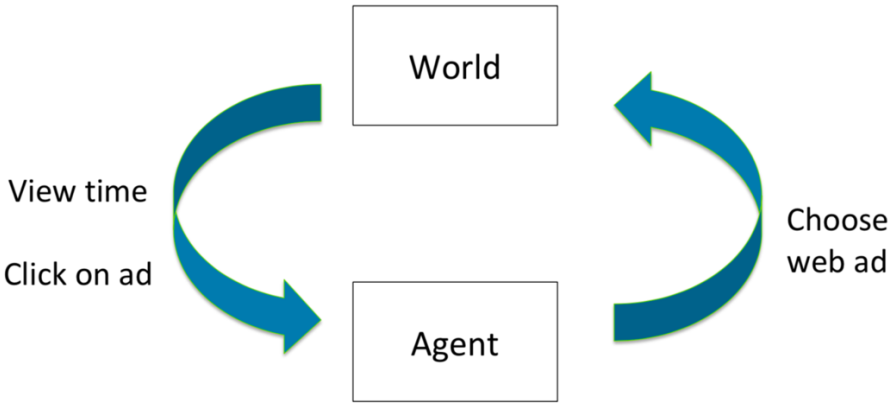
CS 234 Summary

1. Reinforcement Learning

Goals: learn to make good sequences of decisions, which involve optimization, delayed consequences, exploration and generalization. Optimization is to let agent to find an optimal way to make decisions; delayed consequences tells us some particular decisions have a great impact on future reward; exploration is to learn about how world works by repetitively making decisions, though some of them may not invertible (medicine trial); when there are a huge number of states, generalization can help(function approximation).
Sequential Decision Making: Formally, in the discrete setting, the agent will make a sequence of actions $\{a_0, a_1, \cdots, a_t\}$, then the real world will return a sequence of observations $\{o_0, o_1, \cdots, o_t\}$ and corresponding rewards $\{r_0, r_1, \cdots, r_t\}$. Agent state is defined as the function of total history: $s_t = f(h_t)$, and our goal is to find a sequence of decisions that maximize the total expected future rewards.



Basics:
Markov assumption: we consider a transition dynamics that the future state s_{t+1} is only dependent on the current state s_t and corresponding action a_t. i.e.

$$P(s_{t+1}|s_t, a_t, \cdots, s_0, a_0) = P(s_{t+1}|s_t, a_t).$$

Reward function: to predict the expected future rewards given a state and(or) an action at a particular time step(can be either stochastic or deterministic).

$$R(s) = \mathbb{E}[r_t|s_t = s]; R(s) = \mathbb{E}[r_t|s_t = s, a_t = a]$$

Model-based vs Model-free: A model is a mathematical representation of the real world which consists of transition dynamics and reward function. The main difference is whether an agent is making a planning(model-based) and get the highest reward by learning(model-free).
Policy: a function that maps from state space to action space:

$$\pi: \boldsymbol{s} \rightarrow \boldsymbol{a}$$

Value function: a value function V^π is the expected sum of discounted rewards under a policy π.

$$V^\pi = \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s]$$

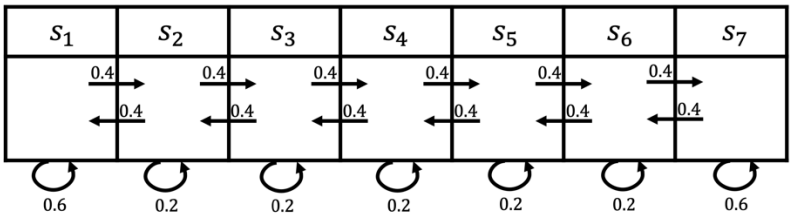
Where $\gamma \in [0,1]$ is the discount factor.

2. Model-based RL

Requirements: The transition dynamics as well as reward function are both known, what the agent to do is make a best planning to maximize the expected rewards.

Markov process: The simplest model which only consists of a sequence of states with randomness. To formalize it, the model has a set of states ($s \in S$) and transition probability which specifies $p(s_{t+1} = s'|s_t = s)$ but no actions and reward function. In most cases, we assume the number of states is finite and the transition probability is not time-varied, i.e. $p(s_{i+1} = s'|s_i = s) = p(s_{j+1} = s'|s_j = s),\ \forall\, s, s' \in S,\ \forall\, i, j = 1,2,\cdots$. In order to compact these single probability, we often have a transition matrix P, whose entry P_{ij} of i-th row and j-th column is specified as $P_{ij} = P(s_i|s_j)$.

e.g.



$$P = \begin{pmatrix} 0.6 & 0.4 & 0 & 0 & 0 & 0 & 0 \\ 0.4 & 0.2 & 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0.2 & 0.4 & 0 & 0 & 0 \\ 0 & 0 & 0.4 & 0.2 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0.2 & 0.4 & 0 \\ 0 & 0 & 0 & 0 & 0.4 & 0.2 & 0.4 \\ 0 & 0 & 0 & 0 & 0 & 0.4 & 0.6 \end{pmatrix}$$

Markov Reward Process: the combination of Markov process and reward function, but still no actions available. In particular, we have a finite set of states ($s \in S$), a transition probability matrix P, a reward function $R(s_t = s) = \mathbb{E}(r_t|s_t = s)$ and discount factor $\gamma \in [0, 1]$. In most cases, we assume the reward gained is stationary, i.e. $r_i = r_j$.

Horizon: the number of time steps in each episode (can be infinite).

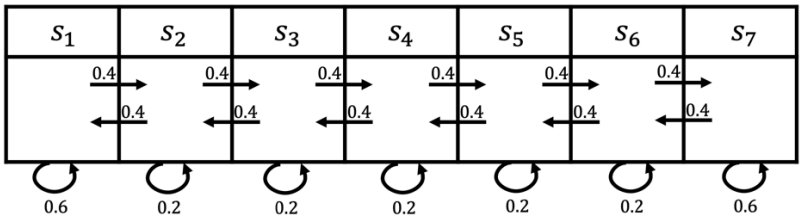
Return: The discounted sum of rewards from time step t to horizon.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

Value function: The expected return from state s.

$$V(s) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots |s_t = s]$$

e.g.



- Reward: $+1$ in s_1, $+10$ in s_7, 0 in all other states
- Value function: expected return from starting in state s

$$V(s) = \mathbb{E}[G_t|s_t = s] = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots |s_t = s]$$

- Sample returns for sample 4-step episodes, $\gamma = 1/2$
  - s_4, s_5, s_6, s_7: $0 + \frac{1}{2} \times 0 + \frac{1}{4} \times 0 + \frac{1}{8} \times 10 = 1.25$
  - s_4, s_4, s_5, s_4: $0 + \frac{1}{2} \times 0 + \frac{1}{4} \times 0 + \frac{1}{8} \times 0 = 0$
  - s_4, s_3, s_2, s_1: $0 + \frac{1}{2} \times 0 + \frac{1}{4} \times 0 + \frac{1}{8} \times 1 = 0.125$
- $V = [1.53\ 0.37\ 0.13\ 0.22\ 0.85\ 3.59\ 15.31]$

Bellman Equation: the value of a state is equal to the immediate reward plus the discounted sum of future reward. To represent the equation by a matrix form, we have:

$$\boldsymbol{V} = \boldsymbol{R} + \gamma \boldsymbol{P}\boldsymbol{V}$$

where $\boldsymbol{V} \in \mathbb{R}^{|S|}$ and $\boldsymbol{R} \in \mathbb{R}^{|S|}$ are vectorized representation of value function and reward function for all states. Thus we can find the analytic solution:

$$\boldsymbol{V} = (\boldsymbol{I} - \gamma \boldsymbol{P})^{-1}\boldsymbol{R}$$

However, the time complexity for analytic solution is $O(|S|^3)$, which is not a desirable algorithm. Iterative algorithm is based on dynamic programming, which initializes $V_0(s) = 0$ for all states s. After that, we iteratively update our value function for all states s:

$$V_k(s) = R(s) + \gamma \sum_{s' \in \boldsymbol{S}} P(s'|s) V_{k-1}(s')$$

For finite horizon, we only iterate the above equation H times, while for infinite horizon, the loop is over when the infinite norm of two consecutive values are smaller than a constant ϵ.

$$\|V_k - V_{k-1}\|_\infty < \epsilon$$

Hence the complexity of iterative algorithm to compute value function is $O(|S|^2)$ for each iteration.

Markov Decision Process: The model is the Markov Reward Process + actions. Concretely, its components have a finite set of states ($s \in S$) and actions ($a \in A$), a transition dynamics for each state-action pair, which specifies $P(s_{t+1} = s'|s_t = s, a_t = a)$, a reward function $R(s_t = s, a_t = a) = \mathbb{E}(r_t|s_t = s, a_t = a)$ and discount factor $\gamma \in [0, 1]$. In most cases, we incorporate policy that maps from state space to action space (can be deterministic or stochastic). Thus, we consider MDP as MRP + policy, where

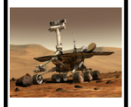
$$R^\pi(s) = \sum_{a \in A} \pi(a|s) R(s, a)$$

$$P^\pi(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s, a)$$

The Bellman equation for MDP is similar with MRP:

$$V_k(s) = R\big(s, \pi(s)\big) + \gamma \sum_{s' \in \boldsymbol{S}} P\big(s' \big| s, \pi(s)\big) V_{k-1}(s')$$

e.g.

s_1	s_2	s_3	s_4	s_5	s_6	s_7
						

- Two actions
- Reward: for all actions, $+1$ in state s_1, $+10$ in state s_7, 0 otherwise
- Let $\pi(s) = a_1 \ \forall s.\ \gamma = 0$.
- What is the value of this policy?
- Recall iterativ e

$$V_k^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s' \in S} p(s'|s, \pi(s)) V_{k-1}^\pi(s')$$

$V^\pi = [1\ 0\ 0\ 0\ 0\ 0\ 10]$

Policy Evaluation:
state value function of a policy:

$$V^{\pi}(s) = R\big(s, \pi(s)\big) + \gamma \sum_{s' \in \boldsymbol{S}} P\big(s' \big| s, \pi(s)\big) V^{\pi}(s')$$

state-action value function of a policy:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in \boldsymbol{S}} P(s'|s,a) V^{\pi}(s')$$

MDP Control: The goal is to find an optimal policy π^* that maximizes expected sum of rewards. Specifically, we have:

$$\pi^* = arg\max_{\pi} V^{\pi}(s)$$

All we need to do is search a best policy, which can be deterministic or stochastic (a probability distribution over actions). In general, if a policy is deterministic, the total number of possible policies is $|A|^{|S|}$, which is inapplicable to search exhaustively.

Policy Iteration:

procedure POLICY ITERATION(M, ϵ)
  $\pi \leftarrow$ Randomly choose a policy $\pi \in \Pi$
  while true **do**
    $V^{\pi} \leftarrow$ POLICY EVALUATION (M, π, ϵ)
    $\pi^* \leftarrow$ POLICY IMPROVEMENT (M, V^{π})
    if $\pi^*(s) = \pi(s)$ **then**
      break
    else
      $\pi \leftarrow \pi^*$
  $V^* \leftarrow V^{\pi}$
  return $V^*(s)$, $\pi^*(s)$ for all $s \in S$

Policy iteration has a monotonically improvement: $V^{\pi^{i+1}}(s) \geq V^{\pi^i}(s), \forall\, s \in S$, thus the Q function does not change again when it is optimal.

Value Iteration:

procedure VALUE ITERATION(M, ϵ)
  For all states $s \in S$, $V'(s) \leftarrow 0$, $V(s) \leftarrow \infty$
  while $||V - V'||_{\infty} > \epsilon$ **do**
    $V \leftarrow V'$
    For all states $s \in S$, $V'(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V(s')\right]$
  $V^* \leftarrow V$ for all $s \in S$
  $\pi^* \leftarrow \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s')\right]$, $\forall\, s \in S$
  return $V^*(s)$, $\pi^*(s)$ for all $s \in S$

In some cases, we often use Bellman backup operator to substitute value function.

$$B^{\pi}V(s) = R^{\pi}(s) + \gamma \sum_{s' \in S} P^{\pi}(s'|s) V(s')$$

Suppose the horizon is infinite, and $\gamma < 1$, Bellman backup operator B is a contraction operator which specifies:

$$\|BV - BV'\|_{\infty} \leq \|V - V'\|_{\infty}$$

Model-based RL: Given a set of experiences, model-based RL first learn a model and then use model-based algorithm to make a good planning. The benefits of that are we can use supervised learning methods to learn a effective model and reason about model uncertainty, but there are two sources of errors (model learning and value function approximation) which is a drawback.

3. Model-free RL

Dynamic Programming: the real-world model is unknown for agent, and the agent should experience to learn a best policy, i.e. needs a large number of simulation to get an optimal policy. Thus, the naïve dynamic programming bootstraps the rest of expected return by the value estimate V_{k-1}.

$$V^{\pi}(s) \approx \mathbb{E}_{\pi}[r_t + \gamma V_{k-1} | s_t = s]$$

procedure POLICY EVALUATION(M, π, ϵ)
  For all states $s \in S$, define $R^{\pi}(s) = \sum_{a \in A} \pi(a|s) R(s,a)$
  For all states $s, s' \in S$, define $P^{\pi}(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s,a)$
  For all states $s \in S$, $V_0(s) \leftarrow 0$
  $k \leftarrow 0$
  while $k = 0$ or $||V_k - V_{k-1}||_{\infty} > \epsilon$ **do**
    $k \leftarrow k + 1$
    For all states $s \in S$, $V_k(s) = R^{\pi}(s) + \gamma \sum_{s' \in S} P^{\pi}(s'|s) V_{k-1}(s')$
  return V_k

Monte Carlo Policy Evaluation: The value function is evaluated by averaging discounted sum of rewards for all states. There is no bootstrapping, no requirement of Markov assumption and each trajectory must be episodic, in other words, each trajectory is a finite MDP. Concretely, the first step is to execute a rollout following a policy π to generate K trajectories, then record the return G_t we observe for each state $s \in S$, and finally, we average them to get Monte Carlo value estimation.

First-visit MC: the return G_t and counting $N(s_t)$ for each state is only computed for the first time we encounter.

procedure FIRST-VISIT-MONTE-CARLO($h_1, \ldots, h_j$)
  For all states s, $N(s) \leftarrow 0$, $S(s) \leftarrow 0$, $V(s) \leftarrow 0$
  for each episode h_j **do**
    for $t = 1, \ldots, L_j$ **do**
      if $s_{j,t} \neq s_{j,u}$ for $u < t$ **then**
        $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
        $S(s_{j,t}) \leftarrow S(s_{j,t}) + G_{j,t}$
        $V^{\pi}(s_{j,t}) \leftarrow S(s_{j,t}) / N(s_{j,t})$
  return V^{π}

Every-visit MC: the return G_t and counting $N(s_t)$ for each state is computed every time we encounter.

procedure EVERY-VISIT-MONTE-CARLO($h_1, \ldots, h_j$)
  For all states s, $N(s) \leftarrow 0$, $S(s) \leftarrow 0$, $V(s) \leftarrow 0$
  for each episode h_j **do**
    for $t = 1, \ldots, L_j$ **do**
      $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
      $S(s_{j,t}) \leftarrow S(s_{j,t}) + G_{j,t}$
      $V^{\pi}(s_{j,t}) \leftarrow S(s_{j,t}) / N(s_{j,t})$
  return V^{π}

By comparison, first-visit MC has an unbiased estimator, while every-visit MC is biased, but both of them are consistent estimator when the number of trajectories goes to infinity, have a high variance.

Alternatively, the value estimator can be modified as the form of moving average.

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

Where $\frac{1}{N(s)}$ can be substituted to learning rate α to adjust the weight of older data. e.g. when $\alpha > \frac{1}{N(s)}$, the value estimator has a larger weight on previous data.

Off-policy MC Policy Evaluation: When there is a probability distribution $p(x)$ that is hard to be sampled, if we know another distribution $q(x)$ that is easy to get those samples, we can reweight them to compute empirical average.

$$\begin{aligned} V^{\pi_1}(s) &= \mathbb{E}_{h\sim\pi_1}[G(h)] \\ &= \mathbb{E}_{h\sim\pi_2}\left[\frac{p(h|\pi_1,s)}{p(h|\pi_2,s)}G(h)\right] \\ &\approx \frac{1}{n}\sum_{j=1}^{n}\frac{p(h_j|\pi_1,s)}{p(h_j|\pi_2,s)}G(h_j) \\ &= \frac{1}{n}\sum_{j=1}^{n}G(h_j)\prod_{t=1}^{L_j}\frac{\pi_1(a_{j,t}|s_{j,t})}{\pi_2(a_{j,t}|s_{j,t})} \end{aligned}$$

Temporal Difference Policy Evaluation: TD learning is a combination of sampling and bootstrapping, so our value function estimation equation is:

$$V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha\big(r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\big)$$

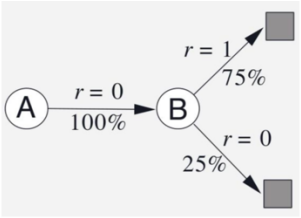
where $r_t + \gamma V^{\pi}(s_{t+1})$ is sampled from the real world.

procedure TDLEARNING(step size α, number of trajectories n)
  For all states s, $V^{\pi}(s) \leftarrow 0$
  while $n > 0$ **do**
    Begin episode E at state s
    while $n > 0$ and episode E has not terminated **do**
      $a \leftarrow$ action at state s under policy π
      Take action a in E and observe reward r, next state s'
      $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(R + \gamma V^{\pi}(s') - V^{\pi}(s))$
      $s \leftarrow s'$
  return V^{π}

TD learning is a low-biased but smaller variance method.

Batch MC and TD: applies to finite dataset, where given a set of K episodes, the algorithm repeatedly sample an episode from the dataset, then update MC and TD based on the episode.

e.g.



Suppose we have 8 trajectories:

$$h_1 = (A, 0, B, 0)$$
$$h_{2-7} = (B, 1)$$
$$h_8 = (B, 0)$$

Then the value of state B for both MC and TD is 0.75 ($V(B) = 0.75$), while $V(A)$ is different, for MC $V(A) = 0$, but for TD $V(A) = 0.75$, since $V(A) = 0 + \gamma V(B) = 0.75$. In terms of convergence, MC in batch setting converges to minimum mean square error, while TD learning converges to maximum likelihood estimates:

$$\hat{P}(s'|s,a) = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{L_k-1}\mathbb{1}\left(s_{k,t} = s, a_{k,t} = a, s_{k,t+1} = s'\right)$$

$$\hat{r}(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{L_k - 1} \mathbb{1}(s_{k,t} = s, a_{k,t} = a) r_{k,t}$$

Pros & Cons of DP, MC and TD:

	Dynamic Programming	Monte Carlo	Temporal Difference
Model Free?	No	Yes	Yes
Non-episodic domains?	Yes	No	Yes
Non-Markovian domains?	No	Yes	No
Converges to true value	Yes	Yes	Yes
Unbiased Estimate	N/A	Yes	No
Variance	N/A	High	Low

Model-free Control: the goal is to learn a best policy based on value or Q function by sampling trajectories or bootstrapping and then estimating state value for each state. In some cases, when MDP model is known but it is computationally infeasible, sample-based methods can be applied to address that issue.

Exploration: policy iteration algorithm always chooses an action that maximizes Q function for each step, being stuck in local optima, and less chance to explore other policies that have smaller immediate reward but substantial future rewards. Hence, we have ϵ-greedy policies which specifies:

$$\pi(a|s) = \begin{cases} arg\max_a Q^{\pi}(s,a) & with\ probability\ 1-\epsilon \\ random\ action\ a & with\ probability\ \frac{\epsilon}{|A|} \end{cases}$$

ϵ-greedy policy is a monotonic improvement:

$$\begin{aligned} Q^{\pi_i}(s, \pi_{i+1}(s)) &= \sum_{a \in A} \pi_{i+1}(a|s) Q^{\pi_i}(s,a) \\ &= \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s,a) + (1-\epsilon) \max_{a'} Q^{\pi_i}(s,a') \\ &= \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s,a) + \sum_{a \in A} (\pi_i(a|s) - \frac{\epsilon}{|A|}) \max_{a'} Q^{\pi_i}(s,a') \\ &\geq \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s,a) + \sum_{a \in A} (\pi_i(a|s) - \frac{\epsilon}{|A|}) Q^{\pi_i}(s,a) \\ &= \sum_{a \in A} \pi_i(a|s) Q^{\pi_i}(s,a) \\ &= V^{\pi_i}(s) \end{aligned}$$

Greedy in the Limit of Exploration (GLIE): To ensure the balance between exploration and exploitation policy iteration converge, we assume that the algorithm satisfies the following two properties:

1. All state-action pairs are visited an infinite number of times, i.e.
$$\lim_{i \to \infty} N_i(s,a) \to \infty$$
2. The behavior policy converges to greedy policy with respect to learned Q function.
$$\lim_{i \to \infty} \pi_i(a|s) = arg\max_a Q(s,a)\ \ with\ the\ probability\ of\ 1$$

Monte Carlo Online Control: use Monte Carlo to estimate Q function, and ϵ-greedy method to improve policy.

procedure ONLINE MONTE CARLO CONTROL
  Initialize $Q(s,a) = 0$, $Returns(s,a) = 0$ for all $s \in S, a \in A$
  Set $\epsilon \leftarrow 1$, $k \leftarrow 1$
  loop
    Sample kth episode $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \ldots, s_T)$ under policy π
    for $t = 1, \ldots, T$ **do**
      if First visit to (s,a) in episode k **then**
        Append $\sum_{j=t}^{T} r_{kj}$ to $Returns(s_t, a_t)$
        $Q(s_t, a_t) \leftarrow average(Returns(s_t, a_t))$
    $k \leftarrow k + 1$, $\epsilon = \frac{1}{k}$
    $\pi_k = \epsilon$-greedy with respect to Q (policy improvement)
  Return Q, π

When the number of sampled trajectories goes to infinity and $\epsilon = \frac{1}{n}$, it satisfies GLIE and hence GLIE Monte Carlo control converges to optimal state-action value function $Q(s,a) \rightarrow Q^*(s,a)$.

Temporal Difference Learning Control: use TD learning to estimate state-action value function (can be on-policy or off-policy) and then improve policy by applying ϵ-greedy.

SARSA: the bootstrapping step for estimated future reward $\gamma Q(s_{t+1}, a_{t+1})$ is sampled and experienced from the real-world, thus SARSA is an on-policy policy iteration. Each time the action used to evaluation and take is guaranteed to be the same.

procedure SARSA(ϵ, α_t)
  Initialize $Q(s,a)$ for all $s \in S, a \in A$ arbitrarily except $Q(terminal, \cdot) = 0$
  $\pi \leftarrow \epsilon$-greedy policy with respect to Q
  for each episode **do**
    Set s_1 as the starting state
    Choose action a_1 from policy $\pi(s_1)$
    loop until episode terminates
      Take action a_t and observe reward r_t and next state s_{t+1}
      Choose action a_{t+1} from policy $\pi(s_{t+1})$
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
      $\pi \leftarrow \epsilon$-greedy with respect to Q (policy improvement)
      $t \leftarrow t + 1$
  Return Q, π

SARSA is guaranteed to converge if the sequence of policies satisfies GLIE and step size α_t satisfies Robbins Munro sequence such that:

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

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

Q-learning: An off-policy policy improvement algorithm. For each step, the future reward estimator always selects an action that maximizes Q function, so the action taken under the current policy and the maximum-value action is not always the same.

procedure Q-LEARNING(ϵ, α, γ)
  Initialize $Q(s,a)$ for all $s \in S, a \in A$ arbitrarily except $Q(terminal, \cdot) = 0$
  $\pi \leftarrow$ ϵ-greedy policy with respect to Q
  for each episode **do**
    Set s_1 as the starting state
    $t \leftarrow 1$
    loop until episode terminates
      Sample action a_t from policy $\pi(s_t)$
      Take action a_t and observe reward r_t and next state s_{t+1}
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$
      $\pi \leftarrow$ ϵ-greedy policy with respect to Q (policy improvement)
      $t \leftarrow t + 1$
  return Q, π

Q-learning is guaranteed to converge to optimal Q^* if the step size α_t satisfies Robbins Munro sequence such that:

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

Guaranteed to converge to optimal policy π^* if the sequence of policies satisfies GLIE.

Maximization Bias: When applying Q-learning, sometimes choosing an action that maximizes estimated Q function will overestimate value function, thus decreasing the performance of Q-learning. E.g. Consider a MDP with a single state and 2 actions. Both of them have random rewards with 0-means ($\mathbb{E}(r|a = a_1) = \mathbb{E}(r|a = a_1) = 0$), thus $Q(s, a_1) = Q(s, a_1) = V(s) = 0$. If we use sample-based method (e.g. Monte Carlo) to estimate state-action value function, then we get $\hat{Q}(s, a_1)$, $\hat{Q}(s, a_2)$. Let $\hat{\pi} = arg\max_a \hat{Q}(s, a)$ be the greedy policy with respect to estimated $\hat{Q}$. Due to the randomness and limited number of samples, the value function estimate under the policy $\hat{\pi}$ can be overestimated.

$$\begin{aligned}\hat{V}(s) &= \mathbb{E}\left[max\left(\hat{Q}(s, a_1), \hat{Q}(s, a_2)\right)\right] \\ &\geq max\left(\mathbb{E}\left[\hat{Q}(s, a_1)\right], \mathbb{E}\left[\hat{Q}(s, a_2)\right]\right) \\ &= max(0, 0) \\ &= 0 = V^*(s)\end{aligned}$$

Double Q-learning: The key idea is instead of a single Q function, double Q-learning uses two separate Q functions $\hat{Q}_1(s, a_i)$, $\hat{Q}_2(s, a_i)$ $\forall$ a. For each iteration, the policy improvement stage is taken based on the maximum Q value of the addition of $\hat{Q}_1$ and $\hat{Q}_2$, while each Q value estimator is updated by choosing an action that maximizes current Q function, but being evaluated by another Q function.

procedure DOUBLE Q-LEARNING(ϵ, α, γ)
  Initialize $Q_1(s,a), Q_2(s,a)$ for all $s \in S, a \in A$, set $t \leftarrow 0$
  $\pi \leftarrow$ ϵ-greedy policy with respect to $Q_1 + Q_2$
  loop
    Sample action a_t from policy π at state s_t
    Take action a_t and observe reward r_t and next state s_{t+1}
    if (with 0.5 probability) **then**
      $Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_t + \gamma Q_2(s_{t+1}, \arg\max_{a'} Q_1(s_{t+1}, a')) - Q_1(s_t, a_t))$
    else
      $Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha(r_t + \gamma Q_1(s_{t+1}, \arg\max_{a'} Q_2(s_{t+1}, a')) - Q_2(s_t, a_t))$
    $\pi \leftarrow$ ϵ-greedy policy with respect to $Q_1 + Q_2$ (policy improvement)
    $t \leftarrow t + 1$
  return $\pi, Q_1 + Q_2$

4. Value Function Approximation

Motivation: we know that each state-action value must be stored into a table. However, when the state space goes to infinite or a huge number, it is undesirable to record all state-action pair into a table and choose the maximum one. To remedy that, we use an alternative Q function parameterized by weights $\boldsymbol{w}$ to generalize true values.

$$V(s) \approx \hat{V}(s;\boldsymbol{w})$$
$$Q(s,a) \approx \hat{Q}(s,a;\boldsymbol{w})$$

Where the parameters $\boldsymbol{w}$ can be a vector (linear combination) or a bunch of vectors (neural network).

Linear Feature Representation: for each state s, we use a feature vector to represent it:

$$x(s) = [x_1(s)\ x_2(s) \cdots x_n(s)]$$

The approximated value function of a particular state is the linear combination of $x(s)$ and $\boldsymbol{w}$.

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

So the distance between estimated Q value and true value is evaluated by mean square error:

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

Gradient with respect to each parameter of weight vector $\boldsymbol{w}$ is:

$$\begin{aligned}\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &= \mathbb{E}_\pi\left[2\left(V(s) - \hat{V}(s;\boldsymbol{w})\right)\right]\nabla_{\boldsymbol{w}} V(s) \\ &= \mathbb{E}_\pi\left[2\left(V(s) - \hat{V}(s;\boldsymbol{w})\right)\right]x(s)\end{aligned}$$

Updating parameter $\boldsymbol{w}$ by a small step $-\frac{1}{2}\alpha$:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\mathbb{E}_\pi\left[V(s) - \hat{V}(s;\boldsymbol{w})\right]x(s)$$

VFA with Oracle: each true state value $V^\pi(s)$ can be referred in an Oracle. Our goal is to learn a set of parameters $\boldsymbol{w}$ to minimize the mean square error, such that

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

VFA with MC: If there is no Oracle, the true state value $V^\pi(s)$ would be substituted to unbiased estimate G_t, so the objective function is:

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

VFA with TD: TD learning with bootstrapping is another approximator to the true state value. Hence, $V^\pi(s)$ is replaced with $r_t + \gamma\hat{V}(s';\boldsymbol{w})$:

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[(r_t + \gamma\hat{V}(s';\boldsymbol{w}) - \hat{V}(s;\boldsymbol{w}))^2]$$

Convergence Guarantees for VFA: Let define the mean square error of a linear value function approximation be the naïve VFA weighted with stationary distribution over states.

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

Monte Carlo policy evaluation with VFA converges to weights $\boldsymbol{w}_{MC}$ which has minimum mean square error:

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

TD learning policy evaluation with VFA converges to weights $\boldsymbol{w}_{TD}$ which has minimum mean square error within a constant factor:

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

VFA with Control: use function approximation to represent each state-action pair. Similarly, each state-action pair is represented as a feature vector:

$$x(s,a) = [x_1(s,a)\ x_2(s,a) \cdots x_n(s,a)]$$

Then apply it to estimate a value function and use gradient descent to update parameter $\boldsymbol{w}$.

Object function for different methods:

① state-action VFA control with an Oracle

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

② state-action VFA control with MC

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

③ state-action VFA control with on-policy TD (SARSA)

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[(r_t + \gamma\hat{Q}(s',a';\boldsymbol{w}) - \hat{Q}(s,a;\boldsymbol{w}))^2]$$

④ state-action VFA control with off-policy TD (Q-learning)

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[(r_t + \gamma \max_{a'} \hat{Q}(s',a';\boldsymbol{w}) - \hat{Q}(s,a;\boldsymbol{w}))^2]$$

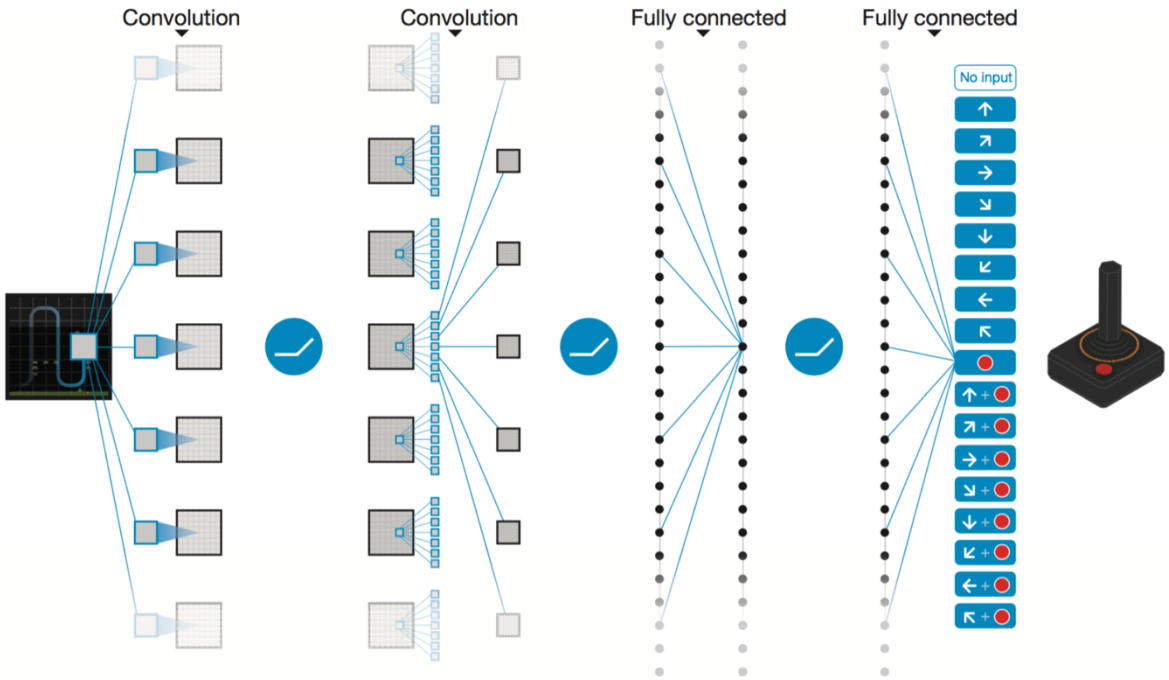
Note that the convergence of all these algorithms are guaranteed except Q-learning.

Algorithm	Tabular	Linear VFA	Nonlinear VFA
Monte-Carlo Control	Yes	(Yes)	No
SARSA	Yes	(Yes)	No
Q-learning	Yes	No	No

5. Deep Q-learning

Motivation: sometimes a single linear combination of feature vector and weight parameter cannot approximate a complex function, Deep Q-learning is one method that represent a function by using a deep neural network (multi-layer perceptron).

Deep Q-Network: In general, Deep Q-Network has several components: First, we need a feature extractor (CNN) that extracts features from a pile of raw pixels (current state); then apply Q-learning to select the best valid action from action space.



However, Deep Q-learning with VFA can diverge when each time choosing an action with maximum Q value. The reason causing this issue is correlations between samples and non-stationary targets. To address these challenges, experience replay and fixed Q-targets are applied.

Experience Replay: To remove correlation issue, each time the agent experience, we store the tuple (s, a, r, s') into a replay buffer $\mathcal{D}$. Then perform replay stage which repeatedly sample the previous experience tuple from replay

buffer: $(s, a, r, s') \sim \mathcal{D}$, then compute target value of state s: $r_t + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w})$, and finally use SGD to update function parameter $\boldsymbol{w}$ by: $\Delta \boldsymbol{w} = \alpha(r_t + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$.

Fixed Q-targets: To help improve stability during training period, we fix estimated Q-target value $r_t + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}^-)$, where $\boldsymbol{w}^-$ is another parameter that keep unchanged for a certain period of time. Thus, the parameter updating is written as: $\Delta \boldsymbol{w} = \alpha(r_t + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}^-) - \hat{Q}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$.

Initialize replay memory D with a fixed capacity
Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
Initialize target action-value function $\hat{q}$ with weights $\mathbf{w}^- = \mathbf{w}$
for episode $m = 1, \ldots, M$ **do**
    Observe initial frame x_1 and preprocess frame to get state s_1
    for time step $t = 1, \ldots, T$ **do**
        Select action $a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a \hat{q}(s_t, a, \mathbf{w}) & \text{otherwise} \end{cases}$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Preprocess s_t, x_{t+1} to get s_{t+1}, and store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample uniformly a random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from D
        Set $y_j = r_j$ if episode ends at step $j + 1$;
        otherwise set $y_j = r_j + \gamma \max_{a'} \hat{q}(s_{j+1}, a', \mathbf{w}^-)$
        Perform a stochastic gradient descent step on $J(\mathbf{w}) = \frac{1}{N} \sum_{j=1}^{N} (y_j - \hat{q}(s_j, a_j, \mathbf{w}))^2$ w.r.t. parameters $\mathbf{w}$
        Every C steps reset $\mathbf{w}^- = \mathbf{w}$

Double DQN: The extensive idea to DQN, where we use two separate Q function to compute the derivative of current network parameter $\boldsymbol{w}$. The current Q-network $\boldsymbol{w}$ is to select actions while the older Q-network $\boldsymbol{w}^-$ is to evaluate actions:

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

Prioritized Experience Replay: The order of episodic replay updates has an impact on performance, since Q-learning only bootstraps the next state's Q value, thus the order of choosing prior experiences will be considered. Concretely, let p_i be the TD error of state s_i which is proportional to priority.

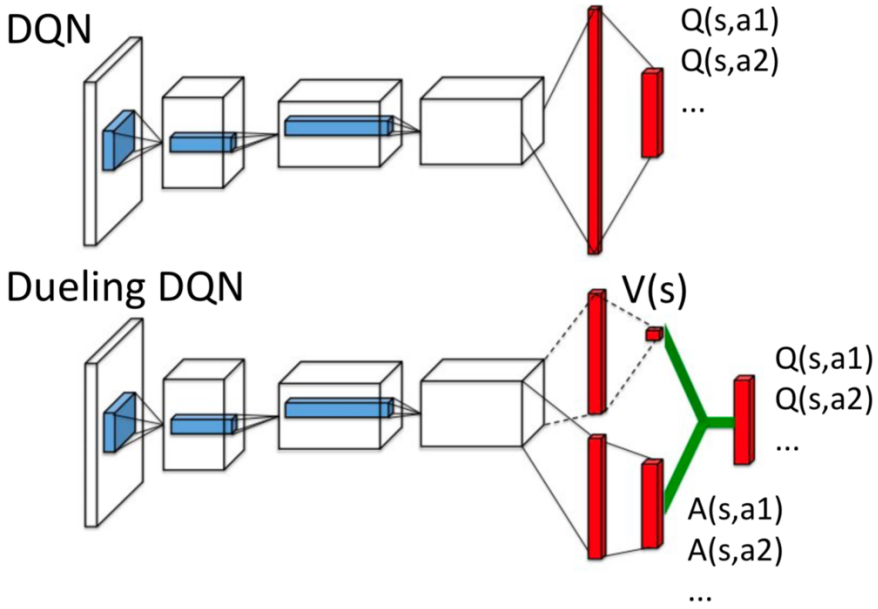
$$p_i = \left| r_t + \gamma \max_{a'} Q(s_{i+1}, a'; \boldsymbol{w}^-) - Q(s_i, a_i; \boldsymbol{w}) \right|$$

The probability for each tuple being selected is normalized to:

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

where α is a hyperparameter, e.g. when $\alpha = 0$, the distribution is uniform.

Dueling DQN: Rather than predicting each Q value for each action of a state, dueling DQN separates the DQN into two components. One is output a value function $V(s)$, and another is predict advantage function $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$, which relates value and Q functions.



Nevertheless, the advantage function $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$ is not identifiable, i.e., we cannot recover $Q^{\pi}(s,a)$ and $V^{\pi}(s)$, so we cannot tell $Q^{\pi}(s,a)$ and $V^{\pi}(s)$ are in good performance only based on a single advantage function. Thus, we modify our Q value function to either the following two forms:
Form 1:

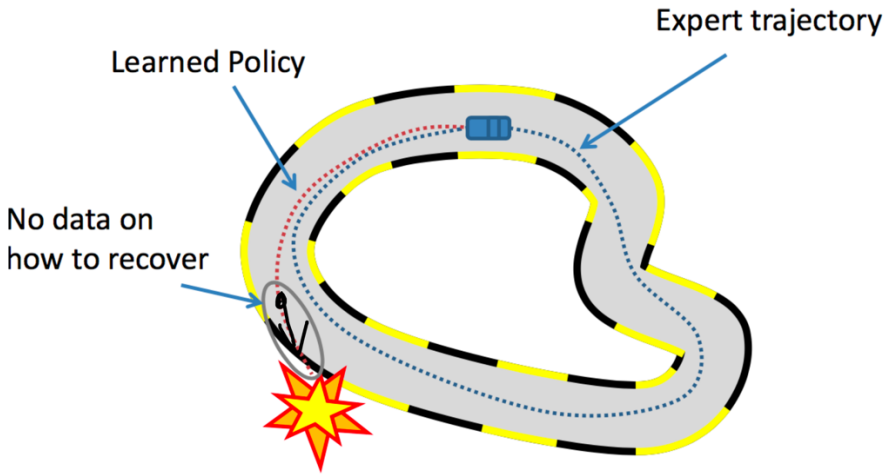
$$\hat{Q}(s,a;\boldsymbol{w}) = \hat{V}(s;\boldsymbol{w}) + (\hat{A}(s,a;\boldsymbol{w}) - \max_{a' \in A} \hat{A}(s,a';\boldsymbol{w}))$$

Form 2:

$$\hat{Q}(s,a;\boldsymbol{w}) = \hat{V}(s;\boldsymbol{w}) + (\hat{A}(s,a;\boldsymbol{w}) - \frac{1}{|A|}\sum_{a'} \hat{A}(s,a';\boldsymbol{w})$$

6. Imitation Learning

Motivation: Previously, we use reward model to learn an optimal policy. However, when our rewards are very sparse or having a complicated reward function, or not tolerable to fail (autonomous vehicle), imitation learning is an approach to let our agent learn a good policy directly from a preset supervisor, i.e., learning from demonstration trajectories. Hence, it is unnecessary for the expert demonstration to specify a reward to be generated in such behavior and the explicit desired policy.
Prerequisite: a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition model $P(s'|s,a)$, a set of expert demonstrations $(s_0, a_0, s_1, a_1, \cdots)$ drawn from expert's best policy, no explicit reward function.
Behavioral Cloning: In behavioral cloning, learning an optimal policy can be treated as a supervised learning problem, where given a state-action pair dataset $\{(s_0, a_0), (s_1, a_1), \cdots\}$, the agent is required to learn a policy that minimizes training error. However, in supervised learning, data is assumed to be i.i.d., which does not hold in MDP. In such cases, when an error occurs, the agent may be in a state that the expert never visited, which may cause catastrophe mistakes (compounding error).



To mitigate compounding errors, one solution is aggregate dataset by providing more labels of the expert action taken by current behavior policy.

Initialize $\mathcal{D} \leftarrow \emptyset$
Initialize $\hat{\pi}_1$ to any policy in Π
for $i = 1$ to N **do**
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$
    Sample T-step trajectories using π_i
    Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by π_i and actions given by expert
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$
    Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$
return best $\hat{\pi}_i$ on validation

Inverse RL: The goal is to infer a reward function R based on the previous setting (state space, action space, transition model and a set of teacher trajectories), assuming that teacher's policy is optimal. To this end, the simplest idea is that consider the reward of a state is the linear combination of features:

$$R(s) = \boldsymbol{w}^T x(s)$$

Therefore, the value function can be represented as:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t) \,|s_0 = s] \\ &= \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t \boldsymbol{w}^T x(s) \,|s_0 = s] \\ &= \boldsymbol{w}^T \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t x(s) \,|s_0 = s] \\ &= \boldsymbol{w}^T \mu(\pi) \end{aligned}$$

where $\mu(\pi)$ is the discounted weighted frequency of state features $x(s)$. To satisfy the preset assumption, the value of teacher's policy π^* is better than any other policies π.

$$\mathbb{E}_{\pi^*}[\sum_{t=0}^{\infty} \gamma^t R^*(s_t) \,|s_0 = s] \geq \mathbb{E}_{\pi}[\textstyle\sum_{t=0}^{\infty} \gamma^t R(s_t) \,|s_0 = s] \;\; \forall\, \pi$$

where $R^*(\cdot)$ denotes an optimal reward function. Thus, we can find an optimal weight vector $\boldsymbol{w}^*$ such that:

$$\boldsymbol{w}^{*T} \mu(\pi^* | s_0 = s) \geq \boldsymbol{w}^{*T} \mu(\pi | s_0 = s)$$

Apprenticeship Learning: Given the optimal reward function, find a policy π that performs equally well to optimal policy π^*.

1: Initialize policy π_0
2: **for** $i = 1, 2, \ldots$ **do**
3: Find reward function weights w such that the teacher maximally outperforms all previous controllers:

$$\begin{aligned} &\arg\max_{w} \max_{\gamma} \gamma \\ &\text{s.t. } w^T \mu(\pi^* \mid s_0 = s) \geq w^T \mu(\pi \mid s_0 = s) + \gamma, \ \forall \pi \in \{\pi_0, \pi_1, \ldots, \pi_{i-1}\}, \forall s \\ &\quad \|w\|_2 \leq 1 \end{aligned}$$

4: Find optimal policy π_i for current w
5: **if** $\gamma \leq \epsilon/2$ **then return** π_i

7. Policy Gradient

Motivation: Value-based function approximator $V^{\pi}(s; \boldsymbol{w})$ or $Q^{\pi}(s, a; \boldsymbol{w})$ is undesirable when there are huge state and action space, so a direct parameterized policy estimator $\pi_{\boldsymbol{\theta}}(s, a) = \mathbb{P}(a|s; \boldsymbol{\theta})$ is applied to find a best policy π^* that maximizes value function.

Pros & Cons: Policy gradient has better convergence properties, can be used in high-dimensional or continuous action spaces and learn stochastic policies; but typically it tends to converge to local optima and has a high variance during evaluation.

Objective Function:

episodic case: the value of a policy in terms of episodic case is the value of the start state.

$$J_1(\boldsymbol{\theta}) = V^{\pi_{\boldsymbol{\theta}}}(s_1)$$

continuous case (average value):

$$J_{avV}(\boldsymbol{\theta}) = \sum_s d^{\pi_{\boldsymbol{\theta}}}(s) V^{\pi_{\boldsymbol{\theta}}}(s)$$

continuous case (average reward):

$$J_{avV}(\boldsymbol{\theta}) = \sum_s d^{\pi_{\boldsymbol{\theta}}}(s) \sum_a \pi_{\boldsymbol{\theta}}(a|s) R(s, a)$$

Policy Optimization: find a parameter $\boldsymbol{\theta}$ that maximizes $V^{\pi_{\boldsymbol{\theta}}}$, the optimization method can be gradient-based that goes to the local optima along the steepest gradient of loss function. For each time step, $\Delta\boldsymbol{\theta} = \alpha \nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta})$, where α is the learning rate and $\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta})$ is the gradient with respect to each parameter.

① Finite difference: perturb $\boldsymbol{\theta}$ in k-th dimension by a small amount to measure the rate of change in that dimension, thus we can estimate the gradient.

$$\frac{\partial V^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k} \approx \frac{V^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\theta} + \epsilon u_k) - V^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\theta})}{\epsilon}$$

where u_k is a unit vector with 1 in k-th dimension and 0 otherwise.

This method does not require objective function being differentiable but inaccurate and computationally expensive.

② Analytics: if objective function $V^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\theta})$ is differentiable, we can solve the gradient analytically. We have our objective function:

$$V^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\theta}) = \mathbb{E}_{(s_t, a_t) \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{\infty} R(s_t, a_t) \right] = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[R(\tau)] = \sum_{\tau} P(\tau; \boldsymbol{\theta}) R(\tau)$$

where τ is a trajectory following policy $\pi_{\boldsymbol{\theta}}$, $R(\tau)$ is the sum of rewards for a trajectory and $P(\tau; \boldsymbol{\theta})$ represents the probability over trajectories under the policy $\pi_{\boldsymbol{\theta}}$. In such case, our goal is to find an optimal parameter such that:

$$\boldsymbol{\theta}^* = arg\, \underset{\boldsymbol{\theta}}{max} \sum_{\tau} P(\tau; \boldsymbol{\theta}) R(\tau)$$

When applying gradient descent algorithm to value function we get:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} V^{\pi_\theta}(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \sum_{\tau} P(\tau;\boldsymbol{\theta}) R(\tau) \\ &= \sum_{\tau} \nabla_{\boldsymbol{\theta}} P(\tau;\boldsymbol{\theta}) R(\tau) \\ &= \sum_{\tau} \frac{P(\tau;\boldsymbol{\theta})}{P(\tau;\boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} P(\tau;\boldsymbol{\theta}) R(\tau) \\ &= \sum_{\tau} \nabla_{\boldsymbol{\theta}} \log P(\tau;\boldsymbol{\theta})\, P(\tau;\boldsymbol{\theta}) R(\tau) \\ &\approx \frac{1}{n} \sum_{i=1}^{n} R(\tau^{(i)})\, \nabla_{\boldsymbol{\theta}} \log P\left(\tau^{(i)};\boldsymbol{\theta}\right)\end{aligned}$$

where $\nabla_{\boldsymbol{\theta}} \log P\left(\tau^{(i)};\boldsymbol{\theta}\right)$ is called score function and the last equation is to estimate gradient by empirical average. The gradient with respect to score function can be decomposed to:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \log P\left(\tau^{(i)};\boldsymbol{\theta}\right) &= \nabla_{\boldsymbol{\theta}} \log \left(\mu(s_0) \prod_{t=0}^{T-1} \pi_{\boldsymbol{\theta}}(a_t|s_t) P(s_{t+1}|s_t, a_t) \right) \\ &= \nabla_{\boldsymbol{\theta}} \left(\log \mu(s_0) + \sum_{t=0}^{T-1} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) + \log P(s_{t+1}|s_t, a_t) \right)\end{aligned}$$

where $\mu(s_0)$ is initial state distribution and $P(s_{t+1}|s_t, a_t)$ is dynamics model which does not contribute to the gradient, thus we have:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\theta}) &= \frac{1}{n} \sum_{i=1}^{n} R(\tau^{(i)})\, \nabla_{\boldsymbol{\theta}} \log P\left(\tau^{(i)};\boldsymbol{\theta}\right) \\ &= \frac{1}{n} \sum_{i=1}^{n} R(\tau^{(i)}) \sum_{t=0}^{T-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\left(a_t^{(i)}|s_t^{(i)}\right)\end{aligned}$$

Even though the above equation is unbiased, the results can be noisy. To mitigate this issue, we use temporal structure. Therefore, the gradient estimator for a single reward $r_{t'}$ is:

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}[r_{t'}] = \mathbb{E}\left[r_{t'} \sum_{t=0}^{t'} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \right]$$

Then extend to all t':

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

REINFORCE Algorithm: Use Monte Carlo method to sample trajectories and apply policy gradient to find an optimal policy.

procedure REINFORCE(α)
  Initialize policy parameters θ arbitrarily
  for each episode $\{s_1, a_1, r_2, \cdots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T-1$ **do**
      $\theta \leftarrow \theta + \alpha \cdot G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$
  return θ

Other Differentiable Classes:

Softmax:

$$\pi_{\boldsymbol{\theta}}(a|s) = \frac{e^{\phi(s,a)^T \boldsymbol{\theta}}}{\sum_{a'} e^{\phi(s,a')^T \boldsymbol{\theta}}}$$

$$\nabla_{\boldsymbol{\theta}} log\, \pi_{\boldsymbol{\theta}}(a|s) = \phi(s,a) - \mathbb{E}_{a' \sim \pi_{\boldsymbol{\theta}}(a'|s)}[\phi(s,a')]$$

Gaussian:

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

which means policy is sampled from Gaussian distribution. The mean value is a linear combination of state features:

$$\mu(s) = \phi(s)^T \boldsymbol{\theta}$$

gradient:

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

Baseline: To reduce high variance of Monte-Carlo method, we introduce baseline, which is a function of a state $b(s)$ so the gradient of objective function is modified to:

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[R(\tau)] = \sum_{t=0}^{T-1} \nabla_{\boldsymbol{\theta}} log\, \pi_{\boldsymbol{\theta}}(a_t|s_t) \left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t) \right)$$

Since the additional term $b(s)$ does not introduce any bias, we let $b(s)$ be value function $V^\pi(s)$, then $\sum_{t'=t}^{T-1} r_{t'} - b(s_t)$ is the advantage function $A(t)$ of time step t.

N-step Estimators: Another approach to reduce variance is use TD method but bootstrap multiple steps instead.

$$\hat{A}_t^{(i)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^i r_{t+i} - V(s_t)$$

when $i = 1$, the advantage function is a pure TD estimate, which has low variance but high bias; the larger of i, the lower bias and higher variance, so we can choose an intermediate value k to balance between bias and variance.

Vanilla Policy Gradient:

procedure POLICY GRADIENT(α)
  Initialize policy parameters θ and baseline values $b(s)$ for all s, e.g. to 0
  for iteration $= 1, 2, \ldots$ **do**
    Collect a set of m trajectories by executing the current policy π_θ
    for each time step t of each trajectory $\tau^{(i)}$ **do**
      Compute the *return* $G_t^{(i)} = \sum_{t'=t}^{T-1} r_{t'}$
      Compute the *advantage estimate* $\hat{A}_t^{(i)} = G_t^{(i)} - b(s_t)$
    Re-fit the baseline to the empirical returns by updating $\mathbf{w}$ to minimize

$$\sum_{i=1}^{m} \sum_{t=0}^{T-1} \| b(s_t) - G_t^{(i)} \|^2$$

    Update policy parameters θ using the policy gradient estimate $\hat{g}$

$$\hat{g} = \sum_{i=1}^{m} \sum_{t=0}^{T-1} \hat{A}_t^{(i)} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})$$

    with an optimizer like SGD ($\theta \leftarrow \theta + \alpha \cdot \hat{g}$) or Adam
  return θ and baseline values $b(s)$

Suppose the baseline values $b(s)$ is an estimated function parameterized by $\boldsymbol{w}$.

Step Size Tuning: Step size is crucial because a long step may cause a very bad policy, so that it affects future data collection and exploration and exploitation trade-off and the bad data leads to an even worse policy. When evaluating a policy, we often calculate the expected return of a policy.

$$V^{\pi}(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t); \pi_{\boldsymbol{\theta}}\right]$$

Expressing another policy is the addition of the value of the current policy and expected advantage function.

$$\begin{aligned} V^{\tilde{\pi}}(\tilde{\boldsymbol{\theta}}) &= V^{\pi}(\boldsymbol{\theta}) + \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t); \pi_{\boldsymbol{\theta}}\right] \\ &= V^{\pi}(\boldsymbol{\theta}) + \sum_{s} \mu_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a|s) A_{\pi}(s, a) \end{aligned}$$

However, the discounted weighted frequency of state s is unknown.

Local Approximation: Redefine our objective function by substituting $\mu_{\tilde{\pi}}(s)$ to discounted weighted frequency of current policy $\mu_{\pi}(s)$.

$$L_{\pi}(\tilde{\pi}) = V^{\pi}(\boldsymbol{\theta}) + \sum_{s} \mu_{\pi}(s) \sum_{a} \tilde{\pi}(a|s) A_{\pi}(s, a)$$

Note that $L_{\pi}(\pi) = V^{\pi}(\boldsymbol{\theta})$.

Lower Bound: Consider a new policy is a mixture of an old policy and a different policy with probability α and $1 - \alpha$:

$$\pi_{new}(a|s) = (1 - \alpha)\pi_{old}(a|s) + \alpha\pi'(a|s)$$

In this case, the lower bound of new policy π_{new} is:

$$V^{new} \geq L_{\pi_{old}}(\pi_{new}) - \frac{2\epsilon\gamma}{(1-\gamma)^2}\alpha^2$$

where $\epsilon = \max_{s} \left|\mathbb{E}_{a\sim\pi'(a|s)}[A_{\pi}(s, a)]\right|$. Replacing to KL divergence, we have:

$$V^{new} \geq L_{\pi_{old}}(\pi_{new}) - \frac{4\epsilon\gamma}{(1-\gamma)^2} D_{KL}^{max}(\pi_{old}, \pi_{new})$$

where $D_{KL}^{max}(\pi_{old}, \pi_{new}) = \max_{s} D_{KL}(\pi_1(\cdot\,|s), \pi_2(\cdot\,|s))$.

Trust Region: replace $\frac{4\epsilon\gamma}{(1-\gamma)^2}$ to a coefficient C which is also called penalty. In order to take a large step, our goal is to maximize the first term $L_{\pi_{old}}(\pi_{new})$ and constrain the second term subject to:

$$D_{KL}^{s\sim\mu_{\boldsymbol{\theta}_{old}}}(\pi_{old}, \pi_{new}) \leq \delta$$

Note that KL divergence term has replaced from choosing the maximum to average.

TPRO algorithm:

1: **for** iteration=1, 2, . . . **do**
2: Run policy for T timesteps or N trajectories
3: Estimate advantage function at all timesteps
4: Compute policy gradient g
5: Use CG (with Hessian-vector products) to compute $F^{-1}g$ where F is the Fisher information matrix
6: Do line search on surrogate loss and KL constraint
7: **end for**

8. Fast Reinforcement Learning

Motivation: Empirically, we use asymptotic convergence rates to measure an algorithm's performance, which is not desirable in many real applications (e.g. education, healthcare, robotics). To achieve good real-world performance, we want rapid convergence to best policies.

Multi-Armed Bandits (MAB): An MAB is represented as a tuple $(\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ is a set of actions and $\mathcal{R}$ denotes to a collection of unknown probability distribution, where each action corresponds a particular distribution over rewards $\mathcal{R}^a(r) = \mathbb{P}(r|a)$. Each time the agent chooses an action and observes a reward which is sampled from distribution $r_t \sim \mathcal{R}^{a_t}$, and the goal is to maximize cumulative rewards $\sum_{t=1}^{T} r_t$. Let $Q(a)$ be the expected reward $\mathbb{E}[r|a]$ for action a, V^* be optimal value for action a $V^* = \max_{a \in \mathcal{A}} Q(a)$, and we define regret as the opportunity loss for one step:

$$l_t = \mathbb{E}[V^* - Q(a_t)]$$

The total loss is defined as:

$$L_t = \mathbb{E}\left[\sum_{\tau=1}^{t} V^* - Q(a_\tau)\right]$$

To evaluate total regret, we define the count $\bar{N}_t(a)$ as the expected number of selections for action a and Δ_a as the difference in value between action a and optimal value V^*, thus we have:

$$\begin{aligned} L_t &= \mathbb{E}\left[\sum_{\tau=1}^{t} V^* - Q(a_\tau)\right] \\ &= \sum_{a \in \mathcal{A}} \mathbb{E}[N_t(a)]\big(V^* - Q(a)\big) \\ &= \sum_{a \in \mathcal{A}} \bar{N}_t(a)\Delta_a \end{aligned}$$

Greedy Algorithm: Since the expected reward $Q(a)$ for action a is unknown, we estimate the value of each action by Monte-Carlo evaluation:

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

The greedy algorithm chooses an action that maximizes estimated Q value:

$$a_t^* = arg\max_{a \in \mathcal{A}} \hat{Q}_t(a)$$

To avoid locking into a sub-optima, we use ϵ-greedy algorithm that select the optimal action with the probability of $1 - \epsilon$ and ϵ probability to choose a random action.

Regret Bounds: It is desirable to guarantee the regret of some algorithm can be quantified and bounded. There are several algorithms whose total regret grows linearly with the increase of time step (e.g. greedy, ϵ-greedy), but what we really desire is sublinear regret (e.g. decaying ϵ-greedy, optimistic initialization).

Lower Bound: We use a lower bound to determine how hard the problem is, in particular, the similarity between optimal arm and other arms, which is measured by gap Δ_a and KL divergence between two distributions $D_{KL}(\mathcal{R}^a || \mathcal{R}^{a^*})$.

$$\lim_{t \to \infty} L_t \geq log\, t \sum_{a | \Delta_a > 0} \frac{\Delta_a}{D_{KL}(\mathcal{R}^a || \mathcal{R}^{a^*})}$$

which guarantees the regret is sublinear.

Optimism in the Face of Uncertainty

This method is based on upper confidence bound algorithm. First, we estimate upper bound $U_t(a)$ for each arm such that $Q(a) \leq U_t(a)$ with high probability, which is dependent on the number of times action a has been selected. Based on Hoeffding's Inequality, we have the relation between true mean and empirical mean such that:

$$\mathbb{P}[\mathbb{E}[X] > \bar{X}_t + u] \leq exp\,(-2nu^2)$$

where $\mathbb{E}[X]$ is the true mean and $\bar{X}_t$ is empirical mean, thus we rewrite upper confidence bound as:

$$U_t(a_t) = \hat{Q}(a_t) + \sqrt{\frac{1}{2N_t(a)} log\,\frac{t^2}{\delta}}$$

with high confidence. Here, we let $exp\,(-2nu^2)$ be δ/t^2, so $u = \sqrt{\frac{1}{2N_t(a)} log\,\frac{t^2}{\delta}}$.

Then for each time step, we choose the action with the highest upper confidence bound $U_t(a)$.

$$a_t = arg\,\max_{a \in \mathcal{A}} \left[\hat{Q}(a) + \sqrt{\frac{1}{2N_t(a)} log\,\frac{t^2}{\delta}} \right]$$

Lastly, the UCB algorithm ensures sublinear regret.

$$\lim_{t \to \infty} L_t = 8\,log\,t \sum_{a|\Delta_a > 0} \Delta_a$$

Bayesian Bandits: Suppose we have assumed a probability distribution (prior distribution) over rewards for some reason, then observe real rewards based on prior distribution and compute posterior distribution over rewards to guide exploration.

e.g. let the reward of arm i be a probability distribution that depends on parameter ϕ_i, we have initial prior over ϕ_i, $p(\phi_i)$. Pull arm i and observe reward r_i, then compute posterior distribution $p(\phi_i|r_i)$ using Bayesian rule such that:

$$p(\phi_i|r_i) = \frac{p(r_i|\phi_i)p(\phi_i)}{p(r_i)} = \frac{p(r_i|\phi_i)p(\phi_i)}{\int_{\phi_i} p(r_i|\phi_i)p(\phi_i)d\phi_i}$$

Thompson Sampling: Each time the action a is selected according to the probability that a is the optimal action:

$$\pi(a|h_t) = \mathbb{P}[Q(a) > Q(a'), \forall a' \neq a|h_t]$$

Initialize prior over each arm a, $p(\mathcal{R}_a)$
loop
  For each arm a **sample** a reward distribution $\mathcal{R}_a$ from posterior
  Compute action-value function $Q(a) = \mathbb{E}[\mathcal{R}_a]$
  $a_t = \arg\max_{a \in \mathcal{A}} Q(a)$ ⟵
  Observe reward r
  Update posterior $p(\mathcal{R}_a|r)$ using Bayes law
end loop

Probability Approximately Correct (PAC): All algorithms seek to achieve regret bound in terms of time step T, but this does not tell us the magnitude of mistakes (small but frequent errors vs large but infrequent errors). PAC algorithm cares about the bound of large mistakes, which means the algorithm will choose an action whose values is ϵ-optimal:

$$Q(a) \geq Q(a^*) - \epsilon$$

Optimistic Initialization: In model-free RL case, the systematic exploration can be easily encouraged by optimistic initialization of value function. Concretely, we initialize $Q(s,a) = \frac{r_{max}}{1-\gamma}$ $\forall s, \forall a$ and then apply algorithms like Q-learning, SARSA, etc. The optimistic initialization ensures for each state and each action can be explored equally likely.

MBIE-EB: The main idea of MBIE-EB is to add a bonus term $\beta = \frac{1}{1-\gamma}\sqrt{0.5\, ln(2|S||A|m/\delta)}$ each time we update Q value.

Given ϵ, δ, m
$\beta = \frac{1}{1-\gamma}\sqrt{0.5 \ln(2|S||A|m/\delta}$
$n_{sas}(s, a, s') = 0\ s \in S,\ a \in A,\ s' \in S$
$rc(s, a) = 0,\ n_{sa}(s, a) = 0,\ \tilde{Q}(s, a) = 1/(1-\gamma)\ \forall\ s \in S,\ a \in A$
$t = 0,\ s_t = s_{init}$
loop
    $a_t = \arg\max_{a \in \mathcal{A}} Q(s_t, a)$
    Observe reward r_t and state s_{t+1}
    $n_{sa}(s_t, a_t) = n(s_t, a_t) + 1,\ n_{sas}(s_t, a_t, s_{t+1}) = n_{sas}(s_t, a_t, s_{t+1}) + 1$
    $rc(s_t, a_t) = \frac{rc(s_t,a_t)n_{sa}(s_t,a_t)+r_t}{(n_{sa}(s_t,a_t)+1)}$
    $\hat{R}(s, a) = \frac{rc(s_t,a_t)}{n(s_t,a_t)}$ and $\hat{T}(s'|s, a) = \frac{n_{sas}(s_t,a_t,s')}{n_{sa}(s_t,a_t)}\ \forall s' \in S$
    while not converged **do**
        $\tilde{Q}(s, a) = \hat{R}(s, a) + \gamma \sum_{s'} \hat{T}(s'|s, a) \max_{a'} \tilde{Q}(s', a') + \underbrace{\frac{\beta}{\sqrt{n_{sa}(s,a)}}}_{\text{reward bonus}}\ \forall\ s \in S,\ a \in A$
    end while
end loop

9. Batch Reinforcement Learning

Motivation: Deploying a bad policy is dangerous and costly, since a bad policy produces bad data, and bad data induces an even worse policy. Therefore, we need a safe batch reinforcement learning algorithm to guarantee a new policy is not worse than current policy with the probability $1-\epsilon$ and not contingent on tuning any hyperparameters. In general:

$$\mathbb{P}\left(V^{\mathcal{A}(\mathcal{D})} \geq V_{min}\right) = 1 - \epsilon$$

where $\mathcal{A}$ is a learning algorithm and $\mathcal{D} = \{T_1, T_2, \cdots, T_n\}$ is historical data which is a collection of trajectories.

Off-policy Policy Evaluation (OPE): Given historical data $\mathcal{D}$ and proposed policy π_e, the goal is to estimate the expected value of π_e.

$$V^{\pi_e} \approx \frac{1}{n}\sum_{i=1}^{n}\left(\prod_{t=1}^{L}\frac{\pi_e(a_t|s_t)}{\pi_b(a_t|s_t)}\right)\left(\sum_{t=1}^{L}\gamma^t R_t^i\right)$$

High-confidence Off-policy Policy Evaluation (HCOPE): Use Hoeffding's inequality to convert the estimate of V^{π_e} into the a $1-\delta$ confidence lower bound on V^{π_e}. First, let $X_1, \cdots X_n$ be n independent random variable bounded as $X_i \in [0, b]$.

$$\mathbb{P}\left(\mathbb{E}[X_i] > \frac{1}{n}\sum_{i=1}^{n} X_i + u\right) \leq exp\left(\frac{-2nu^2}{(b-0)^2}\right)$$

Substitute $exp\left(\frac{-2nu^2}{(b-0)^2}\right)$ to δ, we have an inequality with a confidence of $1-\delta$.

$$\mathbb{E}[X_i] > \frac{1}{n}\sum_{i=1}^{n} X_i + b\sqrt{\frac{log\,(1/\delta)}{2n}}$$

Safety Policy Improvement(SPI): Use HCOPE method to improve a new policy by being $1-\delta$ confidence the lower bound of new policy is large than current policy.

Per-decision Importance Sampling: each time the reweighted term $\prod_{t=1}^{L}\frac{\pi_e(a_t|s_t)}{\pi_b(a_t|s_t)}$ is only up to the step the agent gets a reward, so we have the estimate:

$$V_{PDIS}^{\pi} \approx \sum_{t=1}^{L} \gamma^t \frac{1}{n} \sum_{i=1}^{n} \left(\prod_{\tau=1}^{t} \frac{\pi_e(a_\tau|s_\tau)}{\pi_b(a_\tau|s_\tau)} \right) R_t^i$$

Weighted Importance Sampling: For each trajectory, we multiply a normalized term to reweight the importance of this trajectory. Though the value is biased, the resulting value estimate is still consistent.

$$V_{WIS}^{\pi} \approx \frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i \left(\sum_{t=1}^{L} \gamma^t R_t^i \right)$$

where w_i is equal to importance sampling reweighted term $\frac{\pi_e(a_t|s_t)}{\pi_b(a_t|s_t)}$.

Control Variates: Suppose we have a random variable X, which has a true mean $\mu = \mathbb{E}[X]$ and variance $\sigma = Var(X)$, the estimated variable $\hat{\mu}$ has the same mean $\mathbb{E}[\hat{\mu}] = \mu = \mathbb{E}[X]$ and variance $Var(\hat{\mu}) = \sigma = Var(X)$. Now we have another random variable Y, and the current estimated variable $\hat{\mu}$ is $X - Y + \mathbb{E}[Y]$, so the empirical mean of $X - Y + \mathbb{E}[Y]$ is unbiased:

$$\begin{aligned} \mathbb{E}[\hat{\mu}] &= \mathbb{E}\big[X - Y + \mathbb{E}[Y]\big] \\ &= \mathbb{E}[X] - \mathbb{E}[Y] + \mathbb{E}[Y] \\ &= \mathbb{E}[X] \end{aligned}$$

While the variance of $\hat{\mu}$ is different:

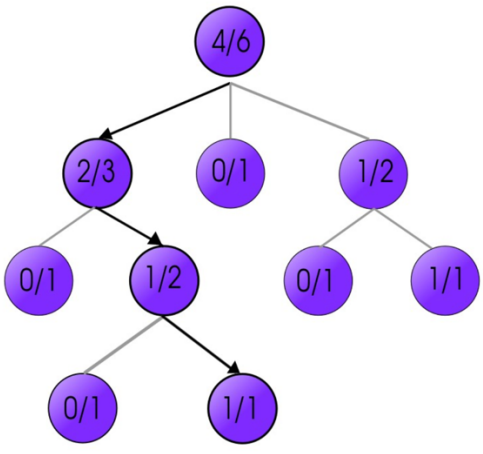
$$\begin{aligned} Var(\hat{\mu}) &= Var(X - Y + \mathbb{E}[Y]) \\ &= Var(X - Y) \\ &= Var(X) + Var(Y) - 2Cov(X, Y) \end{aligned}$$

Thus, if $2Cov(X, Y) > Var(Y)$, we get a lower variance.

10. Monte Carlo Tree Search

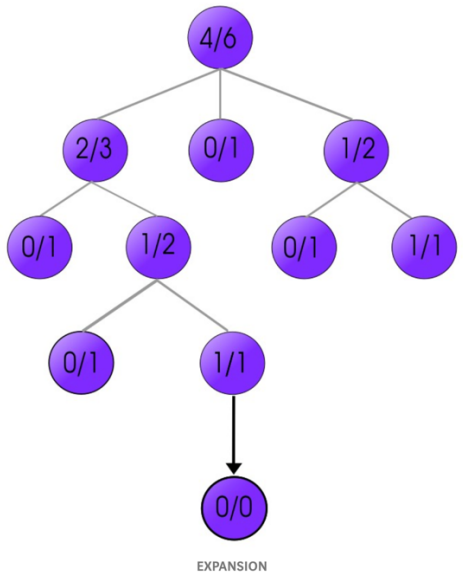
Motivation: Some applications such as Go, there are huge number of states to evaluate, which is not desirable to use traditional tree search algorithm to exhaust statistics of all states then choose the best action for each possible state. Monte Carlo tree search is an alternative heuristic search algorithm that can save computation budget. Monte Carlo tree search consists of four main steps: selection, expansion, simulation and backpropagation.

Selection: Starting from a node we call root, the algorithm selects the child node with the highest probability (or highest UCB) of winning, and this process is done recursively until we reach leaf node but not terminal.



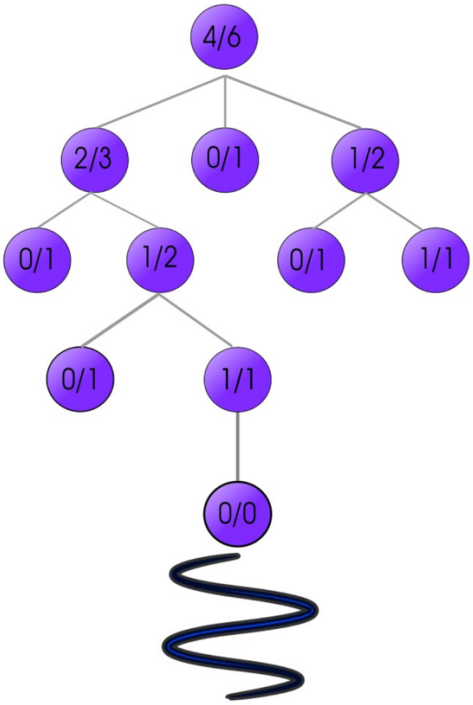
SELECTION

Expansion: Once the algorithm reaches the leaf node we selected, the new node is added into the search tree, thus this node can be chosen in the further move.



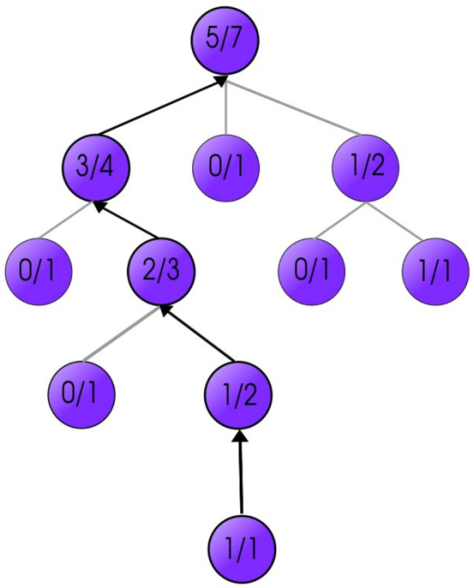
EXPANSION

Simulation: To estimate the expanded node, we use the average reward gained starting from that node under some policy (can be random or specific simulation policy).



SIMULATION

Backpropagation: After getting the reward, we backpropagate the signal to the root node along the path we have experienced.



UPDATING | BACK-PROPAGATION

e.g.