ReinforcementLearning Notes

Avinash Reddy

January 2023

Chapter 1

Introduction

Introduction

Reinforcement Learning is also known as

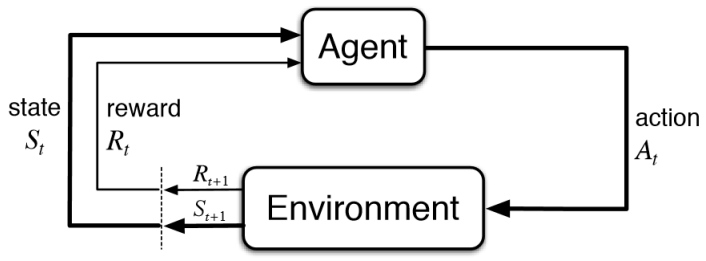
- optimal control
- Approximate Dynamic programming
- Neuro-Dynamic Programming

Reinforcement Learning is an area of machine learning inspired by the behavioural psychology concerned with how software agents ought to take actions in an environment so as to maximise some notion of cumulative reward.
Animal psychology

- negative rewards - pain and hunger
- positive reinforcements - pleasure and food
- reinforcements used to train animals

Applying the similar philosophy to software agents

1.1 Definition



Reinforcement Leaning Application Areas :

- Game Playing
- Operations Research
- Elevator Scheduling
- controls
- spoken dialouge systems
- data center energy consumption
- self managing networks
- autonomous vehicles
- computational finance

Chapter 2

Multi-Armed Bandits

We are not interested in this chapter. Or else we will visit at end of our journey. Just introduced to match the chapters number with Sutton and Barto book chapters.

Chapter 3

Markov Decision Process

Definition

- State $s \in S$
- Action $a \in A$
- reward $r \in \mathbb{R}$
- Transition model $Pr(s_{t+1}|s_t, a_t)$
- Reward model $Pr(r|s_{t+1}, s_t, a_t)$
- Discount Factor $\gamma \in [0, 1]$
- Horizon T

Goal is to find a policy $\pi(a|s)$ that maximises the expectation of discounted return.

How RL differs from MDP solutions

- No Transition Model
- No Reward Model

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

However, we still solve the MDP problem using RL by interacting with the environment by learning the transition and reward models or directly learning the policy.

Types of RL algorithms

- Model Based- if we try to learn the transition and reward models
- Model Free - here, we don't learn any model dynamics. No transition and reward models. Below are the types of model free RL algorithms
- Value Based- if we try to learn the value function $V(s)$ of the state or value function of state-action pair $Q(s, a)$.
- Policy Based- if we try to learn the policy $\pi(a|s) - \pi(s, a)$ directly.
- Policy Gradient- if we try to learn the policy $\pi(a|s) - \pi(s, a)$ directly using gradient ascent.
- Actor Critic - contains both policy $\pi(a|s) - \pi(s, a)$ and value function $V(s) - Q(s, a)$.

Chapter 4

Dynamic Programming

we compute the state value function $V^\pi(s)$ for the policy π. This is called the *policy evaluation* problem. We can write the Bellman equation for the state value function as

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_t = s \right] \\ &= \mathbb{E}_\pi \left[G_t \mid s_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid s_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma V^\pi(s') \right] \end{aligned} \tag{4.1}$$

If enviroment dynamics are known, then $V^\pi(s)$ is a simultaneous linear equations in $|S|$ unknowns. Iterative solutions are most suitable. Assume v_0 as the initial approximation for the state value function. Then update rule is given by the bellman equation

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_k(s') \right] \tag{4.2}$$

This iterative approach converges to $V^\pi(s)$. The policy evaluation problem is solved when $v_k(s) = V^\pi(s)$ for all $s \in S$.

Iterative Policy Evaluation Algorithm

State-Action Value Function

The state-action value function $Q^\pi(s, a)$ is defined as the expected return starting from state s, taking action a, and then following policy π thereafter. We can write the Bellman equation for the state-action value function as

Algorithm 1 Iterative Policy Evaluation

1: Input policy π, initial approximation $v_0(s) \forall s \in S$,
2: Set discount factor γ, $k = 0$, $V(terminal) = 0$
3: $v_k(s) \leftarrow v_0(s)$ for all $s \in S$
4: **while** $|v_k - v_{k-1}| < \theta$ **do**
5: $\quad v_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma v_k(s')]$ for all $s \in S$
6: $\quad k \leftarrow k + 1$
7: **end while**
8: **return** $v_k(s)$ for all $s \in S$

$$
\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_t = s, a_t = a \right] \\
&= \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right] \\
&= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid s_t = s, a_t = a \right] \\
&= \mathbb{E}_\pi \left[R_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a \right] \\
&= \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \sum_{a'} \pi(a' \mid s') Q^\pi(s', a') \right] \\
&= \mathbb{E}_\pi \left[R_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a \right] \\
&= \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V^\pi(s') \right]
\end{aligned}
\tag{4.3}
$$

policy improvement theorem helps us in updating the policy once we found out the value function of a policy by policy evaluation. The theorem states that if we have a policy π and a value function $V^\pi(s)$, then there exists a policy π' such that $V^{\pi'}(s) \geq V^\pi(s)$ for all $s \in S$. This means that we can always improve the value of a policy by following a greedy policy with respect to the value function. The greedy policy with respect to the value function is defined as

$$
\pi'(s) = \arg\max_{a \in \mathcal{A}} Q^\pi(s, a) \tag{4.4}
$$

which leads to $V^{\pi'}(s) \geq V^\pi(s)$ for all $s \in S$.

Policy Improvement Algorithm

Assuming a deterministic policy $\pi(s) = a$

Algorithm 2 Policy Improvement

1: Input policy π, value function $v(s)$ for all $s \in S$,
2: Set discount factor γ
3: Evaluate the policy π to get $v(s)$ for all $s \in S$ using Algorithm 1 Policy Evaluation.
4: Improve the policy by following a greedy policy with respect to the value function.$\pi \leftarrow \arg\max_{a \in \mathcal{A}} Q^\pi(s, a)$ for all $s \in S$

However, policy improvement algorithm involves policy evaluation. So, we need to repeat the policy evaluation and policy improvement until the policy converges. So, Value iteration algorithm updates the value function by acting greedily with respect to the value function and then improves the policy by following a greedy policy with respect to the value function. The algorithm is given below.

Value Iteration Algorithm

Assuming a deterministic policy $\pi(s) = a$

Algorithm 3 Value Iteration

Input policy π, initial approximation $v_0(s) \forall s \in S$,
Set discount factor γ, $k = 0$, $V(terminal) = 0$
$v_k(s) \leftarrow v_0(s)$ for all $s \in S$
while $|v_k - v_{k-1}| < \theta$ **do**
    $v_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s',r} p(s', r \mid s, a) [r + \gamma v_k(s')]$ for all $s \in S$
    $k \leftarrow k + 1$
end while
return $v_k(s)$ for all $s \in S$

Chapter 5

Monte Carlo Methods

Dynamic Programming methods are suitable for MDPs for which the model dynamics are known. In this chapter, we will discuss the methods that can be used to solve MDPs for which the model dynamics are not known. These methods are called Monte Carlo methods. The main idea behind Monte Carlo methods is to estimate the value function by averaging the returns obtained from the episodes. The main advantage of Monte Carlo methods is that they do not require the model dynamics to be known. The main disadvantage of Monte Carlo methods is that they are computationally expensive.

5.1 Monte Carlo Prediction

In earlier chapters, we defined value function as expected sum of discounted rewards. To estimate the expected sum, one way is to average the discounted rewards obtained from the episodes.This is the idea behind the monte carlo methods. Even the core idea of averaging remains same, there are simple variations in the way we consider the rewards for averaging.
Let us discuss two simple algorithms.

- First-visit MC

- Every visit MC

Algorithm 4 First-visit MC

Initialize $V(s)$ arbitrarily for each $s \in S$, and policy π to be evaluated
Initialize $Returns(s), N(S_t)$ as empty list, for each $s \in S$
while each episode **do**:
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \cdots S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    while Loop for each step of episode $t = T-1, \cdots, 2, 1, 0$ **do**:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless S_t is already in $S_0, S_1, \cdots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $N(S_t) \leftarrow N(S_t) + 1$
            $V_{n+1}(S_t) \leftarrow V_n(S_t) + \frac{1}{N(S_t)}[G - V_n(S_t)]$
    end while
end while

First visit MC averages the rewards following from the first visit of the state.

Algorithm 5 Every-visit MC

Initialize $V(s)$ arbitrarily for each $s \in S$, and policy π to be evaluated
Initialize $Returns(s), N(S_t)$ as empty list, for each $s \in S$
while each episode **do**:
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \cdots S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    while Loop for each step of episode $t = T-1, \cdots, 2, 1, 0$ **do**:
        $G \leftarrow \gamma G + R_{t+1}$
        Append G to $Returns(S_t)$
        $N(S_t) \leftarrow N(S_t) + 1$
        $V_{n+1}(S_t) \leftarrow V_n(S_t) + \frac{1}{N(S_t)}[G - V_n(S_t)]$
    end while
end while

Every visit MC averages the rewards following every visit of the state.

One can modify the algorithm to find the state action value function. The algorithm is same as above except that we have to maintain the state action value function instead of state value function.

Monte Carlo methods doesn't bootstrap and the estimated value function is state independent. It means that monte carlo method doesn't use the estimate of other states to estimate the value of a state. It is also called on-policy method because it uses the same policy to generate the episodes and to evaluate the policy.

5.2 Monte Carlo Control

We now have the algorithm to evaluate any given policy π in unknown dynamics MDP. We can use those algorithm to evaluate the policy π and then improve the policy. This is the idea behind Monte Carlo control.Using the same strategy used in policy improvement algorithm, we find the values function for a given policy and then act greedy with respect to the value function which improves the earlier policy. We repeat this process until we find the optimal policy.

There are inherent assumptions in the above algorithms. One assumption is exploring starts, all the state-action pairs are explored enough to average the rewards. Another assumption is we have to sample infinite episodes to converge. The secon assumption we can relax by following the general policy improvement algorithm.

We will see modified version of first visit MC algorithm to find the optimal policy. The algorithm is same as first visit MC algorithm except that we use the greedy policy with respect to the value function to generate the episodes.

Algorithm 6 First Visit MC with policy improvement

Initialize $Q(s,a)$ arbitrarily for each $s \in S, a \in A$, and policy π to be evaluated
Initialize $Returns(s,a), N(s,a)$ as empty list, for each $s \in S, a \in A$
while each episode **do**:
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \cdots S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    while Loop for each step of episode $t = T-1, \cdots, 2, 1, 0$ **do**:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless S_t is already in $S_0, S_1, \cdots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $N(S_t) \leftarrow N(S_t) + 1$
            $Q_{n+1}(S_t, A_t) \leftarrow Q_n(S_t, A_t) + \frac{1}{N(S_t)} [G - Q_n(S_t, A_t)]$
            $\pi(S_t) \leftarrow argmax_a Q_{n+1}(S_t, A_t)$
    end while
end while

Now, we will apply the same policy improvement idea on every visit MC algorithm.

Algorithm 7 Every Visit MC with policy improvement

Initialize $Q(s,a)$ arbitrarily for each $s \in S, a \in A$, and policy π to be evaluated
Initialize $Returns(s,a), N(s,a)$ as empty list, for each $s \in S, a \in A$
while each episode **do**:
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \cdots S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    while Loop for each step of episode $t = T-1, \cdots, 2, 1, 0$ **do**:
        $G \leftarrow \gamma G + R_{t+1}$
        Append G to $Returns(S_t, A_t)$
        $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
        $Q_{n+1}(S_t, A_t) \leftarrow Q_n(S_t, A_t) + \frac{1}{N(S_t)} [G - Q_n(S_t, A_t)]$
        $\pi(S_t) \leftarrow argmax_a Q_{n+1}(S_t, A_t)$
    end while
end while

From sutton and barto, the convergence to an optimal policy is inevitable. However, there is no formal proof to show the convergence.

To relax the first assumption, we should make sure that policy is epsilon soft. $\pi(\cdot|S_t) > 0$. This is acheived by using ϵ-greedy policy.In contrast to the greedy approach, we do not choose action greedily rather we choose action with probability ϵ randomly and with probability $1 - \epsilon$ greedily.

This is called ϵ-greedy policy. We will see the algorithm to find the optimal policy using ϵ-greedy policy.

Algorithm 8 Every Visit MC with ϵ-greedy policy improvement

Initialize $Q(s,a)$ arbitrarily for each $s \in S, a \in A$, and policy π to be evaluated
Initialize $Returns(s,a), N(s,a)$ as empty list, for each $s \in S, a \in A$
while each episode **do**:
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \cdots S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    while Loop for each step of episode $t = T-1, \cdots, 2, 1, 0$ **do**:
        $G \leftarrow \gamma G + R_{t+1}$
        Append G to $Returns(S_t, A_t)$
        $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
        $Q_{n+1}(S_t, A_t) \leftarrow Q_n(S_t, A_t) + \frac{1}{N(S_t)} [G - Q_n(S_t, A_t)]$
        $\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & \text{if } a = argmax_a Q_{n+1}(S_t, A_t) \\ \frac{\epsilon}{|A|} & \text{otherwise} \end{cases}$
    end while
end while

From sutton and barto, the above epsilon-greedy policy improvement algorithm also converges to an optimal policy. Till now, we have seen the on-policy algorithms to find the optimal policy. On-policy algorithms suffer from disadvantage that it can't reuse the generated episodes. To overcome this disadvantage. On-policy approach means we use a policy to generate episodes and use the same policy to improve the policy. Off-policy approach means we use a policy called *behaviour policy* to generate episodes and use another policy called *target policy* to improve continously.

we first solve the prediction problem of value function using off-policy approach. We will use the same algorithm as we used for on-policy approach. The only difference is that we will use the behaviour policy to generate episodes and use the target policy to improve the policy. we make an assumption that $b(a|s) > 0$ to facilitate the exploring starts behaviour. On the otherhand, target policy can be deterministic. While the behaviour policy must be stochastic to facilitate the exploration.

Importance Sampling is the trick followed by all off-policy algorithms. With the help of importance sampling, we can reuse the generated episodes. Instead of accumulting the discounted sum of rewards, we will accumulate the weighted discounted sum of rewards. The weight is the ratio of the probability of the action taken by the behaviour policy to the probability of the action taken by the target policy.

Given a starting state S_0 under a stochastic policy π, the resulting trajectory will have a probability distribution

$$P_\pi(S_0, A_0, R_1, S_1, A_1, R_2, \cdots, S_{T-1}, A_{T-1}, R_T) = \prod_{t=0}^{T-1} \pi(A_t|S_t) \cdot P(S_{t+1}|S_t, A_t) \cdot R_{t+1}$$

We define the relative probability of the trajectory under the target policy π and the behaviour policy b as

$$\begin{aligned}\rho_{t:T-1} &= \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) Pr(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) Pr(S_{k+1}|S_k, A_k)} \\ &= \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}\end{aligned}$$

This is called importance sampling ratio.If you sample with behaviour policy b and then estimate $V(s)$ as $\mathbb{E}[G_t|S_t]$ which will result in $V_b(s)$. Nevertheless, we need to estimate $V_\pi(s)$ which is the value function under the target policy. We can use the importance sampling ratio to estimate $V_\pi(s)$ as expected weighted sum of rewards, where the weight is given by the importance sampling ratio. Instead of $\mathbb{E}[G_t|S_t]$, you approximate $\mathbb{E}[\rho_{t:T-1}G_t|S_t] = V_\pi(s)$. ρ is the importance sampling ratio.

Off-policy Every Visit MC Prediction Algorithm needs to track where the state-action pair occurs in the episode. Let it be $T(s, a)$. We store the time steps at which the state-action pair occurs in the episode. This favours us to calculate the importance sampling ratio from those time steps. We will use the importance sampling ratio to calculate the weighted sum of rewards. This will give us the value function under the target policy.

$$Q_\pi(s, a) = \frac{\sum_{t \in T(s,a)} \rho_{t:T(t)-1} G_t}{|T(s, a)|}$$

This is called ordinary importance sampling.

$$Q_\pi(s, a) = \frac{\sum_{t \in T(s,a)} \rho_{t:T(t)-1} G_t}{\sum_{t \in T(s,a)} \rho_{t:T(t)-1}}$$

This is an alternative way of calculating the value function under the target policy. This is called weighted importance sampling. Now the question is which algorithm to follow or which algorithm leads to convergence and at what rate.

Let's see in case of first visit algorithms. The differences are generally expressed in terms of biases and variances. Ordinary importance sampling is unbiased wheras weighted importance sampling is biased(although the bias converges to zero finally). Ordinary importance sampling has unbounded variance. On the other hand, weighted importance sampling had lower variance and is the preferred algorithm. Nevertheless, we will use the ordinary importance sampling while using approximate value function methods.

In every visit monte carlo case, both ordinary importance sampling and weighted importance sampling are biased but eventually converge to zero.

We will look at the implementation of off-policy every visit monte carlo prediction algorithm.

we are estimating

$$Q_\pi(s, a) = \frac{\sum_{t \in T(s,a)} \rho_{t:T(t)-1} G_t}{\sum_{t \in T(s,a)} \rho_{t:T(t)-1}}$$

we approximate it with the weight parameter W_k.

$$Q_\pi(s, a) = \frac{\sum_{t \in T(s,a)} W_k G_t}{\sum_{t \in T(s,a)} W_k}$$

An incremental update rule for $Q_\pi(s, a)$ is

$$Q_{n+1}(s,a) = Q_n(s,a) + \frac{W_n}{C_n}\left[G_n - V_n\right]$$

where $C_{n+1} = C_n + W_{n+1}$

Off policy Every Visit Monte Carlo Prediction Algorithm

Algorithm 9 Off policy Every Visit Monte Carlo Prediction Algorithm

Input an arbitrary policy π and a behaviour policy b
Initialize $Q(s,a)$ arbitrarily for all $s \in S$ and $a \in A(s)$
Initialize $C(s,a)$ with zeros for all $s \in S$ and $a \in A(s)$
while each episode **do**
    Generate an episode $S_0, A_0, R_1, S_1, A_1, R_2, \cdots, S_{T-1}, A_{T-1}, R_T$ under behaviour policy b
    $G \leftarrow 0$
    $W \leftarrow 1$
    for $t = T-1$ to 0 **do**
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}\left[G - Q(S_t, A_t)\right]$
        $W \leftarrow W\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
    end for
end while

we now the algorithm to evaluate a state value function under a target policy π and any other behaviour policy b. To improve the policy π, we follow the general policy improvement approach by acting greedy.

$$\pi'(s) = \arg\max_{a \in A(s)} Q_\pi(s,a)$$

such that $\pi'(s) \geq \pi(s)$ for all $s \in S$.

(Off policy Every Visit Monte Carlo Control Algorithm)

Algorithm 10 Off policy Every Visit Monte Carlo Control Algorithm

Initialize $Q(s,a)$ arbitrarily for all $s \in S$ and $a \in A(s)$
Initialize $C(s,a)$ with zeros for all $s \in S$ and $a \in A(s)$
Input an arbitrary soft behaviour policy b ▷ e.g. ϵ-soft
$\pi \leftarrow argmax_a Q(s,a)$
while each episode **do**
    Generate an episode $S_0, A_0, R_1, S_1, A_1, R_2, \cdots, S_{T-1}, A_{T-1}, R_T$ under behaviour policy b
    $G \leftarrow 0$
    $W \leftarrow 1$
    for $t = T-1$ to 0 **do**
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $\pi'(s) \leftarrow \arg\max_{a \in A(s)} Q_\pi(s,a)$
        $\pi \leftarrow \pi'$
        if $A_t \neq \pi(S_t)$ **then**
            Break
        end if
    end for
    $W \leftarrow W \frac{1}{b(A_t|S_t)}$
end while

Chapter 6

Temporal Difference Learning

Temporal Difference learning is a combination of Monte Carlo methods and dynamic Programming. It is a model-free control. You don't bootstrap in Temporal Difference learning unlike monte carlo. Temporal Difference methods depends on other state estimates. The simple change in the update equation of the value function is the only difference between Monte Carlo and Temporal Difference methods.

Monte Carlo Update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t)\right]$$

Temporal Difference Update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

In TD learning methods, you don't need to wait for the end of the episode to update the value function. You can update the value function at every time step. You need the next state and next reward to update the value function. This is the main difference between TD learning and Monte Carlo methods.

We follow the same procedure as we did in earlier chapters. We will start with the prediction or evaluation problem. Consider a policy π to be evaluate and develop and algorithm to find the value function or state-action value function under the given policy. TD learning bootstraps from its earlier estimates.

TD(0) Algorithm

One can change the algorithm for estimating the State-Action value function. The only difference is that we need to use the action value function instead of the state value function. Here we call the update $\delta_t \coloneqq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is call the *TD-error*. TD learning methods also converge to value function under the policy π. In TD learning methods, the error is compared with earlier estimate rather the cumulative reward as in monte carlo methods. TD learning methods are computationally advantageous over monte Carlo methods. TD(0) is a special case of TD(λ) where $\lambda = 0$.

Although both monte Carlo methods and TD methods converge, TD methods converge faster than Monte Carlo methods. However, there is no formal proof to show that TD methods converge faster and converge, but from the empirical studies, it is observed that TD methods converge faster than Monte Carlo methods.

TD(0) batch update algorithm

Algorithm 11 TD(0) Algorithm

Input: Initialize a policy π to be evaluated, step-size parameter α, discount factor γ
Initialize: $V(S)$ arbitrarily for all $S \in S$ except for $V(terminal) = 0$
while for each episode **do**
    $S \leftarrow$ initial state S_0
    for for each step of episode **do**
        Choose A from S using policy derived from V
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha \left[R + \gamma V(S') - V(S)\right]$
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
    end for
end while

Algorithm 12 TD(0) Batch Update Algorithm

Input: Initialize a policy π to be evaluated, step-size parameter α, discount factor γ
Initialize: $V(S)$ arbitrarily for all $S \in S$ except for $V(terminal) = 0$
while each episode **do**
    $S \leftarrow$ initial state S_0
    for for each step of episode **do**
        Choose A_t from S_t using policy π derived from V
        Take action A, observe R_{t+1}, S_{t+1}
        Store the transition $(S_t, A_t, R_{t+1}, S_{t+1})$ in a memory
        Store the update $\delta_t \coloneqq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ in a memory
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
        $V(S) \leftarrow V(S) + \frac{\alpha}{|T|} \sum_0^{T-1} \delta_t$
    end for
end while

A state-action value evaluate has the same algorithm as the state value function. The only difference is that we need to use the action value function instead of the state value function.

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

with the above change in the $TD(0)$ algorithm is called *on-policy SARSA* algorithm.

SARSA on-policy TD algorithm

Algorithm 13 SARSA on-policy TD algorithm

Input: Initialize a state-action value funcciton $Q(S, A)$ arbitrarily, step-size parameter α, discount factor γ
Input: Initialize a policy π to be evaluated from Q.
while for each episode **do**
    $S \leftarrow$ initial state S_0
    Choose A from S using policy derived from Q
    for for each step of episode **do**
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$
        $S \leftarrow S'$
        $A \leftarrow A'$
        if S is terminal **then**
            break
        end if
    end for
end while

we have seen the policy evaluation algorithms using TD methods. Now, its time to derive the TD control algorithms to update the policy to acheivve the optimal policy

off policy TD control

Algorithm 14 Off-policy TD control or Q-learning algorithm

Input: Initialize a state-action value funcciton $Q(S, A)$ arbitrarily, step-size parameter α, discount factor γ
Input: Initialize a policy π to be evaluated from Q.
while for each episode **do**
    $S \leftarrow$ initial state S_0
    Choose A from S using policy derived from Q ▷ using ϵ-greedy policy
    for for each step of episode **do**
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma max_a Q(S', a) - Q(S, A)\right]$
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
    end for
end while

with the update step,

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

state-action value function directly approximates the optimal state-action value function Q^*. As the algorithm directly learns optimal Q, this algorithm is popularly called as *Q-learning* algorithm.

Based on the update equation, there are many variants of *Q-learning* or *SARSA* algorithm. One of them is *Expected SARSA*. The update equation for Expected SARSA is given below.

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

Expected SARSA

Algorithm 15 Expected SARSA algorithm

Input: Initialize a state-action value funcciton $Q(S, A)$ arbitrarily, step-size parameter α, discount factor γ
Input: Initialize a policy π to be evaluated from Q.
while for each episode **do**
    $S \leftarrow$ initial state S_0
    Choose A from S using policy derived from Q ▷ using ϵ-greedy policy
    for for each step of episode **do**
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \sum_a \pi(a|S') Q(S', a) - Q(S, A)\right]$
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
    end for
end while

Both Q-learning and Expected SARSA are off-policy algorithms. In Q-learning the target policy is greedy whereas the behaviour policy is ϵ-greedy. In Expected SARSA, behaviour policy is ϵ-greedy but the target policy is the actual stochastic policy.

Maximization bias

In Q-learning or SARSA, we update using choosing the maxmimum action value over the states. The true values of Q could be zero for most of the states. In this case, the algorithm will never learn the true values of Q as it will always choose the maximum action value. This is called *maximization bias*.

A *Double Q-learning* algorithm is proposed to overcome this problem. In Double Q-learning, we use two action value functions Q_1 and Q_2 and update them alternatively. The update equation is given below.

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, argmax_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)\right]$$

or

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_1(S_{t+1}, argmax_a Q_2(S_{t+1}, a)) - Q_2(S_t, A_t)\right]$$

During any iteration, we update either Q_1 or Q_2 and final policy is derived by using the sum of Q_1 and Q_2 either greedily or ϵ-greedily.

Double Q-learning

Algorithm 16 Double Q-learning algorithm

Input: Initialize a state-action value funcciton $Q_1(S, A)$ and $Q_2(S, A)$ arbitrarily, step-size parameter α, discount factor γ
Input: Initialize a policy π to be evaluated from Q_1 and Q_2.
while for each episode **do**
    $S \leftarrow$ initial state S_0
    Choose A from S using policy derived from Q_1 and Q_2 ▷ using ϵ-greedy policy
    for for each step of episode **do**
        Take action A, observe R, S'
        Take a unbiased coin toss
        if Heads **then**
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left[R + \gamma Q_2(S', argmax_a Q_1(S', a)) - Q_1(S, A)\right]$
        else
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left[R + \gamma Q_1(S', argmax_a Q_2(S', a)) - Q_2(S, A)\right]$
        end if
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
    end for
end while

How does this elimimate maximization bias? Here, one Q table provides the optimal action and other Q table provides the optimal action value. So, the algorithm will not be biased towards choosing the maximum action value. Thus, it elimimates the maximization bias.

Chapter 7

n-Step Bootstrapping

We presented Monte Carlo and TD learning methods till now. To solve RL problems, there is no one single solution that works for every problem. You need an algorithm that sits between Monte Carlo and TD Learning. N-Step Bootstrapping is the exactly that. Unlike in TD learning, you sample upto n steps and record the rewards sum and then bootstrap using the approximate value function.

As we did in the earlier chapters, we proceed to solve the prediction problem. we will evaluate a given policy π using the n-step TD algorithm. **n-step TD algorithm**

Algorithm 17 N-Step TD Algorithm

Input: π - policy, α - step size, n - number of steps
Initialize: $V(s)$ arbitrarily for all $s \in S$
Initialise a buffer which can store upto n states and rewards
while for each episode **do**
    Initialise $S_0 \neq terminal$
    for $t = 0$ to $T - 1$ **do**
        Take action $A_t \sim \pi(A_t|S_t)$
        Observe R_{t+1} and S_{t+1}
        Store $S_t, A_t, R_{t+1}, S_{t+1}$ in the buffer
        if $t \geq n - 1$ **then**
            $\tau \leftarrow t - (n - 1)$
            $G_\tau = R_{\tau+1} + \gamma R_{\tau+2} + \gamma^2 R_{\tau+3} + \cdots + \gamma^{n-1} R_{\tau+n} + \gamma^n V(S_{\tau+n})$
            $V(S_\tau) \leftarrow V(S_\tau) + \alpha(G_\tau - V(S_\tau))$
        end if
    end for
end while

Theoritically, n-step returns will better estimate the expection of discounted sum of rewards. n-step TD methods also converge. Possibly, better that TD(0) and MC methods. We have a solution for the prediction problem using n-step TD. We can also solve the control problem using n-step TD. Recall the *SARSA* algorithm in the previous chapter. SARSA is an on-policy TD control algorithm. We can modify SARSA to use n-step returns. The algorithm is given below.

Algorithm 18 N-Step SARSA Algorithm

Initialise $Q(s, a)$ arbitrarily for all $s \in S, a \in A(s)$
Initialise a buffer which can store upto n states, actions and rewards
Initialize policy π to be $\epsilon - greedy$ with respect to Q
while for each episode **do**
    Initialise $S_0 \neq terminal$
    for $t = 0$ to $T - 1$ **do**
        Take action $A_t \sim \pi(A_t|S_t)$
        Observe R_{t+1} and S_{t+1}
        Store $S_t, A_t, R_{t+1}, S_{t+1}$ in the buffer
        if $t \geq n - 1$ **then**
            $\tau \leftarrow t - (n - 1)$
            $G_\tau = R_{\tau+1} + \gamma R_{\tau+2} + \gamma^2 R_{\tau+3} + \cdots + \gamma^{n-1} R_{\tau+n} + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$
            $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha(G_\tau - Q(S_\tau, A_\tau))$
        end if
    end for
end while

N-step *SARSA* algorithm converge given that all other states are suffiently explored. The algorithm is also on-policy. The algorithm is also called *SARSA(n)*.

Off-policy version of n-step SARSA is also possible using the importance sampling technique. when using the importance sampling technique, the behaviour policy is different from the target policy. we learn the target policy while sampling from the behaviour policy. You are not estimating the expected sum of discounted rewards but the expected sum of discounted rewards weighted by the importance sampling ratio. Such change is reflected in the update rule.

Off-policy n-step SARSA algorithm

Algorithm 19 Off-policy N-Step SARSA Algorithm

Initialise $Q(s, a)$ arbitrarily for all $s \in S, a \in A(s)$
Initialise a buffer which can store upto n states, actions and rewards
Initialize policy π to be $\epsilon - greedy$ with respect to Q
Initialize behaviour policy b arbitrarily
while for each episode **do**
    Initialise $S_0 \neq terminal$
    for $t = 0$ to $T - 1$ **do**
        Take action $A_t \sim b(A_t|S_t)$
        Observe R_{t+1} and S_{t+1}
        Store $S_t, A_t, R_{t+1}, S_{t+1}$ in the buffer
        if $t \geq n - 1$ **then**
            $\tau \leftarrow t - (n - 1)$
            $\rho \leftarrow \prod_{i=\tau+1}^{\tau+n} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$
            $G_\tau = R_{\tau+1} + \gamma R_{\tau+2} + \gamma^2 R_{\tau+3} + \cdots + \gamma^{n-1} R_{\tau+n} + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$
            $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho(G_\tau - Q(S_\tau, A_\tau))$
        end if
    end for
end while

Given the idea of *SARSA(n)*, we can also define $expected-SARSA(n)$ of both off-policy and on-policy versions. However, we can have a off-policy version of $SARSA(n)$ without using importance sampling. What is off-policy? The behaviour policy is different from the target policy. We learn the target policy while sampling from the behaviour policy. Earlier, we use the behaviour policy to sample the action, instead take the expected vaue of target policy. This algorithm is called *n-step Tree Backup* algorithm.

Algorithm 20 Off-policy N-Step Tree Backup Algorithm

Initialise $Q(s,a)$ arbitrarily for all $s \in S, a \in A(s)$
Initialise a buffer which can store upto n states, actions and rewards
Initialize policy π to be $\epsilon - greedy$ with respect to Q
while for each episode **do**
    Initialise $S_0 \neq terminal$
    for $t = 0$ to $T - 1$ **do**
        Take action $A_t \sim \pi(A_t|S_t)$
        Observe R_{t+1} and S_{t+1}
        Store $S_t, A_t, R_{t+1}, S_{t+1}$ in the buffer
        if $t \geq n - 1$ **then**
            $\tau \leftarrow t - (n-1)$
            $G \leftarrow R_{\tau+1} + \gamma \sum_a \pi(a|S_{\tau+1})Q(S_{\tau+1}, a)$
            for $i = t$ to τ **do**
                $G \leftarrow R_i + \gamma G \sum_{a \neq A_i} \pi(a|S_{\tau+1})Q(S_i, a) + \gamma\pi(A_i, S_i)G$
            end for
            $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha(G - Q(S_\tau, A_\tau))$
        end if
    end for
end while