Reinforcement Learning

Shuhei Watanabe

March 31, 2021

1 Introduction

Definition 1

1. **Time steps**: $t \in \mathbb{N}$
2. **States**: $s \in \mathcal{S}$ *where* $\mathcal{S}$ *is a finite set of states*
3. **Actions**: $a \in \mathcal{A}$ *where* $\mathcal{A}$ *is a finite set of actions*
4. **Rewards**: $r \in \mathcal{R}$ *where* $\mathcal{R}$ *is a finite set of scalar rewards*
5. **Policies**: *For deterministic policies* $\pi(s) = a$ *and for stochastic policies* $\pi(a|s) = p(a|s)$
6. **Value function**: $v_\pi(s_t) = \mathbb{E}_\pi[G_t|s_t]$ *measures the value of policy with a given state*
7. **Action-value function**: $q_\pi(s_t, a_t) = \mathbb{E}_\pi[G_t|s_t, a_t]$ *measures the value of policy given a pair of state and action*

In Reinforcement learning (RL), the goal is to maximize the expected value of cumulative sum of a received reward. This is supported by **Reward Hypothesis**. Plus, we often introduce **discounted rewards** to reduce the future rewards which are more uncertain [1]. The difference is the following:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots = \sum_{k=0}^{\infty} r_{t+k+1}$$
$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $\gamma \in [0, 1]$ is called discount rate. The main interest in RL is to find **policies** which balance so-called **the exploration-exploitation trade-off** [2] and maximize the cumulative rewards. Additionally, there is a connection between value function and action-value function as follows:

$$\begin{aligned} \text{Stochastic}: \ & v_\pi(s) = \mathbb{E}_\pi[q_\pi(s, a)] \\ \text{Deterministic}: \ & v_\pi(s) = q_\pi(s, \pi(s)) \end{aligned}$$

For the computation scheme of value and action-value function, the following **Bellman equation*** is used:

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

The optimality of policies are defined as follows:

[1] This is especially required when the time limit is infinity and we do not need it if the task is episodic.

[2] Exploration is to take the actions, which agents do not have much experience. On the other hand, exploitation is to use previous knowledge to choose better options.

Definition 2
A policy $\pi_\star$ is optimal iff:

$$\forall \pi, v_{\pi_\star}(s) \geq v_\pi(s)$$

The policy $\pi_\star$ satisfies the following equations:

$$v_{\pi_\star}(s) = \max_a \sum_{s',r} (r + \gamma v_{\pi_\star}(s')) p(s', r|s, a)$$
$$q_{\pi_\star}(s, a) = \sum_{s',r} (r + \gamma \max_{a'} q_{\pi_\star}(s', a')) p(s', r|s, a)$$

2 Markov decision process (MDP)

In MDP, we compute the transition probability $p(s', r|s, a)$ ($P : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$) of the next state and the corresoponding reward with given a current state s and an action a. To compute the transition probability, we assume the following **Markov Property**:

Assumption 1
A state-reward pair (s', r) has the Markov property iff:

$$p(s', r|s_t, a_t) = p(s', r|s_t, a_t, \cdots, s_0, a_0)$$

In other words, the transition probability p only depends on the current state and action, but not on more previous data. One concrete example is the following:

- Imagine a house cleaning robot. It can have three charge levels: high, low and none. At every point in time, the robot can decide to recharge or to explore unless it has no battery. When exploring, the charge level can reduce with probability ρ. Exploring is preferable to recharging, however it has to avoid running out of battery.

The formulation of this problem is as follows:

$$\mathcal{S} = \{\text{high, low, none}\}$$
$$\mathcal{A} = \{\text{explore, recharge}\}$$
$$\mathcal{R} = \{+1, -1, -100\}$$

The transition probability is as follows:

$$p(s', r|s, a) = \begin{cases} 1 \ (\{\text{high}, -1, \text{low}, \text{recharge}\}, \{\text{high}, -1, \text{high}, \text{recharge}\}, \{\text{none}, 0, \text{none}, \cdot\}) \\ \rho \ (\{\text{low}, +1, \text{high}, \text{explore}\}, \{\text{none}, -100, \text{low}, \text{explore}\}) \\ 1 - \rho \ (\{\text{high}, +1, \text{high}, \text{explore}\}, \{\text{low}, +1, \text{low}, \text{explore}\}) \end{cases}$$

MDP assumes that it can fully understand the current state of the environment. However, in the real setting, it is not necessarily true. Therefore, partially observable MDP, which is one variant of MDP introduces the observations $\omega \in \Omega$ and the probability $p(s|\omega)$ of state under a given observation.

3 Multi-armed Bandits (MAB)

3.1 Preliminaries

The problem setting of MAB is the situation where you have several options (e.g. K slot machines). Ideally, we would like to keep on choosing the best option. The reward distribution with a given option (or action) is described as $p(r|a)$. Note that general MAB **does not have the concept of state**. Tha main goal of MAB is also to maximize the cumulative reward. Since there is at least one ideal option among the options, we rewrite the objective function to the following:

$$\mathbb{E}\biggl[\sum_{t=1}^{T}(q(a_\star) - q(a_t))\biggr]$$

This metric is called **total regret** and the minimization of the total regret leads to the maximization of the cumulative rewards.

3.2 ϵ–greedy

This method literally takes the best option greedily with a certain probability $1 - \epsilon$. To obtain the best option, we have to estimate the action-value function as follows:

$$q_t(a) = \frac{1}{N_t(a)}\sum_{t=1}^{T} r_t \mathbb{1}(a_t = a)$$

where $N_t(a)$ is the number of times a was taken in t time steps and α is called step size or learning rate. For more aggressive knowledge exploitation, we can decay ϵ as $\epsilon_{t+1} = (1 - t/T)\epsilon_{\text{init}}$. Using this cumulative $q_t(a)$, the current best option can be computed as $a_t = \text{argmax}_a q_t(a)$. Practically, $q_t(a)$ can be computed in the following manner:

$$\begin{aligned} q_{n+1} &= \frac{(n-1)q_n + r_n}{n} = q_n + \frac{r_n - q_n}{n} \\ q_{n+1} &= q_n + \alpha(r_n - q_n) \end{aligned} \tag{1}$$

where $q_n = q_t(a), r_n = r_t$ s.t. $N_t(a) = n - 1, N_{t-1}(a) = n - 2$. When we use Eq. (1), we can update $q_t(a)$ by the time complexity of $O(1)$. The first equation is the correct way to compute $q_t(a)$. However, in practice, especially when the probability $p(r|a)$ dynamically changes, the second equation works well, because when n goes to inifinity, $1/n$ vanishes in the former equation and not in the latter. That is why the latter formulation can exploit more recent observations and it handles real-world problems better. Note that if we set q_{init} higher, it promotes the exploration and it is effective for non-stationary problems [3]. However, it is unstable in the beginning, because

1. $q_t(a)$ reduces when the action a is chosen and the least change is the best action $a_\star$
2. The best action $a_\star$ is likely to be chosen and this leads to the decrease of regret
3. However, this also leads to the decrease of $q_t(a_\star)$
4. Therefore, another action will be taken until $q_t(a)$ becomes suffciently low compared to $q_t(a_\star)$ and the regret is unstable in the beginning.

3.3 Softmax action selection

Instead of choosing as ϵ–greedy does, we can pick actions proportional to their value. This is called **softmax action selection** and modeled as follows:

$$p(a_t = a) = \frac{\exp(q_t(a)/\tau)}{\sum_{a' \in \mathcal{A}} \exp(q_t(a')/\tau)}$$

where τ is a control parameter. As τ grows, it approaches random selection. On the other hand, as τ becomes smaller, it becomes more aggressive sampling. The advantage of this method is the robustness in the parallel computational situation and the disadvantage is that this model is not objective [4].

[3]The problems which do not converge.

[4]Thompson sampling is not mentioned in the class. However, Thompson sampling can handle in a more objective manner.

Algorithm 1 ϵ-greedy algorithm

```
   $\epsilon, q_{\text{init}}$                    ▷ a control parameter and the initial value of $q_t(a)$
1: function $\epsilon$-GREEDY
2:     Initialization: $q_1(a) = q_{\text{init}}$ for all $a$
3:     for $t = 1, \cdots, T$ do
4:         $r$ = random-number(0, 1)
5:         if $r \leq \epsilon$ then
6:             Choose one option randomly
7:         else
8:             Choose the best option $a_t = \text{argmax}_a q_t(a)$
9:         Update $q_t(a)$ as in Eq (1)
```

3.4 UCB

In ϵ–greedy, the best option will be taken based on **previous experience**. However, some observations are probably underestimated and we should take this possibility into account. One solution is to use Upper Confidence Bound (UCB) of $q_t(a)$. Since this method uses UCB, it is called **the optimism in the face of uncertainty principle**. The queried action will be decided based on the following score [5]:

$$a_t = \text{argmax}_a \left(\underbrace{q_t(a)}_{\text{Greedy Term}} + c \underbrace{\sqrt{\frac{\log\ t}{N_t(a)}}}_{\text{Optimism Term}} \right)$$

Basically, if a given option is not taken so much, this score promotes to take the option optimistically.

3.5 Contextual Bandits

Bandit strategies for solving the exploration-exploitation issue for RL are suboptimal, because they do not take the state into account and they ignore the sequence of actions to be made. To address the issue, the contextual bandits has been introduced. The contextual bandits has the concept of state s. In this problem, there are unknown distribution $p(s)$ (context distribution) and $p(r|s, a)$. The difference is with or without s. In each time step t, the environment generates s_t (e.g. User preferences) and the agent selects action a_t. If actions are allowed to affect the next state s_{t+1} as well as the immediate reward, then this is a full reinforcement learning problem.

4 Policy iteration and value iteration

Policy and value iteration find optimal policies $\pi_\star$ based on the Bellman equation. Since the optimal solution can be decomposed into subproblems, dynamic programming, i.e. tabular format policies, can be applied. Each iteration is composed as follows:

1. **Policy iteration**: Interleave policy evaluation and improvement
2. **Value iteration**: Apply the Bellman optimality equation iteratively

Note that since it is typical to have the curse of dimensionality, we often substitute it with sample-based methods mentioned later.

[5] This formulation can be derived easily if we know the definitions of consistency of policies and Höffding's Inequality. However, I will skip this, because the professor did not talk about them.

Algorithm 2 Policy Iteration (the case of a deterministic policy)

```
    $\theta > 0$                                    ▷ The parameter controling accuracy of estimation
 1: function POLICY ITERATION
 2:     while true do
 3:         while $\Delta \geq \theta$ do           ▷ Policy Evaluation (evaluate a value function for given $\pi$)
 4:             $\Delta = 0$
 5:             for $s \in \mathcal{S}$ do
 6:                 $v = v_\pi(s)$
 7:                 $v_\pi(s) = \sum_{s',r}(r + \gamma v_\pi(s'))p(s', r|s, \pi(s))$      ▷ Bellman expected equation[8]
 8:                 $\Delta = \max(\Delta, |v - v_\pi(s)|)$
 9:         same policy = true
10:         for $s \in \mathcal{S}$ do       ▷ Policy Improvement (improve a policy for given value function $V$)
11:             $a = \pi(s)$
12:             $\pi(s) = \mathrm{argmax}_{a'} \sum_{s',r}(r + \gamma v_\pi(s'))p(s', r|s, a')$      ▷ Bellman optimality equation
13:             if $a \neq \pi(s)$ then
14:                 same policy = false
15:         if same policy then
16:             return $\pi \simeq \pi_\star$
```

4.1 Policy iteration

Policy iteration is the algorithm, which combines **policy evaluation** and **policy improvement**. Note that evaluation step is called **prediction** and improvement step is called **control**. In other words, the prediction is the prediction of the value function and the control is the control of actions or a policy or to update the action-value function. Note that **prediction requires either a model or a policy** and **control does not require either of them**. This algorithm relies on the following theorem:

Theorem 1
Let π and π' be any pair of deterministic policies. If, $\forall s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s),$$

then the policy π' must be as good as, or better than, π, i.e. $\forall s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s)$$

Based on this theorem, we can compare policies and improve policies iteratively. In the algorithm, we first evaluate **the value function for each state** given a policy π [6]. Then, we check if the new policy is successfully improved [7]. The procedure will continue until we get the policy $\pi \simeq \pi_\star$.

4.2 Value iteration

Since policy iteration requires policy evaluations at each improvement step, it often takes substantial amount of time. In fact, this iterative evaluation is not necessarily required [9]. Therefore, value iteration

[6]This can be represented as linear system and solved by the inverse matrix.

[7]We check the optimality in the greedy manner.

[8]Only the first iteration is random policy and we take greedy policy from the second iteration in the case of a deterministic policy.

[9]Because the optimal value is unique, but the optimal policy is not unique. For example, when $\forall s, s' \in \mathcal{S}, v_\star(s) = v_\star(s')$ holds, an arbitrary policy is the optimal policy.

Algorithm 3 Value Iteration

```
   $\theta > 0$                                        ▷ The parameter controling accuracy of estimation
1: function VALUE ITERATION
2:     while $\Delta \geq \theta$ do
3:         $\Delta = 0$
4:         for $s \in \mathcal{S}$ do
5:             $v = v_\pi(s)$
6:             $v_\pi(s) = \max_a \sum_{s',r}(r + \gamma v_\pi(s'))p(s', r|s, a)$      ▷ Bellman optimality equation
7:             $\Delta = \max(\Delta, |v - v_\pi(s)|)$
8:     return $\pi$ s.t. $\pi(s) = \mathrm{argmax}_a \sum_{s',r}(r + \gamma v_\pi(s'))p(s', r|s, a)$      ▷ deterministic policy
```

omits this evaluation and evaluates the value function in the greedy manner once. Then, it returns the deterministic argmax policy for each state. The complexity of each iteration is $O(|\mathcal{S}|^2|\mathcal{A}|)$.

5 Monte-Carlo reinforcement learning

Monte-Carlo (MC) based learning is **model-free**, i.e. it does not require an MDP model and observes the the transitions directly from the real environment ; therefore **lower-biased** method. On the other hand, since MC yields high variance, MC usually requires more samples to converge. In this section, we handle the following:

1. **MC prediction**: Take the average of returns over each generated episode
2. **On-policy MC control**: Control a given policy using the greedy policy on action value function
3. **Off-policy MC control**: Control a given policy using a policy called **behavior policy** and the policy of interest called **target policy** [10].

Note that on-policy methods **use the same policy for both update of value function and generating episodes** while off-policy **separates the policies**. In the case of MC methods, on-policy methods use the current target policy for both and off-policy methods use the current target policy for update and behavior policy for generating episodes. Since the same policy yields similar experience, off-policy methods can reuse experience generated from old policies and thus they are more **sample-efficient** [11]. On the other hand, off-policy methods is less stable. Recall that episodes are generated as follows:

1. Set an initial state
2. Decide the next action given the current state and a policy
3. Observe the next state and reward directly from the real environment
4. Repeat the procedure until we reach the termination or run out the budget

5.1 General setting of MC

We use MC sampling when we cannot examine all possible pairs of a state and an action. Since each episode, i.e. a sequence of pairs of a state, an action and a reward $\{\{(s_t, a_t, r_t)\}_{t=0}^{T_i}\}_{i=1}^{N}$, is generated from the environment directly, we do not need any MDP models which are required in value or policy

[10] Learning its value function is the target of the learning process, so we call it target policy. Target policy is typically deterministic policy while behavior policy is stochastic.

[11] In other words, the training convergence requires less samples.

iteration. Note that an agent learns value function from the average of sampled complete episodes [12] and since it uses the actual data and MC can cover only a fraction of choices [13], the training result is likely to have low bias and high variance. The formal formulation of the MC method is the following:

Definition 3
Given a policy π, the objective is to estimate the value function

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

where $G_t = \sum_{\tau=1}^{T-t} \gamma^{\tau-1} r_{t+\tau}$. The MC method replaces the value function with the empirical value function as follows:

$$v_\pi(s) = \mathbb{E}_{\{\{(s_t,a_t,r_t)\}_{t=0}^{T_i}\}_{i=1}^{N} \sim \pi} G_{i,t}$$

Note that the update of the value function $v_\pi(s)$ can be performed by $O(1)$ using the following:

$$v_\pi(s_t) = v_\pi(s_t) + \frac{1}{N(s_t)}(G_{i,t} - v_\pi(s_t))$$

where $N(s_t)$ is the state-visitation counter. Instead of $N(s_t)$, we can also use the fixed **step size** or **learning rate** α to deal with the infinite episodes or the non-stationary problems [14] as follows:

$$v_\pi(s_t) = v_\pi(s_t) + \alpha(G_{i,t} - v_\pi(s_t))$$

Note that when we use a fixed learning rate, we do not yield the convergence.

5.2 Monte-Carlo prediction

The First-visit MC prediction is shown in Algorithm 4. There are two ways to update value function in MC prediction:

1. **First-visit MC**: Estimate the value function $v_\pi(s)$ as the average of the returns at **the first visits** to s in each episode.
2. **Every-visit MC**: Estimate the value function $v_\pi(s)$ as the average of the returns at **all the visits** to s in each episode.

The former ignores trajectories from s except the first visit in each episode.

5.3 Monte-Carlo control

5.3.1 On-policy MC control

When we perform the policy iteration using value function, we need an MDP model $p(s', r|s, a)$. However, we do not have the model in MC methods, so we generate transitions from a pair of a state and an action using **action-value function** $q_\pi(s, a)$ instead of an MDP model. The policy improvement is guaranteed for any ϵ-greedy policy when we set the next policy to the ϵ-greedy policy with respect to the current action-value function. The proof is given below.

[12] We do not need bootstrapping for MC methods. In this context, bootstrapping is to approximate the target value functions based on existing value functions.
[13] The number of possible trajectories from one state is massive and that is why depending on the trajectories they take, the performance changes significantly.
[14] This is the case for real-world problems. The environments keep changing.

Algorithm 4 First-visit MC prediction

```
    $\pi$                                                        ▷ A policy to be evaluated
 1: function FIRST VISIT MC PREDICTION
 2:     Initialize $v_\pi(s) \in \mathbb{R}$
 3:     while The budget is left do
 4:         $G = 0$, Generate an episode following $\pi : \{(s_t, a_t, r_t)\}_{t=0}^{T}$
 5:         visit[$s$] = # of visits to $s$ in the episode for all $s \in \mathcal{S}$
 6:         for $t = T-1, T-2, \cdots, 0$ do
 7:             $G = \gamma G + r_{t+1}, \text{visit}[s_t] = \text{visit}[s_t] - 1$
 8:             if visit[$s_t$] = 0 then        ▷ In every-visit MC, we do not need this condition.
 9:                 $v_\pi(s_t) = v_\pi(s_t) + \frac{1}{N(s_t)}(G - v_\pi(s_t))$
10:                 visit[$s_t$] = true
```

[Proof]
First, the next ϵ-greedy policy is as follows:

$$\pi'(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + 1 - \epsilon \text{ (if } a = \mathrm{argmax}_{a' \in \mathcal{A}}\, q_\pi(s, a')) \\ \frac{\epsilon}{|\mathcal{A}|} \text{ (Otherwise)} \end{cases}$$

Under this policy, the policy improvement guarantees the improvement.

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_{a \in \mathcal{A}} \pi'(a|s) q_\pi(s, a) \\ &= \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \max_{a \in \mathcal{A}}\; q_\pi(s, a) \\ &\geq \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \sum_{a \in \mathcal{A}} \underbrace{\frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}|}}{1 - \epsilon}}_{1 \text{ or } 0} q_\pi(s, a) \\ &= \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) - \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) + \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) \\ &= v_\pi(s) \end{aligned}$$

where the right and left hand sides are equal if and only if $\pi(a|s) = \pi'(a|s)$. It implies the new policy π' improves from the previous policy π based on Theorem 1.

5.3.2 Off-policy MC control

In on-policy MC control, the action value function is learned from episodes generated based on the target policy $\pi(a|s)$. On the other hand, off-policy MC control learns the **target policy** π from episodes generated from the **behavior policy** b. Since the behavior policy is different from the target policy and can be stochastic, off-policy methods can **promote more exploration**. This policy b can be an arbitrary policy such as a random policy or a greedy policy. One example of the off-policy methods using MC is **importance sampling**. The main idea of the importance sampling is to transfer the knowledge obtained from the behavior policy b into the target policy π. The probability of subsequent trajectory $\{(s_\tau, a_\tau)\}_{\tau=t}^{T}$ under a given policy π and a state s_t is the following:

$$p(a_t, s_{t+1}, a_{t+1}, \cdots, s_T | s_t, \pi) = \prod_{\tau=t}^{T-1} \pi(a_\tau | s_\tau) p(s_{\tau+1} | s_\tau, a_\tau)$$

The importance relative sampling ratio is defined using the probabilities above as follows:

Algorithm 5 On-policy first-visit MC control

```
    $\epsilon > 0$                                                         ▷ Hyperparameter
 1: function ON-POLICY FIRST VISIT MC CONTROL
 2:     Initialize $q_\pi(s, a) \in \mathbb{R}$
 3:     while The budget is left do
 4:         $G = 0$, Generate an episode following $\pi : \{(s_t, a_t, r_t)\}_{t=0}^{T}$
 5:         visit$[s, a]$ = # of visits to $(s, a)$ in the episode for all $(s, a) \in \mathcal{S} \times \mathcal{A}$
 6:         for $t = T-1, T-2, \cdots, 0$ do
 7:             $G = \gamma G + r_{t+1}$, visit$[s_t, a_t]$ = visit$[s_t, a_t] - 1$
 8:             if visit$[s_t, a_t] = 0$ then          ▷ In every-visit MC, we do not need this condition.
 9:                 $q_\pi(s_t, a_t) = q_\pi(s_t, a_t) + \frac{1}{N(s_t, a_t)}(G - q_\pi(s_t, a_t))$
10:                 $a^\star = \mathrm{argmax}_a q_\pi(s_t, a)$
11:                 visit$[s_t, a_t]$ = true
12:                 for all $a \in \mathcal{A}(s_t)$ do
13:
                        $\pi(a|s_t) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & (\text{if } a = a^\star) \\ \frac{\epsilon}{|\mathcal{A}|} & (\text{Otherwise}) \end{cases}$
```

Definition 4

$$\rho_{t:T-1} = \frac{\prod_{\tau=t}^{T-1} \pi(a_\tau|s_\tau) p(s_{\tau+1}|s_\tau, a_\tau)}{\prod_{\tau=t}^{T-1} b(a_\tau|s_\tau) p(s_{\tau+1}|s_\tau, a_\tau)} = \frac{\prod_{\tau=t}^{T-1} \pi(a_\tau|s_\tau)}{\prod_{\tau=t}^{T-1} b(a_\tau|s_\tau)}$$

Since we take the ratio, **the MDP terms are canceled out** and we do not need any models for this method as well. The goal is to estimate the expected return given a state and it can be transformed using the relative ratio as follows:

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t|s_t] = \mathbb{E}_b[\rho_{t:T-1} G_t|s_t]$$

Since it uses the probabilities from a different policy, it is likely to include both larger variance and larger bias and this is the major problem of this method. In practice, we use the following weighted average to reduce variance:

$$v_\pi(s) = \frac{\sum_{\tau \in \mathcal{T}(s)} \rho_{t:\tau} G_t}{\sum_{\tau \in \mathcal{T}(s)} \rho_{t:\tau}}$$

where $\mathcal{T}(s)$ is a set of the time steps when we visit the state s from t to $T-1$.

6 Temporanl difference learning

In MC methods, we use complete episodes for each update of value function. On the other hand, temporal difference (TD) methods use an immediate reward from the enviroment for each update.

6.1 Temporal difference prediction

First, let's take the update of value iteration in the temporal difference method:

$$v_\pi(s_t) = v_\pi(s_t) + \alpha\big(r_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)\big)$$

Algorithm 6 Off-policy MC control (importance sampling)

```
    $q(s,a) \in \mathbb{R}, W(s,a) = 0, \pi(s) = \mathrm{argmax}_a\ q_\pi(s,a)$          ▷ objective function
 1: function OFF-POLICY IMPORTANCE SAMPLING MC
 2:     while The budget is left do
 3:         $G = 0$, Generate an episode following the behavior policy $b : \{(s_t, a_t, r_t)\}_{t=0}^{T}$
 4:         $G = 0, \rho = 1$
 5:         for $t = T-1, T-2, \cdots, 0$ do
 6:             $G = \gamma G + r_{t+1}$
 7:             $W(s_t, a_t) = W(s_t, a_t) + \rho$
 8:             $q_\pi(s_t, a_t) = q_\pi(s_t, a_t) + \frac{\rho}{W(s_t, a_t)}(G - q_\pi(s_t, a_t))$          ▷ Averaging
 9:             $\pi(s_t) = \mathrm{argmax}_a\ q_\pi(s_t, a)$
10:             if $a_t \neq \pi(s_t)$ then break          ▷ Reduce bias by avoiding larger or smaller weights
11:             $\rho = \rho \frac{1}{b(a_t|s_t)}$          ▷ Since $\pi$ is a greedy policy, $\pi(\cdot|a) = 1$.
```

Table 1: The comparison of TD and MC.

-	Temporal difference (TD)	Monte carlo (MC)
Update	every step	at the end of each episode
Final outcome	not required[15]	required
Bias	higher	lower
Variance	lower	higher
Bootstrapping	Yes	No
Solution	maximum likelihood of observed MDP[16]	maximum likelihood of observations

On the other hand, the update in the Monte-Carlo methods is the following:

$$v_\pi(s_t) = v_\pi(s_t) + \alpha\big(G_t - v_\pi(s_t)\big)$$

where $G_t = \sum_{\tau=t+1}^{T} \gamma^{\tau-t-1} r_\tau$. As seen in the equations, the temporal difference only uses the next reward and the value function of the next state. That is why it is called **temporal** difference. Note that $\delta_t = r_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)$ is called **TD error** and $r_{t+1} + \gamma v_\pi(s_{t+1})$ is called **TD target**. Basically, the error in MC can be derived using TD error as follows:

$$\begin{aligned} G_t - v_\pi(s_t) &= \sum_{\tau=t+1}^{T} \gamma^{\tau-t-1} r_\tau - v_\pi(s_t) \\ &= \sum_{\tau=t+1}^{T} \gamma^{\tau-t-1} \delta_\tau \end{aligned}$$

As seen in the equation, the cumulated TD error is equivalent to the MC error. The differences are listed in Table 1. In some cases, TD rather than MC yields the correct value function as seen in Figure 1.

[16]It means it can deal with infinite horizon tasks

[16]If this Markov model were exactly correct, the estimate of the value function would be exactly correct. This is called **certainiy equivalence**.

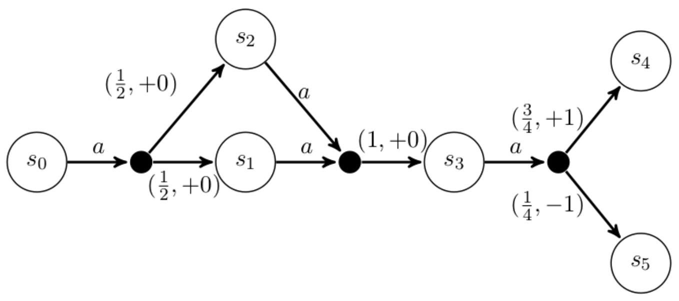

Figure 1: When there are only two kind of trajectoies $t_1 : s_0, s_1, s_3, s_4$ and $t_2 : s_0, s_2, s_3, s_5$ and we yield t_1 for $i \sim i + 2$ iteration and t_2 for $i + 3$ iteration for all $i = 0, 4, \cdots$. While MC yields biased result, TD yields the correct answer due to the effect of bootstrapping at s_3.

Algorithm 7 TD(0) prediction

```
   $\pi, \alpha \in (0, 1]$          ▷ The policy to be evaluated and hyperparameter
1: function TD(0) PREDICTION
2:     while The budget is left do
3:         Sample $s$
4:         while The episode is not finished do
5:             $a \sim \pi(a|s)$ and observe $r, s'$
6:             $v_\pi(s) = v_\pi(s) + \alpha\big(r + \gamma v_\pi(s') - v_\pi(s)\big), s = s'$
```

6.2 Temporal difference control

6.2.1 State-action-reward-state-action (SARSA)

At each iteration, given a state s_t, we take an action a_t chosen in the previous iteration. Then we observe the corresoponding reward r_{t+1} and the following state s_{t+1}. Finally, we choose a future action a_{t+1} based on the current action-value function $q_\pi(s, a)$ using soft policy (e.g. ϵ-greedy) [17]. The action-value function will be updated by the following:

$$q_\pi(s_t, a_t) = q_\pi(s_t, a_t) + \alpha\big(r_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1}) - q_\pi(s_t, a_t)\big)$$

Since both policy improvement and episode generation use the same policy, we call it **on-policy**.

6.2.2 Q-learning

At each iteration, given a state s_t, we take an action a_t based on $q_\pi(s, a)$ using soft policy (e.g. ϵ-greedy). Then we observe the corresoponding reward r_t and the following state s_{t+1}. The action-value function will be updated by the following:

$$q_\pi(s_t, a_t) = q_\pi(s_t, a_t) + \alpha\big(r_{t+1} + \gamma \max_a q_\pi(s_{t+1}, a) - q_\pi(s_t, a_t)\big)$$

Since target policy is greedy algorithm and bahavior policy is a soft policy, we call it **off-policy**.

6.2.3 Expected SARSA

This method takes the expected value of action-value function at each iteration.

$$q_\pi(s_t, a_t) = q_\pi(s_t, a_t) + \alpha\big(r_{t+1} + \gamma \mathbb{E}_{a \sim \pi(a|s_{t+1})}[q_\pi(s_{t+1}, a)] - q_\pi(s_t, a_t)\big)$$

[17] Soft policy is a policy which takes all possible actions with a certain probability.

Since we do not use an actual policy for target policy, it is called **off-policy**. If the target policy π is deterministic and greedy, it is equivalent to Q-learning.

6.2.4 Double Q-learning

In most control algorithms, value functions are created by the maximization of value function. Therefore, we keep taking the maximum of maximum [18] and this leads to the so-called **overestimation**. To alleviate this problem, double Q-learning has been introduced. This algorithm improves two Q-value functions and updates one of them with the probability of 0.5 and the other with the other 0.5 probability as follows:

$$
\begin{aligned}
q_1(s_t, a_t) &= q_1(s_t, a_t) + \alpha\biggl(r_{t+1} + \gamma q_2(s_{t+1}, \mathrm{argmax}_a q_1(s_{t+1}, a)) - q_1(s_t, a_t)\biggr) \\
q_2(s_t, a_t) &= q_2(s_t, a_t) + \alpha\biggl(r_{t+1} + \gamma q_1(s_{t+1}, \mathrm{argmax}_a q_2(s_{t+1}, a)) - q_2(s_t, a_t)\biggr)
\end{aligned}
$$

The major problems of Q-learning are to choose both the next action and action-value function greedily. Due to the operations, action-value function includes noise from the same policy and this noise is correlated to the policy. On the other hand, when we use a different policy to evaluate an action-value function from the other policy, since the noise cummulated in two action-value functions are decorrelated, it solves the overestimation issue. Furthermore, both action-value functions show the similar output in the end.

7 Model-based reinforcement learning and its combination

Up to now, the main focus is to approximate value functions. However, it is sometimes easier to estimate the underlying MDP model, e.g. solving maze, directly. Since model-based methods do not require the observation through the real environment and they can simulate using an MDP model, it is **more sample-efficient**. On the other hand, the learning can be biased depending on the accuracy of a given model, so the combination of both model-free and model-based methods called **Dyna** has been introduced. This method first learns from the real environment, then updates action-value function using experiences from previous observations, i.e. a model-based simulation. As another major example, we will discuss **Monte-Carlo tree search** (MCTS). This model constructs a tree containing previously visited pairs of a state and an action. It first chooses a promising path and then expands and randomly simulates an episode from the end node of the path. The action-value function is updated using this trajectory.

7.1 Model learning

First, the differences among various learnings are the following:

1. **Learning a model** $p(s', r|s, a)$: Data-efficient and can apply supervised learning, but hard to extract an optimal policy
2. **Learning a value function** $v_\pi(s), q_\pi(s, a)$: Less data-efficient, but easier to extract an optimal policy
3. **Learning a policy** π: Data-inefficient, but can estimate an optimal policy directly

Since **model-based RL** does not use the environment directly, we call it *indirect methods* and call **model-free RL** *direct methods*. Indirect methods introduce **additional bias** and it is **sometimes hard to model properly**, but they are **more data-efficient** in most cases. The scheme of model learning is the following:

[18]Since the target policy updates the action-value function by $a_{t+1} = \max_a q_\pi(s_{t+1}, \mathrm{argmax}_a q_\pi(s_{t+1}, a))$ and the existing action-value function are already biased, it is likely to cumulate additional noise.

Algorithm 8 Random-sample one-step tabular Q-planning

```
1: function Q Planning
2:     while The budget is left do
3:         $s \sim \mathcal{S}, a \sim \mathcal{A}$ at random
4:         $s', r \sim p(s', r|s, a)$
5:         $q_\pi(s,a) = q_\pi(s,a) + \alpha\Bigl(r + \gamma \max_{a'} q_\pi(s',a') - q_\pi(s,a)\Bigr)$
```

Algorithm 9 Tabular Dyna-Q

```
    Initialize $q_\pi(s,a)$ and model$(s,a)$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
1: function Dyna Q
2:     while The budget is left do
3:         pick $s$ from the enviroment                                ▷ Learning
4:         $a = \epsilon-$greedy$(s, q_\pi)$
5:         $s', r \sim p(s', r|s, a)$                 ▷ Sample from the real environment
6:         $q_\pi(s,a) = q_\pi(s,a) + \alpha\Bigl(r + \gamma \max_{a'} q_\pi(s',a') - q_\pi(s,a)\Bigr)$
7:         model$(s,a) = s', r$                ▷ In this case, the model is deterministic
8:         for $i = 1, \cdots, n$ do                                   ▷ Planning
9:             pick $(s,a)$ from previous observations randomly
10:            $s', r =$ model$(s,a)$
11:            $q_\pi(s,a) = q_\pi(s,a) + \alpha\Bigl(r + \gamma \max_{a'} q_\pi(s',a') - q_\pi(s,a)\Bigr)$
```

1. Generate simulated experiences [19] based on an estimated model
2. Compute value function by backup operations applied to simulated experience
3. Extract a policy (**planning**)

In state-space planning, planning is viewed primarily as a search of an optimal policy or a path to a goal through the state space. In other words, **planning** is to extract a policy using simulated experience generated by a model and **learning** is to extract a policy using real experiences from the environment. Note that the final policy will be the optimal policy in the given model, i.e. a suboptimal policy in the given enviroment. This is because models can be incorrect due to the limited number of samples or dynamic environment. Therefore, it is a good choice **to model the state difference** rather than the transition to a global state, because there can be a local similarities with respect to the state difference. Needless to say, models can be unreliable especially at the region where the agents do not explore much, called **distribution mismatch**.

7.2 Dyna

Dyna is a method combining the benefits of both direct methods, which are less biased, and indirect methods, which are more data-efficient. Roughly speaking, Dyna first learns from the real environment, then iterates a given indirect method to update action-value function using simulated experiences.

One problem of Dyna is distribution mismatch. To address the problem, **Dyna-Q+** has been introduced. In Dyna-Q+, it modifies the internal reward function as follows:

$$r + \kappa\sqrt{\tau}$$

[19] The next state and the corresponding reward or it can be a sequence of states and rewards. The next state and the corresponding reward can be given as a form of a probability distribution. We call such a model **distribution model** and we call a deterministic model **sample model**.

where κ is a coefficient to control the bonus and τ is the number of time steps in which a certain transition has not been visited. The second term promotes the exploration.

Another option is **prioritized sweeping**. This method puts a higher priority on the pair (s, a) with higher absolute temporal difference $r + \gamma \max_{a'} q_\pi(s', a') - q_\pi(s, a)$. The major change in the algorithm is to choose a pair of (s, a) with higher absolute TD error as a priority in the line 9 in Algorithm 9.

7.3 Simulation-based search (Monte-Carlo tree search)

Simulation-based search is a sample-based planning and it simulates experience from the current state with a given model, then applies model-free RL to simulated episodes. One of the variants is **Monte-Carlo tree search** (MCTS). This model is composed of the following procedures:

1. **Selection**: Start at the state of interest and going to a leaf following the **tree policy** (e.g. ϵ-greedy)
2. **Expansion**: Expand the tree by one or multiple child nodes reached from the selected leaf
3. **Simulation**: Simulate an episode following the **rollout policy** (e.g. random policy)
4. **Backup**: Update the action-values for all nodes visited in the tree (called **MC control**)

Note that while we expand the tree by one or more child nodes, we add only one child node from each node during a simulated episode and back-propagate the result in the end as in the MC control. If we replace the MC control with SARSA, it will be **TD search**.

8 Function approximation

In the cases where the state space and the action space is tremendously huge, the tabular value function does not work. One solution for this is to find a parameterized function taking a state and an action as inputs. However, the estimation is usually hard due to the issues not in supervised learning such as non-stationarity, i.e. dynamic environments, bootstrapping, and delayed targets, i.e. the reward for the corresponding action and state will come later. Note that we mainly focus on the representation of value functions $v_\pi(\cdot), q_\pi(\cdot, \cdot)$, but not policies or models here. One method for the function approximation is the following:

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{\alpha}{2} \frac{\partial}{\partial \boldsymbol{w}} [v_\pi(s_t) - \hat{v}_\pi(s_t; \boldsymbol{w}_t)]^2 \\ &= \boldsymbol{w}_t + \alpha [v_\pi(s_t) - \hat{v}_\pi(s_t; \boldsymbol{w}_t)] \frac{\partial \hat{v}_\pi(s_t; \boldsymbol{w}_t)}{\partial \boldsymbol{w}} \end{aligned}$$

This metric is called **mean squared value error**. Since $v_\pi(s_t)$ is not available in most cases, we often use the following two solutions:

1. **Gradient MC**: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[G_t - \hat{v}_\pi(s_t; \boldsymbol{w}_t)] \frac{\partial \hat{v}_\pi(s_t; \boldsymbol{w}_t)}{\partial \boldsymbol{w}}$
2. **Semi-gradient TD(0)**: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[r_{t+1} + \gamma \hat{v}_\pi(s_{t+1}; \boldsymbol{w}_t) - \hat{v}_\pi(s_t; \boldsymbol{w}_t)] \frac{\partial \hat{v}_\pi(s_t; \boldsymbol{w}_t)}{\partial \boldsymbol{w}}$

The gradient MC uses the cumulated reward as a state value function. On the other hand, the semi-gradient TD(0) uses the temporal difference as a value error. Since the value error relies on the predictions of the given parameterized model, but not targets, this is not true gradient and it is called **semi-gradient**. Note that since TD(0) uses $\hat{v}_\pi(s_{t+1}; \boldsymbol{w}_t)$ in the value error function and the accuracy relies heavily on weight vector $\boldsymbol{w}_t$, semi-gradient TD(0) can be **biased**.

8.1 Linear methods

Linear method is to approximate functions using linear combination of feature vector as follows:

$$\hat{v}_\pi(s_t; \boldsymbol{w}_t) = \boldsymbol{w}_t^\top \boldsymbol{x}(s_t)$$
$$\frac{\partial \hat{v}_\pi(s_t; \boldsymbol{w}_t)}{\partial \boldsymbol{w}_t} = \boldsymbol{x}(s_t)$$

where $\boldsymbol{x} = (x_1(s), x_2(s), \cdots, x_d(s))^\top$ and $x_i(s)$ is a mapping of a given state. The linear methods for both gradient MC and semi-gradient TD(0) are **guaranteed to converge**.

8.1.1 The convergence analysis of semi-gradient TD(0)

We give a proof of the convergence of semi-gradient TD(0). The update of weight vector in semi-gradient TD(0) is the following:

$$\begin{aligned}\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t + \alpha[\underbrace{r_{t+1} + \gamma \boldsymbol{w}_t^\top \boldsymbol{x}(s_{t+1}) - \boldsymbol{w}_t^\top \boldsymbol{x}(s_t)}_{\text{Scalar value}}]\boldsymbol{x}(s_t) \\ &= \boldsymbol{w}_t + \alpha \boldsymbol{x}(s_t)[r_{t+1} + \gamma \boldsymbol{x}(s_{t+1})^\top \boldsymbol{w}_t - \boldsymbol{x}(s_t)^\top \boldsymbol{w}_t] \\ &= \boldsymbol{w}_t - \alpha[\boldsymbol{x}(s_t)\boldsymbol{x}(s_t)^\top - \gamma \boldsymbol{x}(s_t)\boldsymbol{x}(s_{t+1})^\top]\boldsymbol{w}_t + \alpha r_{t+1}\boldsymbol{x}(s_t)\end{aligned}$$

If the system converges, we yield $\boldsymbol{w}_t = \boldsymbol{w}_{t+1}$, therefore,

$$\mathbb{E}[\boldsymbol{w}_\star | \boldsymbol{w}_\star] = \boldsymbol{w}_\star + \alpha(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{w}_\star) = \boldsymbol{w}_\star$$
$$\boldsymbol{A}\boldsymbol{w}_\star = \boldsymbol{b} \Longrightarrow \boldsymbol{w}_\star = \boldsymbol{A}^{-1}\boldsymbol{b}$$

where $\mathbb{E}[\boldsymbol{x}(s_t)\boldsymbol{x}(s_t)^\top - \gamma \boldsymbol{x}(s_t)\boldsymbol{x}(s_{t+1})^\top] = \boldsymbol{A}$ and $\mathbb{E}[r_{t+1}\boldsymbol{x}(s_t)] = \boldsymbol{b}$. The transition of $\boldsymbol{w}_t$ can be reformulated as follows:

$$\boldsymbol{w}_{t+1} = (I - \alpha \boldsymbol{A})\boldsymbol{w}_t + \alpha \boldsymbol{b}$$

where I is an identity matrix. If $|\alpha \boldsymbol{A}| \neq 0$ holds, there exists $\boldsymbol{c}$ such that:

$$\boldsymbol{w}_{t+1} - \boldsymbol{c} = (I - \alpha \boldsymbol{A})(\boldsymbol{w}_t - \boldsymbol{c})$$

where $\boldsymbol{c} = \alpha^{-1}\boldsymbol{A}^{-1}(\alpha \boldsymbol{b}) = \boldsymbol{A}^{-1}\boldsymbol{b}$. This transition satisfies the following equation:

$$\boldsymbol{w}_t - \boldsymbol{c} = (I - \alpha \boldsymbol{A})^t(\boldsymbol{w}_0 - \boldsymbol{c})$$

If this equation converges, $\lim_{t\to\infty}(I - \alpha \boldsymbol{A})^t = \boldsymbol{0}$ must be fulfilled [20]. The matrix which satisfies this property is called convergent and if a given matrix is convergent, the following holds [1]:

Theorem 2
A given matrix M is convergent iff:

$$\rho(M) < 1$$

where $\rho(M) := \max\{\|\lambda\| \,|\lambda \in \Lambda(M)\}$ *is the spectral radius, i.e. the maximum absolute eigenvalue, of a given matrix M.*

From this theorem, if $\rho(I - \alpha \boldsymbol{A}) < 1$, the weight vector converges. Let $V\Lambda V^{-1}$ be the eigendecomposition of the matrix A. Then we obtain the following result:

$$\begin{aligned}I - \alpha A &= I - \alpha V\Lambda V^{-1} \\ &= \underbrace{VIV^{-1}}_{=I} - \alpha V\Lambda V^{-1} \\ &= V(I - \alpha\Lambda)V^{-1}\end{aligned} \tag{2}$$

[20] The convergence requires $(I - \alpha \boldsymbol{A})^t \simeq (I - \alpha \boldsymbol{A})^{t+1}$ for a sufficiently large t. For this reason, $(I - \alpha \boldsymbol{A})^t(I - I + \alpha \boldsymbol{A}) = (I - \alpha \boldsymbol{A})^t \alpha \boldsymbol{A} \simeq \boldsymbol{0}$ holds. Since $|A| \neq 0$, $(I - \alpha \boldsymbol{A})^t \simeq \boldsymbol{0}$.

Therefore, $\rho(I - \alpha \boldsymbol{A}) = \|1 - \alpha\lambda_i\| < 1$, i.e. $0 < \lambda_i < \frac{2}{\alpha}$, must be fullfilled for all the eigenvalues. Since α is an arbitrary positive number, if A is positive definite, the system converges. Next, we will prove that $\boldsymbol{A}$ is positive definite.

$$\begin{aligned}
\boldsymbol{A} &= \mathbb{E}[\boldsymbol{x}(s_t)(\boldsymbol{x}(s_t) - \gamma\boldsymbol{x}(s_{t+1}))^\top] \\
&= \sum_s p(s) \sum_a \pi(a|s) \sum_{s',r} p(r, s'|s, a)\boldsymbol{x}(s)(\boldsymbol{x}(s) - \gamma\boldsymbol{x}(s'))^\top \\
&= \sum_s p(s) \sum_{s'} \boldsymbol{x}(s)(\boldsymbol{x}(s) - \gamma\boldsymbol{x}(s'))^\top \sum_a \pi(a|s) \sum_r p(r, s'|s, a) \\
&= \sum_s p(s) \sum_{s'} \boldsymbol{x}(s)(\boldsymbol{x}(s) - \gamma\boldsymbol{x}(s'))^\top p(s'|s) \\
&= \sum_s p(s)\boldsymbol{x}(s)\left(\boldsymbol{x}(s) - \gamma\sum_{s'} p(s'|s)\boldsymbol{x}(s')\right)^\top \\
&= \underbrace{[\boldsymbol{x}(s_1) \ \cdots \ \boldsymbol{x}(s_{|\mathcal{S}|})]}_{=\boldsymbol{X}^\top} \mathrm{diag}(p(s_1) \ \cdots \ p(s_{|\mathcal{S}|})) \left(\boldsymbol{X} - \gamma \begin{bmatrix} p(s_1|s_1) & \cdots & p(s_{|\mathcal{S}|}|s_1) \\ \vdots & \ddots & \vdots \\ p(s_1|s_{|\mathcal{S}|}) & \cdots & p(s_{|\mathcal{S}|}|s_{|\mathcal{S}|}) \end{bmatrix} \boldsymbol{X}\right) \\
&= \boldsymbol{X}^\top \boldsymbol{p}(I - \gamma\boldsymbol{P})\boldsymbol{X}
\end{aligned}$$

where $p(s)$ is a stationary distribution, i.e. the state distribution after a sufficiently long time of simulation and $|\mathcal{S}|$ is the size of state space. Since $\boldsymbol{p}$ ($\because \forall i, p(s_i) > 0$) is positive definite [21], we have to prove that $I - \gamma\boldsymbol{P}$ is positive definite. Let the eigendecomposition of $\boldsymbol{P}$ be $\boldsymbol{P} = V\Lambda V^{-1}$. Then we obtain the following in the same way as in Eq (2):

$$\begin{aligned}
I - \gamma\boldsymbol{P} &= I - \gamma V\Lambda V^{-1} \\
&= V(I - \gamma\Lambda)V^{-1}
\end{aligned}$$

Therefore, the eigenvalues of $I - \gamma\boldsymbol{P}$ are $1 - \gamma\lambda_i$ where λ_i is the eigenvalues of $\boldsymbol{P}$ and $0 < \gamma < 1$. Additionally, the following fact is known as Gershgorin circle theorem:

Theorem 3
Given a probability matrix $\boldsymbol{P}$, $\forall j, \exists i$ s.t. $\|\lambda_j - P_{i,i}\| \leq 1 - P_{i,i}$

Therefore, $\lambda_j < 1$ for all j and $1 - \gamma\lambda_j > 0$, so $\boldsymbol{A}$ is positive definite and **semi-gradient TD(0) is stable** [22].

8.1.2 Least squares TD

While least squares TD requires quadratic runtime for each update, it is more data-efficient. This method directly estimates the following:

$$\begin{aligned}
\hat{\boldsymbol{A}}_{t+1} &= \sum_{\tau=1}^{t} \boldsymbol{x}_\tau(\boldsymbol{x}_\tau - \gamma\boldsymbol{x}_{\tau+1})^\top + \epsilon I \\
\hat{\boldsymbol{b}}_{t+1} &= \sum_{\tau=1}^{t} r_{\tau+1}\boldsymbol{x}_\tau
\end{aligned}$$

[21] If matrix M is positive definite, $\boldsymbol{X}^\top M\boldsymbol{X}$ is also positive definite. Additionally, if a diagonal matrix D is positive definite, DM is positive definite if and only if M is positive definite.

[22] This holds only for TD(0) using linear function and does not hold for TD(0) using non-linear mapping. However, we can make use of the linear combination of non-linear basis.

Algorithm 10 Least squares TD

```
   $\hat{\boldsymbol{A}}^{-1} = \epsilon^{-1} I, \hat{\boldsymbol{b}} = \boldsymbol{0}, \boldsymbol{x} = \boldsymbol{x}(s)$        ▷ Initialization
1: function LEAST SQUARES TD
2:     while The budget is left do
3:         while The episode is not finished do
4:             $a \sim \pi(a|s), s', r \sim p(s', r|s, a), \boldsymbol{x}' = \boldsymbol{x}(s')$
5:             $\boldsymbol{v} = (\hat{\boldsymbol{A}}^{-1})^\top (\boldsymbol{x} - \gamma \boldsymbol{x}')$
6:             $\hat{\boldsymbol{A}}^{-1} = \hat{\boldsymbol{A}}^{-1}(I - \frac{\boldsymbol{x}\boldsymbol{v}^\top}{1 + \boldsymbol{v}^\top \boldsymbol{x}})$
7:             $\hat{\boldsymbol{b}} = \hat{\boldsymbol{b}} + r\boldsymbol{x}, \boldsymbol{w} = \hat{\boldsymbol{A}}^{-1}\hat{\boldsymbol{b}}$
8:             $s = s', \boldsymbol{x} = \boldsymbol{x}'$
```

Note that we can divide by time step size later and the algorithm shown in Algorithm 10 uses the following Woodbury formula:

$$(A + \boldsymbol{u}\boldsymbol{v}^\top)^{-1} = A^{-1} - \frac{A^{-1}\boldsymbol{u}\boldsymbol{v}^\top A^{-1}}{1 + \boldsymbol{v}^\top A^{-1}\boldsymbol{u}}$$

8.2 Coarse coding

Coarse coding is to divide a state space in a set of areas and assigns a feature $x_i(s)$ for a state in each area. For example, we can divide a state space into a set of circles. If the state is inside a circle, then the corresponding feature has the value 1 and otherwise the feature is 0. In other words, the mapping $\boldsymbol{x}(s)$ in this case is computed as follows:

$$\boldsymbol{x}(s) = [\mathbb{1}[s \in \mathcal{C}_1], \mathbb{1}[s \in \mathcal{C}_2], \cdots, \mathbb{1}[s \in \mathcal{C}_d]]$$

where $\mathcal{C}_i$ is the i-th circle. If we train at one point, then the parameters of all circles intersecting this state will be affected. Thus, the approximate value function will be affected at all points within the union of the circles. Features with large receptive fields give broad generalization, but might also limit the learned function to a coarse approximation. On the other hand, narrower receptive fields require more data or samples to approximate the mapping well. Note that if we **have sufficient data points**, we can train neural networks to construct feature space and then we do not need the coarse coding.

8.3 Memory-based function approximation

Memory-based method is a non-parametric method formulated as follows:

$$\hat{v}_\pi(s|\mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s')g(s')$$

where $k(\cdot, \cdot)$ is a kernel function and $g(s')$ is typically the cumulated reward of the state.

8.4 On-policy control with function approximation

By substituting a value function with an action-value function, we can apply semi-gradient method to SARSA as well.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[r_{t+1} + \gamma \hat{q}_\pi(s_{t+1}, a_{t+1}; \boldsymbol{w}_t) - \hat{q}_\pi(s_t, a_t; \boldsymbol{w}_t)] \frac{\partial \hat{q}_\pi(s_t, a_t; \boldsymbol{w}_t)}{\partial \boldsymbol{w}}$$

The only difference between SARSA and semi-gradient SARSA is that the normal SARSA updates the tabular action-value function while semi-gradient SARSA updates the weight vector.

9 Off-policy learning with function approximation

The main purpose of the off-policy learning is to maintain exploration and we discuss the off-policy learning using function approximation in this section.

9.1 Difficulty of off-policy learning with function approximation

In reinforcement learning, the following three elements often lead to stability issues:

1. **Function approximation**: Use parameterized value functions instead of tabulars
2. **Bootstrapping**: Use approximated values to construct value functions such as TD methods
3. **Off-policy learning**: use behavior policy to maintain exploration instead of using target policy to learn its value functions

We can avoid stability issues when we only use two of them. However, when we combine those three, we typically face to stability issues, i.e. divergent values, and we call those three **Deadly Triad**. In fact, semi-gradient off-policy TD(0) yields divergence of weights in the case of Baird's counterexample even though it only requires linear function approximation. Note that the update of the weight vector in off-policy semi-gradient TD(0) is perfomed by the following:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\rho_t\delta_t\frac{\partial\hat{v}(s_t; w_t)}{\partial\boldsymbol{w}}$$

where ρ_t is the importance ratio $\frac{\pi(a_t|s_t)}{b(a_t|s_t)}$ and δ_t is TD(0) $r_{t+1} + \gamma\hat{v}(s_{t+1}; \boldsymbol{w}_t) - \hat{v}(s_t; \boldsymbol{w}_t)$.

9.2 Gradient TD methods

One solution for the instability in semi-gradient TD methods is to use gradient methods, because gradient method yields the **convergence guarantee**. To introduce the gradient TD methods, we first introduce Bellman error.

Definition 5
Bellman error is defined as follows:

$$\begin{aligned}\bar{\delta}_{\boldsymbol{w}}(s) &= \sum_a \pi(a|s)\sum_{s',r} p(s', r|s, a)(r + \gamma\hat{v}(s'; \boldsymbol{w})) - \hat{v}(s; \boldsymbol{w})\\ &= \mathbb{E}_\pi[r + \gamma\hat{v}(s'; \boldsymbol{w})] - \hat{v}(s; \boldsymbol{w})\\ &= B_\pi(s, \hat{v}(\cdot; \boldsymbol{w})) - \hat{v}(s; \boldsymbol{w})\end{aligned}$$

The Bellman error is clearly the expectation of TD error. However, the Bellman error does not lie in the representable space by $\hat{v}(s; \boldsymbol{w})$ [23], the following **projected Bellman error** realizes the optimization of the Bellman error.

[23]When the policy $\pi(a|s)$ is not a random policy, the Bellman error is a weighted sum of the parameterized value function and this weight is considered to be parameterized by $\boldsymbol{w}$, so it is a non-linear mapping and it cannot be represented by $v(\cdot; \boldsymbol{w})$ anymore.

Definition 6
Suppose $\boldsymbol{v_w} \doteq \boldsymbol{v}(\cdot;\boldsymbol{w}) \in \boldsymbol{V_w}$, $B_{\pi,\boldsymbol{w}} \doteq B_\pi(\boldsymbol{v_w}) \in \boldsymbol{V}$ *and* $\boldsymbol{v_w}, B_{\pi,\boldsymbol{w}} \in \mathbb{R}^{|\mathcal{S}|}$, *then the projected Bellman error is defined as:*

$$\|\Pi(B_{\pi,\boldsymbol{w}} - \boldsymbol{v_w})\|^2_{p(s)} = \|\Pi\bar{\delta}_{\boldsymbol{w}}\|^2_{p(s)}$$

where $\|\boldsymbol{v}\|^2_{p(s)} = \sum_{s=1}^{|\mathcal{S}|} p(s)v_s^2$ *and projection matrix satisfies* $\Pi\boldsymbol{v} \doteq \boldsymbol{v_w}$ *where* $\boldsymbol{w} = \mathrm{argmin}_{\boldsymbol{w}}\|\boldsymbol{v} - \boldsymbol{v_w}\|^2_{p(s)}$.

Note that $B_{\pi,\boldsymbol{w}}$ cannot be fully represented by $\boldsymbol{v_w}$, so $\boldsymbol{V} \cap \boldsymbol{V_w} = \emptyset$ [24]. Additionally, orthogonal projections of the objective value function onto the representable function by $\boldsymbol{w}$ is $\Pi = \boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{P}\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{P}$ where $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{S}|\times d}$ is a feature matrix and the stationary distribution of behavior policy $\boldsymbol{P} = \mathrm{diag}[p(s_1), \cdots, p(s_{|\mathcal{S}|})]$. $p(s_i)$ becomes larger when we visit the state s_i more. In other words, predictions for the states which are visited more have to be more precise. Note that there are various possible error metrics for this optimization and we obtain different weights depending on error metrics such as TDE, PBE, BE, VE. The projected Bellman error can be transformed as follows:

$$\begin{aligned}\|\Pi\bar{\delta}_{\boldsymbol{w}}\|^2_{p(s)} &= (\Pi\bar{\delta}_{\boldsymbol{w}})^\top\boldsymbol{P}\Pi\bar{\delta}_{\boldsymbol{w}} \\ &= \bar{\delta}_{\boldsymbol{w}}^\top\boldsymbol{P}^\top\boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{P}\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{P}\boldsymbol{X}(\boldsymbol{X}^\top\boldsymbol{P}\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{P}\bar{\delta}_{\boldsymbol{w}} \\ &= (\boldsymbol{X}^\top\boldsymbol{P}\bar{\delta}_{\boldsymbol{w}})^\top(\boldsymbol{X}^\top\boldsymbol{P}\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{P}\bar{\delta}_{\boldsymbol{w}}\end{aligned}$$

The projected Bellman error lies in the representable space and it can be optimized via the gradient descent. Therefore, the gradient with respect to $\boldsymbol{w}$ is the following:

$$\begin{aligned}\frac{\partial\|\Pi\bar{\delta}_{\boldsymbol{w}}\|^2_{p(s)}}{\partial\boldsymbol{w}} &= 2\left(\frac{\partial\boldsymbol{X}^\top\boldsymbol{P}\bar{\delta}_{\boldsymbol{w}}}{\partial\boldsymbol{w}}\right)^\top(\boldsymbol{X}^\top\boldsymbol{P}\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{P}\bar{\delta}_{\boldsymbol{w}} \\ &= 2\underbrace{\mathbb{E}_b[\rho_t(\gamma\boldsymbol{x}_{t+1} - \boldsymbol{x}_t)\boldsymbol{x}_t^\top]}_{\mathbb{R}^{d\times d}}\underbrace{(\mathbb{E}_b[\boldsymbol{x}_t\boldsymbol{x}_t^\top])^{-1}}_{\mathbb{R}^{d\times d}}\underbrace{\mathbb{E}_b[\rho_t\delta_t\boldsymbol{x}_t]}_{\mathbb{R}^{d\times 1}}\end{aligned} \tag{3}$$

where each expectation is taken over the stationary distribution of the behavior policy and we use the importance sampling $\bar{\delta}_{\boldsymbol{w}} = \mathbb{E}_\pi[B_{\pi,\boldsymbol{w}} - v_{\boldsymbol{w}}] = \mathbb{E}_b[\rho(B_{\pi,\boldsymbol{w}} - v_{\boldsymbol{w}})]$ and the following part considers the target change:

$$\frac{\partial\bar{\delta}_{\boldsymbol{w}}}{\partial\boldsymbol{w}} = \underbrace{\frac{\partial B_{\pi,\boldsymbol{w}}}{\partial\boldsymbol{w}}}_{\text{Target gradient}} - \underbrace{\frac{\partial\boldsymbol{v_w}}{\partial\boldsymbol{w}}}_{\text{semi-gradient}}$$

The second term of this equation complements the lack in the semi-gradient and thus each update enables the projected Bellman error to approach zero. For this reason, we call this method a **gradient method**. In order to save memory and computation, we usually estimate the product of the last second term in Eq. (3) directly as follows:

$$\begin{aligned}\boldsymbol{v} &\simeq (\mathbb{E}_b[\boldsymbol{x}_t\boldsymbol{x}_t^\top])^{-1}\mathbb{E}_b[\rho_t\delta_t\boldsymbol{x}_t] \\ \boldsymbol{v} &= \boldsymbol{v} + \beta\rho_t(\delta_t - \boldsymbol{v}^\top\boldsymbol{x}_t)\boldsymbol{x}_t\end{aligned}$$

where β is a control parameter. Note that the estimation of the expected values are performed via MC sampling. This method and its variants are called gradient TD methods and they are typically more stable than semi-gradient TD methods.

[24]When $d < |\mathcal{S}|$, the rank of $V_{\boldsymbol{w}} = \boldsymbol{w}\boldsymbol{X}$ is smaller than $V = \boldsymbol{w}_\infty\boldsymbol{X}_\infty$, so Π projects higher rank matrix to lower rank matrix.

Algorithm 11 Neural fitted-Q iteration

```
1: function NEURAL FITTED-Q ITERATION
2:     Initialize replay memory $\mathcal{M}$                                    ▷ Unlimited size
3:     for $i = 0, 1, \dots$ do
4:         Generate an episode following $\epsilon$-greedy : $\{(s_t, a_t, r_t)\}_{t=0}^{T}$      ▷ $\hat{q}(s_{T+1}, \cdot; \cdot) = 0$
5:         Initialize weights of network and store transitions $(s_t, a_t, s_{t+1}, r_t)$ in memory $\mathcal{M}$
6:         while Budget is left do
7:             $\mathcal{L}(\boldsymbol{w}) = \frac{1}{|\mathcal{M}|}\sum_{i=1}^{|\mathcal{M}|}(r_i + \gamma \max_{a'} \hat{q}(s'_i, a'; \boldsymbol{w}) - \hat{q}(s_i, a_i; \boldsymbol{w}))^2$
8:             Loss Backward
```

Algorithm 12 DQN

```
1: function DQN
2:     Initialize weights of network and replay memory $\mathcal{M}$ to size $M$
3:     for $i = 0, 1, \dots$ do
4:         for $t = 1, \dots, T$ do
5:             Select $a_t$ using $\epsilon$-greedy on $\hat{q}(\cdot, \cdot; \boldsymbol{w})$
6:             Store transition $(s_t, a_t, s_{t+1}, r_t)$ in $\mathcal{M}$ and remove the oldest one if $|\mathcal{M}| > M$
7:             Average over mini batch of transitions randomly sampled from replay memory
8:             $\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{(s,a,s',r)\sim\mathcal{M}}\left[(r + \gamma \max_{a'} \hat{q}(s', a'; \boldsymbol{w}_{\text{old}}) - \hat{q}(s, a; \boldsymbol{w}))^2\right]$
9:             Loss Backward                    ▷ Update the weight $\boldsymbol{w}$ of the Q-network
10:        $\boldsymbol{w}_{\text{old}} = \boldsymbol{w}$
```

9.3 Deep Q-learning

9.3.1 Neural fitted-Q iteration

Neural fitted-Q iteration (NFQ) [8] is the first Q-learning using neural networks. This method is a model-free off-policy RL algorithm that works on continuous state and discrete action spaces. This algorithm stores all the experience as a memory and uses all of them during each training. Since this method also uses bootstrapped action-value function, this is also a semi-gradient method.

9.3.2 DQN (deep Q-networks)

DQN [7] is the extension of NFQ and it also uses experience replay. The experiences from the past reduces correlations, i.e. the positive bias of certain action values, because the current policy and the past policy are not identical. Additionally, the samples from the past memory prevent catastrophic forget [25]. The main difference of DQN from NFQ is that while NFQ initializes Q-network each iteration and the TD error is computed using only one network, DQN uses only one network throughout the training and uses the fixed weights from the previous iterations for the computation of gradients. We call the fixed network in DQN **target network** $\hat{q}(\cdot, \cdot; \boldsymbol{w}_{\text{old}})$ and update its weights every certain step. Since the target network provides the stationary target, the optimization becomes easier and stable. The update of the weights in the target network is performed by either of the following:

1. **hard update**: update target network every certain steps
2. **slow update**: slowly update weights every step by $\boldsymbol{w}_{\text{old}} = (1 - \tau)\boldsymbol{w}_{\text{old}} + \tau\boldsymbol{w}$

[25]NFQ uses full-batch and DQN uses mini-batch. Furthermore, DQN limits the maximum memory size.

Algorithm 13 n-step TD for prediction

```
1: function N-STEP TD
2:     while The budget is left do
3:         for $t = 0, 1, \dots, T+n-1$ do
4:             if $t < T$ then
5:                 $a_t \sim \pi(a|s_t), s_{t+1}, r_{t+1} \sim p(s', r|s_t, a_t)$
6:             if $t \geq n-1$ then                                    ▷ $\forall t > T, v_\pi(s_t) = 0, r_t = 0$
7:                 $G_{t-n+1:t+1} = v_\pi(s_{t+1}) + \sum_{i=1}^{n} \gamma^{i-1} r_{t-n+i}$
8:                 $v_\pi(s_{t-n+1}) = v_\pi(s_{t-n+1}) + \alpha(G_{t-n+1:t+1} - v_\pi(s_{t-n+1}))$
9:     return $v_\pi$
```

Algorithm 14 Off-policy n-step SARSA for control

```
1: function OFF-POLICY N-STEP TD
2:     while The budget is left do
3:         $a_0 \sim b(a|s_0)$
4:         for $t = 0, 1, \dots, T+n-1$ do
5:             if $t < T$ then
6:                 $s_{t+1}, r_{t+1} \sim p(s', r|s_t, a_t), a_{t+1} \sim b(a|s_{t+1})$
7:             if $t \geq n-1$ then                                    ▷ $\forall t > T, v_\pi(s_t) = 0, r_t = 0$
8:                 $\rho_{t-n+2:t+1} = \prod_{i=2}^{n+1} \frac{\pi(a_{t-n+i}|s_{t-n+i})}{b(a_{t-n+i}|s_{t-n+i})}$
9:                 $G_{t-n+1:t+1} = q_\pi(s_{t+1}) + \sum_{i=1}^{n} \gamma^{i-1} r_{t-n+i}$
10:                $q_\pi(s_{t-n+1}) = q_\pi(s_{t-n+1}) + \alpha\rho_{t-n+2:t+1}(G_{t-n+1:t+1} - q_\pi(s_{t-n+1}))$
11:    return $v_\pi$
```

10 n-step bootstrapping and eligibility trace

n-step bootstrapping and the eligibility trace are the methods to generalize the TD and MC.

10.1 n-step TD

The target of n-step bootstrapping is defined as follows:

$$G_{t:t+n} = \gamma^n v_\pi(s_{t+n}) + \sum_{i=1}^{n} \gamma^{i-1} r_{t+i}$$

The case of $n = 1$ is equivalent to TD and that of $n = \infty$ is equivalent to MC. This technique can be expanded to control and one example is the n-step SARSA as follows:

$$q_{t+n}(s_t, a_t) = q_{t+n-1}(s_t, a_t) + \alpha(G_{t:t+n} - q_{t+n-1}(s_t, a_t))$$

Since **action-value function is updated n steps after from the first visit** and the action values until $t + n - 1$ step are available, we use them for the update. The algorithm of off-policy SARSA is shown in Algorithm 14. Note that if we take a non-greedy policy for the target policy π, we have to make sure $a_t = \text{argmax}_a q_\pi(s_t, a)$ at each update. Additionally, it is typically hard to apply n-step TD to Q-learning since **the subtrajectory are not sampled from the target policy**. For this reason, we often use importance sampling for off-policy learning and it usually does not exhibit good performance because of both high bias and high variance. The major issue of n-step TD is the **trade-off between variance and bias**. Larger n leads to lower bias and higher variance.

10.2 Forward-view TD(λ) (offline λ-return algorithm)

To balance the trade-off between bias and variance, we can take the average of different n-step TD. TD(λ) or λ-return takes the weighted average of different n-step TD as follows:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{T-t-1}G_t$$

where T is the length of each episode and $(1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}+\lambda^{T-t-1} = 1$ and $0 \leq \lambda \leq 1$. This cumulated reward is called λ-return and $\lambda = 0, 1$ correspond to TD(0) and MC methods, respectively. Since n-step TD allows relatively immediate update compared to MC and more reliable target compared to TD, n-step TD is efficient. The update of value function is made according to the following:

$$\text{Function approximation} : \boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha(G_t^\lambda - \hat{v}(s_t;\boldsymbol{w}_t))\frac{\partial \hat{v}(s_t;\boldsymbol{w}_t)}{\partial \boldsymbol{w}}$$
$$\text{Others} : v_\pi(s_t) = v_\pi(s_t) + \alpha(G_t^\lambda - v(s_t))$$

This target reward G_t^λ can be reformulated as the weighted sum of TDs from different time steps and we show this now. In other words, we prove that $G_t^\lambda = \sum_{n=0}^\infty \lambda^n \delta_{t+n+1}$. First, the following holds:

$$(1-\lambda)\sum_{n=1}^\infty \lambda^{n-1}\sum_{i=1}^n r_{t+i} = \sum_{n=1}^\infty \lambda^{n-1}r_{t+n} \left(\because \sum_{n=1}^\infty \lambda^{n-1}\sum_{i=1}^n r_{t+i} = \sum_{i=1}^\infty r_{t+i}\sum_{n=i}^\infty \lambda^{n-1}\right) \tag{4}$$

$$(1-\lambda)\sum_{n=1}^\infty \lambda^{n-1}v(s_{t+n}) = v(s_t) + \sum_{n=1}^\infty \lambda^{n-1}(v(s_{t+n}) - v(s_{t+n-1})) \tag{5}$$

For the simplicity, we ignore γ first.

$$\begin{aligned} v(s_t) &= (1-\lambda)\mathbb{E}\left[\sum_{n=1}^\infty \lambda^{n-1}G_{t:t+n}\right] = (1-\lambda)\mathbb{E}\left[\sum_{n=1}^\infty \lambda^{n-1}\left(\sum_{i=1}^n r_{t+i} + v(s_{t+i})\right)\right] \\ &= \mathbb{E}\left[\sum_{n=1}^\infty \lambda^{n-1}\Big(r_{t+n} + v(s_{t+n}) - v(s_{t+n-1})\Big) + v(s_t)\right] \ (\because \text{Eqs. (4), (5)}) \\ &= \mathbb{E}\left[\sum_{n=1}^\infty \lambda^{n-1}\delta_{t+n} + v(s_t)\right] \end{aligned}$$

By replacing $v(s_{t+n}), r_{t+n}$ with $\gamma^n v(s_{t+n}), \gamma^{n-1}r_{t+n}$, we can trivially show the discounted version of the formulation. Using the result, the update is performed as follows:

$$\text{Function approximation} : \boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\sum_{n=1}^\infty(\lambda\gamma)^{n-1}\delta_{t+n}\frac{\partial \hat{v}(s_t;\boldsymbol{w}_t)}{\partial \boldsymbol{w}}$$
$$\text{Others} : v_\pi(s_t) = v_\pi(s_t) + \alpha\sum_{n=1}^\infty(\lambda\gamma)^{n-1}\delta_{t+n}$$

Since this update requires the future rewards, we have to **compute the value function from completed episodes** and this computation is performed in **offline**[26] ; therefore, this method is called offline λ-return or **forward-view TD(λ)**. In the same vein, offline λ-return can be applied to SARSA as follows:

$$q_\pi(s_t) = q_\pi(s_t) + \alpha\sum_{n=1}^\infty(\lambda\gamma)^{n-1}\delta_{t+n}$$

[26]As shown in Algorithm 14, the update for the time step t will happen at time step $t+n-1$ and we have to wait until this time step in the forward-view. In this case, the complete episode has the length of n.

Algorithm 15 Backward TD(λ)

```
1: function Backward TD($\lambda$)
2:     while The budget is left do
3:         $\boldsymbol{z}_{-1} = \boldsymbol{0}$
4:         for $t = 0, 1, \dots, T$ do
5:             $a_t \sim \pi(a|s_t), s_{t+1}, r_{t+1} \sim p(s', r|s_t, a_t)$
6:             $\boldsymbol{z}_t = \gamma\lambda \boldsymbol{z}_{t-1} + \frac{\partial \hat{v}(s_t; \boldsymbol{w}_t)}{\partial \boldsymbol{w}}$
7:             $\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}; \boldsymbol{w}_t) - \hat{v}(s_t; \boldsymbol{w}_t)$
8:             $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{z}_t$
9:     return $v_\pi$
```

Basically, the reward from far future is less important compared to that from near future and the TDs using far future are exponetially decayed by the factor of $\gamma\lambda$. In this sense, it does not make much difference when we cut off the far future and it can yield faster convergence. Such method is called **truncated TD (TTD)** and the update is the following:

$$G^{\lambda}_{t:t+n} = (1-\lambda)\sum_{i=1}^{n} \lambda^{i-1} G_{t:t+n} + \lambda^n G_{t:t+n} = \hat{v}(s_t; \boldsymbol{w}_t) + \sum_{i=1}^{n} (\lambda\gamma)^{i-1} \delta_{t+i}$$
$$\boldsymbol{w}_{t+n} = \boldsymbol{w}_{t+n-1} + \alpha(G^{\lambda}_{t:t+n} - \hat{v}(s_t; \boldsymbol{w}_{t+n-1}))\frac{\partial \hat{v}(s_t; \boldsymbol{w}_{t+n-1})}{\partial \boldsymbol{w}}$$

10.3 Backward-view TD(λ)

To compute the value function in an online manner, we use backward-view and then we do not have to wait for the end. This method combines heuristics of **Frequency and recency** to consider the contribution to the prior states' value. This is realized by a λ-decay. The update of this method is as follows:

$$\boldsymbol{z}_t = \underbrace{\gamma\lambda \boldsymbol{z}_{t-1}}_{\text{Recency}} + \underbrace{\frac{\partial \hat{v}(s_t; \boldsymbol{w}_t)}{\partial \boldsymbol{w}}}_{\text{Frequency}}$$
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{z}_t$$

where λ is called trace-decay parameter and the case of $\lambda = 0$ is equivalent to semi-gradient method. Backward-view TD(λ) can also be applied to SARSA in the same vein as Algorithm 15.

11 Policy gradient

Up to the last section, we focus on the estimation of the value functions and approximate the policy $\pi(a|s)$ implicitly. The policy gradient aims to approximate the policy explicitly by parameterizing $\pi(a|s; \boldsymbol{w}_\pi)$ and maximizing the performance by the gradient ascent. The major advantages of the policy gradient are the **handling of large or continuous action spaces**. However, since it usually requires MC sampling, the result is likely to cause **high variance** and requires much **more time for the convergence**.

11.1 The formulation

The formal formulation of the policy gradient in the continuous settings is the following:

$$\pi_\star \in \mathrm{argmax}_\pi \mathcal{P}(\boldsymbol{w}_\pi)$$
$$\mathcal{P}(\boldsymbol{w}_\pi) = \sum_{s \in \mathcal{S}} p(s) v_\pi(s) \tag{6}$$

where $\mathcal{P}$ is the performance measure of the policy, $\boldsymbol{w}_\pi$ is the learnable parameters for the policy $\pi(a|s;\boldsymbol{w}_\pi)$ and $p(s)$ is the stationary distribution of each state. Note that we take the expectation over the possible initial states in the case of episodic problems, i.e. the problems that have the termination for each episode. The following **Policy gradient theorem** holds given the performance measure $\mathcal{P}$.

Theorem 4
$\forall\pi(a|s,\boldsymbol{w}_\pi) \in C^1$, the policy gradient is:

$$\begin{aligned}\frac{\partial \mathcal{P}}{\partial \boldsymbol{w}_\pi} &= \sum_s p(s) \sum_a q_\pi(s,a) \frac{\partial \pi(a|s;\boldsymbol{w}_\pi)}{\partial \boldsymbol{w}_\pi} \\ &= \sum_s p(s) \sum_a q_\pi(s,a) \pi(a|s;\boldsymbol{w}_\pi) \frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a|s;\boldsymbol{w}_\pi) \\ &= \mathbb{E}_\pi\left[q_\pi(s,a)\frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a|s;\boldsymbol{w}_\pi)\right]\end{aligned} \tag{7}$$

where $v_\pi(s) = \sum_a \pi(a|s;\boldsymbol{w}_\pi) q_\pi(s,a)$

Note that we take the following form to make the transformation to the expectation form easier:

$$\frac{\partial \pi(a|s;\boldsymbol{w}_\pi)}{\partial \boldsymbol{w}_\pi} = \pi(a|s;\boldsymbol{w}_\pi) \underbrace{\frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a|s;\boldsymbol{w}_\pi)}_{\text{score function}} \tag{8}$$

We assume that the expectation value over the policy π is defined as follows:

$$\mathbb{E}_\pi[f] = \sum_{s,a} p(s)\pi(a|s) f(s,a) \tag{9}$$

11.2 REINFORCE

REINFORCE is **an MC policy gradient method**. This method computes the **unbiased** action-value function q_π by MC sampling:

$$q_\pi(s_t,a_t) = G_t = r_{t+1} + \gamma r_{t+2} + \cdots \gamma^{T-t-1} r_T \tag{10}$$

and uses the function for the gradient computation:

$$\mathbb{E}_\pi\left[q_\pi(s,a)\frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a|s;\boldsymbol{w}_\pi)\right] = \alpha \sum_{t=0}^{T-1} \gamma^t G_t \frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a_t|s_t;\boldsymbol{w}_\pi) \tag{11}$$

where α is a learning rate or a constant factor which scale the right hand side. By taking each term iteratively, we update the weights as shown in shown in Algorithm 16. Since it uses MC sampling, it typically suffers from high variance and slow convergence. Therefore, some works have developed the variance reduction methods. One major example is the **variance reduction with baselines**. Since $\sum_a \pi(a|s) = 1$, the derivatives of the summation of the policy over the action space is always zero. Using this fact, the following is used for the variance reduction while maintaining the unbiased value function:

$$\begin{aligned}&\frac{\partial \mathcal{P}}{\partial \boldsymbol{w}_\pi} = \mathbb{E}_\pi\left[(q_\pi(s,a) - f(s))\frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a|s,\boldsymbol{w}_\pi)\right] \\ &\left(\because \sum_s \sum_a p(s) f(s) \underbrace{\pi(a|s,\boldsymbol{w}_\pi)\frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a|s,\boldsymbol{w}_\pi)}_{\frac{\partial \pi(a|s,\boldsymbol{w}_\pi)}{\partial \boldsymbol{w}_\pi}} = \sum_s p(s) f(s) \frac{\partial \sum_a \pi(a|s,\boldsymbol{w}_\pi)}{\partial \boldsymbol{w}_\pi} = 0\right)\end{aligned} \tag{12}$$

Algorithm 16 REINFORCE: Monte-Carlo policy-gradient control

```
    $\alpha_\pi, \alpha_v$                                                        ▷ Learning rate
 1: function REINFORCE
 2:     while The budget is left do
 3:         $G = 0$, Generate an episode following $\pi(a|s; \boldsymbol{w}_\pi) : \{(s_t, a_t, r_t)\}_{t=0}^{T}$
 4:         for $t = 0, 1, \ldots, T-1$ do
 5:             $G = \sum_{\tau=t+1}^{T} \gamma^{\tau-t-1} r_\tau$
 6:             if Use baseline then
 7:                 $\delta = G - \hat{v}(s_t, \boldsymbol{w}_v)$                      ▷ Variance reduction
 8:                 $\boldsymbol{w}_v = \boldsymbol{w}_v + \alpha_v \delta \frac{\partial \hat{v}(s_t, \boldsymbol{w}_v)}{\partial \boldsymbol{w}_v}$
 9:                 $\boldsymbol{w}_\pi = \boldsymbol{w}_\pi + \alpha \gamma^t \delta \frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a_t|s_t; \boldsymbol{w}_\pi)$
10:             else
11:                 $\boldsymbol{w}_\pi = \boldsymbol{w}_\pi + \alpha \gamma^t G \frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a_t|s_t; \boldsymbol{w}_\pi)$
```

Algorithm 17 One-step actor-critic control

```
    $\alpha_\pi, \alpha_v$                                                        ▷ Learning rate
 1: function ONE-STEP ACTOR-CRITIC
 2:     while The budget is left do
 3:         for $t = 0, 1, \ldots, T-1$ do
 4:             $a_t \sim \pi(a|s_t, \boldsymbol{w}_\pi), s_{t+1}, r_{t+1} \sim p(s', r|s_t, a_t)$
 5:             $\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w}_v) - \hat{v}(s_t, \boldsymbol{w}_v)$
 6:             $\boldsymbol{w}_v = \boldsymbol{w}_v + \alpha_v \delta_t \frac{\partial \hat{v}(s_t, \boldsymbol{w}_v)}{\partial \boldsymbol{w}_v}, \boldsymbol{w}_\pi = \boldsymbol{w}_\pi + \alpha_\pi \gamma^t \delta_t \frac{\partial}{\partial \boldsymbol{w}_\pi} \log \pi(a_t|s_t, \boldsymbol{w}_\pi)$
```

Note that the baseline reduces the variance through the reduction of the scale of each value function. In other words, the variance will be relaxed by introducing the scale reduction. The typical selection of the baseline is **parameterized value function** $\hat{v}(s; \boldsymbol{w}_v)$ and we optimize this value function jointly with the policy by gradient MC. We replace G with $G - \hat{v}(s_t; \boldsymbol{w}_v)$ in the update of the weights of the policy.

11.3 Actor-critic methods

While the baseline reduces the variance of policy gradient, it still includes high variance and tends to converge slowly. To address this issue, the actor-critic methods have been introduced [5] and they **replace MC with TD**. They jointly learn policy (**actor**) and value function (**critic**) and these are computed based on TD as shown in Algorithm 17. Note that the actor-critic methods can be applied to n-step methods as well.

11.4 Proximal policy optimization

In the proximal policy optimization [10], we would like to optimize the following local approximation:

$$\mathcal{P}(\boldsymbol{w}_\pi) \simeq \mathcal{L}_{\pi_{\mathrm{old}}}(\boldsymbol{w}_\pi) = \mathcal{P}(\boldsymbol{w}_{\pi_{\mathrm{old}}}) + \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{A}_{\pi_{\mathrm{old}}}(s_t, a_t) \right] \tag{13}$$

where $\mathcal{A}_{\pi_{\mathrm{old}}}(s, a)$ is the advantage function of the new policy π over the older policy π_{old} and the advantage function is defined as follows:

$$\begin{aligned} \mathcal{A}_{\pi_{\mathrm{old}}}(s, a) &= \mathbb{E}_\pi[q_{\pi_{\mathrm{old}}}(s, a) - v_{\pi_{\mathrm{old}}}(s)] \\ &= \mathbb{E}_\pi[r(s, a) + \gamma v_{\pi_{\mathrm{old}}}(s') + v_{\pi_{\mathrm{old}}}(s)] \end{aligned} \tag{14}$$

Intuitively speaking, the advantage function measures the improvement of the policy given value functions of the old policy. By rearanging Eq (13) to sum over states instead of time steps, we obtain:

$$\mathcal{L}_{\pi_{\text{old}}}(\boldsymbol{w}_\pi) - \mathcal{P}(\boldsymbol{w}_{\text{old}}) = \mathbb{E}_{s\sim p_{\text{new}}(s), a\sim\pi}[\mathcal{A}_{\pi_{\text{old}}}(s,a)] \tag{15}$$

From these equations, the performance optimization is achieved by the maximization of the expected advantage function given a state distribution $p_{\text{new}}(s)$ of a new policy π. However, since we **cannot sample from a policy that has not been applied**, we ignore the change in the state distribution and use the following proxy to reuse the old data:

$$\begin{aligned} \boldsymbol{w}_\pi &\in \text{argmax}_{\boldsymbol{w}} \mathbb{E}_{s\sim p_{\text{old}}(s), a\sim\pi_{\text{old}}}\left[\frac{\pi(\cdot|\cdot;\boldsymbol{w})}{\pi_{\text{old}}}\mathcal{A}_{\pi_{\text{old}}}(s,a)\right] \\ \text{where } \mathbb{E}_{s\sim p_{\text{old}}(s), a\sim\pi_{\text{old}}}\left[\frac{\pi}{\pi_{\text{old}}}\mathcal{A}_{\pi_{\text{old}}}(s,a)\right] &= \mathcal{L}_{\pi_{\text{old}}}(\boldsymbol{w}_\pi) - \mathcal{P}(\boldsymbol{w}_{\text{old}}) \end{aligned} \tag{16}$$

This optimization is performed via the **stochastic gradient ascent** over previously obtained N trajectories. Note that the following equation is called improvement theory and it holds between the surrogate function and the original objective:

$$\mathcal{L}_{\pi_{\text{old}}}(\boldsymbol{w}_\pi) \geq \mathcal{P}(\boldsymbol{w}_\pi) - c \cdot \max_s \underbrace{D_{\text{KL}}(\pi_{\text{old}}(a|s)\|\pi(a|s))}_{\text{regularization}} \tag{17}$$

where c is a positive constant number. The statement of this theorem is that the optimization of the surrogate is similar to that of the original one as long as we keep the KL-divergence beween those two sufficiently small. In practice, we apply either **clipping** to the importance to increase the stability or subtracting the KL-divergence from the surrogate objective to **put a penalty** on largely different policies, because $\mathcal{L}_{\pi_{\text{old}}}(\boldsymbol{w}_\pi)$ is a local approximation of the performance $\mathcal{P}(\boldsymbol{w}_\pi)$ and optimistic optimizaitons lead to high bias and high variance.

12 Advanced value-based methods

1. **Prioritized experience replay** [9]: The DQN training can be inefficient when we sample mini-batches uniformly from the experience, so it prioritizes the importance measured by the magnitude of TD error[27]. The mini-batches are sampled by the weighted distribution based on the priorities. We do not use deterministic selections to avoid forgetting the transitions with small priorities and sampling only noisy TD error.

2. **Distributional RL** [2]: It uses long-term value distribution. There is no convergence guarantee, but it can model multi-modalities in value distributions and it may help with non-stationarities; therefore, it can be stable.

3. **Deep deterministic policy gradient (DDPG)** [6]: DDPG is an actor-critic continuous DQN method. In the case of continuous actions, the greedy step is not trivial; therefore, it performs the approximation of deterministic actor $a(s;\theta) = \text{argmax}_a \hat{q}_\pi(s,a;\boldsymbol{w})$ by a NN and update its parameters θ by deterministic policy gradient theorem. DDPG builds the basis for SOTA actor-critic algorithms, but can be unstable and sensitive to its hyperparameters.

4. **TD3** [3]: It introduces clipped double Q-learning, delayed policy updates and target-policy smoothing to the vanilla DDPG. The update is similar to expected SARSA. Clipped double Q-learning takes the minimum of two Q-networks and removes the overestimation bias. Delayed policy updates performs the update every certain steps and removes the overestimation by ignoring the mutual dependency between actor and critic updates. Target-policy smoothing adds gaussian noise to the next action in the target calculation.

[27]This TD errors are fixed once they are stored, so the older memory can be biased.

5. **Soft actor-critic** [4]: It promotes the exploration by adding the entropy of the policy given a state s_t to the reward.

Note that all of them except distributional RL use TD error for the weight update.

References

[1] Bauschke, H.H., Cruz, J., Nghia, T.T., Phan, H.M., Wang, X.: Optimal rates of convergence of matrices with applications. arXiv preprint arXiv:1407.0671 (2014)

[2] Bellemare, M.G., Dabney, W., Munos, R.: A distributional perspective on reinforcement learning. In: International Conference on Machine Learning. pp. 449–458. PMLR (2017)

[3] Fujimoto, S., Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods. In: International Conference on Machine Learning. pp. 1587–1596. PMLR (2018)

[4] Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International conference on machine learning. pp. 1861–1870. PMLR (2018)

[5] Konda, V.R., Tsitsiklis, J.N.: Actor-critic algorithms. In: Advances in neural information processing systems. pp. 1008–1014 (2000)

[6] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971 (2015)

[7] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)

[8] Riedmiller, M.: Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In: European conference on machine learning. pp. 317–328. Springer (2005)

[9] Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. arXiv preprint arXiv:1511.05952 (2015)

[10] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)