Reinforcement Learning

1. Introduction to reinforcement learning

1.1 Why reinforcement learning:
There are problems where we can formulate them as MDP, but we are unsure of its transition model T and reward function R. If we know its T, and R, then we can either run policy iteration or value iteration to obtain an optimal policy. However, when T/R, or T and R, are unknown, we can utilize reinforcement learning.

1. 2 Goal of reinforcement learning:
Agent learns to act optimally, i.e. learn to maximize expected rewards, by receiving feedbacks in the form of rewards.

Remark (Utility)**.** *Agent's utility is defined by the reward function.*

1.3 Types of reinforcement learning:

- **Passive learning**: also referred as 'prediction'. We evaluate a fixed policy, instead of learning good actions.
- **Active learning**: learning to act optimally

Each of the above section, can be further divided into:

- **Model-based learning:** learn the model of MDP, i.e. learn the transition probability model $T(s, a, s')$, and observe $R(s, a, s')$ from experience of (s, a, s'). Compute optimal policy by either value iteration or policy iteration.
- **Model-free learning:** Bypass the need to learn T, R, compute an optimal policy from experience of agent its own, or other policies with a Q-learning table, which records state-action values. Passive model-free learning are TD, MC. Active model-free learning can be classified into:
  - **On-policy learning:** Learn about policy π from experience sampled from π, e.g. SARSA

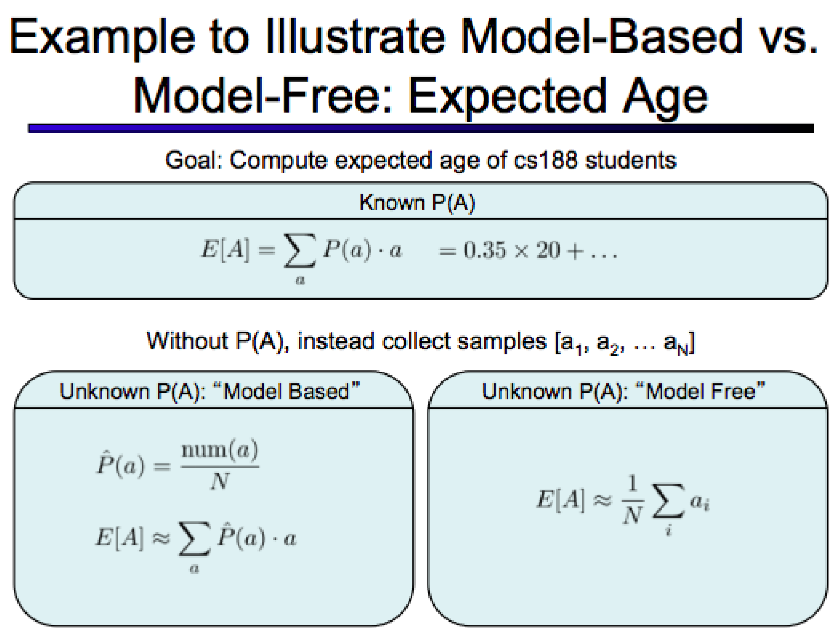
- **Off-policy learning:** Learn about policy π from experience sampled from μ, e.g. Q-learning

Remark (Optimal Policy)**.** *Optimal policy is deterministic.*

Remark (π, μ)**.**

1.4 Definitions

- **Model:** We refer model as transition probability model, T.



2. Passive reinforcement learning

In passive learning, the policy is fixed. Therefore, we only learn the utility value of each state if the policy is run from that state. The **goal** is to learn the value function $U^{\pi}(s)$ from observations when **transition model** $P(s'|s, a)$ and **reward function** $R(s)$ are unknown.

General idea:

- The agent executes a set of trials using the fixed policy π.
- The utility or value for π is $U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)]$

Question: how to compute $U^\pi(s)$?

2.1 Model-based leaning:

- Adaptive Dynamic Programming: learns the T, R model, then solves it.
  - Learn the transition probabilities $P(s'|s, \pi(s))$
  - Learn reward function $R(s)$
  - Calculate the value by solving the **Bellman equation** with linear algebra
    - Can use **modified policy iteration** method of doing k iteration of value updates after each change to model
  - Can learn the model using supervised learning. The pseudo-code below shows learning using maximum likelihood with table representing $P(s|s, a)$

2.2 Model-free leaning:

- Direct Utility Estimation / Monte Carlo Learning
  - **Expected reward-to-go/return:** In a state, the utility or value is the expected total reward from that state onwards
  - Each trial is treated as providing a sample of this quantity for each state visited
  - Keep **a running average** for each state in a table. In infinitely many trial, sample average will converge to expected value.
    - Running average:

$$
\begin{aligned}
U_k(s) &= \frac{1}{k}\sum_{i=0}^{k} G_i(s) \\
&= \frac{1}{k}(G_k(s) + (k-1)G_{k-1}U_{k-1}(s) \\
&= \underbrace{U_{k-1}(s)}_{\text{Estimate after k-1 returns}} + \frac{1}{k}\underbrace{(G_k(s) - U_{k-1}(s))}_{\text{Prediction Error}}
\end{aligned}
\tag{1}
$$

- Monte Carlo Learning is just an instance of supervised learning: each example has state as inout and observed return as output
- Advantage: Simple; Each labeled target/return is an unbiased estimate
- Disadvantage:
  * Need to wait till the end of episode before learning can be done
  * Variance can be high as a return is a sum of many rewards over the sequence. Exploiting the constraint imposed by the Bellman equation may help reduce variance ($\implies$ Temporal-Difference Learning)

- **Temporal difference** learning exploits more of the Bellman equation constraints than Monte Carlo learning. For a transition from state s to s', TD learning does: $U^\pi(s) = U^\pi(s) + \alpha(\underbrace{R(s) + \gamma U^\pi(s') - U^\pi(s))}_{\text{TD Error}}$
  - α is the learning rate
  - $R(s) + \gamma U^\pi(s')$ is called the TD target
  - $R(s) + \gamma U^\pi(s') - U^\pi(s)$ is called the TD error
  - Converges to the expected value if α decreases with the number of times the states has been visited, e.g. $\alpha(n) = O(1/n)$

Remark (Model-free)**.** *TD does not need a transition model to do the updates, only the observed transition.*

2.3 n-step TD

- Let $G_{t:t+n} = R_t + \gamma R_{t+1} + \gamma^2 R(r+2) + ... + \gamma^n \overline{V}(S_{t+n})$, where $\overline{V}(S_{t+n}$ is the estimated value at state S_{t+n}
- n-step TD sets $G_{t:t+n}$ as the target for update

2.4 TD(λ)

- $G_t^\gamma = (1-\lambda)\sum_{n=1}^{\infty} \gamma^{n-1} G_{t:t+n}$, an average TD value over different values of n.
- G_t^γ is a weighted sum of n-step returns, where n-th step is weighted by λ^{n-1}
- When $\lambda = 1$, we get Monte Carlo. When $\lambda = 0$, we get TD.

$$
\begin{aligned}
G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} & \\
= \quad & (1-\lambda)\lambda^0\big(R_t + V(s_{t+1})\big) \\
+ \quad & (1-\lambda)\lambda^1\big(R_t + R_{t+1} + V(s_{t+2})\big) \\
+ \quad & (1-\lambda)\lambda^2\big(R_t + R_{t+1} + R_{t+2} + V(s_{t+3})\big) \\
+ \quad & (1-\lambda)\lambda^3\big(R_t + R_{t+1} + R_{t+2} + R_{t+3} + V(s_{t+4})\big) \\
+ \quad & \cdots \\
= \quad & (1-\lambda)[\quad R_t(\lambda^0 + \lambda^1 + \lambda^2 + \cdots + \lambda^\infty) \\
& \qquad + R_{t+1}(\lambda^1 + \lambda^2 + \lambda^3 + \cdots + \lambda^\infty) \\
& \qquad + R_{t+2}(\lambda^2 + \lambda^3 + \lambda^4 + \cdots + \lambda^\infty) \\
& \qquad + R_{t+3}(\lambda^3 + \lambda^4 + \lambda^5 + \cdots + \lambda^\infty) \\
& \qquad + ...] \\
+ \quad & (1-\lambda)\big[\lambda^0 V(s_{t+1}) + \lambda^1 V(s_{t+2}) + \lambda^2 V(s_{t+3}) + \cdots\big] \\
\\
= \quad & (1-\lambda)(1 + \lambda^1 + \cdots + \lambda^\infty)[R_t\lambda^0 + R_{t+1}\lambda^1 + R_{t+2}\lambda^2 + \cdots] \\
+ \quad & (1-\lambda)\big[\lambda^0 V(s_{t+1}) + \lambda^1 V(s_{t+2}) + \lambda^2 V(s_{t+3}) + \cdots\big] \\
\\
= \quad & (1-\lambda)\cdot\frac{1}{1-\lambda}\cdot[R_t\lambda^0 + R_{t+1}\lambda^1 + R_{t+2}\lambda^2 + \cdots] \\
+ \quad & (1-\lambda)\big[\lambda^0 V(s_{t+1}) + \lambda^1 V(s_{t+2}) + \lambda^2 V(s_{t+3}) + \cdots\big] \\
\\
= \quad & R_t\lambda^0 + R_{t+1}\lambda^1 + R_{t+2}\lambda^2 + R_{t+3}\lambda^3 + \cdots \\
+ \quad & (1-\lambda)\big[\lambda^0 V(s_{t+1}) + \lambda^1 V(s_{t+2}) + \lambda^2 V(s_{t+3}) + \cdots\big]
\end{aligned}
$$

$$
\begin{aligned}
&if\ \lambda \to 0 \quad G_t^\lambda = R_t + V(s_{t+1}) && (1 - stepTD) \\
&if\ \lambda \to 1 \quad G_t^\lambda = R_t + R_{t+1} + R_{t+2} + \cdots && (MonteCarlo)
\end{aligned}
$$

2.5 Summary

TD vs. ADP

- ADP learns the model then solves for $U^\pi(s)$ for each state
- TD/MC does not need a model. They can work with measurements from the real world, or a simulator.
- ADP tend to be more data efficient - requires less data from the real world
- TD does not need to compute expectation, does not need to solve the system of linear equations - tend to be computationally more efficient.

TD vs. MC

- TD can learns/updates after every step. MC learns/updates after each episode.
- TD target depends only on one measured reward. MC target $G(s)$depends on the sum of many rewards.
  - TD target has low variance but is biased
  - MC target is unbiased but has higher variance
- TD usually converges faster than MC in practice because TD exploits constraints of the Bellman equation

Model-based vs. Model-free

- Model-based RL
  - First act in MDP and learn T, R
  - Then value iteration or policy iteration with learned T, R
  - Advantage: efficient use of data
  - Disadvantage: requires building a model for T, R
- Model-free RL
  - Bypass the need to learn T, R
  - Methods to evaluate V^π, the value function for a fixed policy π without knowing T, R:
    - (i) Direct Evaluation
    - (ii) Temporal Difference Learning
  - Method to learn π^*, Q^*, V^* without knowing T, R
    - (iii) Q-Learning

3. Active reinforcement learning

Actions in reinforcement learning not only gain rewards but also help learn a better model. By improving the model, greater reward may potentially be obtained. Therefore there is a need between:

- Exploitation: maxmize value as reflected by the current estimate/current utility function
- Exploration: learn more about the model/model of the world to potentially improve long term well being

A scheme for balancing exploration and exploitation must:

- Try each action in each state an unbounded number of times to avoid a finite probability of missing an optimal action
- Eventually become greedy in the limit of infinite exploration

Such schemes are greedy in the limit exploration (**GILE**), e.g. ϵ-greedy is GILE if ϵ reduces to 0 at $\epsilon_k = \frac{1}{k}$.

Remark (ϵ-greedy exploration)**.** *Choose the greedy action with probability (1 - ϵ), and an action from the policy with probability ϵ.*

Remark (ϵ)**.** *The decreasing of ϵ is important to the convergence of an optimal value.*

3.1 Greedy action selection with $U^+(s)$

GLIE based ϵ-greedy convergence may be slow, an alternative way to balance exploration and exploitation is to use greedy action selection with respect to an **optimistic** estimate of the utility $U^+(s)$.

$$U^+(s) = R(s) + \gamma \arg\max_a f(\textstyle\sum_{s'} P(s'|s,a)U^+(s'), N(s,a))$$

$N(s,a)$ is the number of times action a has been tried in state s.

3.2 Learning Q-value and Q-function

Instead of learning teh utility function, we can learn an action-utility function $Q(s, a)$, the value of doing action a in state s.

> Q-values are related to utility values by: $U(s) = \arg\max_a Q(s, a)$
>
> The Q-function similarly satisfies a version of the Bellman equations: $Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \arg\max_{a'} Q(s', a')$

- Model-based: take ADP approach, learn P(s' — s, a), then use an iterative method to compute the Q-function, given the estimated model.
  - ADP with Q-learning still need model for learning but not to take actions
  - Using MC/TD do not need model for learning and taking actions
- Model-free: an agent that learns a Q-function does not need a model of the form P(s' — s, a) for action selection.

3.3 GLIE ϵ-greedy MC control

3.3.1 ϵ-greedy exploration

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability / every action has a non-zero probability to be tried
- With probability ϵ choose an action at random:

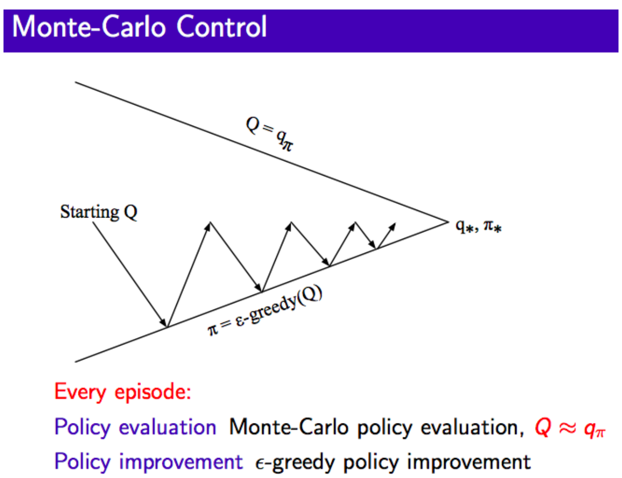
$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon, & \text{if } a^* = \arg\max_{a \in A} Q(s, a) \\ \epsilon/m, & \text{otherwise} \end{cases} \tag{2}$$

> ϵ-Greedy Policy Improvement:
>
> Theorem: For any ϵ-greedy policy π, the ϵ-greedy policy π' with respect to q_π is an improvement.

3.3.2 GLIE ϵ-greedy MC control

General Idea:

- Sample kth episode using π: $\{S_1, A_1, R_1, ..., S_T\} \quad \pi$
- For each state S_t and action A_t in the episode,
  - $N(S_t, A_t) = N(S_t, A_t) + 1$
  - $Q(S_t, A_t) = Q(S_t, A_t) + \dfrac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$
- Improve policy based on new action-value function
  - $\epsilon = \dfrac{1}{k}$
  - $\pi = \epsilon - greedy(Q)$



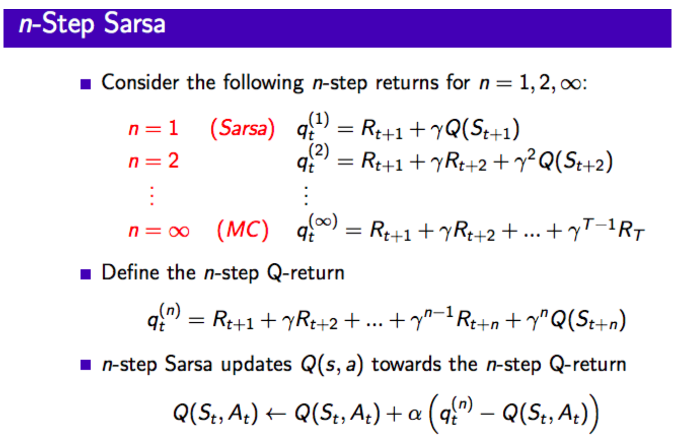
> Theorem: GLIE Monte-Carlo control converges to the optimal action-value function, $Q(s, a) \Rightarrow q_*(s, a)$

3.4 Q-learning

- With TD update, we have what is called Q-learning:
  $Q(s, a) = Q(s, a) + \alpha(R(s) + \gamma \arg\max_{a'} Q(s', a') - Q(s, a))$
- Q-learning is **off-policy**. Works regardless of policy for generating the trajectory, e.g. random policy

3.4 SARSA

- $Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$ where a' is the action actually taken, whereas in Q-learning, a is the estimated action across different state-action pairs.
- In comparison, Q-learning uses the max over possible actions a'.
- SARSA is **on policy**. If agent policy is always exploring, learns to take exploration into account as well.
- When greedy agent that takes action with best Q-value is used, Q-learning is the same with SARSA.
- n-step SARSA

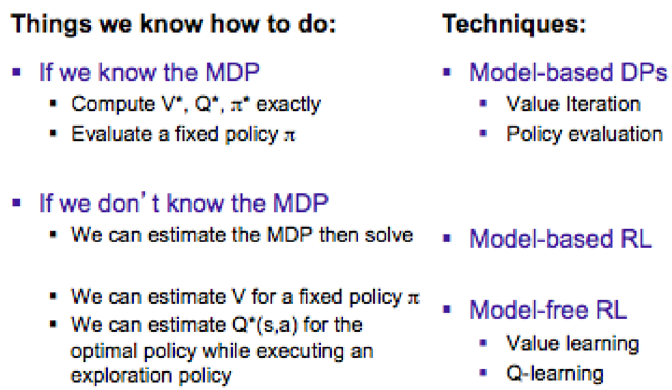


3.5 Summary

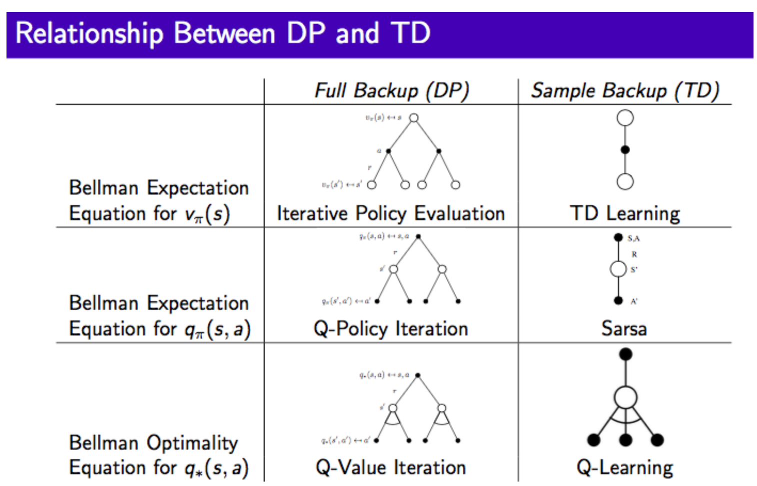
- On-policy: given a policy, while agent follow the given policy, agent evaluate the policy and update the policy on the go.
- Off-policy: given a behavioural policy, agent evaluates other policies/target policy $\pi(a|s)$ to compute $Q_\pi(s,a)$
  - Learn from observing humans or other agents
  - Re-use experience generated from old policies $\pi_1, \pi_2, \ldots$
  - Learn about optimal policy while following exploratory policy

  - Learn about multiple policies while following one policy

- MDP & RL



- DP & TD



4. Function approximation

Function approximation helps to scale reinforcement learning by discarding the Q-table, and generalizing from what the agent has seen to unseen. Here we focus on the **linear combinations of features** as it is differentiable therefore easier to adjust its parameters by looking at its **gradient**.

For example, $\overline{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s)$, where $\overline{U}_\theta(s)$ is the function approximation that uses n parameters in θ to weight n features of a state to represent a very large state space. Reinforcement learning agent is supposed to learn $\theta_1, \theta_2 ...$ to approximate the $U(s)$.

4.1. Function approximation with Monte Carlo learning

- Obtain a set of training samples: $(((x_1, y_1), u_1), ((x_2, y_2), u_2), ..., ((x_n, y_n), u_n))$, where u_j is the measured utility of the j-th example. Note, u_j is unbiased
- This gives a **supervised learning** problem.
- With squared error and linear function, we get a standard linear regression problem.
- The squared error can be minimized with **online learning**/stochastic gradient descent.
- For j-th example, the error can be represented as:

$$E_j(s) = (\overline{U}_\theta(s) - u_j(s))^2 \tag{3}$$

- i-th parameter θ_i can be updated as:

$$\begin{aligned} \theta_i(s) &= \theta_i \underbrace{-}_{\text{Negative gradient}} \alpha \frac{\partial E_j}{\partial \theta_i} \\ &= \theta_i + \alpha(\overline{U}_\theta(s) - u_j(s)) \frac{\overline{U}_\theta(s)}{\partial \theta_i} \end{aligned} \tag{4}$$

4.2 Online learning with temporal difference learning :

- Obtain a set of training data: $((s_1, (R(s_2) + \gamma U_\theta(s_2))), (s_2, (R(s_3) + \gamma U_\theta(s_3))), ..., (s_n, (R(s_{n+1}) + \gamma U_\theta(s_{n+1}))))$

- For TD: $\theta_i = \theta_i + \alpha(R(s) + \gamma \overline{U}_\theta(s') - \overline{U}_\theta(s)) \frac{\partial \overline{U}_\theta(s)}{\partial \theta_i}$

- For Q-learning: $\theta_i = \theta_i + \alpha(R(s) + \gamma \arg\max_{a'} \overline{Q}_\theta(s', a') - \overline{Q}_\theta(s, a)) \frac{\partial \overline{Q}_\theta(s, a)}{\partial \theta_i}$

5. Policy search

5.1. Policy representation and search

Policy search adjusts θ to improve the policy. Policy search tries to find a policy, e.g. represented as Q-functions that does well, so $Q^*/10$ can given the same optimal actions as Q^*. In contrast, Q-learning with function approximation tries to find a value of θ such that $\overline{Q}_\theta$ that is close to Q^*.

- Use **stochastic policy** $\pi_\theta(s, a)$ that specifies the probability of selecting action a in state s. This solves the problem that policy as a function of action is discontinuous. For example, the **softmax function**:

$$\underbrace{\pi_\theta(s, a)}_{\text{Probability distribution of actions in s}} = \frac{e^{\overline{Q}_\theta(s,a)}}{\underbrace{\sum_{a'} e^{\overline{Q}_\theta(s,a')}}_{\text{Nomalization}}}$$

- We can specify the **policy value** as $\rho(\theta)$. $\rho(\theta)$ can be optimized by:
  - Taking a step in the direction of the **policy gradient** $\nabla_\theta \rho(\theta)$/hill climbing (positive gradient descent), if $\rho(\theta)$ is differentiable (if we specify the policy in softmax function).
  - Look for a local optimal

- For stochastic environment and/or policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of gradient at θ, $\nabla_\theta \rho(\theta)$ directly from results of trials executed at θ.

- Consider **single action from single state** s_0:
  $\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a \nabla_\theta(\pi_\theta(s_0, a)) R(a)$

Then we approximate the summation using **samples** generated from $\pi_\theta(s_0, a)$/ sample N samples (a_n) from one state:

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \frac{\nabla_\theta(\pi_\theta(s_0, a))R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{n} \frac{\nabla_\theta(\pi_\theta(s_0, a_j))R(a_j)}{\pi_\theta(s_0, a_j)}$$

5.2. Policy search: REINFORCE algorithm

- From **single state**, we generalize this idea to **sequential case**: state s_i and action a_{ij}, $\pi_\theta(s_i, a_{ij})$.

  $$\nabla_\theta \rho(\theta) \propto \sum_s \rho_{\pi_\theta}(s) \sum_a \frac{\pi_\theta(s_0, a)\nabla_\theta \pi_\theta(s_0, a) Q_{\pi_\theta}(s, a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_i^N \sum_j^i \frac{\nabla_\theta \pi_\theta(s_i, a_{ij}) G_{ij}(s_i)}{\pi_\theta(s_i, a_{ij})},$$

  for each state s_i is visited, a_{ij} is executed on j-th trial, and G_{ij} is the total reward/return received from state s_i onwards on j-th trial.

- Use online learning/update, we get **REINFORCE** algorithm:

  $$\theta_{j+1} = \theta_j + \alpha G_j \frac{\nabla \theta \pi_\theta(s, a_j)}{\pi_\theta(s, a_j)}$$

  As $\nabla_\theta \ln(\pi_\theta(s, a_j)) = \dfrac{\nabla \theta \pi_\theta(s, a_j)}{\pi_\theta(s, a_j)}$, we have rewrite our update function as follows:

  $\theta_{j+1} = \theta_j + \alpha G_j \nabla_\theta \ln(\pi_\theta(s, a_j))$, because usually the gradient of natural logarithm of the policy function is easier to look for than directly from the policy function itself.

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T-1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

- Reduce variance with a **Baseline**

  We are estimating $\nabla\rho(\theta) = \sum_s p_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s, a_j) Q_{\pi_\theta}(s, a)$

$$
\begin{aligned}
\nabla\rho(\theta) &= \sum_s p_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s, a_j) Q_{\pi_\theta}(s, a) \\
&= \sum_s p_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s, a_j)(Q_{\pi_\theta}(s, a) - B(s)) \\
&= \sum_s p_{\pi_\theta}(s)(\sum_a \nabla_\theta \pi_\theta(s, a_j) Q_{\pi_\theta}(s, a) - (\sum_a \nabla_\theta \pi_\theta(s, a_j) B(s))) \\
&= \sum_s p_{\pi_\theta}(s)(\sum_a \nabla_\theta \pi_\theta(s, a_j) Q_{\pi_\theta}(s, a) - B(s)(\underbrace{\sum_a \nabla_\theta \pi_\theta(s, a_j)}_{\text{Distribution sums up to 1}})) \\
&= \sum_s p_{\pi_\theta}(s)(\sum_a \nabla_\theta \pi_\theta(s, a_j) Q_{\pi_\theta}(s, a) - B(s)(\nabla_\theta 1))
\end{aligned}
\tag{5}
$$

  - Using a baseline function B(s) can reduce variance
  - Use an **advantage function** in place of $Q_{\pi_\theta}(s, a)$: $A_{\pi_\theta}(s, a) = Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s)$, where $V_{\pi_\theta}(s)$ is the baseline function

- REINFORCE uses a Monte Carlo estimate of the advantage function, which has higher variance. To reduce variance, an alternative is to use TD method. The advantage function is: $Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s) = E[r + \gamma V_{\pi_\theta}(s')] - V_{\pi_\theta}(s)$. It is also common to use multiple steps of rewards instead of one step in TD.

- **actor-crtitic method** :
  - **Critic**: Learns a value or Q-function, that is to update parameter w, that is used only for evaluation
  - **Actor**: Learns a policy (actor) that takes action, that is to update parameters θ in direction suggested by critic
  - The critic is solving a familiar problem: policy evaluation

– We use a critic to estimate the action-value function, $Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$

Algorithm 1: Adaptive Dynamic Programming

---

function PASSIVE-ADP-AGENT(percept) returns an action ;

Input : percept, a percept indicating the current state s' and reward signal r'

Output: a

Persistent: π, a fixed policy; mdp, an MDP with model P, reward R, discount γ; U, a table of utilities, initially empty; N_{sa}, a table of frequencies for state-action pairs, initially zero; $N_{s'|sa}$, a table of outcome frequencies given state-action pairs, initially zero; s, a, the previous state and action, initially null ;

if s' *is new* **then**
    $U[s'] = r'$;
    $R[s'] = r'$;
end
if s *is not null* **then**
    increment $N_{sa}[s,a]$ and $N_{s'|sa}[s',s,a]$;
    foreach *t such that* $N_{t|sa}[t,s,a]$ *is nonzero* **do**
        $P(t|s,a) = N_{s'|sa}[s',s,a] \;/\; N_{sa}[s,a]$;
    end
end
U = POLICY-EVALUATION (π, U, mdp);
if s' *is Terminal* **then**
    s, a = null ;
    $R[s'] = r'$;
else
    s, a = s', $\pi[s']$
end
return a

Algorithm 2: Temporal-Difference Learning

function PASSIVE-TD-AGENT(percept) returns an action ;

Input : percept, a percept indicating the current state s' and reward signal r'

Output: a

Persistent: π, a fixed policy; mdp, an MDP with model P, reward R, discount γ; U, a table of utilities, initially empty; N_{sa}, a table of frequencies for state-action pairs, initially zero; s, a, r, the previous state, action, and reward, initially null ;

if s' *is new* **then**
    $U[s'] = r'$;
end
if s *is not null* **then**
    increment $N_{sa}[s, a]$;
    $U^{\pi}[s] = U^{\pi}[s] + \alpha(N_{sa}[s])(R(s) + U\pi[s'] - U\pi[s])$
end
if s' *is Terminal* **then**
    $s, a, r =$ null ;
else
    s, a, r = s', $\pi[s']$, r
end
return a

Algorithm 3: Q Learning

function Q-LEARNING-AGENT(percept) returns an action ;
Input : percept, a percept indicating the current state s' and reward signal r'
Output: a
Persistent: Q, a table of action values indexed by state and action, initially zero; N_{sa}, a table of frequencies for state-action pairs, initially zero; s, a, r, the previous state, action, and reward, initially null ;
if s' *is Terminal* **then**
    $Q[s', None] = r'$
end
if s *is not null* **then**
    increment $N_{sa}[s, a]$;
    $Q(s, a) = Q(s, a) + \alpha(N_{sa}[s, a])(R(s) + \gamma \arg\max_{a'} Q(s', a') - Q(s, a))$
end
s, a, r = s', $\arg\max_{a'} f(Q(s', a'), N(s', a'))$, r' ;
return a