Reinforcement Learning

Seungcheol Oh

I. Introduction

Reinforcement learning (RL) is a sub-field of machine learning that has a unique feature that sets itself apart from other machine learning fields, such as supervised learning, and unsupervised learning. Unlike these methods, which rely on pre-sampled data, reinforcement learning involves an agent learning to decide the optimal actions by interacting with an environment. The agent explores the environment, receives feedback in the form of rewards, and uses this feedback to learn and improve its sequential decision-making over time. Therefore, the objective of reinforcement learning is for the agent to learn to take actions that maximizes the expected total reward. In this paper, we will discuss in detail of what this means.

II. Markov Chain

As discussed, RL learns to take optimal actions by interacting with an environment. This environment is formally modeled as Markov decision process (MDP). To start understanding MDP, we first need to understand Markov property (chain), which states, *the future is independent of the past given the present.*

We start first with stochastic process, because Markov chain is stochastic process with a special property. Stochastic process is a collection of random variables indexed by a time set. For instance, discrete time stochastic process can be described as

$$S_0, S_1, \ldots, S_t, S_{t+1}, \ldots \tag{1}$$

where the random variables $S_0, S_1, \ldots$ represent states at different time steps. The specific distribution of each random variable is less important than the fact that they form an ordered sequence over time. Similarly, a continuous time random process can be expressed as

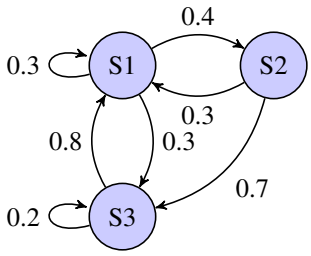
$$\{S_t | t \geq 0\}. \tag{2}$$

Markov chain is stochastic process with a special property; i.e., a stochastic process is Markov chain if it satisfy the following condition:

$$P(S_{t+1} = s_{t+1} | S_t = s_t) = P(S_{t+1} = s_{t+1} | S_t = s_t, \ldots, S_0 = s_0), \tag{3}$$

where $P(S_{t+1} = s_{t+1})$ is the probability that future state S_{t+1} takes the value of s_{t+1}, given that the current state is $S_t = s_t$. This means that the probability of the future state depends only on the current state and not on the sequence of past states. We denote that $P(S_{t+1} = s' | S_t = s)$ is the transition probability from current state s to future state s'. To this end, we can define Markov chain as a tuple (S, P). Here, S represents a set of states, and P represents a transition probability matrix.

1) Markov Chain Example: Consider an example, we have a Markov chain represented by this graph.



This graph can be described by transition matrix P, expressed as

$$P = \begin{bmatrix} 0.3 & 0.4 & 0.0 \\ 0.3 & 0.0 & 0.7 \\ 0.8 & 0.0 & 0.2 \end{bmatrix}. \tag{4}$$

Let us look at an example, consider that S_t (current state) is S_2. Given this information, the probability of next state being S_3 is $P(S_{t+1} = S_3 | S_t = S_2) = 0.7$.

III. Markov Decision Process

As explained, MDP is a mathematical model of the environment with which an agent interacts. MDP is very similar to Markov chain but the difference is that it is defined by a 5-tuple (S, A, P, R, γ), where

- S: state space, set of all possible states
- A: action space, set of all possible actions
- P: state transition probability, which gives the probability of transitioning from state s to state s' given action a,
- R: reward function, which gives the immediate reward received after transitioning from state s to state s' under action a,
- γ: discount factor.

We will discuss all these terms in detail in the following subsections.

A. State, Action and State Transition Probability

In this section, we describe how state and action is used to describe the transition probability. Just like Markov chain, MDP transition probability uses current state to predict the next state with the additional information of current action. Therefore, the transition probability is defined as

$$P^a_{ss'} = P(s'|s,a) = P(S_{t+1} = s'|S_t = s, A_t = a), \tag{5}$$

where the future state depends only on current state and the action.

B. Reward

Reward, denoted as R_t, is a scalar feedback indicating how well the agent is making decisions at step t. As mentioned in Sec. I, the underlying objective in reinforcement learning is to *find optimal actions that would maximize the expected cumulative rewards.* There are two important points to note about this statement: first, notice how the goal for the agent is to maximize the *expected* cumulative reward. What does it mean to have an expected reward? We describe this by considering an example depicted by Fig. 1. As shown, there are two states s' and s'' that we can transition to by taking action a. For each transition

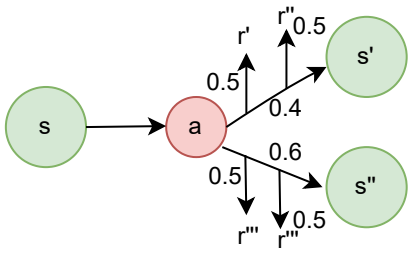


Fig. 1. Simple Markov Decision Process

from the action to the states, there are two possible rewards and we need to take in the account of all the rewards that we can get from a single state-action pair. This is why the agent is interested in maximizing the *expected* cumulative reward. The expression of expected reward is described as

$$R^a_s = r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} P(s', r|s, a). \tag{6}$$

Second important aspect to take note of is that we consider a *cumulative* reward. This means that the agent considers the rewards accumulated over many, or even all, time steps. This accumulated reward is referred to as the return, which will be explained in the next subsection.

1) Example: Transition Probability with MDP: Consider an environment that is described by MDP shown in Fig. 1. Given s and a, the transition probability of getting s' is described by (5). With law of total probability (62), this can be expressed in terms of reward as

$$P^a_{ss'} = P(s'|s,a) = \sum_{r\in\mathcal{R}} P(s', r|s, a), \tag{7}$$

where $\mathcal{R}$ is the set of all possible rewards. Now, conditional joint probability $P(s', r|s, a)$ can be decomposed to $P(s', r|s, a) = P(s'|s, a, r)P(r|a, s)$. For example, transition probability of next state being s' given current state s, and action a would be

$$P(s'|s,a) = \sum_{r\in\mathcal{R}} P(s'|s,a,r)P(r|s,a) = P(s'|s,a,r')P(r'|s,a) + P(s'|s,a,r'')P(r''|s,a) = 0.4\cdot 0.5 + 0.4\cdot 0.5 = 0.4. \tag{8}$$

C. Return

While reward is the immediate scalar feedback from the environment to the agent after the agent takes one action from a particular state, return is the total *discounted* reward from time step t. This is described as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{9}$$

where $\gamma \in [0, 1]$ is the discount factor. This awards the immediate actions and compensates the far actions. Most MDPs are discounted because of the following reasons,

- mathematically convenient,
- uncertainty of the future and
- immediate rewards may earn more interest than delayed rewards.

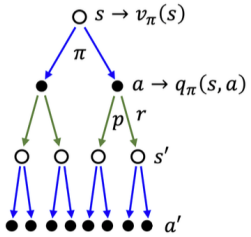


Fig. 2. Backup Diagram for Value Function

IV. Policy and Value Functions

A. Policy

Policy, denoted as π, is a *probability distribution* over actions for given states. This is expressed as

$$\pi(a|s) = P(A_t = a|S_t = s). \tag{10}$$

We note that (10) describing stochastic policy, which has a probability distribution of actions when given a state. On the other hand, deterministic policy has a defined action for a given state. Therefore, deterministic policy is described as

$$\pi(s) = a. \tag{11}$$

B. Value Function

Value function measures the goodness of each state s (or state-action pair (s, a)) when following a policy π in terms of the expectation of returns G_t. Note that we are now computing the *expected* returns. Reason for utilizing the expected returns can be explain with Fig. 2. Notice that the tree starts out with a state s. Because there are multiple actions you can take from s with π, it branches out to many possible actions and states. When the sequences of actions terminate, and the agent ends up in a final state, an episode has terminated. All episode ends up with returns denote as G_t^j, where $j = 1, ..., N$, and N is total number of episodes. We have to take into account of all the episodes and the corresponding returns to measure the goodness of one state; which justifies our reason for using the expected return value to define the value function. The state value function can then be described as

$$v_\pi(s) = G_t^1 P(G_t^1|s) + G_t^2 P(G_t^2|s) + \cdots + G_t^N P(G_t^N|s) = \mathbb{E}_\pi[G_t|S_t = s], \tag{12}$$

where $P(G_t^j|s)$ is probability of G_t^j happening when state s is given.

There are two types of value functions: *state-value* and *action-value*. We will describe them in the following subsections.

C. State-Value Function

The state-value function, denoted as $v_\pi(s)$, measures the quality of the state when following the policy π. The function is described as

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]. \tag{13}$$

Consider a backup diagram illustrated by Fig. 2. We are given a state and based on the state, the policy π has two actions. Each action transitions to different states with transition probability p with reward r. Then the process continues until the episode ends. Each episode outputs a return and expected value of each episode given a state is the state-value function.

D. Action-Value Function

Action-value function is the expected returns starting from state s, taking action a, and following policy π. This is described as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = a, A_t = a]. \tag{14}$$

Only difference between action-value function and state-value function is that action value considers a state-action pair. Therefore, complexity to solve for action-value function is greater than that of state-value function because action-value has to consider all the actions derived from each state. Action-value function is also known as Q-function.

We can express state-value function in terms of action-value function as

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a). \tag{15}$$

Further, we can express action-value function in terms of state-value function as

$$A_\pi(s, a) = q_\pi(s, a) - v_\pi(s). \tag{16}$$

V. Bellman Equation

A. Bellman Expectation Equation

Bellman expectation equation is a recursive equation decomposing state-value function $v_\pi(s)$ into immediate reward R_{t+1} and discounted next state-value $\gamma v_\pi(S_{t+1})$. Derivation for this expression is described as follows

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] && (17)\\
&= \sum_a \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \underbrace{P(A_t = a|S_t = s)}_{\pi(a|s)} && (18)\\
&= \sum_a \pi(a|s)\mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a], && (19)\\
&= \sum_a \pi(a|s) \sum_{s'}\sum_r \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r]P(s', r|s, a) && (20)\\
&= \sum_a \pi(a|s) \sum_{s'}\sum_r P(s', r|s, a)\big[r + \gamma \underbrace{\mathbb{E}[G_{t+1}|S_{t+1} = s']}_{v_\pi(s')}\big] = \sum_a \pi(a|s) \sum_{s'}\sum_r P(s', r|s, a)\big[r + \gamma v_\pi(s')\big] && (21)\\
&= \sum_a P(a|s)\Bigg[\sum_{s'}\sum_r rP(s', r|s, a) + \sum_{s'}\sum_r \gamma v_\pi(s')P(s', r|s, a)\Bigg] && (22)\\
&= \sum_a P(a|s)\Bigg[\sum_r rP(r|s, a) + \sum_{s'} \gamma v_\pi(s')P(s'|s, a)\Bigg] && (23)\\
&= \sum_a P(a|s)\big[\mathbb{E}[R_{t+1}|S_t = s, A_t = a] + \mathbb{E}[\gamma v_\pi(s')|S_t = s, A_t = a]\big] && (24)\\
&= \sum_a P(a|s)\big[\mathbb{E}[R_{t+1} + \gamma v_\pi(s')|S_t = s, A_t = a]\big] && (25)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(s')|S_t = s]. && (26)
\end{aligned}
$$

Note that (21) can also be expressed as $\sum_a \pi(a|s)[R^a_s + \gamma \sum_{s'} P^a_{ss'} v_\pi(s')]$. Now with (26), we have expressed the state-value function in terms of immediate reward and value-function of next state.

Similarly, we can express the action-function in a same way. This is expressed as

$$
\begin{aligned}
q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] && (27)\\
&= \sum_{s'}\sum_r P(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s')q_\pi(s', a')] && (28)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]. && (29)
\end{aligned}
$$

B. Bellman Optimality Equation

Before we describe the Bellman optimality equation, we first explain the optimal value function and policy. The optimal value function yields maximum value compared to all other value function. The goal of MDP is to find this optimal value function described as

$$v_*(s) = \max_\pi v_\pi(s). \tag{30}$$

Similarly, optimal action-value function is expressed as

$$q_*(s, a) = \max_\pi q_\pi(s, a). \tag{31}$$

We define a partial ordering policy that says $\pi' \geq \pi$ if $v_{\pi'} \geq v_\pi(s)$ for all s. This means that to say a policy π' is greater than or equal to π, the state-value function of π' must be greater than or equal to π for all s.

Fundamental theorem of MDP states that any MDP satisfies the following:

- there exists an *optimal policy* $\pi_* \geq \pi, \forall \pi$,
- all optimal policies achieve optimal state-value function, i.e., $v_{\pi_*} = v_*(s)$,
- all optimal policies achieve optimal action-value function, i.e., $q_{\pi_*}(s, a) = q_*(s, a)$.

Then, most important question narrows down to how we find this optimal policy. This is found by maximizing over $q_*(s, a)$. That is,

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_a q_*(s, a), \\ 0 & \text{otherwise.} \end{cases} \tag{32}$$

Note that since $v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a)$, then, the optimal state-value function is $v_{\pi_*} = \sum_a \pi_*(a|s) q_{\pi_*}(s,a)$. We noted that $v_{\pi_*} = v_*(s)$ and $q_{\pi_*}(s,a) = q_*(s,a)$. Therefore,

$$v_{\pi_*}(s) = \sum_a \pi_*(a|s) q_{\pi_*}(s,a) \tag{33}$$

$$= v_*(s) = \sum_a \pi_*(a|s) q_*(s,a), \tag{34}$$

where optimal policy $\pi_*(a|s)$ will deterministically only take action that yields maximum action-value. Therefore, optimal state-value function is the maximum action-value function, which is described as

$$v_*(s) = \max_a q_*(s,a). \tag{35}$$

We can see from (35), that if we know the optimal action-function, we can immediately find (35). The optimal action-function is described as

$$q_*(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_*(s'), \tag{36}$$

where $r(s,a)$ is described by (6). We can see that from (36), optimal action-value is found if we know optimal state-value for the next state, but only under the condition that transition probability $P(s'|s,a)$ is known. Therefore, if the MDP is known, we can solve for optimal state and action functions based on the known transition probabilities. This can be efficiently solved by dynamic programming. However, if the transition probabilities are not known, we use random sampling to approximate the optimal action-value. This method is solved with reinforcement learning.

We now describe Bellman optimality equation that is used to find the optimal value-function and action-function which then gets used to find the optimal policy. First, we describe the optimal value-function as

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s,a) = \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \tag{37}$$

$$= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \tag{38}$$

$$= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \tag{39}$$

$$= \max_a \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma v_*(s')]. \tag{40}$$

Similarly, optimal action-function is described as

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \tag{41}$$

$$= \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')]. \tag{42}$$

VI. Optimal Policy With Dynamic Programming

A. Value Iteration

In this section, we describe how we find optimal state-value function $v_*(s)$. By finding $v_*(s)$, we can find optimal deterministic policy which is described as

$$\pi_*(s) = \arg\max_a q_*(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_*(s') \tag{43}$$

$$= \arg\max_a \sum_r r \sum_{s'} P(s', r|s, a) + \gamma \sum_r r \sum_{s'} P(s', r|s, a) v_*(s') \tag{44}$$

$$= \arg\max_a \sum_r \sum_{s'} P(s', r|s, a)[r + \gamma v_*(s')]. \tag{45}$$

This shows that optimal policy is a function of optimal state-action function. Now, we see from (35) that to compute optimal state-value function, we need state-value for all possible states. This is impossible to solve if we have a large state space. Therefore, we solve this iteratively as described as follow

$$V_{k+1}(s) \leftarrow \max_a \sum_{s',r} P(s', r|s, a)[r + \gamma V_k(s')]. \tag{46}$$

We computes the (46) until convergence. After we calculate the optimal state-value function, we can compute the optimal policy by solving (45). Complete algorithm to solve for optimal state-value function and policy function is described by Algorithm 1.

There are several drawbacks to using value iteration, they are listed as follows:

- the action argument inducing the max at each state rarely changes, so the policy often converges long before the value converges,
- value iteration is very slow as $O(S^2 A)$ per iteration and needs many iterations to converge.

Algorithm 1 Value Iteration

Hyperparameter: Small threshold $\epsilon > 0$ for the convergence check.
Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$, except $V(\text{terminal}) = 0$.
repeat
  $\Delta \leftarrow 0$
  for each state $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} P(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \epsilon$
Output a deterministic policy $\pi \approx \pi_*$ such that $\pi(s) = \arg\max_a \sum_r \sum_{s'} P(s', r|s, a)[r + \gamma V(s')]$

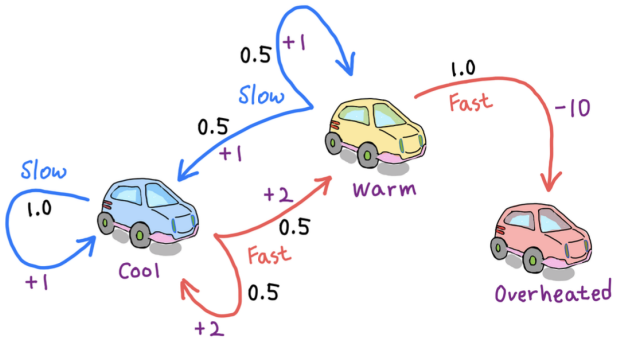


Fig. 3. Example for Value Iteration

B. Example: Markov Decision Process for Car Control

Consider an example depicted by Fig. 3. A robot car wants to travel far and quickly. Here are the required information for this example:

- 3 states: cool, warm, overheated,
- 2 actions: slow, fast,
- rewards: slow=1, fast=2 (but -10 when overheated).

Further, transition probability is given by the figure. As shown, if the current state is cool, it can take two actions: slow and fast. If it decides to take slow action, then there is one deterministic action to only stay at the state cool. However, if it decides to take the action fast, it transitions to warm state with probability of $1/2$. It also transitions to cool again with probability $1/2$.

Using the information, we update (46). We first initialize the state-value of each functions to be 0. Then, using the initialized value, we update state-value for each states. We do this until state-value function converges. We can see how the state-value function updates each iteration by looking at Table I.

Take a closer look at how $V_2(s)$ gets updated. $V_2(\text{warm}) = 0.5(1+2)+0.5(1+1) = 2.5$, when $a = \text{slow}$ and $V_2(\text{warm}) = 1.0(-10+0) = -10$ when $a = \text{fast}$. Therefore, slow action maximize $V_2(\text{warm})$. Hence, the table gets updated. Notice that as the table get updated, actions for each state becomes fixed way before $V(s)$ converges. This is main draw back of value-iteration.

C. Policy Iteration

Policy iteration improves upon value-iteration by mitigating the drawbacks of value-iteration. Policy iteration repeats *policy evaluation* and *policy improvement* until convergence. We start with policy evaluation. Instead of updating the table by

$V(s)$	cool	warm	overheated
$V_2(s)$	3.5	2	0
$V_1(s)$	2	1	0
$V_0(s)$	0	0	0

TABLE I
VALUE ITERATION UPDATE

maximizing over all actions, policy evaluation only compute one value-function V^π from a deterministic policy π. Then, we update $V_{k+1}(s)$, described as

$$V_{k+1}(s) \leftarrow \sum_r \sum_{s'} P(s', r|s, \pi(s))[r + \gamma V_k(s')] \tag{47}$$

for all $V_k(s')$ until convergence. Then, we aim to improve π to π' by greedy policy improvement based on V^π. Description of this is shown as

$$\pi'(s) = \arg\max_a \sum_r \sum_{s'} P(s', r|s, a)[r + \gamma V^\pi(s')] = \arg\max_a Q^\pi(s, a). \tag{48}$$

We then update (47) once more with the updated deterministic policy (48). We repeat this until there is convergence with both policy evaluation and policy improvement. This method is definitely more efficient than value iteration because we use fewer iterations to reach optimality. However, how do we make sure that this method will reach optimality? We check by going over policy improvement theorem.

Policy improvement theorem first consider that there are two policies π and π'. If $Q^\pi(s, \pi'(s)) \geq V^\pi(s)$ for all $s \in S$, then $V^{\pi'}(s) \geq V^\pi(s)$ for all $s \in S$. We prove this in the Appendix C. Complete algorithm for policy iteration is described in Algorithm 2.

Algorithm 2 Policy Iteration

Initialization: $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
Policy Evaluation
repeat
  $\Delta \leftarrow 0$
  for each state $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} P(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \epsilon$ (a small positive number for the convergence check)
Policy Improvement
policy-stable$\leftarrow$ true
for each $s \in \mathcal{S}$ **do**
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} P(s', r|s, a)[r + \gamma V(s')]$
  if *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
end for
if *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

D. Generalized Policy Iteration

In the case of known MDP, it is possible to find a complete value function for policy evaluation, which means that the value function is iterated fully until convergence before policy improvement. On the other hand, if the MDP is not known, we can only approximate the value function for the policy evaluation. Although the value function found via policy evaluation is merely an estimate, we can still alternate between the approximated value function in policy evaluation and policy improvement to find the optimal policy. This is called generalized policy iteration (GPI). In the next sections, we delve into reinforcement learning, which deals with unknown MDPs. In this case, value function in policy evaluation must be approximated; therefore, it is based on GPI.

VII. Monte-Carlo Method

We now delve into reinforcement learning which deals with *model free* MDP environment. Model free environment under MDP setting simply means that it does not explicitly have knowledge of the transition probability; therefore, solving for optimal policy based on dynamic programming is no longer possible. To tackle this problem, we estimate the transition probability. One of the most renown method under this setting is Monte Carlo (MC) method. It utilizes the property of law of large numbers, described in Appendix B, to approximate the action-value function that utilizes the empirical mean to estimate the action-value function. Since approximation of action-value function is utilized, MC cannot use a complete policy to find optimal policy through policy iteration; hence it uses GPI. Further, the distinguishing feature about MC method is that it uses samples from the entire episode to update policy evaluation function $Q(s, a) = q_\pi(s, a)$.

The goal of MC prediction is to learn $q_\pi(s,a)$ (14) from samples of entire episodes of real experience under policy π. To estimate $q_\pi(s,a)$, at time-step t, we increment number of times a particular state action pair (s,a) was visited as $n(s,a) \leftarrow n(s,a)+1$. We further increment the total return as $S(s,a) \leftarrow S(s,a)+G_t$. The action-value then, is estimated as empirical mean return: $Q(s,a) = \frac{S(s,a)}{n(s,a)}$. We can modify this equation slightly by the using the following property

$$\mu_k = \frac{1}{k}\sum_{i=1}^{k} x_i = \frac{1}{k}\big(\sum_{i=1}^{k-1} x_i + x_k\big) \tag{49}$$

$$= \frac{1}{k}\big((k-1)\mu_{k-1} + x_k\big), \tag{50}$$

since $\mu_{k-1} = \frac{1}{k-1}\sum_{i=1}^{k-1} x_i$ which yields $\mu_{k-1}(k-1) = \sum_{i=1}^{k-1} x_i$. Therefore,

$$\mu_k = \frac{1}{k}(\mu_{k-1}(k) - \mu_{k-1} + x_k) \tag{51}$$

$$= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}). \tag{52}$$

We apply this directly to calculate $Q(s,a)$. We can update $Q(s,a)$ after one episode, we increment number of time each state-action pair appears as $n(S_t,A_t) \leftarrow n(S_t,A_t)+1$. As (52), $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{1}{n(S_t,A_t)}[G_t - Q(S_t,A_t)]$. In practice, we can use a step size α to update $Q(S_t,A_t)$ as

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[G_t - Q(S_t,A_t)]. \tag{53}$$

Algorithm 3 Monte Carlo Method

Initialization: $Q(S,A)$, all $S \in \mathcal{S}$, $A \in \mathcal{A}(S)$, arbitrarily and $Q(\text{terminal}) = 0$
  $\text{Returns}(S,A) \leftarrow$ empty list
  $\pi \leftarrow$ arbitrarily ϵ-soft policy (non-empty probabilities)
Policy Evaluation
repeat
  $\Delta \leftarrow 0$
  for each state $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} P(s',r|s,a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \epsilon$ (a small positive number for the convergence check)
Policy Improvement
policy-stable$\leftarrow$ true
for each $s \in \mathcal{S}$ **do**
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} P(s',r|s,a)[r + \gamma V(s')]$
  if *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
end for
if *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

VIII. Temporal Difference Learning

The downside of MC method is that it updates the action value function after an episode terminates. Instead, we can update the action value function based on immediate reward and the next state action value function. To do this, we modify express the return as $G_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ and modify (53) as

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[\underbrace{R_{t+1} + \gamma Q(S_{t+1},A_{t+1})}_{target} - \underbrace{Q(S_t,A_t)}_{behavioral}]. \tag{54}$$

This update is called SARSA because it require state action and reward from current time stamp and state and action from next time stamp to update the value function. We categorize the terms in (54) for further analysis. That is, there is behavioral, denoted as $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ and target expressed as $Q(S_t, A_t)$. The for TD is that we wa Further modification on (54) can be made by modifying it as

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1},A_{t+1})}_{target} - \underbrace{Q(S_t,a)}_{behavioral}]. \tag{55}$$

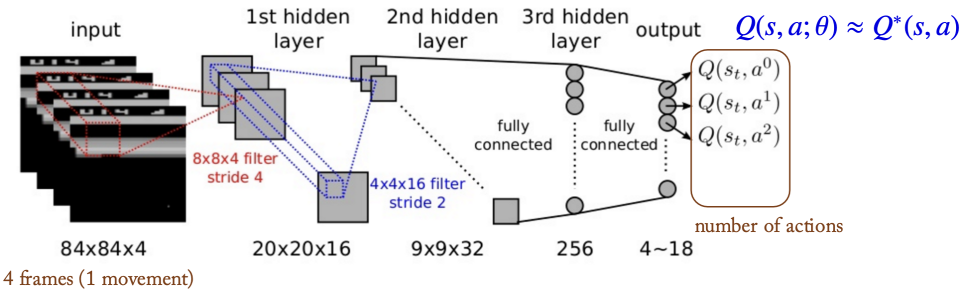


Fig. 4. DQN Model

This modified update is Q-learning method. We note that By comparing (54) and (55), we note important facts. That is, on-policy and off-policy. As shown by (54), behavioral

IX. Deep Reinforcement Learning

Traditional RL is a tabular updating method which updates state-action value $Q(s,a)$ table with Bellman equation. The downside of this method is that state and action space has to be in manageable scale. On the other hand, deep RL (DRL) is abandoning the tabular method and it is approximating the state-action value function with deep neural network (DNN).

X. Deep Q-Network

Deep Q-Network (DQN) is the first DNN based Q-learning model that successfully learned the policies to play Atari games. The input to such a system is naturally 2D images with many number of frames per second. The state space is the representation with intensity of each pixel which makes Atari games to have a very large scale state space. Normally this problem would be challenging to solve with classical RL, because it needs to represent each state and action space explicitly as a table. On the other hand, DQN solves this problem with the aid of convolutional neural network (CNN). Because the CNN reduces the dimension of the input by extracting the important features of a high dimensional image to lower dimension, DQN is able to work with manageable state space.

Fig. 4 shows the overall model depiction of DQN. As shown in the figure, the input to the system is a particular state of an Atari game frames. It takes in 4 consecutive 84 by 84 2D images. If the intensity of each pixel is in the range from 0 to 255, the state space is $(84*84*4)^{256} = 28224^{256}$. This scale is too large for classical RL to handle. On the other hand, with CNN, the DQN model is able to bring down the dimension of the input space to 256 by extracting important features of the images. Then, with the extracted features, it approximates the state action value functions for the corresponding input state. However, note that it still requires to output each corresponding actions to the state. In other words, action space of DQN must still be in manageable scale.

A. Naive DQN

Intuitively, the concept of DQN is quite simple. We update (55) by parameterizing behavior and target action value functions with θ; this is described as

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\big[\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1},a;\theta)}_{target} - \underbrace{Q(S_t,A_t;\theta)}_{behavioral}\big]. \tag{56}$$

We update the parameter θ by setting the loss function as

$$L(\theta) = [\underbrace{r_{t+1} + \gamma \max_a Q(s_{t+1},a;\theta)}_{target} - \underbrace{Q(s_t,a_t;\theta)}_{behavioral}]^2, \tag{57}$$

and updating it with stochastic gradient descent (SGD). This whole procedure is known as naive DQN. However, this method often performs worst or as well as a simple linear model. There are two reasons and there are methods to tackle these problems to make DQN perform far better than a linear model.

The first reason why naive DQN performs poorly is the temporal correlation between inputs. Since an RL agent is interacting with an environment modeled as MDP, input sequence must be temporally correlated. However, as shown in (57), DQN is trained with MSE, which assumes that each data is under i.i.d. assumption. Furthermore, it throws out the past experience that might be very crucial for the model to remember to make better decision. DQN can be reinforced by solving this problem with *experience replay* or replay buffer. This stores experiences that includes 4-tuples of state transitions, actions, and rewards

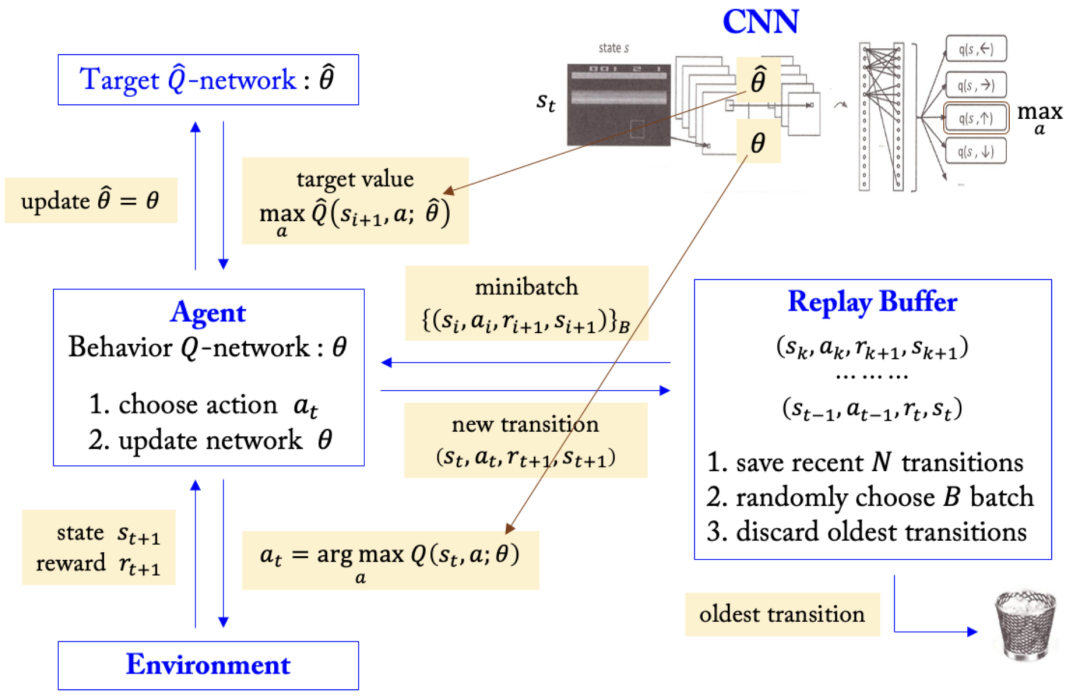


Fig. 5. DQN Model with Experience Replay and Target Network

$\{(s, a, r, s')\}$ to perform Q-learning. Then in actual training, random experiences are selected in mini-batch to train the model using backpropagation. The significance of this technique is stated as follows

- reduces temporal correlation between experiences during training,
- increases learning speed with minibatches,
- increases data efficiency by re-using same samples repeatedly.

Second reason why naive DQN performs poorly is because it is learning from a non-stationary target. As shown in (57), target as well as behavioral is parameterized by θ. This means that with every update on the parameter, the target also changes. This makes training very difficult. A target network technique is used to mitigate this problem. This fixes previous parameters $th\hat{e}ta$ to target $\hat{\theta}$-network and updates parameters θ only on behavior Q-network. Now, with this technique, loss function is expressed as

$$L(\theta) = \frac{1}{B}\sum_i [r_{i+1} + \gamma \max_a \hat{Q}(s_{i+1}, a; \hat{\theta}) - Q(s_i, a_i; \theta)]^2. \tag{58}$$

The $\hat{\theta}$ that parametrizes the target is not always stationary but get updated after many updates in the behavioral function. The complete DQN model incorporating experience replay and target network is depicted by the Fig. 5. As shown, it starts with an agent choosing an action (a_t) based on a state (s_t). The environment sends the next state and rewards pairs (s_{t+1}, r_{t+1}). Agent then sends the gathered the information of $(s_t, a_t, r_{t+1}, s_{t+1})$ to replay buffer. The replay buffer saves the new information and discards the oldest information. We choose B minibatch and send it to the agent which approximates the action value by learning it with target $\hat{Q}$ network.

B. Dueling DQN

We now explain dueling DQN (DDQN), which is a variant of DQN. This variant was developed to improve upon DQN's inability to measure the goodness state s apart from the action a. This is because the output of DQN is state value function corresponding to all actions. Therefore, DDQN is particularly useful in states where its actions do not affect the environment. To this end the DDQN outputs two streams: advantage $A(s, a)$ and action-independent state value $V(s)$. The advantage function is defined as

$$A(s, a) = Q(s, a) - V(s). \tag{59}$$

As shown by (59), advantage function subtracts state action value by the state value function. Since state value represents the *goodness* of a particular state s, and state action value measures the goodness of a state action pair s, a, subtracting them will measure the goodness of the action a. Therefore, intuitively we can justify advantage function as measuring the merit of taking action a. On the other hand, $V(s)$ is simply a state value function which measures the values of states.

XI. Policy Gradient

The downside of DQN is that it cannot work with large or continuous action space because each output is allocated to action value function corresponding to an action. For example, consider a robotics control problem where the objective is to make a robot walk. A robot might have four joints and the angle of each joint is the action. The joints can be controlled from an integer value angle from -90 to 90. This alone gives 181 possible angles for each joint, which yields 181^4 action space. If this was to be handled by DQN, it would require 181^4 outputs, which is far too complex. Policy gradient mitigates this problem by directly approximating the policy instead of action value function. It can assign an action variable for each joint, which reduces the output dimension from 181^4 to 4. Then, it learns the distribution of action for each action variable. This makes policy gradient to be able to handle very large or continuous action space.

APPENDIX A
APPENDIX A: LAW OF TOTAL PROBABILITY

We have a sample space S, it consists of n disjoint events $B_1, B_2, \cdots, B_n$ which makes up the sample space S. We also have an event A, in the sample space, which is made up of parts in $B_1, B_2, \cdots, B_n$. Then, event A can be described as

$$A = (A \cap B_1) \cup (A \cap B_2) \cup \cdots \cup (A \cap B_n). \tag{60}$$

The probability of event A happening then is expressed as

$$P(A) = P(A \cap B_1) + P(A \cap B_2) + \cdots + P(A \cap B_n), \tag{61}$$

where we use sum rule of probability to sum the probabilities because $B_1, B_2, \cdots, B_n$ are disjoint. Then, using the conditional probability, we can express $P(A \cap B) = P(A|B)P(B)$. Therefore, (61) can be expressed as

$$P(A) = \sum_n P(A|B_n)P(B_n). \tag{62}$$

Now, this can be applied to expected value where expected value of random variable X can be expressed in Y as

$$\mathbb{E}[X] = \sum_y \mathbb{E}[X|Y = y]P(Y = y), \tag{63}$$

where $\mathbb{E}[X] = \sum_x xP(X = x)$. This can be extended to conditional expected value, where it is expressed as

$$\mathbb{E}[X|Z = z] = \sum_y \mathbb{E}[X|Y = y, Z = z]P(Y = y|Z = z). \tag{64}$$

APPENDIX B
APPENDIX B: LAW OF LARGE NUMBERS

Law of large number states that the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed. Consider n number of i.i.d random samples, $X_1, ..., X_n$.

$$\bar{X} = \frac{1}{n}(X_1 + ... + X_n) \to \mathbb{E}[X] \quad \text{as} \quad n \to \infty. \tag{65}$$

APPENDIX C
APPENDIX C: POLICY IMPROVEMENT THEOREM