')$ were equal to the true state-action value. However, $Q(s', a')$ is just an estimate of true state-action function. This implies that $Q(s', a')$ can be considered as the sum of the true state-action function and a noise. Thus, for the state s', the action for the maximum

of $Q(s', a')$ may not be the optimal action due to a positive noise. This leads to a significant positive bias for state-action value estimation, called maximization bias. For example, consider a single state s where there are many actions a whose true values, q(s,a), are all zero but the current estimated values Q(s,a) are distributed around zero (above or below zero). The maximum of true values is zero, but the maximum of the estimated values is positive.

To deal with this maximization bias, double Q-learning maintains two different state-action functions, Q_1(s,a) and Q_2(s,a), to decouple the maximal action selection from the target value calculation. We use e-greedy policy to generate episodes, based on the average of the two function values, and then learn the two Q functions alternatively in a probability of 0.5 for each. When learning one function, we pick the maximum value action from the other function to calculate the target. The double Q-learning algorithm is described below.

Initialize $Q_1(s, a)$ and $Q_2(s, a)$, $\forall s \in S, a \in A$ $t = 0$, initial state $s_t = s_0$
loop
  Select a_t using ϵ-greedy $\pi(s) = \arg\max_a Q_1(s_t, a) + Q_2(s_t, a)$
  Observe (r_t, s_{t+1})
  if (with 0.5 probability True) **then**

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_t + Q_1(s_{t+1}, \arg\max_{a'} Q_2(s_{t+1}, a')) - Q_1(s_t, a_t))$$

  else

$$Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha(r_t + Q_2(s_{t+1}, \arg\max_{a'} Q_1(s_{t+1}, a')) - Q_2(s_t, a_t))$$

  end if
  $t = t + 1$
end loop

We extend this idea of double Q-learning to DQN. Double Q-Learning implementation with Deep Neural Network is called Double Deep Q Network (Double DQN). The Q-network w is used to select the action for the maximal value of the next state while the target network w^- is used to evaluate the action value – target value.

$$\Delta w = \alpha \left(r + \gamma \hat{Q} \left(s', arg \max_{a'} \hat{Q}(s', a'; w) ; w^- \right) - \hat{Q}(s, a; w) \right) \nabla_w \hat{Q}(s, a; w)$$

In double DQN, two neural networks (Q network and target network) are involved in calculating the target

$$r + \gamma \hat{Q} \left(s', arg \max_{a'} \hat{Q}(s', a'; w) ; w^- \right)$$

Specifically, the Q network selects the action with the maximal value for the next state while the target network calculates the value for the next state given the action recommended by the Q network. The block diagram of double DQN is shown in Fig.3.

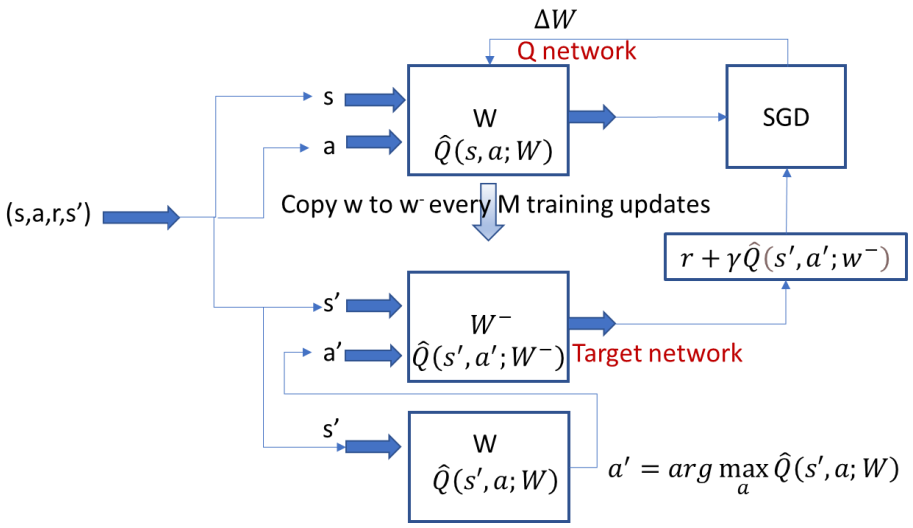


Fig.3 Double DQN

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

[From "Dueling Network Architectures for Deep Reinforcement Learning" Z. Wang, et.al.]

6.3 **Prioritized Replay** (Stanford CS234 lecture 6)

A recent innovation in prioritized experience replay (Schaul et al., 2016) built on top of DDQN and further improved the state-of-the-art. Their key idea was to increase the replay probability of experience tuples that have a high expected learning progress (as measured via the proxy of absolute TD-error). This led to both faster learning and to better final policy quality across most games of the Atari benchmark suite, as compared to uniform experience replay.

Let i be the index of the i-th tuple of experience (s_i, a_i, r_i, s_{i+1}). We sample tuples for update using priority function. The priority of a tuple i is proportional to DQN error

$$p_i = \left| r + \gamma \max_{a'} Q(s_{i+1}, a'; \boldsymbol{w}^-) - Q(s_i, a_i; \boldsymbol{w}) \right|$$

We can define the probability of selecting the i-th tuple as

$$P(i) = \frac{p_i^\beta}{\sum_k p_k^\beta}$$

where β is a hyperparameter.

6.4 **Dueling DQN** (Wang's paper "Dueling Network Architectures for Deep Reinforcement Learning")

In some applications, it is unnecessary to estimate the value of each action choice for many states. In some states, it is of paramount importance to know which action to take, but in many other states the choice of action has no repercussion on what happens. For example, in the Enduro game setting, knowing whether to move left or right only matters when a collision is eminent.

For an agent behaving according to a stochastic policy π, the values of the state-action pair (s, a) and the state s are defined as follows

$$Q^\pi(s,a) = \mathbb{E}[G_t | s_t = s, a_t = a, \pi]$$
$$V^\pi(s) = \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)]$$

The optimal Q function is defined as

$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

Under the deterministic policy $a = arg\max_{a'} Q^*(s,a')$, it follows that

$$V^*(s) = \max_a Q^*(s,a)$$

Intuitively, the value function V measures how good it is to be in a particular state s. The Q function, however, measures the value of choosing a particular action when in this state. A third function, called advantage function, is defined to measure a relative value of an action compared to other actions in the state

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

Note that $\mathbb{E}_{a\sim\pi}[A^\pi(s,a)] = 0$.

Wang et al. proposed the dueling architecture which explicitly separates the representation of state values and state-dependent action advantages via two separate streams, as illustrated in Fig. 4. The lower layers of the dueling network are convolutional as in the original DQNs (Mnih et al., 2015). However, instead of following the convolutional layers with a single sequence of fully connected layers, we instead use two sequences (or streams) of fully connected layers. The streams are constructed such that they have the capability of providing separate estimates of the value and advantage functions. Finally, the two streams are combined to produce a single output Q. By explicitly separating two estimators, the dueling architecture

can learn which states are (or are not) valuable, without having to learn the effect of each action for each state.

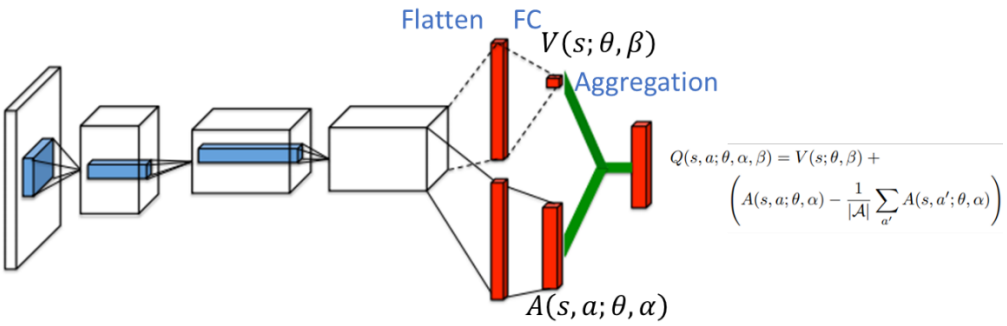


Fig.4 Dueling DQN architecture

Now, let us consider the dueling network shown in Fig.4, where we make one stream of fully connected layers output a scalar V (s; θ, β), and the other stream output an |A|-dimensional vector A(s, a; θ, α). Here, θ denotes the parameters of the convolutional layers, while α and β are the parameters of the two streams of fully connected layers.

It is straightforward to obtain the Q function as the sum of state value function and advantage function

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

However, this equation is unidentifiable in the sense that given Q we cannot recover V and A uniquely. To address this issue of identifiability, we can force the advantage function estimator to have zero advantage at the chosen action. To achieve this, we let the last module of the network implement the forwarding aggregation mapping

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a'} A(s,a';\theta,\alpha)\right)$$

Now, we obtain

$$Q(s,a^*;\theta,\alpha,\beta) = V(s;\theta,\beta)$$

Thus, the stream V provides an estimate of the value function, while the other stream produces an estimate of the advantage function. An alternative aggregation module is suggested by replacing the max operator with an average, as shown in Fig.4,

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

The average aggregation loses the original meaning of V and A because they are now off-target by a constant, but it increases the stability of the optimization because the advantages only need to change as fast as the mean, instead of having to compensate any change to the optimal action's advantage. Note that while subtracting the mean helps identifiability, it does not change the relative rank of the A values, preserving any greedy or ϵ-greedy policy based on Q values.

Training of the dueling architectures, as with standard Q networks (e.g. the deep Q-network of Mnih et al. (2015)), requires only back-propagation. The estimates V (s; θ, β) and A(s, a; θ, α) are computed automatically without any extra supervision or algorithmic modifications.

6.5 Example: Mountain Car

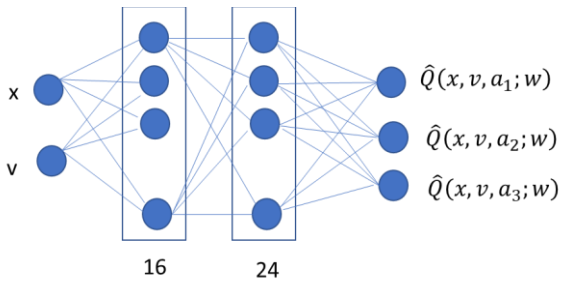
In Example 7 in Chapter 4, we solved the mountain car problem by tabular Q-learning. Now, Instead of updating directly the Q(s,a) table for all possible (s,a) combinations, we approximate the Q-table using a parameterized function $\hat{Q}(s,a;W)$ that is implemented by a neural network (called Q net) with parameter W. To learn the Q net, we generate a training dataset in which each sample is a four tuple (s,a,r,s') from sampled episodes. Each episode is generated through an e-greedy policy with respect to the current Q net (i.e. current policy). A target network, a copy of Q net, is used to generate the estimated ground truth i.e. target, $r+\gamma \max_{a'} \hat{Q}(s',a';W^{-})$. But the target network is updated less frequently for training stability. The diagram of the reinforcement learning is shown in Fig.2.

State space: (x,v), and action space: a set of (0,1,2): 0—left, 1—no-act, 2—right

```
#!/usr/bin/env python
# coding: utf-8
# In[1]:

#This is a bunch of initialization
import gym
import random
import torch
import torch.nn as nn
from torch.utils.data import Dataset
env = gym.make('MountainCar-v0') #action = (0,1,2) = (left, no_act, right)
print(env.observation_space)
print(env.action_space)

Box(-1.2000000476837158, 0.6000000238418579, (2,), float32)
Discrete(3)
```

Q neural network

Input layer:
Hidden layer1: (2, 16) + LeakyReLU
Hidden layer2: (16, 24) + LeakyReLU
Output layer: (24, 3), Loss：MSE
The dimension of the input layer is based on the dimension of state, and the environment observes several dimensions for several inputs. The hidden layer is arbitrary, after all, the function to be fitted should be relatively simple. The output layer dimension is based on action.
In other words, the network input is state, and the output is the Q value of each action.
In the code, we list two implementations of the Q net. You only need to run one of them.



```
# In[]:

#Simple linear model
def GetModel():
    #In features:2(state) ,out:3 action q
    return nn.Sequential(nn.Linear(2, 16),
                         nn.LeakyReLU(inplace=True),
                         nn.Linear(16,24),
                         nn.LeakyReLU(inplace=True),
                         nn.Linear(24,3))
```

```
# In[2]:

class GetModel(nn.Module):
    def __init__(self):
        super(GetModel, self).__init__()
        self.fc1 = nn.Linear(2,16)
        self.ReLU1 = nn.LeakyReLU(inplace=True)
        self.fc2 = nn.Linear(16,24)
        self.ReLU2 = nn.LeakyReLU(inplace=True)
        self.fc3 = nn.Linear(24,3)
    def forward(self, x):
        out = self.ReLU1(self.fc1(x))
        out = self.ReLU2(self.fc2(out))
        out = self.fc3(out)
        return out
```

Create dataset

Define a subset of class Dataset: **RLDataset( )**

1) __init__ function: specify ***samples*** (original format from env) as input, ***self.samples*** (torch tenor type for observation) as its output. A function self.transform(self, samples) will be defined to implement this transform from env state format to tensor.
2) __getitem__ function: get one example from self.samples, so that in the later main training loop we can load the dataset as

```
# trainset = RLDataset(train_samples)
# trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
#                shuffle=True, num_workers=0,pin_memory=True)
```

3) __len__ function: each call requests a sample index for which the upper bound is specified in the ***__len__*** method.
4) Transform(self, samples) function: implement a transform for observations from evn format to torch tensor format.

```
# In[3]:

#Create data set
class RLDataset(Dataset):
    def __init__(self, samples, transform = None, target_transform = None):
        #samples = [(s,a,r,s_), ...]
        self.samples = self.transform(samples)
    def __getitem__(self, index):
        #if self.transform is not None:
```

```
        #    img = self.transform(img)
        return self.samples[index]
    def __len__(self):
        return len(self.samples)
    def transform(self, samples):
        transSamples = []
        for (s,a,r,s_) in samples:
            sT = torch.tensor(s,).float()
            sT_ = torch.tensor(s_).float()
            transSamples.append((sT, a, r, sT_))
        return transSamples
```

Generating samples from env

Given the current Q net (q values), epsilon, the number of episodes, max steps of one episode, we will sample the episodes from env using e-greedy policy.

```
#   GetSamplesFromEnv(env, model, epoch, max_steps, drop_ratio = 0.8)
```

where

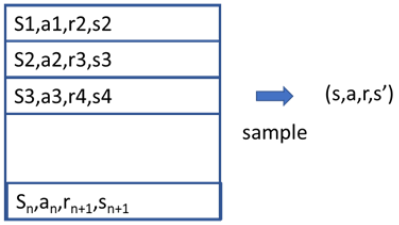
`model` : Q net
`epoch`: the number of episodes to be generated, e.g. 10
`max_steps`: the maximum of time steps for one episode, e.g. 200, even if the episode does not end.
`drop_ratio`: random action probability

we modify the reward: increasing the reward when it is approaching the flag.
1. When the car is between -0.2~-0.15, an additional reward of 0.2 will be given for each step
2. When the car is between -0.15~-0.1, an additional reward of 0.5 will be given for each step
3. When the car is at -0.1~*, an additional reward of 0.7 will be given for each step

How to use this function? We call this function, and add the samples to the old train_samples (list for example (3900, 4)). If the resulting train_samples has a length more than 4000, we discard the oldest samples to keep the training set size <4000 samples. The number of discarded samples is equal to the newly added samples. Then the updated train_samples can be used by RLDataset class.

For example,

```
#    tmpSample = GetSamplesFromEnv(env,net_eval, sample_times, 200, drop_ratio)
#    train_samples += tmpSample
#    #dataset contains the latest samples not exceeding 4000
#    if len(train_samples) > 4000:
#        train_samples = train_samples[len(tmpSample):len(train_samples)]
#    trainset = RLDataset(train_samples)

#    import numpy as np
#    np.shape(train_samples)
#    (3980,4)
```



```
# In[4]:


#Sampling environment function, you can set the probability of random operation. The
#focus is on the design of the reward


def GetSamplesFromEnv(env, model, epoch, max_steps, drop_ratio = 0.8):
    train_samples = []
    each_sample = None
    env.reset()
    observation_new = None
    observation_old = None
    model.eval()
    for i_episode in range(epoch):
        observation_new = env.reset()
        observation_old = env.reset()
        for t in range(max_steps):
            #env.render()
            #print(observation)
            if random.random() > 1-drop_ratio: # if drop_ratio is <0, no random action
                action = env.action_space.sample()
            else:
                inputT = torch.tensor(observation_new).float()
                action = torch.argmax(model(inputT)).item()
                #print(action)
            observation_new, reward, done, info = env.step(action)
            #print(reward)
            #We record samples.
            if t > 0 :
                if observation_new[0] > -0.1:
                    reward += 0.7
                elif observation_new[0] > -0.15:
                    reward += 0.5
                elif observationnew[0] > -0.2:
                    reward += 0.2
                each_sample = (observation_old, action, reward, observation_new)
                train_samples.append(each_sample)

            observation_old = observation_new

            if done:
              #Failed samples are not printed
                if t != 199:
                    print("Episode finished after {} timesteps".format(t+1))
                break
    return train_samples
```

TrainNet function

Now, we define a function to train the Q net for a certain number of epochs. The format is

```
#  TrainNet(net_target, net_eval, trainloader, criterion, optimizer,
             device,epoch_total, gamma)
```

Input:

`net_target`: target network that calculates the target $r + \gamma \max_{a'} \hat{Q}(s', a'; W^{-})$

`net_eval`: Q network that is to be trained, $\hat{Q}(s', a'; W)$

`trainloader`: RLDataset loader, can be enumerated for accessing each sample.
`criterion`: specifies the loss
`optimizer`: specifies the optimizer
`device`: CPU or GPU
`epoch_total`: the total number of epochs for training, e.g. we use 10.

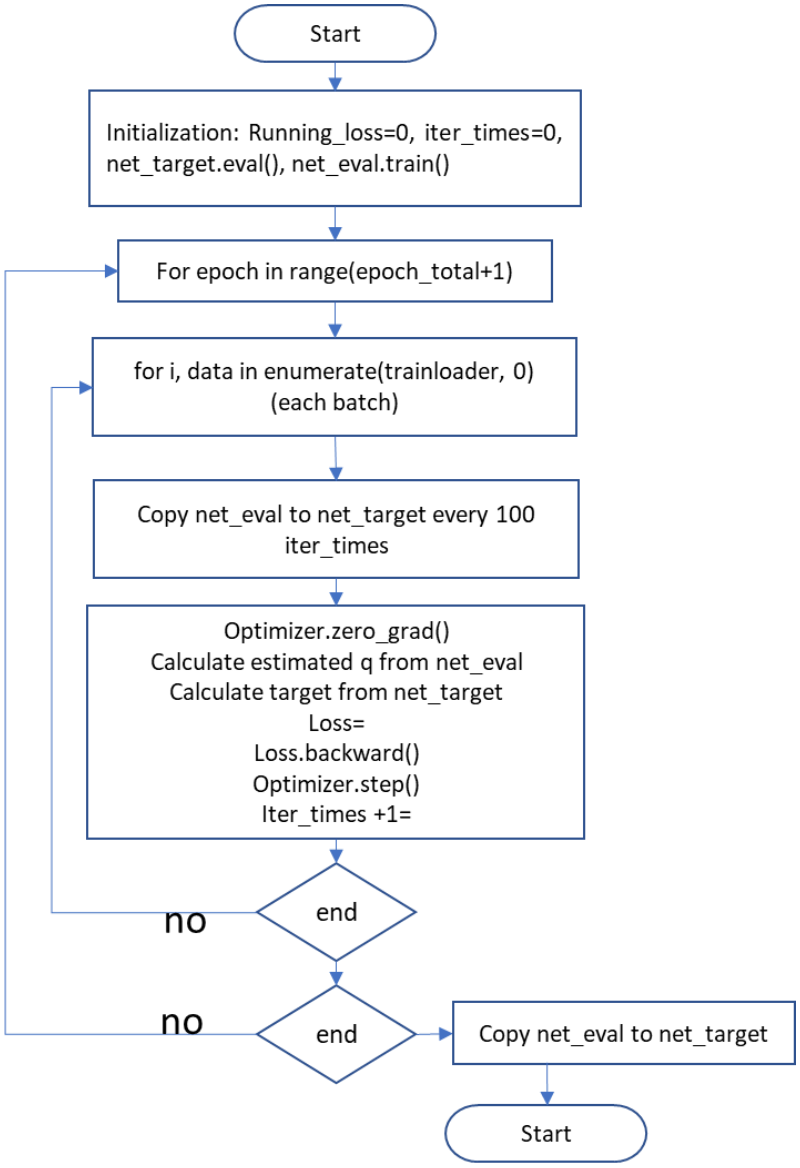
Result/output:

`net_eval.state_dict()`: net_eval is updated at each batch
`net_target.load_state_dict(net_eval.state_dict())`: net_target is updated ever 100 batch
But at the end of this training period, net_eval and net_target are synchronized (i.e. the same).

Code:

```
# In[5]:


#Training network. The gather function may be more convoluted here, and the dual
#network update is more difficult to understand. Ignore these, the same as the normal
#training cycle
#gamma is the attenuation factor in Bellman equation

def TrainNet(net_target, net_eval, trainloader, criterion, optimizer, device,
epoch_total, gamma):
    running_loss = 0.0
    iter_times = 0
    net_target.eval()
    net_eval.train()
    for epoch in range(epoch_total + 1):
        #if epoch > 0:
            #print('epoch %d, loss %.5f' % (epoch, running_loss))
        running_loss = 0.0
        if epoch == epoch_total:
            break
        for i, data in enumerate(trainloader, 0):
            if iter_times % 100 == 0:
                net_target.load_state_dict(net_eval.state_dict())
            s,a,r,s_ = data
            optimizer.zero_grad()

            #output = Q_predicted.
            q_t0 = net_eval(s)
            q_t1 = net_target(s_).detach()
            #q_t1 = gamma * (r + torch.max(q_t1,dim=1)[0])
            q_t1 = r + gamma* torch.max(q_t1,dim=1)[0]

            loss = criterion(q_t1, torch.gather(q_t0, dim=1,
                       index=a.unsqueeze(1)).squeeze(1))
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            iter_times += 1
    net_target.load_state_dict(net_eval.state_dict())    # synchronization
    #print('Finished Training')
```

Flowchart for understanding TrainNet function:

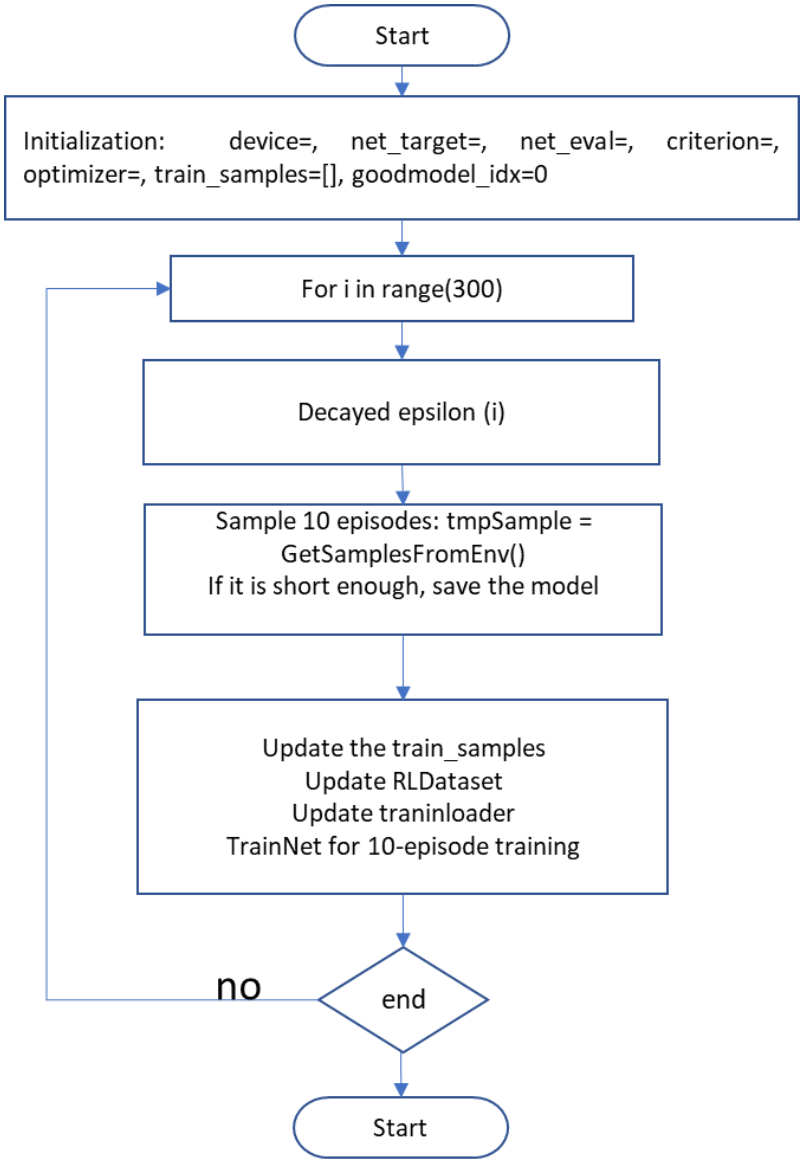


Main code for training

The following is the main code for our training. We call TrainNet function 300 times (in our case). Before we call TrainNet each time, we generate new 10 (for our case) episodes into the training set but keep the total training set < 400 samples by discarding oldest samples. Please note that we train the Q-net for 10 epochs by one single call TrainNet.

After running, whenever a group of 10 sampled episodes is short enough (less than 160x10 time steps for our case), the model is saved. In general, there are multiple model files saved.

Flowchart:




```
#Finally is a lot of main loop
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net_target, net_eval = GetModel(), GetModel()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(net_eval.parameters(),lr=0.01)
train_samples = []

# In[6]:
```

```
#This pile is for testing to see the effect
#PATH = 'mountain_model/goodmodel42.pth'
#net_eval.load_state_dict(torch.load(PATH))
#net_target.load_state_dict(torch.load(PATH))
#GetSamplesFromEnv(env,net_eval, 20, 200, 0)
goodmodel_idx = 0
for i in range(300):
    print(i)
    drop_ratio = 0.8 - 0.0052*i    # 0.0052 is decay factor
                                   # drop_ratio is the probability of random action
    #drop_ratio = 0.8
    sample_times = 10
    tmpSample = GetSamplesFromEnv(env,net_eval, sample_times, 200, drop_ratio)
    train_samples += tmpSample
    # if the number of time seps is smaller, the model is better, and save it.
    if len(tmpSample) < sample_times * 160:
        print("good model!save it!")
        torch.save(net_eval.state_dict(), "goodmodel" + str(goodmodel_idx) + ".pth")
        goodmodel_idx += 1
    #dataset contains the latest samples not exceeding 4000
    if len(train_samples) > 4000:
        train_samples = train_samples[len(tmpSample):len(train_samples)]
    trainset = RLDataset(train_samples)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True,
            num_workers=0,pin_memory=True)
    TrainNet(net_target, net_eval, trainloader, criterion, optimizer, device, 10, 0.9)
    #PATH = "model/model"+str(i)+".pth"
    #torch.save(net_eval.state_dict(), PATH)
env.close()
```

```
…
Episode finished after 136 timesteps
277
278
Episode finished after 133 timesteps
Episode finished after 133 timesteps
Episode finished after 133 timesteps
Episode finished after 137 timesteps
Episode finished after 132 timesteps
Episode finished after 133 timesteps
Episode finished after 130 timesteps
Episode finished after 132 timesteps
Episode finished after 133 timesteps
Episode finished after 136 timesteps
good model!save it!
279
…
```

Testing

Finally, we test our saved models by visualizing the game env.render() for 10 episodes

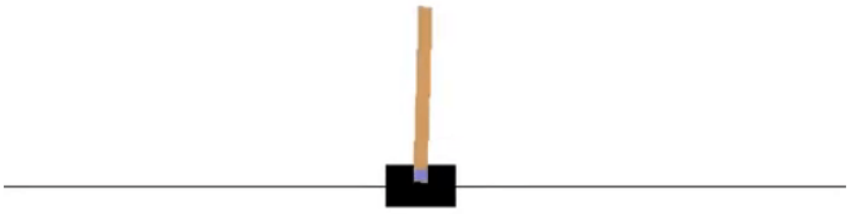
```
# In[12]:


test_model = GetModel()
test_model.load_state_dict(torch.load("goodmodel1.pth"))
test_model.eval()
with torch.no_grad():
```

```
    for i in range (10):
        done = False
        observation = env.reset()
        t=0
        while not done:
            t += 1
            env.render()
            inputT = torch.tensor(observation).float()
            action = torch.argmax(test_model(inputT)).item()
            observation_new, reward, done, info = env.step(action)
            observation = observation_new
            if done:
                if t <= 199:
                    print("success at time step", t, "episode", i)

                break
env.close()
```

```
success at time step 113 episode 0
success at time step 120 episode 1
success at time step 122 episode 2
success at time step 114 episode 3
success at time step 123 episode 4
success at time step 120 episode 5
success at time step 132 episode 6
success at time step 118 episode 7
success at time step 121 episode 8
success at time step 123 episode 9
```

6.6 Example: Cart pole

We will be using the CartPole-v0 environment provided by OpenAI GYM. A pole is attached by a joint to a cart, which moves along a frictionless track, shown below. The pendulum starts upright, and the goal is to prevent it from falling over by adjusting the cart's velocity.

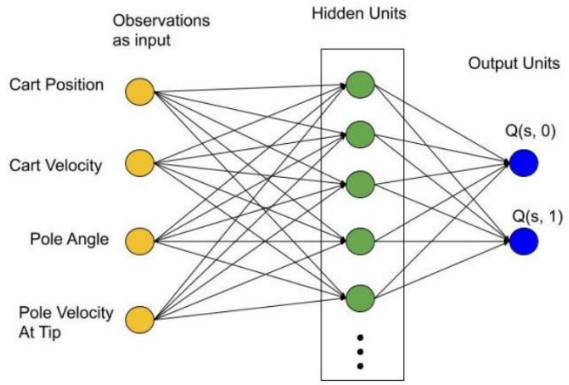


State Space: the observation of this environment is a four tuple: (cart position, cart velocity, pole angle, pole velocity at tip).
Action space: 0—left, 1—right.
Reward: the reward is 1 for every step taken, including the termination step.
Episode Termination: (one of the follows is met)

1) Pole Angle is more than ±12°
2) Cart Position is more than ±2.4 (center of the cart reaches the edge of the display)
3) Episode length is greater than 200 (500 for v1).

Solved Requirements (we can define various standards for solution, here is one)
Considered solved when the average reward is greater than or equal to 195.0 over 100 consecutive trials.
The python code is almost the same as the mountain car project, except:

1) Q-net (4 inputs and 2 outputs)
   Input layer:
   Hidden layer1: (4, 16) + LeakyReLU
   Hidden layer2: (16, 24) + LeakyReLU
   Output layer: (24, 2)



2) Reward: for each time step, the reward is 1. To speed up training, we modify the reward for the end of episode is -1.
3) The condition to end the training: if the average length of 100 consecutive episodes is equal or more than 199.

6.7 Atari Games

In the previous examples, such as mountain car and cart pole, the state observed from the environment is represented by a few quantities. Thus, a simple neural network (not deep neural network) is sufficient to learn the Q function. However, if the state is represented by images or videos, DQN is required to learn the features of images.

Mnih et al. (2015) introduce DQN through experiments on Atari games, and made a milestone in Deep reinforcement learning. Their work demonstrates that the DQN agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. In this section, we introduce Mnih's work.

Fig.5 shows the deep convolutional neural network which learns an approximate value function $Q(s, a; \theta_i)$, where θ_i are the parameters of the Q-network at iteration i. To perform experience replay we store the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t in a data set $D_t = \{e_1, \dots, e_t\}$. During learning, we apply Q-learning updates, on samples (or minibatches) of experience $(s, a, r, s') \sim U(D)$, drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

where γ is the discount factor, θ_i are the parameters of the Q-network at iteration i and θ_i^- are the parameters of the target network at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters θ_i every C steps and are held fixed between individual updates.

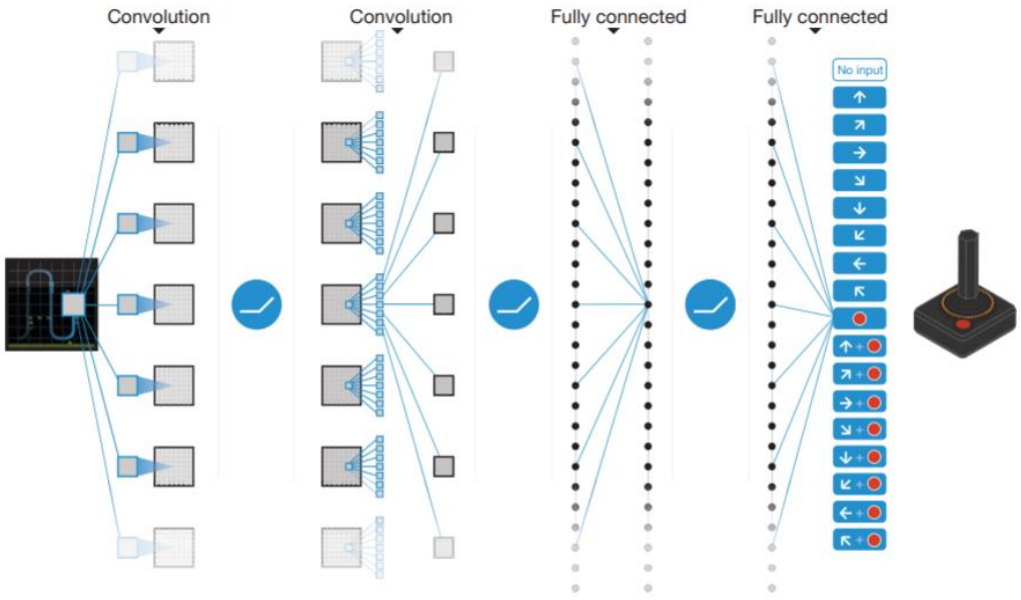


Fig. 5 Schematic illustration of the convolutional neural network for Q(s,a) function. (from Mnih 2015)

A preprocessing ϕ is applied to raw images which are 210x160 images with a 128-colour palette. First, to encode a single frame we take the maximum value for each pixel color value over the frame being encoded and the previous frame. This was to remove flickering that is present in games where some objects appear only in even frames while other objects appear only in odd frames. Second, we extract the Y channel as luminance from the RGB frame and rescale it to 84x84. Finally, the *m* most recent frames are stacked to produce the input the Q-network. (m=4). Thus, the input to the Q-network consists of an 84x84x4 image produced by the preprocessing.

The Q-network is composed of three convolutional layers and two fully connected layers. Each hidden layer is followed by a ReLU. The first hidden layer convolves 32 filters of 8x8 with stride 4 with the input image. The second hidden layer convolves 64 filters of 4x4 with stride 2. The third hidden layer convolves 64 filters of 3x3 with stride 1. The final hidden layer is fully connected and consists of 512 units. The output layer is a fully connected linear layer with a single output of each valid action. The number of valid actions varied between 4 and 18 on the games.

Algorithm: deep Q-learning with experience replay.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode =1,M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For t=1,T **do**
        With probability ε select a random action a_t
        otherwise select $a_t = arg\max_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
Set

$$y_j = \begin{cases} r_j & if\ episode\ termnates\ at\ step\ j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta_i^-) & otherwise \end{cases}$$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
Every C steps reset $\hat{Q} = Q$

End For

End For

For more details on this work, please refer to (Mnih 2015, Human-level control through deep reinforcement Learning, *Nature*)

6.8 Summary

In this chapter, we present the integration of neural networks with Q-learning, which leads to deep Q-networks (DQNs). The neural network approximates the Q function. By interacting with the environment, an agent samples the episodes and trains the neural network. The neural network can generalize for unseen samples (state-action pairs). Thus, DQN can handle the reinforcement learning problems with large discrete state space or continuing state space.

For training stability, we introduce two techniques for DQN training: experience replay and fixed target Q-network. In experience replay, we store the transitions (s,a, r, s') in a memory, and then randomly choose the transitions in minibatch to train the DQN in a general supervised learning way. This randomization removes the correlation among training samples. In fixed target Q-network, we keep the target Q-network unchanged during a certain number of iterations, so that the target computed by the target Q-network is relatively stable, which is essential for training convergence.

Further improvements have been discussed, including double DQN, prioritized replay, and dueling DQN.

Finally, we demonstrate the DQNs through a few examples. The first example is mountain car. The detailed Python codes are provided for this example. The problem of cart pole is very similar to mountain car. The differences are identified. As a breakthrough and milestone in deep reinforcement learning, the work by Mnih 2015 on Atari games is briefly introduced.

Chapter 7 Policy Gradient Methods

(resources: Stanford CS234 lecture 8, Sutton book ch.13, David Silver slides)

Learning Objectives

Upon the completion of this chapter, one should be able to

- Understand the difference between value-based RL methods and policy-based RL methods
- Understand the policy gradient theorem
- Understand and implement the following policy gradient methods
  REINFORCE
  Q-actor-critic
  Advantage actor critic

7.1 Policy-Based RL

In the previous chapter, we approximated the value or action-value function use parameters θ,

$$V(s;\theta) \approx V^{\pi}(s)$$
$$Q(s,a;\theta) \approx Q^{\pi}(s,a)$$

We learned the functions from the episodes, and then generated a policy (e.g. ϵ-greedy) directly from the learned function. For example, Q-learning learns an optimal policy through policy iteration that involves evaluating Q function. The Q values are dictating the agent how to behave, and thus defining the policy. This can be written as

$$\pi(s) = arg\max_{a} Q(s,a)$$

Which mean that the result of policy π, at every state, is the action with the largest Q.

In this chapter, we consider methods that learn a parameterized policy directly without consulting a value function. In other words, a policy gradient method is a policy iteration approach where policy is directly manipulated to reach the optimal policy that maximizes the expected return. Specifically, the policy can be represented by

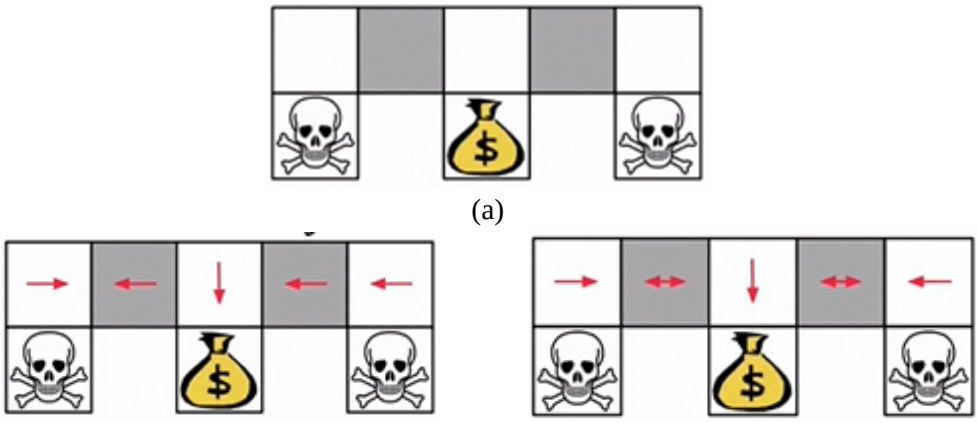
$$\pi_{\theta}(s,a) = \mathbb{P}[A_t = a | S_t = s; \theta_t = \theta]$$

which is the probability of taking action a at time t given the environment is in state s at time with parameter θ. The goal is to find a policy π which leads to the highest value function V^{π}. To achieve this goal, we need to learn the parameter θ based on the gradient of some scalar performance measure $J(\theta)$, called objective function. All methods directly learning the policy are called **policy gradient methods** or policy-based

reinforcement learning. The methods we learned in previous chapters, such as TD, SARSA, and Q-learning, are called as **value-based methods**.

Why do we want to learn stochastic policies? In tabular MDP, if there exists a policy which is deterministic and optimizes the value function, we don't need to learn a stochastic policy. However, in some situations, a stochastic policy is required to achieve an optimal value function. For example, in the two-player game of rock-paper-scissors, a uniform random policy is optimal. Another example is the aliased grid-world, shown in Fig.1. The agent cannot distinguish between the two grey states. An optimal deterministic policy will either move W in both grey states (illustrated in Fig.1(b)) or move E in both grey states (not illustrated). In either optimal deterministic policy, it is possible for the agent to get stuck and never reach the money. Value-based RL learns a near-deterministic policy (e.g. greedy or e-greedy), but it will traverse the corridor for a long time. An optimal stochastic policy will randomly move E or W in grey states so that it will reach the goal state (money) in a few steps with high probability. Policy-based RL can learn the optimal stochastic policy.

$$\pi_\theta(wall\ to\ N\ and\ S, move\ E) = 0.5$$
$$\pi_\theta(wall\ to\ N\ and\ S, move\ W) = 0.5$$



(a)

(b) optimal deterministic policy (c) an optimal stochastic policy

Fig.1 Aliased grid-world

Another reason why policy-based RL is attractive is due to environments with a large number of actions, or in the extreme case with continuous action space.

The advantages of policy-based RL include better convergence properties, effective in high-dimensional or continuous action spaces, and learning stochastic policies. The challenges include: they typically converge to a local rather than global optimum, and evaluating a policy is typically inefficient and high variance.

7.2 Policy Objective Functions

To learn policy directly from experience, we need to define some scalar performance measure or objective function J(θ) with respect to its argument θ. Given policy $\pi_\theta(s, a) = \mathbb{P}[a|s; \theta]$, the goal is to find the parameter θ for the best performance. The question is, how do we measure the quality of a policy π_θ in

terms of J(θ)? The episodic and continuing cases define the performance measure differently, and thus have to be treated separately.

In episodic cases, we can use the value of the start state s_0, by assuming every episode starts in some particular state (non-random) s0 without losing any meaningful generality

$$J_0(\theta) = V^{\pi_\theta}(s_0) \tag{1.a}$$

In continuing cases, we can use the average value

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s) \tag{1.b}$$

Or the average reward per time-step

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a \tag{1.c}$$

where $d^{\pi_\theta}(s)$ is probability distribution of states under the policy π_θ. Thus, policy-based reinforcement learning is an optimization problem: find θ that maximizes J(θ). We can use gradient to solve the optimization problem.

7.3 Policy Gradient

Let J(θ) be any policy objective function. Policy gradient algorithms search for a local maximum in J(θ) by ascending the gradient of the policy, w.r.t. parameter θ

$$\theta = \theta + \Delta\theta = \theta + \alpha \nabla_\theta J(\theta) \tag{2}$$

where $\nabla_\theta J(\theta)$ is the **policy gradient**, and α is a step-size parameter.

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix} \tag{3}$$

We can compute the policy gradient numerically by **finite differences**. For each dimension θ_k, we estimate kth partial derivative of objective function w.r.t. θ_k by perturbing θ_k by small amount ϵ in kth dimension

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon} \tag{4}$$

where u_k is unit vector with 1 in kth component, 0 elsewhere. In general, this method works for arbitrary policies, even if policy is not differentiable, but is noisy and inefficient.

7.3.1 Likelihood Ratio Policy Gradient

Consider the state value of the start state s_0, given a parameterized policy $\pi_\theta(s,a)$

$$V(s_0, \theta) = \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{T} R(s_t, a_t); \pi_\theta, s_0] \tag{5}$$

This state value is the expectation of the return, i.e. the rewards accumulated from the start state s_0 to the end of the episode by following the policy. It can also be represented in different ways. For example,

$$V(s_0, \theta) = \sum_a \pi_\theta(a|s_0)\, Q(s_0, a; \theta) \tag{6}$$

Let $\tau = (s_0, a_0, r_0 \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$ be a state-action trajectory, $P(\tau;\theta)$ be the probability over trajectory τ when executing policy $\pi_\theta(s,a)$ starting in state s_0, and $R(\tau) = \sum_{j=0}^{T-1} r_j$ be the sum of rewards for a trajectory τ. Then

$$V(s_0,\theta) = \mathbb{E}_{\pi_\theta}[R(\tau)] = \sum_\tau P(\tau;\theta)\, R(\tau) \tag{7}$$

For a concise notation, we drop s_0 in $V(s_0,\theta)$, without a meaning change. Thus, our goal is to find the policy parameter θ (or policy $\pi_\theta(s,a)$) that maximizes the expected return of a trajectory generated by the policy, i.e.,

$$arg\max_\theta V(\theta) = arg\max_\theta \sum_\tau P(\tau;\theta)\, R(\tau) \tag{8}$$

The optimal θ can be searched based on the gradient with respect to θ. The gradient can be represented as

$$\nabla_\theta V(\theta) \triangleq \nabla_\theta V(s_0,\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[R(\tau)] = \nabla_\theta \sum_\tau P(\tau;\theta)\, R(\tau) = \sum_\tau \nabla_\theta P(\tau;\theta)\, R(\tau)$$
$$= \sum_\tau P(\tau;\theta)\, R(\tau) \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} = \sum_\tau P(\tau;\theta)\, R(\tau) \nabla_\theta \log P(\tau;\theta) = \mathbb{E}_{\pi_\theta}[R(\tau)\nabla_\theta \log P(\tau;\theta)] \tag{9}$$

This gradient can be approximated with *m* sample trajectories under policy $\pi_\theta(s,a)$

$$\nabla_\theta V(\theta) \approx \frac{1}{m}\sum_{i=1}^{m} R\left(\tau^{(i)}\right)\nabla_\theta \log P\left(\tau^{(i)};\theta\right) \tag{10}$$

Furthermore, the term in the right-hand side of the above equation, $\nabla_\theta \log P\left(\tau^{(i)};\theta\right)$, can be calculated based on the policy (not the environment) only

$$\nabla_\theta \log P(\tau;\theta) = \nabla_\theta \log\left[\mu(s_0)\prod_{t=0}^{T-1} \pi_\theta(a_t|s_t)\, P(s_{t+1}|s_t,a_t)\right]$$
$$= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{11}$$

This term is called log likelihood. It measures the likelihood of the observed trajectory.

Thus,

$$\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau)\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)] \approx \frac{1}{m}\sum_{i=1}^{m}\left[R\left(\tau^{(i)}\right)\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right] \tag{12}$$

Score function: Assume policy is differentiable whenever it is non-zero, and we know the gradient $\nabla_\theta \pi_\theta(s,a)$. The **score function** is defined by

$$\nabla_\theta log \pi_\theta(s,a) = \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \tag{13}$$

Note that two expressions $\pi_\theta(s,a)$ and $\pi_\theta(a|s)$ are the same thing and interchangeable.

7.3.2 Two Policy Functions

In the subsequent discussions, we assume that the policy be represented by a parameterized function with parameters θ. Two functions, softmax and Gaussian, are typically used for discrete action space and continuous action space, respectively.

Softmax policy: The probability of action is proportional to exponential of linear combination of features

$$\pi_\theta(s,a) = \frac{e^{\phi(s,a)^T\theta}}{\sum_b e^{\phi(s,b)^T\theta}} \tag{14}$$

where $\phi(s,a)$ is the feature function. Then the score function is

$$\nabla_\theta log\pi_\theta(s,a) = \phi(s,a) - \sum_b \pi_\theta(s,b)\,\phi(s,b) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)] \tag{15}$$

Gaussian policy: in continuous action spaces, a Gaussian policy is natural. The policy follows a Gaussian distribution with mean equal to the linear combination of state features and variance fixed or parameterized:

$$a \sim \mathcal{N}(\mu(s), \sigma^2)$$
$$\mu(s) = \phi(s)^T\theta$$

The score function is

$$\nabla_\theta log\pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2} \tag{16}$$

In summary, the policy gradient is the expectation of the product of the trajectory return and its accumulated score function. In the rest of this chapter, we will describe how to estimate the gradient in practice.

7.3.3 Policy Gradient Theorem

In this section, we will present the policy gradient theorem and give a detailed proof. The policy gradient theorem plays a core role in policy gradient methods. Before we present policy gradient theorem, we consider a simple class of one-step MDPs, which start in state s with distribution d(s), and terminate after one time-step reward $r = R_{s,a}$. The objective function is

$$J(\theta) = \mathbb{E}_{\pi_\theta}[r] = \sum_{s\in S} d(s) \sum_{a\epsilon A} \pi_\theta(s,a) R_{s,a} \tag{17}$$

We can compute the policy gradient using likelihood ratios (i.e. score functions)

$$\nabla_\theta J(\theta) = \sum_{s\in S} d(s) \sum_{a\epsilon A} \pi_\theta(s,a) \nabla_\theta log\pi_\theta(s,a) R_{s,a} = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) r] \tag{18}$$

To apply the likelihood ratio approach to multi-step MDPs, we just need to replace the instantaneous reward r with a long-term value $Q^\pi(s,a)$. This generalized approach to multi-step MDPs is called policy gradient theorem. Furthermore, policy gradient theorem applies to three different objective functions: start state objective, average reward objective and average value objective.

https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#a2c

[Policy Gradient Theorem] For any differentiable policy $\pi_\theta(a|s)$, and for any of the policy objective functions, $V(\theta) = J_0(\theta), J_{avR}(\theta), or \frac{1}{1-\gamma} J_{avV}(\theta)$, the policy gradient is

$$\nabla_\theta J(\theta) \propto \sum_s [d^\pi(s) \sum_a Q^{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s)] \tag{19}$$

The proof of the theorem can be found in Sutton's book. The proof is presented below for those who are interested. We first start with the gradient of the state value function:

$$\nabla_\theta V^\pi(s)$$
$$= \nabla_\theta \left(\sum_a \pi_\theta(a|s) Q^\pi(s,a) \right)$$
$$= \sum_a \left(Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) + \pi_\theta(a|s) \nabla_\theta Q^\pi(s,a) \right) \quad \text{;Derivative product rule}$$
$$= \sum_a \left(Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) + \pi_\theta(a|s) \nabla_\theta \sum_{s',r} P(s',r|s,a)\left(r + V^\pi(s')\right) \right) \quad \text{;Bellman backup one step}$$
$$= \sum_a \left(Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) + \pi_\theta(a|s) \sum_{s',r} P(s',r|s,a) \nabla_\theta \left(V^\pi(s')\right) \right) \quad \text{; constant derivative}$$
$$= \sum_a \left(Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta \left(V^\pi(s')\right) \right) \quad \text{; } P(s'|s,a) = \sum_r P(s',r|s,a)$$

Now we have

$$\nabla_\theta V^\pi(s) = \sum_a \left(Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta \left(V^\pi(s')\right) \right)$$

This equation has a recursive form for the gradient of the state value function. The future state value function gradient $\nabla_\theta \left(V^\pi(s')\right)$ can be repeated unrolled by the following the same equation.

Let's consider the following transition sequence and label the probability of transition from state s to state x with policy π_θ after k step as $\rho^\pi(s \to x, k)$

$$s \xrightarrow{a \sim \pi_\theta(a|s)} s' \xrightarrow{a \sim \pi_\theta(a|s'')} s'' \xrightarrow{a \sim \pi_\theta(a|s'')} \cdots$$

To understand $\rho^\pi(s \to x, k)$, consider three scenarios:

1) k=0, $\rho^\pi(s \to s, k = 0) = 1$
2) k=1, $\rho^\pi(s \to s', k = 1) = \sum_a \pi_\theta(a|s) P(s'|s,a)$, one step back.
3) Transition from s to x after k+1 steps by following the policy. We can first transition from s to a middle point s' after k steps and then go to the final state x by the last step. The overall transition probability can be represented recursively

$$\rho^\pi(s \to x, k+1) = \sum_{s'} \rho^\pi(s \to s', k) \rho^\pi(s' \to x, 1)$$

Now, we go back to unroll the recursive representation of the gradient. To simplify the math notation, we let

$$\phi(s) = \sum_a \left(Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) \right)$$

We have

$$\nabla_\theta V^\pi(s)$$
$$= \phi(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta \left(V^\pi(s')\right)$$
$$= \phi(s) + \sum_{s'} \sum_a \pi_\theta(a|s) P(s'|s,a) \nabla_\theta \left(V^\pi(s')\right)$$
$$= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \nabla_\theta \left(V^\pi(s')\right)$$

$$= \phi(s) + \sum_{s'} \rho^{\pi}(s \to s', 1) \left[\phi(s') + \sum_{s''} \rho^{\pi}(s' \to s'', 1) \nabla_{\theta} \left(V^{\pi}(s'') \right) \right]$$

$$= \phi(s) + \sum_{s'} \rho^{\pi}(s \to s', 1) \phi(s') + \sum_{s''} \rho^{\pi}(s \to s'', 2) \nabla_{\theta} \left(V^{\pi}(s'') \right)$$

$$= \phi(s) + \sum_{s'} \rho^{\pi}(s \to s', 1) \phi(s') + \sum_{s''} \rho^{\pi}(s \to s'', 2) \phi(s'') + \sum_{s'''} \rho^{\pi}(s \to s''', 3) \nabla_{\theta} \left(V^{\pi}(s''') \right)$$

$$= \cdots repeat\ unrolling\ to\ k \to \infty$$

$$= \sum_{x \in S} \sum_{k=0}^{\infty} \rho^{\pi}(s \to x, k) \phi(x)$$

The policy gradient is defined as the gradient of the start state value with respect to θ

$\nabla_{\theta} J(\theta) = \nabla_{\theta} V^{\pi}(s_0)$

$= \sum_{s \in S} \sum_{k=0}^{\infty} \rho^{\pi}(s_0 \to s, k) \phi(s)$; let $\eta(s) = \sum_{k=0}^{\infty} \rho^{\pi}(s_0 \to s, k)$ the probability from s_0 to s ; $\eta(s)$ is the probability of s showing up

$$= \sum_{s \in S} \eta(s) \phi(s)$$

$= (\sum_{s'} \eta(s')) \sum_{s \in S} \frac{\eta(s)}{\sum_{s'} \eta(s')} \phi(s)$; normalize $\eta(s)$ to be a probability distribution

$$\propto \sum_{s \in S} \frac{\eta(s)}{\sum_{s'} \eta(s')} \phi(s) = \sum_{s \in S} d^{\pi}(s) \sum_{a} \left(Q^{\pi}(s, a) \nabla_{\theta} \pi_{\theta}(a|s) \right)$$

where $d^{\pi}(s)$ is stationary state distribution. In the episode case, the constant of proportionality, $\sum_s \eta(s)$, is the average length of an episode; in the continuing case, it is 1.

The theorem implies that the gradient of performance with respect to the policy parameters can be computed analytically without involving the derivative of the state distribution, and that the gradient value is proportional to the expectation value of the product of score function multiplied by the state-action value function. The gradient can be further written as

$$\nabla_{\theta} J(\theta) \propto \sum_{s} \left[d^{\pi}(s) \sum_{a} Q^{\pi_{\theta}}(s, a) \nabla_{\theta} \pi_{\theta}(a|s) \right] = \sum_{s} \left[d^{\pi}(s) \sum_{a} \pi_{\theta}(a|s) Q^{\pi_{\theta}}(s, a) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} \right]$$

$$= \mathbb{E}_{\pi}[Q^{\pi}(S_t, A_t) \nabla_{\theta} log \pi_{\theta}(A_t|S_t)]$$

where $\mathbb{E}_{\pi}$ refers to $\mathbb{E}_{s \sim d(s), a \sim \pi_{\theta}}$. The policy gradient theorem lays the theoretical foundation for various policy gradient algorithms. This vanilla policy gradient update has no bias but high variance. We will present the variations which were proposed to reduce the variance while keeping the bias unchanged.

Note that the policy gradient is proportional to the expectation of the product of state-action value function and score function. Since the policy gradient is scaled by a learning rate α when updating the parameter θ,

the constant of proportionality in the policy gradient theorem is not important and thus can be ignored. Therefore, we simply use the following notation,

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(S_t, A_t)\nabla_\theta log\pi_\theta(A_t|S_t)]$$

The policy gradient defines the direction in which we need to change the parameters to improve the policy in terms of the accumulated total reward. The gradient is proportional to the value of the action taken, and the gradient of log probability of the action taken. This implies that we are trying to increase the probability of actions that result in good total reward, and decrease the probability of actions with bad total reward. Expectation in the equation just means that we average the gradient of several steps that we have taken in the environment.

7.4 Monte-Carlo Policy Gradient: REINFORCE Algorithms

7.4.1 Policy update based on multiple trajectories

According the equation (12), we can update the policy parameters based on the total rewards of episodes R(τ). A straightforward way is to update the parameter once for one trajectory. We estimate the policy gradient by one example trajectory

$$\nabla_\theta V(\theta) \approx R\left(\tau^{(i)}\right) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} | s_t^{(i)}\right) \tag{20}$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta V(\theta)$$

The estimate is unbiased but very noisy (i.e. has high variance).

To reduce the variance, we can estimate the policy gradient using *m* trajectories under the current policy,

$$\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] \approx \frac{1}{m} \sum_{i=1}^{m} \left[R\left(\tau^{(i)}\right) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} | s_t^{(i)}\right)\right] \tag{21}$$

and then update parameters by gradient ascent

$$\theta \leftarrow \theta + \alpha \nabla_\theta V(\theta)$$

This results in our first algorithm, call REINFORCE. REINFORCE is acronym for "REward Increment = Nonnegative Factor Offset Reinforcement Characteristic Eligibility". We will introduce its variants in the subsequent sections in this chapter.

Algorithm 1 Monte Carlo Policy Gradient Algorithm: REINFORCE with episode returns

---

REINFORCE with episode returns

Initialize θ arbitrarily

Sample *m* trajectories by following current policy π_θ

Compute the returns of the m trajectories: $R\left(\tau^{(i)}\right)$, i=1,2,3…,*m*

Evaluate the gradient using these m trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \left[R\left(\tau^{(i)}\right) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} | s_t^{(i)}\right)\right]$$

Update the policy parameters:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Alternatively, we can update the parameters at each time step as follows.

Algorithm 1(b) REINFORCE (time-step update) with episode returns

---

Function REINFORCE with step-returns G_t
  Initialize θ arbitrarily
  for each episode $\{S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T\} \sim \pi_\theta$ do
    $G \leftarrow \sum_{k=2}^{T} \gamma^{k-1} R_k$ (consider discount)
    for t=1 to T-1 do
      $\theta \leftarrow \theta + \alpha G \nabla_\theta log \pi_\theta(S_t, A_t)$
    end for
  end for
  return θ
end function

7.4.2 Temporal structure of rewards for policy gradient

To update the policy more frequently (e.g. at each time-step), we will use the temporal structure to estimate the policy gradient, instead of the entire return of a trajectory. Equation (12) can be understood as the expected policy gradient for an entire trajectory,

$$\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{22}$$

We can derive the gradient estimate for a single step reward at t' in the same way we derived equation (22).

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[r_{t\prime}] = \mathbb{E}_{\pi_\theta}[r_{t\prime} \sum_{t=0}^{t\prime} \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{23}$$

Thus, the policy gradient can be obtained by summing up the single step reward over all time steps for a trajectory

$$\begin{aligned} \nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[R] &= \nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t'=0}^{T-1} r_{t'}\right] \\ &= \mathbb{E}_{\pi_\theta}[\sum_{t\prime=0}^{T-1}(r_{t\prime} \sum_{t=0}^{t\prime} \nabla_\theta \log \pi_\theta(a_t|s_t))] \end{aligned} \tag{24}$$

Be reorganizing the sum operators in (24), we obtain

$$\begin{aligned} \nabla_\theta V(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_{t'=0}^{T-1}\left(r_{t'} \sum_{t=0}^{t'} \nabla_\theta \log \pi_\theta(a_t|s_t)\right)\right] \\ = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T-1} r_{t'}\right)\right] &= \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{T-1}(\nabla_\theta \log \pi_\theta(a_t|s_t)\, G_t)] \end{aligned} \tag{25}$$

The policy gradient can be estimated using *m* trajectories

$$\nabla_\theta V(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \left[\sum_{t=0}^{T-1} G_t^{(i)} \nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right] \tag{26}$$

Let's compare equation (12) and equation (26). In equation (12), the score function at each time-step is weighted by the return of the entire trajectory while the score function in (26) is weighted by the return received after the time-step.

It would be not computationally efficient to estimate the policy gradient based on *m* trajectories according to equation (26). Instead we can update the policy gradient at every time step for a more frequent θ update

but with a higher variance. This every time step update idea results in algorithms 2 and 3, called REINFORCE, for policy gradient methods. In this algorithm, we update parameters by stochastic gradient ascent using the score function and return at each time step. This algorithm can be justified by policy gradient theorem with return Gt as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$. The issue of high variance will be addressed by its generalization in the subsequent sections, called Baseline.

Algorithm 2(a) REINFORCE (batch update) with time-step returns

---

REINFORCE with episode returns

Initialize θ arbitrarily

Sample *m* trajectories by following current policy π_θ

Compute the returns $G_t^{(i)}$ at the time step t in the ith trajectories for all i=1,2,..,M, t=1,2…, $T^{(i)}$

Evaluate the gradient using these m trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{M}\sum_{i=1}^{M}\left[\sum_{t=0}^{T-1} G_t^{(i)} \nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right]$$

Update the policy parameters:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Algorithm 2(b) REINFORCE (time-step update) with time-step returns

---

Function REINFORCE with step-returns G_t

Initialize θ arbitrarily

for each episode $\{S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T\} \sim \pi_\theta$ do

for t=1 to T-1 do

$G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-1} R_k$ (consider discount)

$\theta \leftarrow \theta + \alpha G_t \nabla_\theta log \pi_\theta(S_t, A_t)$

end for

end for

return θ

end function

7.5 Implementation of REINFORCE using Neural Networks

The goal of policy-based reinforcement learning is to a learn a stochastic policy $\pi_\theta(a|s)$, a probability distribution mapping from state to action, that will maximize the objective function J(θ). It is natural to use a neural network parameterized by θ to approximate the policy. For the scenario of discrete act space, the distribution is usually categorical with a softmax over the action logits. For continuous action space, the output of the neural network is parameters of a continuous distribution (e.g. the mean and variance of a Gaussian). Fig.2 shows an example neural network for the policy with 4-action space. Each output of the neural network represents the probability of one action for the state specified by the input s.

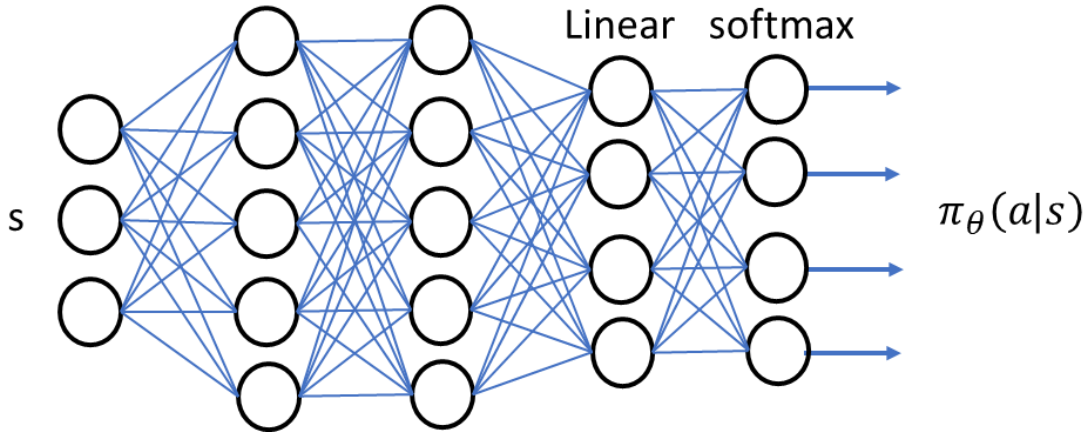


Fig.2 An example of neural network for policy

To gain some intuition regarding the neural network for policy gradient methods, let's consider the discrete action space shown in Fig.2, and compare it to a supervised learning classification task with K classes. We assume both networks (one for policy and another for classification) using softmax layer to normalize the logits to a probability distribution.

$$q_i(z) = \frac{e^{z_i}}{\sum_{j=0}^{K-1} e^{z_j}} \tag{27}$$

where z_i , i=0, 1, …, K-1, are the inputs of the softmax layer, called logits.

In a supervised learning classification task, we commonly use the cross-entropy function on the softmax output as a loss function. Given an input image x, we use a 1-hot encoded vector for the true distribution p , where the 1 is at the index of the true label (y):

$$p_i(x) = \begin{cases} 1 & if\ y = i \\ 0 & otherwise \end{cases}$$

The corresponding softmax output is a K-dimensional vector $q(x) = [q_0(x), q_i(x), \dots q_{K-1}(x)]^T$, representing the estimated probability distribution over K classes. The cross-entropy loss function for $p(x)\ and\ q(x)$ is define as

$$L(p, q) = -\sum_{i=0}^{K-1} p_i(x) log(q_i(x)) \tag{28}$$

For example, for an image classification task with K=4 classes (cat, dog, bird, car) with index mapping {0: cat, 1:dog, 2:bird, 0:car}, if the input image x is "dog", then the ground truth (or true distribution) is a vector p=[0, 1, 0, 0]^T. For a specific value of parameter θ, the output of the softmax layer may be [0.12, 0.61, 0.11, 0.16]^T, Then, the cross-entropy loss value is

$$L(p, q) = -0 \log(0.12) - 1 \log(0.61) - 0 \log(0.11) - 0 \log(0.16) = -\log(0.61)$$

Note that the loss function can be written as $-\log(q_y(x))$, where y is the index of the true label. The loss is equal to 0 when the classifier assigns the corresponding probability as 1 (i.e. 100% prediction for the true class) and is infinite when the classifier assign 0 probability to the true class (i.e. totally wrong prediction).

The reason why we use the cross-entropy loss function is that the derivatives (or gradients) of the loss function with respect to parameters θ is relatively simple for backward propagation during the neural network training process.

If we consider the contribution of a single training example x_i (i.e. a single image) to the gradient, the objective is to maximize the log-likelihood $logq_i(x_i)$, which is equivalent to minimizing the cross entropy loss -$logq_i(x_i)$. Thus, the standard SGD can be applied to the cross-entropy loss for the parameter updating

$$\theta_{j+1} = \theta_j + \alpha\nabla_\theta\big(logq_i(x_i)\big) = \theta_j - \alpha\nabla_\theta(-logq_i(x_i)) = \theta_j - \alpha\nabla_\theta L(\theta) \tag{29}$$

If we look at the PG gradient methods, a single transition contribution to the gradient is $G_t\nabla_\theta log\pi_\theta(A_t|S_t)$, and the update to the network parameters θ for a maximization task can be represented as gradient ascent

$$\theta \leftarrow \theta + \alpha G_t\nabla_\theta log\pi_\theta(A_t|S_t) = \theta - \alpha G_t\nabla_\theta(-log\pi_\theta(A_t|S_t)) \tag{30}$$

which is the same as the classification task when Gt has a constant value of 1. In general, the reward Gt determines the sign and magnitude of the gradient. We are using this sampled action as the true label, like image classification. The true probability distribution of policy will be a one-hot encoded vector in which the one is at the index of the sample action. However, we don't assume it is the optimal action and trust that the reward Gt which multiplies this gradient, to guide the gradient by both sign and magnitude.

From a point of practice view, the classifier specified by (29) can be easily implemented by mini-batch SGD training, illustrated in Fig.3. A batch of images, X, are provided as the input to the neural network that computes the corresponding predicted probability distributions over classes, q(X). The cross-entropy loss function block calculates the average loss over the batch, given the image labels. The loss will be used to update the parameters θ. It is very convenient to implement this training process by popular machine learning frameworks, such as PyTorch.

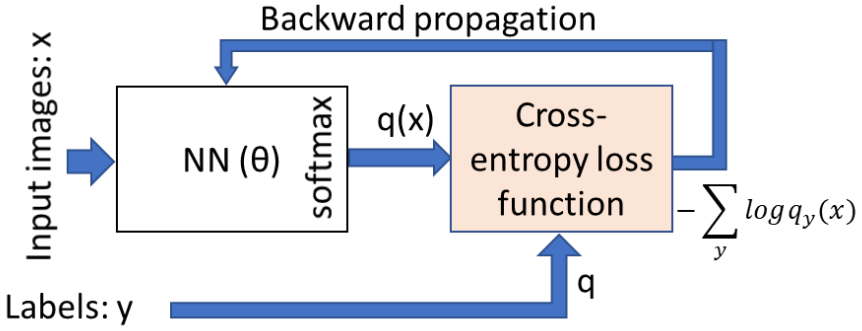


Fig.3 Training of a general image classifier

Be comparing equations (29) and (30), we find out that we can train the policy neural network in the same way as we do for the image classifier, except that we need to modify the loss by multiplying the log probabilities with return Gt. To generate the training examples (s_t, a_t, G_t), we have to interact with the environment based on the current policy. [Question: should we keep the policy (for example generation) fixed during a certain number of parameter updates? We did this for DQN for training stability.] Fig. 4 shows how we can train the policy neural network.

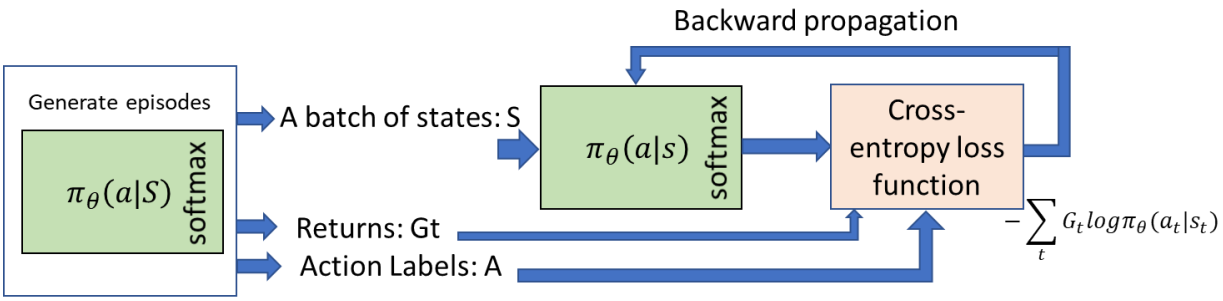


Fig.4 Block diagram of training neural network in REINFORCE

Thus, the REINFORCE algorithm can be equivalently described as the following steps:

1. Initialize the network with random weights θ
2. Generate a batch of full episodes (e.g. M episodes), saving all states, actions, and rewards.
3. For every step t of every episode i, calculate the discounted return $G_{m,t} = \sum_{i=t}^{T_m-1} \gamma^{i-t} R_{m,i+1}$, where γ is the discounted rate, $R_{m,i+1}$ is the reward at time step i (following the state at time i) of episode m, T_m is the time step of the terminal in episode m.
4. Calculate the average cross-entropy loss over all transitions in the batch $\mathcal{L} = -\frac{1}{\sum T_m} \sum_m \sum_t G_{m,t} \log\left(\pi_\theta\left(a_{m,t}|s_{m,t}\right)\right)$, where the log() is the output of the softmax layer for the state-action pair at time step t, $s_{m,t}$ and $a_{m,t}$, $\sum T_m$ is the total number of transitions for the batch.
5. Perform an SGD update of weights θ by an optimizer (e.g. Adam) that minimizes the loss.
6. Repeat from step 2 until converged.

As an example, we implement the REINFORCE algorithm for the cartpole task by PyTorch as follows.

```
#!/usr/bin/env python
# coding: utf-8

# In[42]:


import numpy as np
import matplotlib.pyplot as plt
import gym
import sys
import torch
```

```
from torch import nn
from torch import optim
print("PyTorch:\t{}".format(torch.__version__))


# In[91]:


class policy_estimator():
    def __init__(self, env):
        self.n_inputs = env.observation_space.shape[0] #for CatPole-v0: 4
        self.n_outputs = env.action_space.n # for CatPole-v0:2

        # Define network
        self.network = nn.Sequential(
            nn.Linear(self.n_inputs, 16),
            nn.ReLU(),
            nn.Linear(16, self.n_outputs),
            nn.Softmax(dim=-1))

    def predict(self, state):
        action_probs = self.network(torch.FloatTensor(state))
        return action_probs


# In[103]:


def discount_rewards(rewards, gamma=0.99):
    res = []
    sum_r = 0.0
    for r in reversed(rewards):
        sum_r *= gamma
        sum_r += r
        res.append(sum_r)
    return list(reversed(res))


# In[125]:


def reinforce(env, policy_estimator, num_episodes=3000,
              batch_size=10, gamma=0.99):
    # Set up lists to hold results
    total_rewards = []
    batch_rewards = []
    batch_actions = []
    batch_states = []
    batch_counter = 1
    avg_rewards_saved =[]
    # Define optimizer
    optimizer = optim.Adam(policy_estimator.network.parameters(),
                           lr=0.01)

    action_space = np.arange(env.action_space.n)
    ep = 0
    while ep < num_episodes:
```

```
s_0 = env.reset()
states = []
rewards = []
actions = []
done = False
while done == False:
    #visualize every 100 episodes
    if ep%100 == 0:
        env.render()

    # Get actions and convert to numpy array
    action_probs = policy_estimator.predict(
        s_0).detach().numpy()
    action = np.random.choice(action_space,
        p=action_probs)
    s_1, r, done, _ = env.step(action)

    states.append(s_0)
    rewards.append(r)
    actions.append(action)
    s_0 = s_1
    # visualize

    # If done, batch data
    if done:
        batch_rewards.extend(discount_rewards(
            rewards, gamma))
        batch_states.extend(states)
        batch_actions.extend(actions)
        batch_counter += 1
        total_rewards.append(sum(rewards))

        # If batch is complete, update network
        if batch_counter == batch_size:
            optimizer.zero_grad()
            state_tensor = torch.FloatTensor(batch_states)
            reward_tensor = torch.FloatTensor(
                batch_rewards)
            # Actions are used as indices, must be
            # LongTensor
            action_tensor = torch.LongTensor(
               batch_actions)

            # Calculate loss
            logprob = torch.log(
                policy_estimator.predict(state_tensor))
            selected_logprobs = reward_tensor * logprob.gather(1,
                           action_tensor.unsqueeze(-1)).squeeze()
            # perform gather on 0 for state, on 1 for action
            loss = -selected_logprobs.mean()

            # Calculate gradients
            loss.backward()
            # Apply gradients
            optimizer.step()

            batch_rewards = []
```

```
                    batch_actions = []
                    batch_states = []
                    batch_counter = 1

                avg_rewards = np.mean(total_rewards[-100:])
                 #Print running average
                print("\rAverage rewards of last 100 episodes at " +
                      "{:.0f} : {:.5f}".format(
                      ep + 1, avg_rewards), end="")
                avg_rewards_saved.append(avg_rewards)
                if ep%100 ==0:
                    env.close()
                ep += 1

    return avg_rewards_saved


# In[126]:


env = gym.make('CartPole-v0')
policy_est = policy_estimator(env)
rewards = reinforce(env, policy_est)


# In[127]:


plt.plot(rewards)
plt.title('REINFORCE training for CartPole-v0')
plt.xlabel('episodes')
plt.ylabel('Avg rewards of the last 100 episodes')
plt.show()
```

Summary of settings:

The neural network for $\pi_{\theta}(a|s)$: 4 inputs (i.e. 4-dimensional state), the first hidden layer with 16 nodes and ReLU activation for each node. The second hidden layer with 2 nodes (i.e. 2 actions) is linear, and followed by the softmax layer as the output layer.

Hyperparameters: γ=0.99, α=0.01, batch size=10, total number of training episodes = 3000.

To visualize the training results, we plot the average return over the recent 100 episodes during the training process, and display the animation of cart pole once every 100 training episodes. We ran the program 6 times, and the plots are shown in Fig.5.

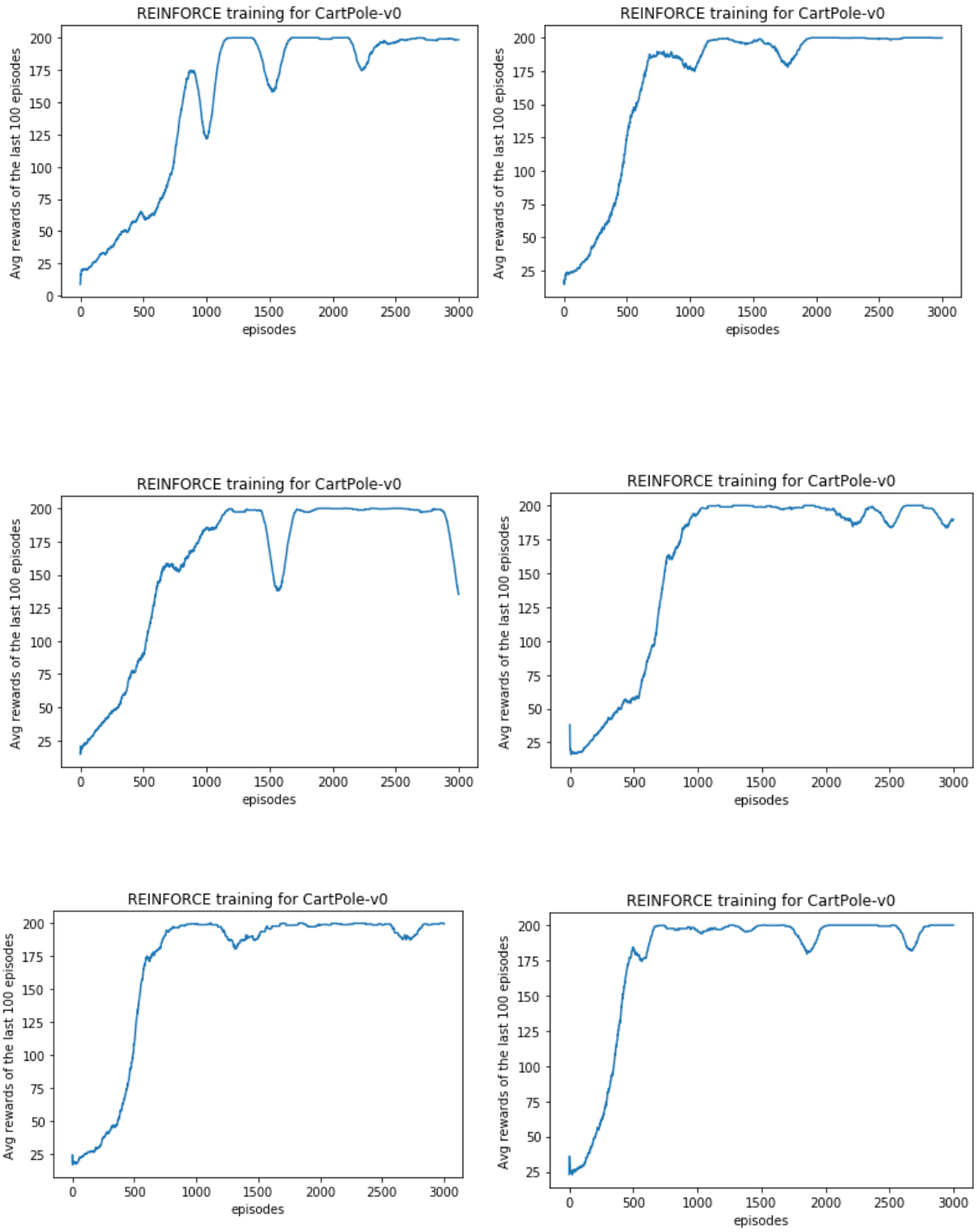


Fig. 5 average returns versus episodes for 6 independent runs (Algorithm 2): ch7_reinforce_cartpole.ipynb

Now let's look at the performance of algorithm 1. To implement algorithm 1, we replace every time step return Gt with the episode return R by redefining the discount_rewards function as follows. It has big variance and does not converge.

```
# calculate the return for the episode R()
 def discount_rewards(rewards, gamma=0.99):
     discounts = [gamma ** i for i in range(len(rewards) + 1)]
     R = sum([a * b for a,b in zip(discounts, rewards)])
     return [R]* len(rewards)
```

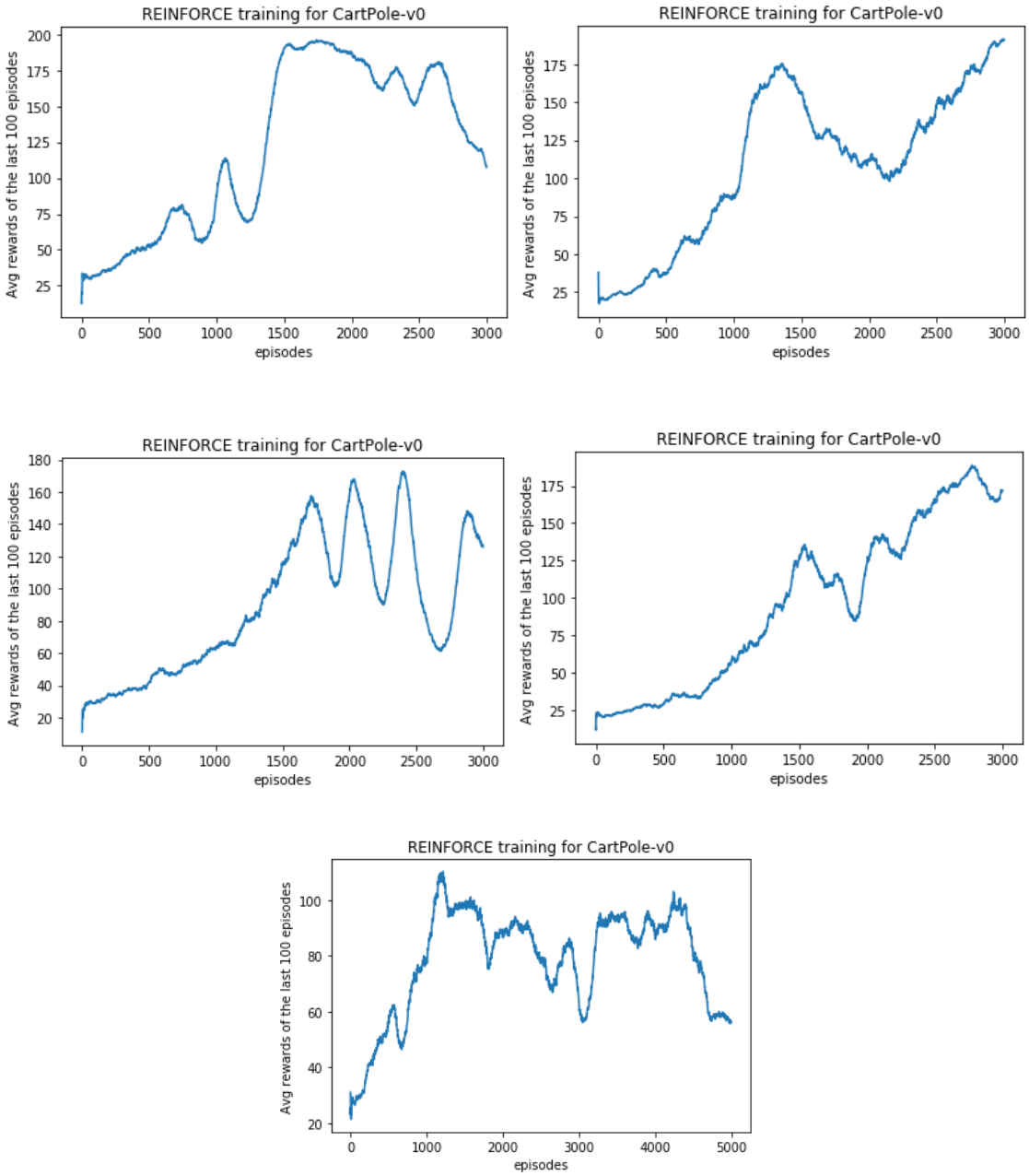


Fig. 6 Average returns of 100 episodes (Algorithm 1): ch7_reinforce_cartpole_total_reward.ipynb

7.6 REINFORCE with Baseline

https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients/

7.6.1 why baseline?

In the preliminary REINFORCE described in the previous section, the estimator of the policy gradient is unbiased but has a high variance. One efficient way to reduce the variance is to subtract a baseline function $b(s_t)$ from G_t. It can be shown that subtracting the baseline will not change the expectation of gradient estimate. With the baseline, the policy gradient theorem can be generalized as

$$
\begin{aligned}
&\nabla_\theta J(\theta) = \mathbb{E}_\pi\big[\big(Q^\pi(S_t,A_t) - b(S_t)\big)\nabla_\theta log\pi_\theta(A_t|S_t)\big] \\
&= \sum_s \mu(s) \sum_a \pi_\theta(a|s)\big(Q^\pi(s,a) - b(s)\big)\nabla_\theta log\pi_\theta(a|s) \\
&= \sum_s \mu(s) \sum_a \pi_\theta(a|s) Q^\pi(s,a)\nabla_\theta log\pi_\theta(a|s) - \sum_s \mu(s) \sum_a \pi_\theta(a|s) b(s)\nabla_\theta log\pi_\theta(a|s) \\
&= \mathbb{E}_\pi[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)] - \sum_s \mu(s)b(s) \sum_a \pi_\theta(a|s)\nabla_\theta log\pi_\theta(a|s) \\
&= \mathbb{E}_\pi[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)] - \sum_s \mu(s)b(s) \sum_a \nabla_\theta \pi_\theta(a|s) \\
&= \mathbb{E}_\pi[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)] - \sum_s \mu(s)b(s)\nabla_\theta \sum_a \pi_\theta(a|s) \\
&= \mathbb{E}_\pi[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)] - \sum_s \mu(s)b(s)\nabla_\theta\,(1) \\
&= \mathbb{E}_\pi[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)]
\end{aligned}
$$

The variance of the gradient estimate $\big(Q^\pi(S_t,A_t) - b(S_t)\big)\nabla_\theta log\pi_\theta(A_t|S_t)$ is usually much smaller than the variance of original estimate, $Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)$, given an appropriate baseline $b(S_t)$.

$$
\begin{aligned}
&Var\big[\big(Q^\pi(S_t,A_t) - b(S_t)\big)\nabla_\theta log\pi_\theta(A_t|S_t)\big] \\
&= \mathbb{E}\left[\big(Q^\pi(S_t,A_t) - b(S_t)\big)^2\big(\nabla_\theta log\pi_\theta(A_t|S_t)\big)^2\right] - \big(\mathbb{E}\big[\big(Q^\pi(S_t,A_t) - b(S_t)\big)\nabla_\theta log\pi_\theta(A_t|S_t)\big]\big)^2 \\
&= \mathbb{E}\left[\big(Q^\pi(S_t,A_t) - b(S_t)\big)^2\big(\nabla_\theta log\pi_\theta(A_t|S_t)\big)^2\right] - (\mathbb{E}[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)])^2 \\
&\approx \mathbb{E}\left[\big(Q^\pi(S_t,A_t) - b(S_t)\big)^2\right]\mathbb{E}\left[\big(\nabla_\theta log\pi_\theta(A_t|S_t)\big)^2\right] - (\mathbb{E}[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)])^2
\end{aligned}
$$

The approximation is applied because we assume the Q-function and baseline are independent of the policy. If $Q^\pi(S_t,A_t)$ varies dramatically with state, it is very effective to reduce the variance by including the baseline (for example, the average of $Q^\pi(S_t,A_t)$ over actions), i.e.

$$
\mathbb{E}\left[\big(Q^\pi(S_t,A_t) - b(S_t)\big)^2\right] < \mathbb{E}\left[\big(Q^\pi(S_t,A_t)\big)^2\right]
$$

Intuitively, for a state where all actions have high values, we need a high baseline, while for a state where all actions have low values, we need a low baseline.

7.6.2 Choice of baseline

As long as the baseline is independent of actions, it has no effect on the gradient estimate. To reduce the variance of gradient estimate, a natural choice for baseline is $\hat{V}(s_t, w)$ which is the estimate of the value function at the current state, w is the weights parameterizing $\hat{V}$. The state value function is a good choice of a baseline because it adjusts accordingly based on the state so that the variation of $\left(Q^{\pi}(S_t, A_t) - b(S_t)\right)^2$ is smaller than that of $\left(Q^{\pi}(S_t, A_t)\right)^2$ in a sense of statistical average.

7.6.3 REINFORCE algorithm with baseline

Algorithm 2 can be extended by including a baseline. The algorithm described in Sutton book is straightforward.

Algorithm 3 (a): (Sutton book)
Input: differentiable policy $\pi_\theta(a|s)$
Input: differentiable state-value function $\hat{V}(s, w)$
Constants: learning rate α_θ, α_w
Initialize: policy parameter θ, state-value function weights w
Loop forever (for each episode):
Generate an episode following $S_0 A_0 R_1 \ldots S_{T-1} A_{T-1} R_T$ the current policy $\pi_\theta(a|s)$
Loop for each timestep t:

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$
$$\delta \leftarrow G - \hat{V}(S_t, w)$$
$$w \leftarrow w + \alpha_w \delta \nabla_w \hat{V}(S_t, w)$$
$$\theta \leftarrow \theta + \alpha_\theta \gamma^t \delta \nabla_\theta log\pi_\theta(A_t|S_t)$$

(My note: $\theta \leftarrow \theta + \alpha_\theta \delta \nabla_\theta log\pi_\theta(A_t|S_t)$)

Algorithm3 (b): Vanilla policy gradient algorithm (Stanford CS234, lecture9) (batch update)
Initialize policy parameter θ, baseline b
for loop iteration (forever) =1,2,…. (policy):
Collect a set of trajectories by following the current policy
Loop for each trajectory τ^i:
Loop for each timestep t in τ^i:
Return $G_t^i = \sum_{t'=t}^{T-1} r_{t'}^i$
Advantage estimate $\hat{A}_t^i = G_t^i - b(s_t)$
Re-fit the baseline by minimizing $\sum_i \sum_t \left\| G_t^i - b(s_t) \right\|^2$
Update the policy, using a policy gradient estimate $\hat{g}$,
which is the sum of terms $\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t$
(plug $\hat{g}$ into SGD or ADAM)

Alternatively, we have "vanilla" policy gradient algorithm from Stanford CS234 lecture 9. Please note the different notations.

Equivalently we can use the following pseudo code for Vanilla algorithm—**Algorithm 3 (c) (batch-update)**

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

  Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

  or via another gradient ascent algorithm like Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

(from https://spinningup.readthedocs.io/zh_CN/latest/algorithms/vpg.html)

7.7 Actor-Critic Policy Gradient

REINFORCE is unbiased and will converge asymptotically to a local minimum, but like all other Monte-Carlo methods it tends to learn slowly due to high variance estimate and to be inconvenient to implement online or for continuing problems due to the requirement of an entire trajectory. To eliminate these inconveniences, we can update the gradient using estimated state-action value function or advantage value function, called **critic,** so that both gradient and value function can be updated at each timestep via bootstrapping (updating the value estimate for a state from the estimated values of subsequent states). In practice both the critic and the policy are modeled by neural networks. These methods are called **actor-critic** methods, which maintain two sets of parameters: w, **critic** that updates action-value function; and θ, **actor** that updates policy in direction suggested by critic.

7.7.1 Q Actor-Critic method

(From http://www.cs.cmu.edu/~rsalakhu/10703/Lectures/Lecture_PG2.pdf)

According to the policy gradient theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(S_t, A_t)\nabla_\theta log\pi_\theta(A_t|S_t)]$$

the policy can be updated by

$$\theta \leftarrow \theta + \alpha Q(S_t, A_t)\nabla_\theta log\pi_\theta(S_t, A_t)$$

In Q actor-critic methods, we use a **critic** (neural network) to estimate the action-value function,

$$Q_w(s, a) \approx Q^{\pi_\theta}(s, a)$$

Thus,

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s, a)Q_w(s, a)]$$

$$\theta \leftarrow \theta + \Delta\theta = \theta + \alpha\nabla_\theta log\pi_\theta(s, a)Q_w(s, a)$$

The critic is a policy evaluation problem addressed earlier: how good is policy π_θ for current parameters θ? We can use MC policy evaluation, TD learning, or TD(λ), etc. To make it concrete, we can implement the idea as follows. Assume we use linear value function to approximate action-value function

$$Q_w(s, a) = \phi(s, a)^T w$$

and linear TD(0) for critic to update w and policy gradient for actor to update θ. Without waiting till the end of episodes, we update the parameters every single step. Note that there is no e-greedy policy involved here. The policy is directly determined by the current parameters.

Algorithm 4 Q actor-critic

Initialize: s, θ, w and learning rates α,β

Sample a~$\pi_\theta(a|s)$

for each step t=1,2,…,T **do**

sample reward $r = R_s^a$; next state $s' \sim P_s^a$

sample action $a' \sim \pi_\theta(s', a')$; sample tuple {s,a,r,s',a'}

$\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$; TD(0) error for Q at time t

$\theta = \theta + \alpha\nabla_\theta log\pi_\theta(s, a)Q_w(s, a)$; update actor (policy)

$w \leftarrow w + \beta\delta\nabla_w Q_w(s, a) = w + \beta\delta\phi(s, a)$; update critic

$a \leftarrow a', s \leftarrow s'$

end for

Since approximating the policy gradient introduces a bias and a biased policy gradient may not find the right solution, we need to choose value function approximation carefully to avoid significant bias, i.e., we

can still follow the exact policy gradient. This issue can be resolved by **compatible function approximation theorem.** Specifically, if the following two conditions are satisfied:

1) Value function approximator is **compatible** to the policy
$$\nabla_w Q_w(s,a) = \nabla_\theta log\pi_\theta(s,a)$$
2) Value function parameters ***w*** minimize the mean-squared error
$$\varepsilon = \mathbb{E}_{\pi_\theta}\left[\left(Q^{\pi_\theta}(s,a) - Q_w(s,a)\right)^2\right]$$

Then the policy gradient is exact,
$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) Q_w(s,a)]$$

[proof]

If w is chosen to minimise mean-squared error, gradient of ε w.r.t. w must be zero,

$$\nabla_w \varepsilon = 0$$
$$\mathbb{E}_{\pi_\theta}\left[(Q^\theta(s,a) - Q_w(s,a))\nabla_w Q_w(s,a)\right] = 0$$
$$\mathbb{E}_{\pi_\theta}\left[(Q^\theta(s,a) - Q_w(s,a))\nabla_\theta \log \pi_\theta(s,a)\right] = 0$$
$$\mathbb{E}_{\pi_\theta}\left[Q^\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)\right] = \mathbb{E}_{\pi_\theta}\left[Q_w(s,a)\nabla_\theta \log \pi_\theta(s,a)\right]$$

So $Q_w(s,a)$ can be substituted directly into the policy gradient,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)\right]$$

7.7.2 Advantage Actor-Critic Method: Baseline for Reducing Variance

(From http://www.cs.cmu.edu/~rsalakhu/10703/Lectures/Lecture_PG2.pdf)

A) Advantage function

To reduce variance, as we discussed earlier, we subtract a baseline function b(s) from the policy gradient, without changing expectation

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) Q^{\pi_\theta}(s,a)] = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a)(Q^{\pi_\theta}(s,a) - b(s))]$$

Theoretically the baseline function b(s) could be arbitrary for keeping the policy gradient unchanged. But, for a reduced variance, a good baseline function is the state value function. i.e.,

$$b(s) = V^{\pi_\theta}(s)$$

The difference between the state-action function and the state value function is called the **advantage function**,

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$

B) Advantage actor-critic algorithm (A2C)

The advantage function tells us how much better than usual when we take action ***a***. We can rewrite the policy gradient using the advantage function as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a)A^{\pi_\theta}(s,a)] = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a)(Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s))]$$

Critic

Now, the question is how to estimate the advantage function. One way is to estimate both state-action function and state function using two function approximators and two parameter vectors,

$$V_v(s) \approx V^{\pi_\theta}(s)$$
$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$
$$A_{w,v}(s,a) = Q_w(s,a) - V_v(s)$$

and then we update both value function by e.g. TD learning. This method requires two sets of parameters and two separate approximators.

A more convenient way is to directly estimate the advantage function based on state-value function only via TD error. For the **true** value function $V^{\pi_\theta}(s)$, the TD error δ^{π_θ}, represented by

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

is an unbiased estimate of the advantage function because

$$\mathbb{E}_{\pi_\theta}[\delta^{\pi_\theta}|s,a] = \mathbb{E}_{\pi_\theta}[r + \gamma V^{\pi_\theta}(s')|s,a] - V^{\pi_\theta}(s) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) = A^{\pi_\theta}(s,a)$$

Thus, we can use the TD error to compute the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a)\delta^{\pi_\theta}]$$

In practice, we can use an approximate TD error

$$\delta_V = r + \gamma V_v(s') - V_v(s)$$

This method only requires one set of critic parameters *v*. The parameter of the critic can be updated from various targets, such as Gt (MC target) or TD target $r + \gamma V(s')$

$$\text{Monte Carlo:} \quad v \leftarrow v + \alpha\big(G_t - V_v(s_t)\big)\nabla_v V_v(s_t)$$
$$\text{TD(0):} \quad v \leftarrow v + \alpha\big(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)\big)\nabla_v V_v(s_t)$$

However, Monte Carlo target requires a complete episode while TD(0) or TD-n step target only needs a portion of episode Note that TD target estimate is biased because a bootstrapping is applied. We will implement these two ways in the cartpole example later.

Actor

The policy gradient can be estimated in different way depending how the advantage are obtained. For example, Monte-Carlo policy gradient uses error from complete return

$$\Delta\theta = \alpha\big(G_t - V_v(s_t)\big)\nabla_\theta log\pi_\theta(s_t,a_t)$$

Actor-critic policy gradient uses the one-step TD error or n-step TD error

$$\Delta\theta = \alpha\big(r + \gamma V_v(s_{t+1}) - V_v(s_t)\big)\nabla_\theta log\pi_\theta(s_t,a_t)$$

Algorithm 5 advantage actor-critic: one step TD

Input: policy function $\pi_\theta(s,a)$, estimated state-value function $V(s,w)$

Initialize: θ, w and learning rates α,β

Repeat forever:

Initialize s (first state of episode)

$I \leftarrow 1$ (initial discount)

While s is not terminal:

sample a~$\pi_\theta(a|s)$

take action a, and observe reward $r = R_s^a$, next state s'~P_s^a

$\delta = r + \gamma V(s',w) - V(s,w)$; if s' is terminal, then $V(s',w) = 0$

$w \leftarrow w + \beta\delta\nabla_w V(s,w)$; update critic

$\theta = \theta + \alpha \cdot \mathrm{I} \cdot \delta \cdot \nabla_\theta log\pi_\theta(s,a)$; update actor (policy)

$I \leftarrow \gamma I, s \leftarrow s'$; why γ accumulated?**

**explanation:

The policy gradient can be evaluated in two options:

Option1: $\nabla_\theta V(\theta) \approx \frac{1}{m}\sum_{i=1}^{m}\left[R\left(\tau^{(i)}\right)\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right]$ (definition)

Option2: $\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(S_t,A_t)\nabla_\theta log\pi_\theta(A_t|S_t)]$ (policy gradient theorem)

7.7.3 Implementation of A2C Algorithm

A) A2C using one-step TD error

The A2C algorithm can be implemented online or in batch. They are summarized as follows.

Online A2C algorithm (repeat the following steps):

1. Take action a~$\pi_\theta(a|s)$, get (s,a,s',r)
2. Evaluate $A = r + \gamma V(s',w) - V(s,w)$
3. $\theta \leftarrow \theta + \alpha \cdot \mathrm{A} \cdot \nabla_\theta log\pi_\theta(s,a)$
4. update parameter w using target $r + \gamma V(s',w)$

Batch A2C algorithm (repeat the following steps):

1. sample a set $\{s_i,a_i,r_i\}$ from $\pi_\theta(a|s)$
2. evaluate the advantage $A_i = r_i + \gamma V(s_{i+1},w) - V(s_i,w)$
3. $\theta \leftarrow \theta + \alpha \cdot \frac{1}{N}\sum_i A_i \nabla_\theta log\pi_\theta(s_i,a_i)$
4. Fit $V(s,w)$ to the sampled set with loss function MES loss (sum) $\sum_i\|A_i\|^2$ or MSE loss (mean) $\frac{1}{N}\sum_i\|A_i\|^2$

The architecture of online implementation is shown in Fig.7. First, we initialize two neural networks for policy and value function, respectively. Then we generate samples by following the current policy. Based on the samples, the policy network will predict the probability distribution of action and the value network will estimate the advantage. Two loss functions, cross entropy function and MSE loss function, are used to train the policy network and the value network, respectively. For a batch implementation, the loss functions for should be replaced by the sum of all individual losses in the batch.

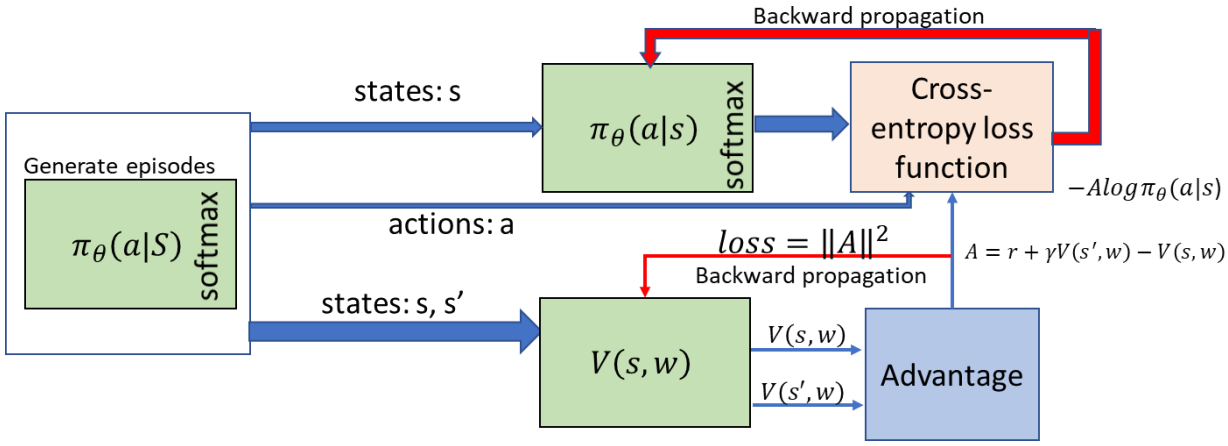


Fig.7 A2C implementation diagram (one-step TD error)

We implement A2C with one-step TD return for cartpole task. The Python code is attached below. (inspired by Ref: https://github.com/floodsung/a2c_cartpole_pytorch/blob/master/a2c_cartpole.py)

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:

# ch7_A2C_2_cartpole.ipynb:
# on-step TD error
# A=r+rV(st+1,w)-V(st,w)
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import matplotlib.pyplot as plt
import numpy as np
import math
import random
import os
import gym


# In[49]:
```

```
# Hyper Parameters
STATE_DIM = 4
ACTION_DIM = 2
STEP = 4000
SAMPLE_NUMS = 30

# In[3]:

class ActorNetwork(nn.Module):

    def __init__(self,input_size,hidden_size,action_size):
        super(ActorNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size,hidden_size)
        self.fc2 = nn.Linear(hidden_size,hidden_size)
        self.fc3 = nn.Linear(hidden_size,action_size)

    def forward(self,x):
        out = F.relu(self.fc1(x))
        out = F.relu(self.fc2(out))
        out = F.log_softmax(self.fc3(out), dim=-1)
        return out

# In[4]:


class ValueNetwork(nn.Module):

    def __init__(self,input_size,hidden_size,output_size):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size,hidden_size)
        self.fc2 = nn.Linear(hidden_size,hidden_size)
        self.fc3 = nn.Linear(hidden_size,output_size)

    def forward(self,x):
        out = F.relu(self.fc1(x))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out


# In[5]:


def roll_out(actor_network,task,sample_nums,value_network,init_state):

    states = []   # store all states for this roll_out [s0, s1, ...sT], or
                  # [s0, s1,...Ssample_numbers]
    actions = []  # all actions in one-hot code
    rewards = []  # all rewards [r0, r1, r2...]
    #values = []   # values for states[v(s0,w), v(s1,w),...]
    is_done = False
    final_r = 0   # the reward for the last state: 0 for temerinal, V(st)
                  # if the last state is not a terminal
    state = init_state
```

```
    for j in range(sample_nums):
        states.append(state)
        log_softmax_action =
                actor_network(Variable(torch.Tensor([state])))
        softmax_action = torch.exp(log_softmax_action)
        action =
       np.random.choice(ACTION_DIM,p=softmax_action.cpu().data.numpy()[0])
        one_hot_action = [int(k == action) for k in range(ACTION_DIM)]
        next_state,reward,done,_ = task.step(action)
        #fix_reward = -10 if done else 1
        actions.append(one_hot_action)
        rewards.append(reward)
        final_state = next_state
        state = next_state
        if done:
            is_done = True
            state = task.reset()
            break
    if not is_done:
        final_r =
value_network(Variable(torch.Tensor([final_state]))).cpu().data.numpy()
        return states,actions,rewards,final_r,state

def discount_reward(r, gamma,final_r):
    discounted_r = np.zeros_like(r)
    running_add = final_r
    for t in reversed(range(0, len(r))):
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    return discounted_r


## Now, we iterate the single training step for many times

# In[60]:


task = gym.make("CartPole-v0")
init_state = task.reset()

# init value network
value_network = ValueNetwork(input_size = STATE_DIM,hidden_size =
40,output_size = 1)
value_network_optim =
torch.optim.Adam(value_network.parameters(),lr=0.001)

# init actor network
actor_network = ActorNetwork(STATE_DIM,40,ACTION_DIM)
actor_network_optim = torch.optim.Adam(actor_network.parameters(),lr =
0.001)
steps =[]
task_episodes =[]
test_results =[]
```

```
# In[61]:

for step in range(STEP):
    # geneate SAMPLE_NUMS transitions, and store the results in lists.
    # states: [s0, s1, ...]
    # actions: [a0, a1, ...]   note a0 is one-hot-code
    # rewards: [r0, r1, ...]
    # the lists above will end earlier if episode is done
    # current_state: the starting state for the next roll_out
    # = the state following the last state in the list states if the
    #   episode is not done
    # or = a random reset state if the previous roll_out episode is done.
    #final_r = scalr 0 if the episode is done, final_r=V(current_state, w)

    states,actions,rewards,final_r,current_state =
roll_out(actor_network,task,SAMPLE_NUMS,value_network,init_state)

    init_state = current_state
    # prepare a starting state for the next step roll_out

    actions_var = Variable(torch.Tensor(actions).view(-1,ACTION_DIM))
    # .size(): [*, ACTION_DIM]

    # train actor_network
    # input for actor_network and value_network
    states_var = Variable(torch.Tensor(states).view(-1,STATE_DIM))
    #torch.size(): [*, STATE_DIM]
    actor_network_optim.zero_grad()
    log_softmax_actions = actor_network(states_var)
    # log_softmax_actions.size(): tensor [*,2]
    vs_temp = value_network(states_var).detach()
    vs=vs_temp.view(len(vs_temp))
    # .detach():  vs not need gradient track, vs.size(): tensor [*,1]
    #.view(len(vs)): [*,1] --> [*] (I added)

    # calculate qs
    gamma=0.99
    vs_delay = np.array(vs)
    vs_delay = np.append(vs_delay[1:], final_r)
    qs = torch.Tensor(rewards+gamma*vs_delay)

    advantages = qs - vs        # [*]
    actor_network_loss =
    - torch.mean(torch.sum(log_softmax_actions*actions_var,1)* advantages)
    actor_network_loss.backward()
    torch.nn.utils.clip_grad_norm_(actor_network.parameters(),0.5)
    actor_network_optim.step()

    # train value_network
    value_network_optim.zero_grad()
    target_values = qs
    values = value_network(states_var)
    values = values.view(len(values))
```

```
    criterion = nn.MSELoss()
    value_network_loss = criterion(values,target_values)
    value_network_loss.backward()
    torch.nn.utils.clip_grad_norm_(value_network.parameters(),0.5)
    value_network_optim.step()

    if (step + 1) % 50== 0:
        result = 0
        test_task = gym.make("CartPole-v0")

        for test_epi in range(10):

            state = test_task.reset()
            for test_step in range(200):

                softmax_action = torch.exp
                    (actor_network(Variable(torch.Tensor([state]))))
                        #print(softmax_action.data)
                action = np.argmax(softmax_action.data.numpy()[0])
                next_state,reward,done,_ = test_task.step(action)
                result += reward
                state = next_state

                if done:
                    break

        print("step:",step+1,"test result:",result/10.0)

        steps.append(step+1)
        test_results.append(result/10)


# In[62]:


plt.plot(steps,test_results)
plt.title('A2C (TD(0)) training for CartPole-v0')
plt.xlabel('steps')
plt.ylabel('Avg rewards of the last 10 episodes')
plt.show()
```

B) A2C using n-step TD return

In Fig.7, we use TD(0) return to estimate the Q(s,a). Since TD(0) return target estimate is biased, the training process is slow to converge, but can update the parameters at every transition (online method) or at every batch (batch method). In practice, to improve the accuracy of advantage estimate especially at the beginning of the training and thus speed up the train convergence, we can use the combination of Monte Carlo and TD(0) to estimate the Q(s,a). For example, in a batch implementation, we generate and store a trajectory of N transitions $(s_i, a_i, r_i), i = 0,1,2 \ldots N\text{-}1$ or a shorter trajectory if the episode is "done", for one batch. The parameters in both networks are updated once based on one batch data. If the last transition in the batch reaches the terminal state, the Q-value for (s_i, a_i) can estimated as (Monte Carlo)

$$Q(s_i, a_i) = \sum_{j=i}^{N_T} \gamma^{j-i} r_j$$

where $N_T \leq N, \quad N_T$ is the total number of transitions in the batch if the episode is "done".

If the last transition does not reach the terminal state, a bootstrapping is applied to estimate the Q-value

$$Q(s_i, a_i) = \sum_{j=i}^{N} \gamma^{j-i} r_j + \gamma V(s_N, w)$$

where s_N is the state following the last state of the current batch and will used as the initial state for the next batch. Thus, each batch does not necessarily restart an episode. A new episode starts only when "done" from the environment is true. However, a true "done" may end the batch sampling earlier.

Then the advantages in the batch can be obtained

$$A_i = Q(s_i, a_i) - V(s_i, w)$$

The loss of the policy network is

$$L_\pi(\theta) = -\frac{1}{N} \sum_i A_i \nabla_\theta log \pi_\theta(s_i, a_i)$$

The loss of the value network is

$$L_V(w) = \frac{1}{N} \sum_i \|A_i\|^2$$

The implementation for cartpole is attached below.

```
#!/usr/bin/env python
# coding: utf-8

# In[106]:


# ch7_A2C_cartpole.ipynb:
# A=Gt-V(st,w)
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import matplotlib.pyplot as plt
import numpy as np
import math
import random
import os
import gym


# In[184]:
```

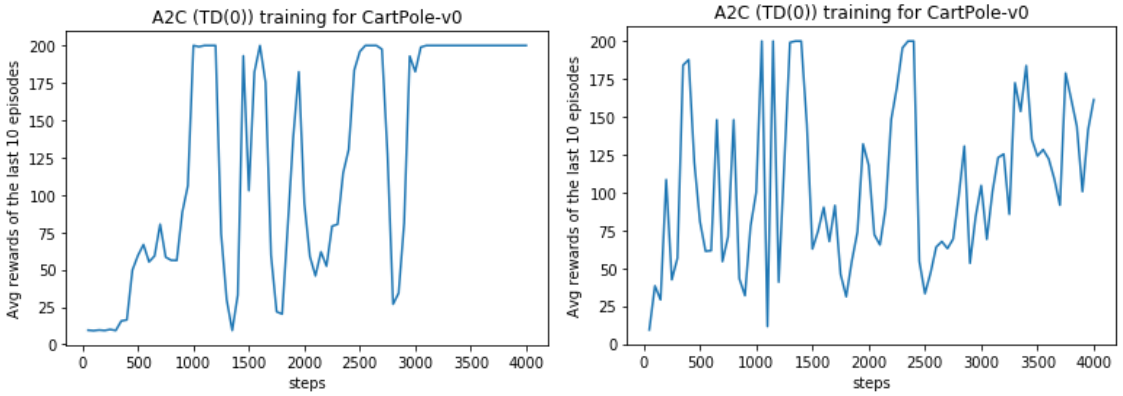
```
# Hyper Parameters
STATE_DIM = 4
ACTION_DIM = 2
STEP = 4000
SAMPLE_NUMS = 30


# In[108]:


class ActorNetwork(nn.Module):

    def __init__(self,input_size,hidden_size,action_size):
        super(ActorNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size,hidden_size)
        self.fc2 = nn.Linear(hidden_size,hidden_size)
        self.fc3 = nn.Linear(hidden_size,action_size)

    def forward(self,x):
        out = F.relu(self.fc1(x))
        out = F.relu(self.fc2(out))
        out = F.log_softmax(self.fc3(out), dim=-1)
        return out



# In[109]:



class ValueNetwork(nn.Module):

    def __init__(self,input_size,hidden_size,output_size):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size,hidden_size)
        self.fc2 = nn.Linear(hidden_size,hidden_size)
        self.fc3 = nn.Linear(hidden_size,output_size)

    def forward(self,x):
        out = F.relu(self.fc1(x))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out


# In[110]:


def roll_out(actor_network,task,sample_nums,value_network,init_state):

    states = []   # store all states for this roll_out [s0, s1, ...sT], or
                  # [s0, s1,...Ssample_numbers]
    actions = []  # all actions in one-hot code
    rewards = []  # all rewards [r0, r1, r2...]
```

```
    is_done = False
    final_r = 0   # the reward for the last state is 0 for temerinal, or
                  # V(st)if the last state is not a terminal
    state = init_state

    for j in range(sample_nums):
        states.append(state)
        log_softmax_action =actor_network(Variable(torch.Tensor([state])))
        softmax_action = torch.exp(log_softmax_action)
        action = np.random.choice(
               ACTION_DIM,p=softmax_action.cpu().data.numpy()[0])
        one_hot_action = [int(k == action) for k in range(ACTION_DIM)]
        next_state,reward,done,_ = task.step(action)
        #fix_reward = -10 if done else 1
        actions.append(one_hot_action)
        rewards.append(reward)
        final_state = next_state
        state = next_state
        if done:
            is_done = True
            state = task.reset()
            break
    if not is_done:
        final_r = value_network(
            Variable(torch.Tensor([final_state]))).cpu().data.numpy()

    return states,actions,rewards,final_r,state



def discount_reward(r, gamma,final_r):
    discounted_r = np.zeros_like(r)
    running_add = final_r
    for t in reversed(range(0, len(r))):
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    return discounted_r

# # Now, we iterate the single training step for many times

# In[189]:

task = gym.make("CartPole-v0")
init_state = task.reset()

# init value network
value_network = ValueNetwork(input_size = STATE_DIM,hidden_size =
40,output_size = 1)
value_network_optim =
torch.optim.Adam(value_network.parameters(),lr=0.001)

# init actor network
actor_network = ActorNetwork(STATE_DIM,40,ACTION_DIM)
```

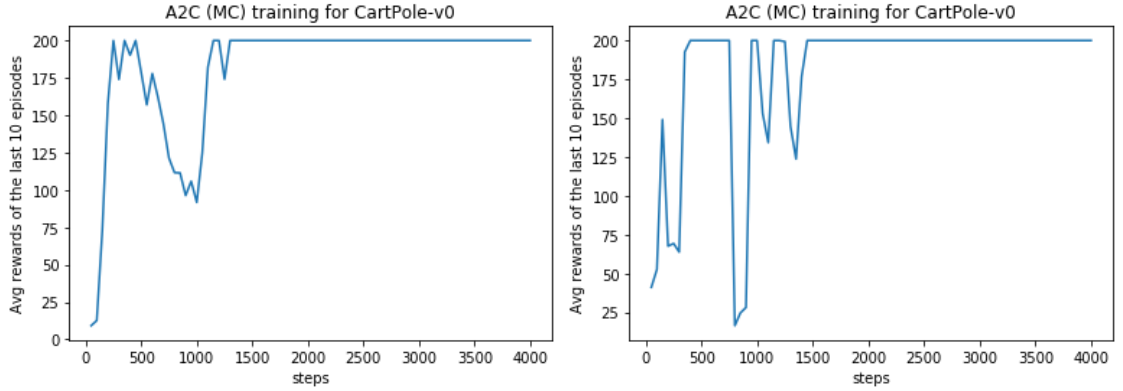
```
actor_network_optim = torch.optim.Adam(actor_network.parameters(),lr =
0.001)
steps =[]
task_episodes =[]
test_results =[]


# In[190]:

for step in range(STEP):
    # geneate SAMPLE_NUMS transitions, and store the results in lists.
    # states: [s0, s1, ...]
    # actions: [a0, a1, ...]   note a0 is one-hot-code
    # rewards: [r0, r1, ...]
    # the lists above will end earlier if episode is done
    # current_state: the starting state for the next roll_out
    # = the state following the last state in the list states if the
    # episode is not done
    # or = a random reset state if the previous roll_out episode is done.
    #final_r = scalr 0 if the episode is done, final_r=V(current_state, w)
    states,actions,rewards,final_r,current_state = roll_out(
             actor_network,task,SAMPLE_NUMS,value_network,init_state)

    init_state = current_state
    # prepare a starting state for the next step roll_out

    actions_var = Variable(torch.Tensor(actions).view(-1,ACTION_DIM))
    # .size(): [*, ACTION_DIM]

    # train actor_network
    # input for actor_network and value_network
    states_var = Variable(torch.Tensor(states).view(-1,STATE_DIM))
    #torch.size(): [*, STATE_DIM]
    actor_network_optim.zero_grad()
    log_softmax_actions = actor_network(states_var)
    # log_softmax_actions.size(): tensor [*,2]
    vs_temp = value_network(states_var).detach()
    vs=vs_temp.view(len(vs_temp))
    # .detach():  vs not need gradient track, vs.size(): tensor [*,1]
    #.view(len(vs)): [*,1] --> [*] (I added)

    # calculate qs
    qs = Variable(torch.Tensor(discount_reward(rewards,0.99,final_r)))
    #qs.size: tensor [*]

    advantages = qs - vs        # [*]
    actor_network_loss = - torch.mean(
         torch.sum(log_softmax_actions*actions_var,1)* advantages)
    actor_network_loss.backward()
    torch.nn.utils.clip_grad_norm_(actor_network.parameters(),0.5)
    actor_network_optim.step()

    # train value_network
    value_network_optim.zero_grad()
```

```
    target_values = qs
    values = value_network(states_var)
    values = values.view(len(values))
    criterion = nn.MSELoss()
    value_network_loss = criterion(values,target_values)
    value_network_loss.backward()
    torch.nn.utils.clip_grad_norm_(value_network.parameters(),0.5)
    value_network_optim.step()

    if (step + 1) % 50== 0:
        result = 0
        test_task = gym.make("CartPole-v0")

        for test_epi in range(10):

            state = test_task.reset()
            for test_step in range(200):

                softmax_action = torch.exp(actor_network(
                      Variable(torch.Tensor([state]))))
                        #print(softmax_action.data)
                action = np.argmax(softmax_action.data.numpy()[0])
                next_state,reward,done,_ = test_task.step(action)
                result += reward
                state = next_state

                if done:
                    break

        print("step:",step+1,"test result:",result/10.0)

        steps.append(step+1)
        test_results.append(result/10)


# In[191]:

plt.plot(steps,test_results)
plt.title('A2C (MC) training for CartPole-v0')
plt.xlabel('steps')
plt.ylabel('Avg rewards of the last 10 episodes')
plt.show()
```

In summary, we implement the A2C algorithm for the cartpole using two different ways of estimating target return. The first way uses one-step TD return while the second way uses Monte Carlo plus one step bootstrapping (if the episode is not “done” in the current batch). It has been noticed that the first method requires a more careful hyperparameter selection and needs more data to converge. For example, learning rate 0.01 does not work for the first method, but both learning rates 0.01 and 0.001 work for the second method. Fig.8 shows the average rewards over 10 test episodes at every 50 training steps (or batches). Thus, the second method (MC return) is recommended.



(a) A2C with TD(0), learning rate 0.001 (b) A2C with TD(0), learning rate 0.01 (not converge)



(c) A2C with 30-step TD, learning rate 0.001 (d) A2C with 30-step TD, learning rate 0.01

Fig. 8 Average rewards of 10 test episodes at every 50 training steps (batches)

7.7.4 Asynchronous Advantage Actor-Critic (A3C) Algorithm

Asynchronous reinforcement learning framework was proposed by Mnih et al. in 2016. The idea of the framework is to use multiple threads to update the neural networks in parallel and asynchronously. Each thread interacts with an instance (or copy) of the environment separately. It is observed that multiple threads running in parallel are likely to be exploring different parts of the environment. Moreover, one can explicitly use different exploration policies in each thread to maximize this diversity. By running different exploration policies in different threads, the overall changes being made to the parameters by multiple threads applying online updates in parallel are likely to be less correlated in time than a single-thread agent applying online updates. Hence, we can rely on this parallelism and batch accumulated updates to perform the stabilizing role undertaken by experience reply in DQN training algorithm.

The asynchronous RL framework can be applied to one-step SARSA, one-step Q-learning and n-step Q-learning and actor-critic learning. The asynchronous A2C algorithm achieves the best performance among these asynchronous methods and can work for both discrete and continuous cases. The architecture of A3C is illustrated in Fig.9. Each thread (worker) is obtained by duplicating the global network but with local parameters. Each worker interacts with an instance of the environment independently, and updates the global network parameters after a period of time steps. Note that while the parameters of the policy and the value function are separate as shown in equations for generality, we always share some of the parameters

in practice. We typically use a convolutional neural network that has one softmax output for the policy π and one linear output for the value function V, with all non-output layers shared.

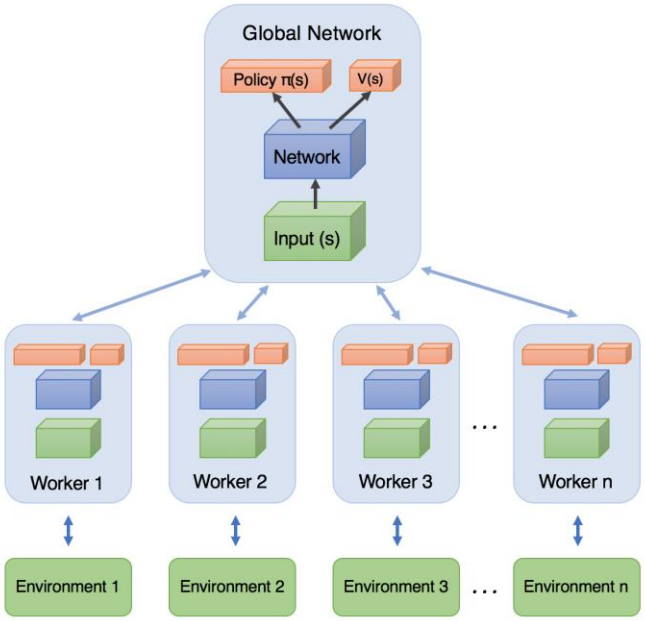


Fig.9 the architecture of A3C

The implementation of A3C is very similar to that of A2C with n-step TD return discussed in the previous section, except that we duplicate A2C to multiple threads. Pseudocode for A3C is shown in the following table. Each thread interacts with its own instance of the environment. It operates in the forward view by explicitly computing n-step returns, instead the more common backward view with eligibility traces. The reason is that using the forward view is easier when training neural networks with momentum-based methods and backpropagation through time. In order to compute a single update, the algorithm first selects actions (as a batch) using its exploration policy for up to t_{max} steps or until terminal state is reached. The agent receives up to t_{max} rewards from the environment since its last update. The algorithm then computes the gradients of the actor (policy) for each state-action pair and the gradient of the critic (value function) for each state encountered since the last update. Each n-step update uses the longest possible n-step return resulting in a one-step update for the last state, a two-step update for the second last state, and so on for a total of up to t_{max} updates. At each time step, the gradients are computed and accumulated. The policy and the value function are updated after every t_{max} time steps or when a terminal state is reached. At time step t, the update for the actor is

$$\nabla_{\theta'} log\pi(a_t|s_t;\theta')\big(R - V(s_t;\theta'_v)\big)$$

where $R - V(s_t;\theta'_v)$ is an estimate of the advantage function, R is the estimate of Q-value for the state-action pair $(s_{t,}a_t)$ given by

$$R = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k},\theta'_v)$$

where k can very from state to state and is upper-bounded by t_{max}, and $V(s_{t+k}, \theta'_v)$ is equal to zero if s_{t+k} is terminal.

Algorithm: Asynchronous advantage actor-critic – pseudocode for each actor-learner thread

// Assume global shared parameter vector θ and θ_v and global shared counter T = 0
// Assume thread-specific parameter vector θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state s_t
    repeat
        Perform a_t according to policy $\pi(a_t|s_t;\theta')$
        Receive reward r_t and new state s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal s_t **or** $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & for\ terminal\ s_t \\ V(s_t, \theta'_v) & for\ non-terminal\ s_t, boostrap\ from\ last\ state \end{cases}$
    for $i\ from\ \{t-1, \dots, t_{start}\}\ do$ // note the descent order
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')\big(R - V(s_i;\theta'_v)\big)$
        Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \frac{\partial\big(R - V(s_i;\theta'_v)\big)^2}{\partial\theta'_v}$
    end for
    Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{max}$

Mnih et al. demonstrated that, for most of Atari games, A3C runs much faster yet performs better than or comparably with DQN, Gorila (Nair et al., 2015), D-DQN, Dueling D-DQN, and Prioritized D-DQN. A3C also succeeds on continuous motor control problems: TORCS car racing games and MujoCo physics manipulation and locomotion, and Labyrinth, a navigating task in random 3D mazes using visual inputs.

7.8 Summary

RL Policy Gradient (PG) methods are model-free methods that try to maximize the RL objective without requiring a value function. Although a value function can be used as a baseline for variance reduction, or in order to evaluate current and successor state's value (Actor-Critic), it is not required for the purpose of action selection. The RL objective, i.e. the performance measure J(θ), is defined as the sum of rewards from the initial state to a terminal state for the episodic task, and the average return for the continuing task, where θ is the parameter that defines the policy $\pi_\theta(a|s)$, a probability distribution of actions at the state s.

To learn the optimal policy, the gradient of the objective with respect to θ, called policy gradient, $\nabla_\theta J(\theta)$, will be estimated based on the interactions between the agent and the environment. The policy gradient theorem provides a theoretical foundation for how to estimate the policy gradient. All the subsequent policy

gradient methods can be derived from the policy gradient theorem. The policy gradient has equivalent forms, each of which corresponds to a particular policy gradient method.

REINFORCE with baseline: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)(G - b(s))]$

Q Actor-Critic: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)Q^w(s,a)]$

Advantage Actor-Critic: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)A^w(s,a)]$

TD Actor-Critic: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)\delta]$

In general, each method consists of two components: actor and critic. The actor is to learn the policy through a stochastic gradient ascent algorithm, which can be implemented by training a neural network with a soft max output layer and cross-entropy loss function. The critic uses policy evaluation (e.g. MC or TD learning) to estimate the value function (highlighted by red), such as MC return, state value function, state-action value function or TD error.

To help readers understand the algorithms, we give the detailed implementations of REINFORCE, A2C for a cartpole task.