Reinforcement Learning: Theories and Algorithms

Wonjun Ko

wjko@korea.ac.kr

Machine Intelligence Laboratory,
Department of Brain and Cognitive Engineering,
Korea University

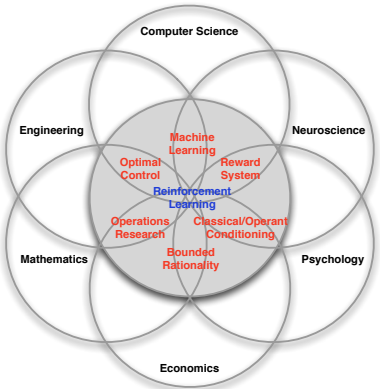
September 19, 2019

Contents

Introduction

Reinforcement Learning (1/2)

- *"Reinforcement is the strengthening of a pattern of a behavior as a result of an animal receiving a stimulus in an appropriate temporal relationship with another stimulus or a response."* - Ivan Pavlov



Many faces of reinforcement learning

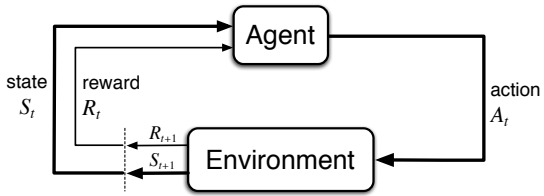
Machine Intelligence Lab

Reinforcement Learning (2/2)

- A master chess player makes a move. The choice is informed both by planning, anticipating possible replies and counterreplies, and by immediate, intuitive judgements of the desirability of particular positions and moves.

- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour.

- A reinforcement learning (RL) problem involves *interaction* between an active decision-making *agent* and its *environment*, within which the agent seeks to achieve a *goal* despite *uncertainty* about its environment.



The agent-environment interaction

Elements of Reinforcement Learning

- Beyond the agent and the environment, one can identify four main subelements.
- A *policy* defines the learning agent's way of behaving at a given time.
  - A policy is a mapping from perceived states of the environment to actions to be taken when in those states.
- A *reward signal* defines the goal of an RL problem.
  - One each time step, the environment sends to the RL agent a single number called the reward.
- Whereas the reward signal indicates what is good in an immediate sense, a *value function* specifies what is good in the long run.
  - The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.
- Optionally, a *model* of the environment mimics the behavior of the environment, or more generally, allows inferences to be made about how the environment will behave.

Limitations and Scope

- Most of recent studies about RL focus fully on the decision-making issues, despite issues of constructing, changing, or learning the state signal are also important.
- Most of the RL methods are structures around estimating value functions, but it is not strictly necessary to do this to solve the problems.
- For instance, solution methods such as genetic algorithms, genetic programming, simulated annealing, and other optimization methods never estimate value functions.
- Evolutionary methods ignore much of the useful structure of the RL problem: they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects.
  - In some cases this information can be misleading (*e.g.*, when states are misperceived), but more often it should enable more efficient search.

Tabular Solution Methods: Multi-Armed Bandits

A k-Armed Bandit Problem (1/2)

- In our k-armed bandit problem, each of the k actions has an expected or mean reward given that action is selected; let us call this the value of that action, *i.e.*, *action-value*.
- The value then of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a], \tag{1}$$

  where action selected on time step t as A_t, and the corresponding reward as R_t.
- If we knew the value of each action, then it would be trivial to solve the problem, but we do not know, therefore we denote the estimated value of action a at time step t as $Q_t(a)$.
  - We would like $Q_t(a)$ to be close to $q_*(a)$.

A k-Armed Bandit Problem (2/2)

- If we maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest, we call these the *greedy* actions.
- When we select one of these actions, we say that we are exploiting our current knowledge of the action-values.
- If instead we select one of the nongreedy actions, then we say we are exploring, because this enables you to improve your estimate of the nongreedy action's value.
- *Exploitation* is the right thing to do to maximize the expected reward on the one step, but *exploration* may produce the greater total reward in the long run.
- In any specific case, whether it is better to explore or exploit depends in a complex way on the precise values of the estimates, uncertainties, and the number of remaining way on the precise values of the estimates, uncertainties, and the number of remaining steps.

Action-Value Methods

- One natural way to estimate this is by averaging the rewards actually received:

$$Q_t(a) \doteq \frac{\text{sum of reward when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}, \tag{2}$$

  where $\mathbb{1}_{\text{state}}$ denotes the random variable that is 1 if state is true and 0 if it is not.

- We write the *greedy* action selection method as

$$A_t \doteq \arg\max_a Q_t(a). \tag{3}$$

- A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ε, instead select randomly from among all the actions with equal probability, independently of the action-value estimates, which is called an *ε-greedy* method.

The 10-Armed Testbed (1/4)

- A tutorial code of 'Multi-Armed Bandits' is implemented in Google Colab link[1].

```
# Import APIs
import numpy as np
import matplotlib.pyplot as plt

# Define functions
def multi_armed_bandits(k):
  mab_reward_mean = np.random.RandomState(seed=951014).rand(k)*10-5
  mab_reward_stdev = np.random.RandomState(seed=5930).rand(k)
  return mab_reward_mean, mab_reward_stdev

def pull_bandits(k, pull):
  mean, std = multi_armed_bandits(k)
  reward = np.random.normal(loc=mean[pull], scale=std[pull])
  return reward

num_arms = 10

action_list = np.arange(num_arms)
mean_reward, std_reward = multi_armed_bandits(num_arms)
```

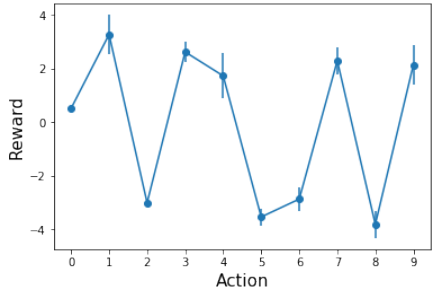
- We first design the 10-armed bandit environment, each arm has different mean and standard deviation for reward distribution.

[1]This code was implemented by Ko.

The 10-Armed Testbed (2/4)

- We plot a graph with errorbar to show each reward distribution for each arm.

```
plt.errorbar(action_list, mean_reward, std_reward, fmt="-o")
plt.xticks(np.arange(num_arms))
plt.xlabel("Action", fontsize=15)
plt.ylabel("Reward", fontsize=15)
plt.show()
```



- For instance, if we pull the 0-th arm, then we can achieve 0.514 ± 0.082 as a reward signal.

Machine Intelligence Lab

The 10-Armed Testbed (3/4)

```
num_step = 1000
steps = np.arange(num_step) + 1 # [1, 2, ..., 1000]
def action_value_method(epsilon, Q_t):
  cumulated_actions = np.zeros(num_arms)
  cumulated_rewards = np.zeros(num_arms)
  expected_rewards = np.zeros(num_step)
  optimal_action = np.zeros(num_step)
  opt_act = 0

  for step in steps:
    # Epsilon-Greedy Exploration-Exploitation
    if epsilon > np.random.uniform(): A_t =
          np.random.randint(num_arms)
    else: A_t = np.argmax(Q_t)

    reward = pull_bandits(k=num_arms, pull=A_t)
    cumulated_actions[A_t] += 1
    cumulated_rewards[A_t] = cumulated_rewards[A_t] +
          np.array(reward)
    Q_t[A_t] = cumulated_rewards[A_t] / cumulated_actions[A_t]

    if A_t == 1: opt_act += 1

    optimal_action[step-1] = opt_act/step
    expected_rewards[step-1] = np.mean(cumulated_rewards)/step

  print("Calculated Q-table for %.3f epsilon:" % epsilon, "\n",
        Q_t)
  print("The number of selected actions:\n", cumulated_actions)
  return expected_rewards, optimal_action
```

```
expected_rewards_1, optimal_action_1 = action_value_method(0.1,
      np.zeros(num_arms))
expected_rewards_2, optimal_action_2 = action_value_method(0.01,
      np.zeros(num_arms))
expected_rewards_3, optimal_action_3 = action_value_method(0,
      np.zeros(num_arms))
```

```
Calculated Q-table for 0.100 epsilon:
[ 0.52441996 3.22500046 -2.99693165 2.80982619
1.76235335 -3.43926102 -3.14397685 2.24216641
-3.94090257 2.14859357]
The number of selected actions:
[ 20. 906. 11. 9. 9. 6. 12. 10. 9. 8.]

Calculated Q-table for 0.010 epsilon:
[ 0.51405025 3.28662018 -3.10154593 2.61439577 0. 0. 0.
0. -4.08473095 2.24996996]
The number of selected actions:
[157. 366. 1. 474. 0. 0. 0. 0. 1. 1.]

Calculated Q-table for 0.000 epsilon:
[0.51681465 0. 0. 0. 0. 0. 0. 0. 0. 0. ]
The number of selected actions:
[1000. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Machine Intelligence Lab

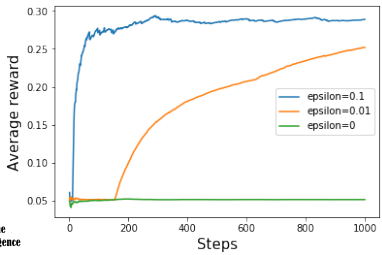
The 10-Armed Testbed (4/4)

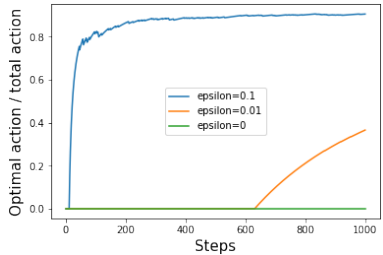
- In the testbed, our action-value function with appropriate ε could find an optimal action, which is pulling 1st arm.

- To show expected rewards and optimal action selection rates of each ε of 0.1, 0.01, and 0, we then plot graphs.

```
plt.plot(steps, expected_rewards_1, label="epsilon=0.1")
plt.plot(steps, expected_rewards_2, label="epsilon=0.01")
plt.plot(steps, expected_rewards_3, label="epsilon=0")
plt.legend()
plt.xlabel("Steps", fontsize=15)
plt.ylabel("Average reward", fontsize=15)
plt.show()
```

```
plt.plot(steps, optimal_action_1, label="epsilon=0.1")
plt.plot(steps, optimal_action_2, label="epsilon=0.01")
plt.plot(steps, optimal_action_3, label="epsilon=0")
plt.legend()
plt.xlabel("Steps", fontsize=15)
plt.ylabel("Optimal action / total action", fontsize=15)
plt.show()
```





Machine Intelligence Lab

Incremental Implementation (1/2)

- We turn the question of how estimating action-value methods can be computed in a computationally efficient manner.

- Let R_i now denote the reward received after the i-th selection of this action, and let Q_n denote the estimate of its action-value. after it has been selected $n-1$ times:

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}. \tag{4}$$

- It is easy to devise incremental formulas for updating Q_n with small and constant computation required to process each new reward.

$$\begin{aligned} Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) = \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\big(R_n + (n-1)Q_n\big) = \frac{1}{n}\big(R_n + nQ_n - Q_n\big) \\ &= Q_n + \frac{1}{n}[R_n - Q_n]. \end{aligned} \tag{5}$$

Machine Intelligence Lab

Incremental Implementation (2/2)

- The general form of the update rule is:

$$NewEstimate \leftarrow OldEstimate + StepSize\ \big[Target - OldEstimate\big]. \tag{6}$$

- In pseudocode form;

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:

$A \leftarrow \arg\max_a Q(a)$ with ε-greedy
$R \leftarrow \text{bandit}(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(a) + \frac{1}{N(A)}[R - Q(A)]$

```
def incremental_implementation(epsilon):
  Q_t = np.zeros(num_arms) # Q-table
  N = np.zeros(num_arms)
  for step in steps:
    # Epsilon-Greedy Exploration-Exploitation
    if epsilon > np.random.uniform():
      A_t = np.random.randint(num_arms)
    else:
      A_t = np.argmax(Q_t)

    reward = pull_bandits(k=num_arms, pull=A_t)

    N[A_t] = N[A_t] + 1
    Q_t[A_t] = Q_t[A_t] +[reward - Q_t[A_t]]/N[A_t]

  print("Q-table, epsilon: %f\n" %epsilon, Q_t)
  return

incremental_implementation(0.1)
```

```
Q-table, epsilon: 0.100000
[ 0.50040965 3.25859375 -3.01852886 2.57983548
1.44310263 -3.58868297 -2.78630141 2.21408633
-3.79171997 2.19869826]
```

Machine Intelligence Lab

Tracking a Nonstationary Problem

- We often encounter RL problems that are effectively nonstationary.
  - In such cases it makes sense to give more weight to recent rewards than to long-pass rewards.

- One of the most popular ways of doing this is to use a constant step-size parameter $\alpha \in (0, 1]$.

$$
\begin{aligned}
Q_{n+1} &\doteq Q_n + \alpha\big[R_n - Q_n\big] && (7)\\
&= \alpha R_n + (1-\alpha)Q_n \\
&= \alpha R_n + (1-\alpha)[\alpha R_{n-1} + (1-\alpha)Q_{n-1}] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i && (8)
\end{aligned}
$$

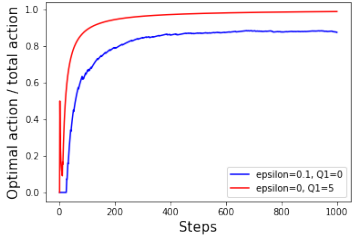
Optimistic Initial Values

- All the methods we have discussed so far are dependent to some extent on the initial action-value estimates, $Q_1(a)$, *i.e.*, these methods are *biased* by their initial estimates.

- Initial action values can also be used as a simple way to encourage exploration.
  - Suppose that instead of setting the initial action values to zero, as we did in the 10-armed testbed, we set them all to $+5$.

```
# With optimistic initial values
_, optimal_action_4 = action_value_method(0,
      5*np.ones(num_arms)) # Q1 = +5

plt.plot(steps, optimal_action_1, label="epsilon=0.1, Q1=0",
      c="blue")
plt.plot(steps, optimal_action_4, label="epsilon=0, Q1=5",
      c="red")
plt.legend()
plt.xlabel("Steps", fontsize=15)
plt.ylabel("Optimal action / total action", fontsize=15)
plt.show()
```

```
Calculated Q-table for 0.000 epsilon:
[ 0.39132487 3.28117093 -2.82206609 2.01569415
1.55402117 -2.97242546 -3.34720604 1.83260197
-3.7599842 2.26621717]
The number of selected actions:
[ 1. 989. 1. 1. 1. 1. 1. 1. 1. 3.]
```



Machine Intelligence Lab

Upper-Confidence-Bound Action Selection (1/2)

- Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates.
- ε-greedy action selection forces the non-greedy actions to be tried, but indiscriminately, with no preference for those that are nearly greedy or particularly uncertain.
- One effective way of that, selecting among the non-greedy actions according to their potential for actually being optimal, is to select actions according to:

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right], \tag{9}$$

where $N_t(a)$ denotes the number of times that action a has been selected prior to time t and the number $c > 0$ controls the degree of exploration.

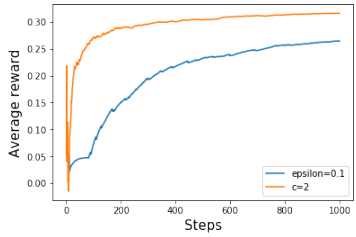
  - If $N_t(a) = 0$, then a is considered to be a maximizing action.

Upper-Confidence-Bound Action Selection (2/2)

- The idea of this *upper confidence bound* (UCB) action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of a's value.
- For UCB method, we just use the following code instead of ε-greedy action selection.

```
## Epsilon-Greedy Exploration-Exploitation
# if epsilon > np.random.uniform():
#   A_t = np.random.randint(num_arms)
# else:
#   A_t = np.argmax(Q_t)

# We use UCB exploration instead of epsilon-greedy method.
if np.min(N_t) == 0:
  A_t = np.argmin(N_t)
  N_t[A_t] +=1
else:
  tmp = Q_t + c * np.sqrt(np.log(step)/N_t)
  A_t = np.argmax(tmp)
  N_t[A_t] += 1
```

```
Calculated Q-table for 2.000 c:
[ 0.51269976 3.28009245 -3.16047003 2.53320825
1.55391985 -3.04609883 -2.4632513 2.44216027 -4.1196314
2.32191526]
The number of selected actions:
[ 4. 900. 1. 33. 8. 1. 1. 27. 1. 24.]
```



Machine Intelligence Lab

Gradient Bandit Algorithms (1/4)

- We consider learning a numerical *preference* for each action a, which we denote $H_t(a)$:

$$\Pr\{A_t = a\} \doteq \frac{\exp H_t(a)}{\sum_{b=1}^{k} \exp H_t(b)} \doteq \pi_t(a), \tag{10}$$

  where $\pi_t(a)$ denotes the probability of taking action a at time t.

- There is a natural learning algorithm based on the idea of stochastic gradient ascent, on each step, after selecting action A_t and receiving the reward R_t, the action preferences are updated by:

$$\begin{aligned} H_{t+1}(A_t) &\doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), \quad \text{and} \\ H_{t+1}(a) &\doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a), \qquad\qquad \text{for all } a \neq A_t, \end{aligned} \tag{11}$$

  where $\alpha > 0$ is a step-size parameter, and $\bar{R}_t \in \mathbb{R}$ is the average of all the rewards up through and including time t.

Gradient Bandit Algorithms (2/4)

- In exact *gradient ascent*, each action preference $H_t(a)$ would be incremented proportional to the increment's effect on performance:

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \tag{12}$$

where $\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x)$.

- Then, first we take a closer look at the exact performance gradient:

$$\begin{aligned} \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \left[\sum_x \pi_t(x) q_*(x) \right] = \sum_x q_*(x) \frac{\partial \pi_t(x)}{\partial H_t(a)} \\ &= \sum_x (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)}, \end{aligned} \tag{13}$$

where B_t, called the *baseline*, can be any scalar that does not depend on x.

Gradient Bandit Algorithms (3/4)

- Next we multiply each term of the sum by $\pi_t(x)/\pi_t(x)$:

$$\begin{aligned}
\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \sum_x \pi_t(x)(q_*(x) - B_t)\frac{\partial \pi_t(x)}{\partial H_t(a)}/\pi_t(x) \\
&= \mathbb{E}\left[(q_*(A_t) - B_t)\frac{\partial \pi_t(A_t)}{\partial H_t(a)}/\pi_t(A_t)\right] \\
&= \mathbb{E}\left[(R_t - \bar{R}_t)\frac{\partial \pi_t(A_t)}{\partial H_t(a)}/\pi_t(A_t)\right] \\
&= \mathbb{E}[(R_t - \bar{R}_t)\pi_t(A_t)(\mathbb{1}_{a=A_i} - \pi_t(a))/\pi_t(A_t)] \\
&= \mathbb{E}[(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a))]. \qquad (14)
\end{aligned}$$

- Then, we can finally get:

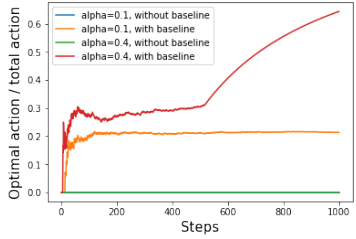
$$\therefore H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a)), \qquad \text{for all } a, \qquad (15)$$

which we may recognize as being equivalent to our original algorithm Eq. (11).

Gradient Bandit Algorithms (4/4)

```
def softmax(vec):
  return np.exp(vec)/np.sum(np.exp(vec))

def gradient_bandit(alpha, baseline, Q_t):
  cumulated_actions = np.zeros(num_arms)
  cumulated_rewards = np.zeros(num_arms)
  expected_rewards = np.zeros(num_step)
  optimal_action = np.zeros(num_step)
  opt_act, R_total = 0, 0
  H_t = np.random.rand(num_arms)
  for step in steps:
    # Calculate the probability distribution
    pi_t = softmax(H_t)
    A_t = np.argmax(pi_t)
    reward = pull_bandits(k=num_arms, pull=A_t)
    R_total = R_total + reward
    R_bar = R_total/step + baseline

    # Gradient Ascent and update the preference, H_t
    H_t = H_t - alpha * (reward-R_bar) * pi_t
    H_t[A_t] = H_t[A_t] + alpha * (reward-R_bar)*(1-pi_t[A_t])

    if A_t == 1:
      opt_act += 1
    optimal_action[step-1] = opt_act/step
    expected_rewards[step-1] = np.mean(cumulated_rewards)/step
  return expected_rewards, optimal_action
```

- The baseline shifts up all the rewards with absolutely no effect on the algorithm.
- If the baseline were omitted, then the gradient would be significantly degraded.



Machine Intelligence Lab

Tabular Solution Methods: Finite Markov Decision Processes

Finite Markov Decision Processes

- *Markov decision processes* (MDPs) are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards.
  - MDPs involve delayed reward and the need to tradeoff immediate and delayed reward.

- In MDPs, we estimate the value $q_*(s, a)$ of each action a in each state s or we estimate the value $v_*(s)$ of each state given optimal action selections, whereas in bandit problems we estimated the value $q_*(a)$ of each action a.
  - These state-dependent quantities are essential to accurately assigning credit for long-term consequences to individual action selections.

- MDPs are a mathematically idealized form of the RL problem for which precise theoretical statements can be made.
  - Key elements of the mathematical structure: returns, value functions, and Bellman equations

The Agent-Environment Interface (1/3)

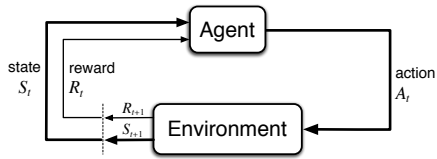
- MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal.
  - The learner and decision maker is called the *agent*.
  - The thing it interacts with, comprising everything outside the agent, is called the *environment*.

- At each time steps, $t = 0, 1, 2, \cdots$, the agent receives some representation of the environment's *state*, $S_t \in \mathcal{S}$, and on that bases selects an *action*, $A_t \in \mathcal{A}(s)$.

- One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}.

- The MDP and agent together thereby give rise to sequence or *trajectory* that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \cdots \tag{16}$$



The Agent-Environment Interface (2/3)

- In a *finite* MDP, the sets of states, actions, and rewards ($\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$) all have a finite number of elements.

- The random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action.

- For particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}, \tag{17}$$

  for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}$.
  - The probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}, and, given them, not at all on earlier states and actions.

- The function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ defines the *dynamics* of the MDP.

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \ \forall s \in \mathcal{S}, a \in \mathcal{A}(s). \tag{18}$$

Machine Intelligence Lab

The Agent-Environment Interface (3/3)

- From the dynamics, we can compute one might want to know about the environment, such as the *state-transition probabilities*, as a three-argument function $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$,

$$p(s'|s, a) \doteq \Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a). \tag{19}$$

- We can also compute the expected rewards for state-action pairs as a two argument function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$:

$$r(s, a) \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a), \tag{20}$$

and the expected reward for state-action-next-state triples as a three-argument function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$:

$$r(s, a, s') \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)}. \tag{21}$$

Goals and Rewards

- In RL, the goal of the agent is formalized in terms of a special signal, called the reward.
  - The agent's goal is to maximize the total amount of reward it receives, in other words we do not maximize immediate reward, but cumulative reward in the long run.
- We can state informal idea as the *reward hypothesis*: *That all of what we mean by goals and purposed can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*
- It is critical that the rewards we set up truly indicate what we want accomplished.
- The reward signal is not the place to impart to the agent prior knowledge about how to achieve what we want it to do.
  - A chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent's pieces or gaining control of the center of the board.
- The reward signal is our way of communicating to the agent what we want it to achieve, not how we want it achieved.

Returns and Episodes (1/3)

- The agent's goal is to maximize the cumulative reward it receives in the long run.
- In general, we seek to maximize the *expected return*, where the return, denoted G_t, is defined as some specific function of the reward sequence:

$$G_t \doteq R_{t+1} + R_{t+2} + \cdots + R_T, \tag{22}$$

  where T is a final time step.
- This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences, which we call *episodes*, *i.e.*, *trials*, such as plays of a game, trips through a maze, or any sort of repeated interaction.
- Each episode ends in a special state called the *terminal state*.
- Even if we think of episodes as ending in different ways, such as winning and losing a game, the next episode begins independently of how the previous one ended, then tasks with episodes of this kind are called *episodic tasks*.

Returns and Episodes (2/3)

- In many cases the agent-environment interaction does not break naturally into identifiable episodes like episodic tasks, but goes on continually without limit, and we call these *continuing tasks*.
- The return formulation Eq. (22) is problematic for continuing tasks, thus we add an additional concept, *discounting*.
- In particular, it chooses A_t to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{23}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

Returns and Episodes (3/3)

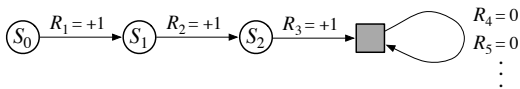
- The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately.
  - If $\gamma < 1$, the infinite sum in Eq. (23) has a finite value as long as the reward sequence $\{R_k\}$ is bounded.
  - If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose A_t so as to maximize only R_{t+1}.
  - A γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

- Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of RL:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{24}$$

Unified Notation for Episodic and Continuing Tasks

- To establish one notation that enables us to discuss precisely about both episodic and continuing tasks simultaneously is useful.
- We defined the return as a sum over a finite number of terms in one case, Eq. (22) and as a sum over an infinite number of terms in the other, Eq. (23).
- These two can be unified by considering episode termination to be the entering of a special *absorbing state* that transitions only to itself and that generates only rewards of zero.



- We can write

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{25}$$

including the possibility that $T = \infty$ or $\gamma = 1$.

Policies and Value Functions (1/3)

- Almost all RL algorithms involve estimating *value functions*-functions of states (or stage-action pairs) that estimate *how good* it is for the agent to be in a given state (or how good it is to perform a give action in a given state).
  - Value functions are defined with respect to particular ways of acting, called policies.
- A *policy* is a mapping from states to probabilities of selecting each possible action.
  - If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.
- RL methods specify how the agent's policy is changed as a result of its experience.

Policies and Value Functions (2/3)

- The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+K+1} \middle| S_t = s\right], \quad \forall s \in \mathcal{S} \tag{26}$$

- We call the function $v_\pi(s)$ the *state-value function* for policy π.

- We also define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

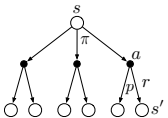
$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]. \tag{27}$$

- We call $q_\pi(s, a)$ the *action-value function* for policy π.

Policies and Value Functions (3/3)

- A fundamental property of value functions used throughout RL and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return Eq. (24).

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\big] \\
&= \sum_a \pi(a|s) \sum_{a', r} p(s', r|s, a)\big[r + \gamma v_\pi(s')\big], \quad \forall s \in \mathcal{S}
\end{aligned} \tag{28}
$$



- Eq. (28) is the *Belmann equation* for v_π, it expresses a relationship between the value of a state and the value of its successor states.
- For graphical summaries, we use a diagram called a *backup diagram*, because the diagram relationship that form the basis of the update or backup operations that are at the heart of RL methods.

Machine Intelligence Lab

Optimal Policies and Optimal Value Functions (1/3)

- Solving an RL task means finding a policy that achieves a lot of reward over the long run.
- We define a policy π which is better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states, *i.e.*, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$, $\forall s \in \mathcal{S}$
  - There is always at least one policy that is better than or equal to all other policies, this is an *optimal policy*, denoted π_*.
  - The optimal policies share the same state-value or action-value function, called the *optimal state-value function*, v_* and *optimal action-value function*, q_* respectively.
- For all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, $v_*(s)$ and $q_*(s, a)$ are defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s), \tag{29}$$

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a). \tag{30}$$

- For the state-action pair (s, a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy, thus we can write q_* in terms of v_*:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]. \tag{31}$$

Optimal Policies and Optimal Value Functions (2/3)

- The optimal state-value function must satisfy the self-consistency condition given by the Bellman equation, Eq. (28).

- This is the Bellman equation for v_*, or the *Bellman optimality equation*.

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) = \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \qquad (32) \\ &= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')]. \qquad (33) \end{aligned}$$

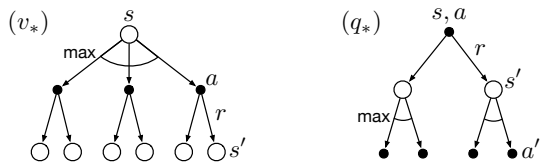
- The Bellman optimality equation for q_* is:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')]. \qquad (34) \end{aligned}$$

Machine Intelligence Lab

Optimal Policies and Optimal Value Functions (3/3)

- For finite MDPs, the Bellman optimality equation has a unique solution independent of the policy.
- Once we have an optimal value function, then it is relatively easy to determine an optimal policy.



- Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the RL problem.
  - However, this solution is rarely directly useful.
  - This solution relies on at least three assumptions that are rarely true in practice:
    (1) we accurately know the dynamics of the environment;
    (2) we have enough computational resources;
    (3) and we have the Markov property.

Machine Intelligence Lab

Tabular Solution Methods: Dynamic Programming

Dynamic Programming (1/2)

- The term *dynamic programming* (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP.
- The key idea of DP, and of RL generally, is the use of value functions to organize and structure the search for good policies.
- We can easily obtain optimal policies once we have found the optimal value functions which satisfy the Bellman optimality equations for all $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$, Eq. (33) and (34).
- DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.
- DP and RL are both solving decision-making problem for given time steps.
  - DP: model-based, *i.e.*, need reward and state transition probability of the environment; *planning*
  - RL: model-based or model-free, *i.e.*, do not need to know the environment; *learning*

Dynamic Programming (2/2)

- DP deals with sequential or temporal component (*dynamic*) of the problem by optimizing a *program*, *i.e.*, a policy.
- DP is a method for solving complex problems by breaking them down into subproblems.
- DP is a very general solution method for problems which have two properties:
  1. Optimal substructure
     - *Principles of optimality* applies.
     - Optimal solution can be decomposed into subproblems.
  2. Overlapping subproblems
     - Subproblems recur many times.
     - Solutions can be cached and reused.
- MDPs satisfy both properties:
  - Bellman equation gives recursive decomposition;
  - Value function stores and reuses solutions.

Machine Intelligence Lab

Policy Evaluation (Prediction) (1/5)

- We consider *policy evaluation* in the DP literature, *i.e.*, *prediction problem*, how to compute the state-value function v_π for an arbitrary policy π.

- Recall the Bellman equation Eq. (28),

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{a',r} p(s', r|s, a) \big[r + \gamma v_\pi(s')\big], \quad \forall s \in \mathcal{S}. \end{aligned}$$

- If the environment's dynamics are completely known, then iterative solution methods are most suitable.

- The initial approximation, v_0, is chosen arbitrarily, and each successive approximation is obtained by using the Bellman equation as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]. \end{aligned} \tag{35}$$

Policy Evaluation (Prediction) (2/5)

- Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case.
- The sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π.
- This algorithm is called *iterative policy evaluation*, and in pseudocode form;

Input π, the policy to be evaluated

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$, arbitrarily except that $V(terminal) = 0$

Loop until $\Delta < \theta$:

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Machine Intelligence Lab

Policy Evaluation (Prediction) (3/5)

- To produce each successive approximation, v_{k+1} from v_k, iterative policy evaluation applies the same operation, called *expected update*, to each state s.
  - It replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated.

- For policy evaluation, we build a 5×5 grid-world example[2].

```
import numpy as np
def state_coord(state, action, grid_size):
  # Define action_code, Up, Down, Left, Right
  code = [(-1, 0), (1, 0), (0, -1), (0, 1)]
  state[0] += code[action][0]
  state[1] += code[action][1]

  if state[0] < 0: state[0] = 0
  elif state[0] > grid_size-1: state[0] = grid_size-1

  if state[1] < 0: state[1] = 0
  elif state[1] > grid_size-1: state[1] = grid_size-1

  return state[0], state[1]
```

[2]This code was implemented by Ko.

Policy Evaluation (Prediction) (4/5)

- We build an the iterative policy evaluation algorithm as a function.

```
# Define policy evaluation function
def policy_evaluation(grid_size, action, policy, iter_num,
      reward=-1, dis=1):
  # Initialize policy table
  value_table = np.zeros((grid_size, grid_size), dtype=float)

  # Iteration
  if iter_num == 0:
    print("Iteration: {} \n{}\n".format(iter_num, value_table))
    return value_table
```

```
  for iteration in range(iter_num):
    tmp = np.zeros((grid_size, grid_size), dtype=float)
    for i in range(grid_size):
      for j in range(grid_size):
        if i == j and ((i == 0) or (i == grid_size-1)):
          value_t = 0
        else:
          value_t = 0
          for act in action:
            i_, j_ = state_coord([i, j], act, grid_size)
            value = policy[i][j][act] * (reward +
                  dis*value_table[i_][j_])
            value_t += value
        tmp[i][j] = round(value_t, grid_size-4)
    iteration += 1
    # Result print
    if (iteration % 10) != iter_num:
      if iteration > 100:
        if (iteration % 20) == 0:
          print("Iteration: {} \n{}\n".format(iteration, tmp))
      else:
        if (iteration % 10) == 0:
          print("Iteration: {} \n{}\n".format(iteration, tmp))
    else:
      print("Iteration: {} \n{}\n".format(iteration, tmp))

    value_table = tmp

  return tmp
```

Machine Intelligence Lab

Policy Evaluation (Prediction) (5/5)

- In our 5×5 grid-world, there are 4 possible actions (up, down, left, and right).
- The initial policy has 0.25 value for each action except for each goal.

```
grid_size = 5
action = [0, 1, 2, 3] # Up, Down, Left, Right
policy = np.empty([grid_size, grid_size, len(action)],
      dtype=float)
for i in range(grid_size):
  for j in range(grid_size):
    for k in range(len(action)):
      if i==j and ((i==0) or (i==grid_size-1)):
        policy[i][j]=0.00
      else:
        policy[i][j]=0.25

policy[0][0] = [0] * len(action)
policy[grid_size-1][grid_size-1] = [0] * len(action)

value = policy_evaluation(grid_size=grid_size, action=action,
      policy=policy, iter_num=100)
```

- Then after 100 iterations, we can have a final evaluated policy which has larger values for nearer states for the goal.

```
Iteration: 100
[[ 0. -21.7 -32.3 -37.3 -39.2]
[-21.7 -29. -34.2 -36.7 -37.3]
[-32.3 -34.2 -35.2 -34.2 -32.3]
[-37.3 -36.7 -34.2 -28.9 -21.6]
[-39.2 -37.3 -32.3 -21.7 0. ]]
```

Machine Intelligence Lab

Policy Improvement (1/5)

- Suppose we have determined the value function v_π for an arbitrary deterministic policy π.
  - For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$.
- We should know which is better between the current policy and the new policy.
  - One way to answer this question is to consider selecting a in s and thereafter following the existing policy, π.

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]. \end{aligned} \tag{36}$$

- The *policy improvement theorem*; let π and π' be any pair of deterministic policies,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s), \qquad \forall s \in \mathcal{S} \tag{37}$$

then $v_{\pi'}(s) \geq v_\pi(s)$, *i.e.*, the policy π' must obtain greater or equal expected return from all states $s \in \mathcal{S}$.

Policy Improvement (2/5)

- Starting from Eq. (37), we keep expanding the q_π side with Eq. (36) and reapplying Eq. (37) until we get $v_{\pi'}(s)$ to prove the policy improvement theorem:
- So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state to a particular action.

$$
\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})] | S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots | S_t = s] \\
&= v_{\pi'}(s).
\end{aligned}
$$

Machine Intelligence Lab

Policy Improvement (3/5)

- To consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $q_\pi(s, a)$ is a natural extension.
- In other words, to consider the new *greedy* policy, π', given by:

$$\begin{aligned}\pi'(s) &\doteq \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')].\end{aligned} \tag{38}$$

- The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*.

Policy Improvement (4/5)

- Suppose the new greedy policy, π', is as good as, but not better than, the old policy π, then $v_\pi = v_{\pi'}$, for all $s \in \mathcal{S}$:

$$\begin{aligned} v_{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi'}(s')]. \end{aligned}$$

- However, this is the same as the Bellman optimality equation, Eq. (33), and therefore, $v_{\pi'}$ must be v_*, and both π and π' must be optimal policies.

- Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

Policy Improvement (5/5)

- To implement the policy improvement in the grid-world, we introduce the Q-table.

```
# Define policy improvement function
def policy_improvement(grid_size, value, action, policy, reward=-1):
  action_id = ["U","D", "L", "R"] # Up, Down, Left, Right
  action_table = []

  # Calculate Q-function
  for i in range(grid_size):
    for j in range(grid_size):
      Q_list = []
      if i==j and ((i==0) or (i==grid_size-1)): action_table.append("G") # Goal
      else:
        for k in range(len(action)):
          i_, j_ = state_coord([i, j], k, grid_size)
          Q_list.append(value[i_][j_])
        max_action = [action_value for action_value, x in enumerate(Q_list) if x ==
              max(Q_list)]

        # Policy update
        policy[i][j] = [0] * len(action) # Initialize Q list
        for y in max_action: policy[i][j][y] = (1/len(max_action))

        # Get action
        idx = np.argmax(policy[i][j])
        action_table.append(action_id[idx])
  action_table = np.asarray(action_table).reshape((grid_size, grid_size))
  print("Updated policy is :\n{}\n".format(policy))
  print("At each state, chosen action is: \n{}".format(action_table))
  return policy
```

- Then we can get a result table, when ‘G’ denotes a goal, and ‘U’, ‘D’, ‘L’, and ‘R’ respectively denote ‘up,’ ‘down,’ ‘left,’ and ‘right’ actions from the improved policy.

```
At each state, chosen action is:
[[‘G’ ‘L’ ‘L’ ‘L’ ‘D’]
[‘U’ ‘U’ ‘L’ ‘D’ ‘D’]
[‘U’ ‘U’ ‘U’ ‘D’ ‘D’]
[‘U’ ‘U’ ‘R’ ‘R’ ‘D’]
[‘U’ ‘R’ ‘R’ ‘R’ ‘G’]]
```

Machine Intelligence Lab

Policy Iteration

- Once a policy π, has been improved using v_π to yield a better policy, π', we can then compute v'_π and improve it again to yield an even better π''.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

  where $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a policy *improvement*.

- This way of finding an optimal policy is called *policy iteration*, and in pseudocode form;

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$, $\forall s \in \mathcal{S}$

2. Policy Evaluation

Loop:

$\Delta \leftarrow 0$
Loop for each $s \in \mathcal{S}$:
$v \leftarrow V(s)$
$V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small threshold)

3. Policy Improvement

$policy - stable \leftarrow true$
For each $s \in \mathcal{S}$:
$old - action \leftarrow \pi(s)$
$\pi(s) \leftarrow$
$\arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
If $old - action \neq \pi(s)$, then
$policy - stable \leftarrow false$

If $policy - stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to **2**.

Value Iteration (1/4)

- The policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration.
- One important special case is when policy evaluation is stooped after just one sweep (one update of each state), and this algorithm is called *value iteration*.

$$
\begin{aligned}
v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')], \qquad \forall s \in \mathcal{S}.
\end{aligned} \tag{39}
$$

Initialize $V(s)$ arbitrarily except that $V(terminal) = 0$
Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
Output a deterministic policy, $\pi \approx \pi_*$, s.t. $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

Machine Intelligence Lab

Value Iteration (2/4)

- Now, we implement the value iteration algorithm in the grid-world.

```
# Define value iteration algorithm function
def value_iteration(grid_size, action, policy, iter_num,
      reward=-1, dis=1):
  # Initialize table
  value_table = np.zeros((grid_size, grid_size), dtype=float)

  # Iteration
  if iter_num == 0:
    print("Iteration: {} \n{}\n".format(iter_num, value_table))
    return value_table

  for iteration in range(iter_num):
    tmp = np.zeros((grid_size, grid_size), dtype=float)
    for i in range(grid_size):
      for j in range(grid_size):
        if i == j and ((i == 0) or (i == grid_size-1)):
          value_t = 0
        else:
          value_t_list = []
          for act in action:
            i_, j_ = state_coord([i, j], act, grid_size)
            value = (reward + dis*value_table[i_][j_])
            value_t_list.append(value)
          tmp[i][j] = max(value_t_list)
    iteration +=1
```

```
# Result print
if (iteration % 10) != iter_num:
  if iteration > 100:
    if (iteration % 20) == 0:
      print("Iteration: {} \n{}\n".format(iteration, tmp))
  else:
    if (iteration % 10) == 0:
      print("Iteration: {} \n{}\n".format(iteration, tmp))
else:
  print("Iteration: {} \n{}\n".format(iteration, tmp))

value_table = tmp

return tmp
```

Machine Intelligence Lab

Value Iteration (3/4)

- Starting from the random policy like policy iteration algorithm implemented earlier, however this value iteration algorithm can reach to convergence faster.

```
grid_size = 5
action = [0, 1, 2, 3] # Up, Down, Left, Right
policy = np.empty([grid_size, grid_size, len(action)],
       dtype=float)
for i in range(grid_size):
  for j in range(grid_size):
    for k in range(len(action)):
      if i==j and ((i==0) or (i==grid_size-1)):
        policy[i][j]=0.00
      else:
        policy[i][j]=0.25

policy[0][0] = [0] * len(action)
policy[grid_size-1][grid_size-1] = [0] * len(action)

value = value_iteration(grid_size=grid_size, action=action,
       policy=policy, iter_num=1)
value = value_iteration(grid_size=grid_size, action=action,
       policy=policy, iter_num=2)
value = value_iteration(grid_size=grid_size, action=action,
       policy=policy, iter_num=3)
value = value_iteration(grid_size=grid_size, action=action,
       policy=policy, iter_num=10)
```

```
Iteration: 1
[[ 0. -1. -1. -1. -1.]
[-1. -1. -1. -1. -1.]
[-1. -1. -1. -1. -1.]
[-1. -1. -1. -1. -1.]
[-1. -1. -1. -1. 0.]]

Iteration: 2
[[ 0. -1. -2. -2. -2.]
[-1. -2. -2. -2. -2.]
[-2. -2. -2. -2. -2.]
[-2. -2. -2. -2. -1.]
[-2. -2. -2. -1. 0.]]

Iteration: 3
[[ 0. -1. -2. -3. -3.]
[-1. -2. -3. -3. -3.]
[-2. -3. -3. -3. -2.]
[-3. -3. -3. -2. -1.]
[-3. -3. -2. -1. 0.]]

Iteration: 10
[[ 0. -1. -2. -3. -4.]
[-1. -2. -3. -4. -3.]
[-2. -3. -4. -3. -2.]
[-3. -4. -3. -2. -1.]
[-4. -3. -2. -1. 0.]]
```

Machine Intelligence Lab

Value Iteration (4/4)

- For fancier exhibition, now we will use another environment[3], which is implemented by 'Open AI GYM' API.
  - We use a toy example, 'Taxi-v2' environment.
  - The agent, taxi (a yellow box)'s goal is to deliver a passenger (a blue colored letter) to a destination (a magenta colored letter).
  - There are 6 possible actions, move N, E, W, S, pickup passenger, and dropoff passenger.
  - For each action, a punishment of -1 and for (right) deliver, a reward of $+20$ is gaven.
  - Further, there is a punishment of -10 for wrong dropoff or pickup.
  - For pair comparison, we set the same initial environment for each random policy and value iteration algorithm.
  - The random policy took 255 timesteps and 58 penalties to achieve terminal step.
  - On the other hand, with $\Delta = 10^{-4}$, the value iteration algorithm took 10 timesteps and 0 penalties.
    - Let us check the taken timesteps and penalties with different values of Δ!

[3] This code was implemented by Ko.

Asynchronous Dynamic Programming

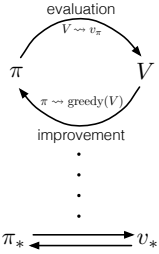
- DP method described so far used *synchronous* backups, *i.e.*, all states are backed up in parallel.
  - $\therefore$ If the state set is very large, then even a single sweep can be prohibitively expensive.
- *Asynchronous DP* backs up states individually, in any order.
  - Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set.
- For each selected state, asynchronous DP applies an appropriate backup, thus it can significantly reduce computational cost.
- There are three simple ideas for asynchronous DP which will be discussed later, (i) in-place DP, (ii) prioritized sweeping, and (iii) real-time DP.

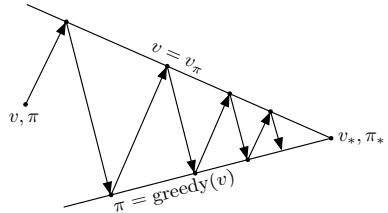
Generalized Policy Iteration

- In policy iteration algorithm, we have two steps:
  - **(i)** policy evaluation; estimate v_π using iterative policy evaluation,
  - **(ii)** policy improvement; generate $\pi' \geq \pi$ using greedy policy improvement.

- On the other hand, *generalized policy iteration* (GPI) method estimates v_π using any policy evaluation algorithm and generate $\pi' \geq \pi$ using any policy improvement algorithm.

- In other words, we use the term GPI to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes.

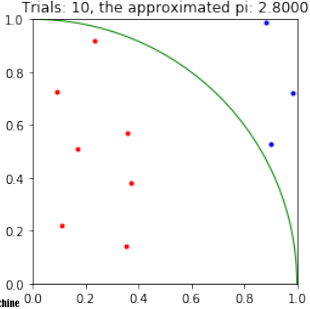


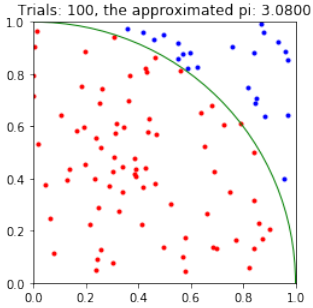


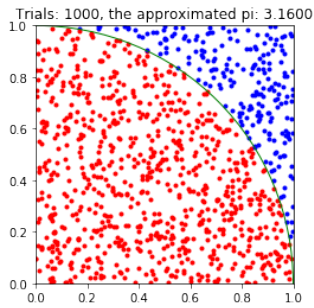
Tabular Solution Methods: Monte Carlo Methods

Monte Carlo Methods

- Unlike DP, here we do not assume complete knowledge of the environment, *i.e.*, we consider our first learning methods for estimating value functions and discovering optimal policies.
- *Monte Carlo* (MC) methods require only *experience*—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment.
- MC methods are ways of solving the RL problem based on averaging sample returns.







Machine Intelligence Lab

Monte Carlo Prediction (1/3)

- We begin by considering MC methods for learning the state-value function for a given policy.
  - Recall that the value of a state is the expected return, expected cumulative future discounted reward, starting from that state.
- An obvious way to estimate the state-value function from experience is simply to average the returns observed after visits to that state.
  - As more returns are observed, the average should converge to the expected value.
- When we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes, obtained by following π passing through s.
  - Each occurrence of state s in an episode is called a *visit* to s.
  - s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the *first visit* to s.
- The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every-visit MC method* averages the returns following all visits to s.

Monte Carlo Prediction (2/3)

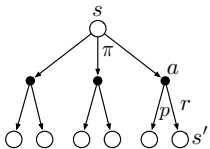
- Both first-visit MC and every-visit MC converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity.
- Now, we focus on first-visit MC because every-visit MC extends more naturally to function approximation and eligibility traces which will be discussed later.
- In pseudocode form, first-visit MC prediction, for estimating $V \approx v_\pi$;

Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$an empty list, for all $s \in \mathcal{S}$
Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \cdots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \cdots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, \cdots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average$(Returns(S_t))$

Monte Carlo Prediction (3/3)

- The general idea of a backup diagram is to show at the top the root node to be updated and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update.

- For MC estimation of v_π, the root is a state node, and below it is the entire trajectory of transitions along a particular single episode, ending at the terminal state.



- Whereas the DP diagram (left) shows all possible transitions and includes only one-step transitions, the MC diagram (right) shows only those sampled on the one episode and goes all the way to the end of the episode.

- Importantly, MC methods do not *bootstrap*, *i.e.*, the estimates for each state are independent.

Machine Intelligence Lab

Monte Carlo Estimation of Action Values (1/2)

- If a model is not available, then it is particularly useful to estimate action values rather than state values.
  - With a model, state values alone are sufficient to determine a policy, however, without a model, state values alone are not sufficient.
  - $\therefore$ One of primary goals for MC methods is to estimate q_*.

- The policy evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π.
  - $\Rightarrow$ The MC methods for this are essentially the same as just presented for state values, except now we talk about visits to a state-action pair rather than to a state.

- The only complication is that many state-action pairs may never be visited, and this is the general problem of *maintaining exploration*.

Monte Carlo Estimation of Action Values (2/2)

- For policy evaluation to work for action values, we must assure continual exploration, and one way to do this is by specifying that the episodes *starts in a state-action pair*, and that every pair has a nonzero probability of being selected as the start.
  - This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes, we call this the assumption of *exploring starts*.

- The assumption of exploring starts is sometimes useful, but of course it cannot be relied upon in general, particularly when learning directly from actual interaction with an environment.
  - In that case, the most common alternative approach to assuring that all state-action pairs are encountered is to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state.

Monte Carlo Control (1/2)

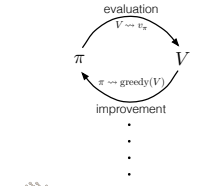
- Like GPI, we can consider a MC version of classical policy iteration:

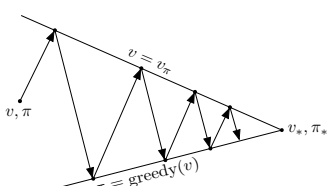
$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

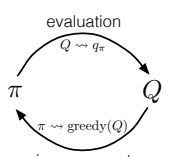
  where $\xrightarrow{\text{E}}$ denotes a policy evaluation and $\xrightarrow{\text{I}}$ denotes a policy improvement.

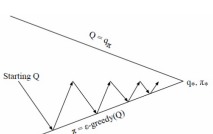
- In this case we have an action-value function, and therefore no model is needed to construct the greedy policy.
- For any action-value function q, the corresponding greedy policy is the one that:

$$\pi(s) \doteq \arg\max_a q(s,a). \tag{40}$$









Monte Carlo Control (2/2)

- For MC policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis.

- A complete simple algorithm along these lines, which we call *MC ES*, for Monte Carlo with Exploring Starts, is given in pseudocode form;

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \cdots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \cdots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \cdots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Monte Carlo Control without Exploring Starts (1/3)

- To ensure that all actions are selected infinitely often without ES, there are two approaches to ensuring this, resulting in what we call *on-policy* methods and *off-policy* methods.
  - On-policy methods attempt to evaluate or improve the policy that is used to make decisions.
    - The MC ES method developed above is an example of an on-policy method.
  - Off-policy methods evaluate or improve a policy different from that used to generate the data.

- Without the assumption of ES, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of nongreedy actions.

- In on-policy control methods, the policy is generally *soft*, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, but gradually shifted closer and closer to a deterministic optimal policy.
  - The ε-greedy policies are examples of ε-soft policies, defined as policies for which $\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\varepsilon > 0$.

Monte Carlo Control without Exploring Starts (2/3)

- For estimating $\pi \approx \pi_*$, the on-policy first visit MC control (for ε-soft policies) algorithm has the form;

Algorithm parameter: small $\varepsilon > 0$
Initialize:
$\pi \leftarrow$an arbitrary ε-soft policy
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
Repeat forever (for each episode):
Generate an episode following $\pi : S_0, A_0, R_1, \cdots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \cdots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \cdots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Monte Carlo Control without Exploring Starts (3/3)

- Let π' be the ε-greedy policy, then the policy improvement theorem is applied:

$$
\begin{aligned}
q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\
&= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a) \\
&\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon} q_\pi(s,a) \\
&= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a) \\
&= v_\pi(s).
\end{aligned}
\tag{41}
$$

- Thus, by the policy improvement theorem, $\pi' \geq \pi$, *i.e.*, $v_{\pi'}(s) \geq v_\pi(s)$, for all $s \in \mathcal{S}$.

Off-policy Prediction via Importance Sampling (1/5)

- All learning control methods face a dilemma: they seek to learn action values conditional on subsequent *optimal* behavior, but they need to behave non-optimally in order to explore all actions (to *find* the optimal actions).
- The on-policy approach is actually a compromise, it learns action values not for the optimal policy, but for a near-optimal policy that still explores.
- A more straightforward approach is to use two policies, one (*target policy*) that is learned about and that becomes the optimal policy, and one (*behavior policy*) that is more exploratory and is used to generate behavior.
  - In this case we say that learning is from data "off" the target policy, and the overall process is termed *off-policy learning*.
- Off-policy methods require additional concepts, and because the data is due to a different policy, off-policy methods are often of greater variance and are slower to converge.
- Off-policy methods are more powerful and general.

Off-policy Prediction via Importance Sampling (2/5)

- Let us consider the *prediction* problem, in which both target and behavior policies are fixed.
  - That is, suppose we wish to estimate v_π or q_π, but all we have are episodes following another policy b, where $b \neq \pi$, *i.e.*, π is the target and b is the behavior policy in this case.

- In order to use episodes from b to estimate values for π, we require that every action taken under π is also taken, at least occasionally, under b.
  - In other words, we require that $\pi(a|s) > 0$ implies $b(a|s) > 0$, and this is called the assumption of *coverage*.

- Almost all off-policy methods utilize *importance sampling*[4], a general technique for estimating expected values under one distribution given samples from another.
  - We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the *importance-sampling ratio*.

[4] There are other sampling methods like *discounting-aware*, *per-decision importance sampling*, *etc.*

Off-policy Prediction via Importance Sampling (3/5)

- Given a starting state S_t, the probability of the subsequent state-action trajectory, $A_t, S_{t+1}, A_{t+1}, \cdots, S_T$, occurring under any policy π is:

$$\begin{aligned}
&\Pr\{A_t, S_{t+1}, A_{t+1}, \cdots, S_T | S_t, A_{t:T-1} \sim \pi\} \\
&\quad = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_t|S_{T-1}, A_{T-1}) \\
&\quad = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k),
\end{aligned}$$

  where p is the state-transition probability function.

- The relative probability of the trajectory under the target and behavior policies, *i.e.*, the importance-sampling ratio, is:

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{42}$$

Off-policy Prediction via Importance Sampling (4/5)

- To estimate the expected returns, *i.e.*, values under the target policy, we call the importance sampling here, because we only have returns G_t due to the behavior policy.
- The ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value:

$$\mathbb{E}[\rho_{t:T-1} G_t | S_t = s] = v_\pi(s). \tag{43}$$

- We define the set of all time steps in which state s is visited, denoted $\mathcal{J}(s)$ (for a first-visit method, $\mathcal{J}(s)$ would only include time steps that were first visits to s within their episodes), then this way is called *ordinary importance sampling*

$$V(s) \doteq \frac{\sum_{t \in \mathcal{J}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{J}(s)|}, \tag{44}$$

where $T(t)$ denote the first time of termination following time t.

- An important alternative is *weighted importance sampling*, which uses a *weighted* average, defined as:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{J}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{J}(s)} \rho_{t:T(t)-1}}. \tag{45}$$

Off-policy Prediction via Importance Sampling (5/5)

- For estimating $Q \approx q_\pi$, in pseudocode form, the off-policy MC prediction (policy evaluation) algorithm is:

  Input: an arbitrary target policy π

  Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

  $Q(s,a) \in \mathbb{R}$ (arbitrarily)

  $C(s,a) \leftarrow 0$

  Loop forever (for each episode):

  $b \leftarrow$ any policy with coverage of π

  Generate an episode following $b : S_0, A_0, R_1, \cdots, S_{T-1}, A_{T-1}, R_T$

  $G \leftarrow 0$

  $W \leftarrow 1$

  Loop for each step of episode, $t = T-1, T-2, \cdots, 0$:

  $G \leftarrow \gamma G + R_{t+1}$

  $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$

  $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$

  $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

  If $W = 0$ then exit For loop

Machine Intelligence Lab

Incremental Implementation

- MC prediction methods can be implemented incrementally, on an episode-by-episode basis, using extensions of the techniques.

- Suppose we have a sequence of returns $G_1, G_2, \cdots, G_{n-1}$, all starting in the same state and each with a corresponding random weight W_i, *e.g.*, $W_i = \rho_{t_i:T(t_i)-1}$.

- We wish to form the estimate

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \qquad n \geq 2, \tag{46}$$

  and keep it up-to-date as we obtain a single additional return G_n.

- Then the update rule for V_n is

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\big[G_n - V_n\big], \qquad n \geq 1, \tag{47}$$

  and, where $C_0 \doteq 0$,

$$C_{n+1} \doteq C_n + W_{n+1}.$$

Off-policy Monte Carlo Control (1/2)

- In off-policy methods these two functions are separated.
  - The policy used to generate behavior, called the behavior policy
  - The policy that is evaluated and improved, called, the target policy
- An advantage of this separation is that the target policy may be deterministic, *e.g.*, greedy, while the behavior policy can continue to sample all possible actions.
- Techniques for off-policy MC control methods follow the behavior policy while learning about and improving the target policy.
- These techniques require that the behavior policy has a nonzero probability of selecting all actions that might be selected by the target policy (coverage).
- To explore all possibilities, we require that the behavior policy be soft, *i.e.*, that it select all actions in all states with nonzero probability.

Off-policy Monte Carlo Control (2/2)

- For estimating $\pi \approx \pi_*$, in pseudocode form, the off-policy MC control algorithm is:

  Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

  $Q(s,a) \in \mathbb{R}$ (arbitrarily)
  $C(s,a) \leftarrow 0$
  $\pi(s) \leftarrow \arg\max_a Q(s,a)$ (with ties broken consistently)

  Loop forever (for each episode):

  $b \leftarrow$ any soft policy
  Generate an episode following $b : S_0, A_0, R_1, \cdots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  Loop for each step of episode, $t = T-1, T-2, \cdots, 0$:

  $G \leftarrow \gamma G + R_{t+1}$
  $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
  $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
  $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken consistently)
  If $A_t \neq \pi(S_t)$ then exit For loop
  $W \leftarrow W \frac{1}{b(A_t|S_t)}$

Tabular Solution Methods: Temporal-Difference Learning

Temporal-Difference Learning

- *Temporal-difference* (TD) learning is a combination of MC ideas and DP ideas.
- Like MC methods, TD methods can learn directly from raw experience without a model, and like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).
- These TD, MC, and DP ideas and methods blend into each other and can be combined in may ways.
  - In particular, there are n-step algorithms and $\mathrm{TD}(\lambda)$ algorithm.
- We start by focusing on the policy evaluation or prediction problem and control problem by some variation of GPI, as usual.

TD Prediction (1/3)

- Shortly, MC methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$, therefore a simple every-visit MC method for nonstationary environment is:

$$V(S_t) \leftarrow V(S_t) + \alpha\big[G_t - V(S_t)\big], \tag{48}$$

  where G_t is the actual return following time t, and α is a constant step-size parameter.

- On the other hand, TD methods need to wait only until the next time step, in other words, at time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$.

- The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha\big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big] \tag{49}$$

  immediately on transition to S_{t+1} and receiving R_{t+1}.

- This TD method is called *TD(0)*, or *one-step TD*, because it is a special case of the $\mathrm{TD}(\lambda)$ and n-step TD methods.

TD Prediction (2/3)

- For estimating v_π, the tabular TD(0) algorithm is:

  Input: the policy π to be evaluated
  Algorithm parameter: step size $\alpha \in (0, 1]$
  Initialize $V(s)$, for all $s \in \mathcal{S}$, arbitrarily except that $V(terminal) = 0$
  Loop for each episode:
    Initialize S
    Loop for each step of episode:
      $A \leftarrow$ action given by π for S
      Take action A, observe R, S'
      $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
      $S \leftarrow S'$
    until S is terminal

- Because TD(0) bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] \tag{50}$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \tag{51}$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_t + 1) | S_t = s] \tag{52}$$

TD Prediction (3/3)

- The value estimate for the state node at the top of the backup diagram is updated on the basis of the one sample transition from it to the immediately following state.

TD(0)

- We refer to TD and MC updates as *sample updates*, which differ from the *expected updates* of DP methods.
- We define *TD error*, δ_t, the quantity in brackets in the $\mathrm{TD}(0)$ update, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$.

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{53}$$

- The TD error at each time is the error in the estimate made at that time.
  - ∵ The TD error depends on the next state and next reward, it is not actually available until one time step later.

Advantages of TD Prediction Methods

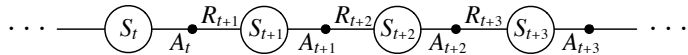
- TD methods update their estimates based in part on other estimates.
- They learn a guess from a guess, *i.e.*, they *bootstrap*.

- TD methods do not require a model of the environment (its reward and next-state probability distributions), whereas DP methods require it.
- TD methods are naturally implemented in an online, fully incremental fashion whereas MC methods are not.

- Furthermore, TD methods guarantee convergence to the correct answer when the methods learn one guess from the next, without waiting for an actual outcome.
  - For any fixed policy π, $\mathrm{TD}(0)$ has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions.

Sarsa: On-policy TD Control (1/3)

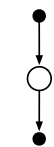
- To tackle the control problem in TD manner, as usual, the first step is to learn an action-value function rather than a state-value function.



- We consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs:

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

- This update uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to the next.
- As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π.


Sarsa

Sarsa: On-policy TD Control (2/3)

- For estimating $Q \approx q_*$, the Sarsa algorithm is;

  Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

  Initialize $Q(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, arbitrarily except that $Q(terminal, \cdot) = 0$

  Loop for each episode:

  Initialize S
  Choose A from S using policy derived from Q (*e.g.*, ε-greedy)
  Loop for each step of episode:

  Take action A, observe R, S'
  Choose A' from S' using policy derived from Q (*e.g.*, ε-greedy)
  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
  $S \leftarrow S'$; $A \leftarrow A'$;

  until S is terminal

- The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q.
  - For instance, one could use ε-greedy policy.

Sarsa: On-policy TD Control (3/3)

- We use 'Taxi-v2' environment again to validate the Sarsa algorithm[5].

```
# Let us implement on-policy TD control, Sarsa algorithm.
# Define hyperparameters
alpha, epsilon, gamma = .4, .1, .0
num_states = env.observation_space.n
num_actions = env.action_space.n
num_iterations = 10000
# Initialize
Q = np.zeros([num_states, num_actions])

# Loop for each episode
for iter in range(num_iterations):
  S = env.reset() # Initialize S
  # Epsilon-greedy XX
  if np.random.rand() < epsilon: A = env.action_space.sample() # take a random action
  else: A = np.argmax(Q[S, :])

  # Loop for each step of episode
  done = False
  while not done:
    S_prime, R, done, info = env.step(int(A))
    # Epsilon-greedy XX
    if np.random.rand() < epsilon: A_prime = env.action_space.sample()
    else: A_prime = np.argmax(Q[S_prime, :])
    Q[S, A] += alpha * (R + gamma * Q[S_prime, A_prime] - Q[S, A])
    S, A = S_prime, A_prime
```

[5]This code was implemented by Ko.

Q-learning: Off-policy TD Control (1/2)

- One of the breakthroughs in RL was the development of off-policy TD control algorithm known as *Q-learning*:

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

- In this case, the learned action-value function Q directly approximates the optimal action-value function q_*, independent of the policy being followed.

- For estimating $\pi \approx \pi_*$, the Q-learning algorithm is;

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (*e.g.*, ε-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    until S is terminal

Q-learning: Off-policy TD Control (2/2)

- We also implement Q-learning algorithm.

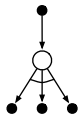
```
# Let us implement off-policy TD control, Q-learning algorithm.
# Define hyperparameters
alpha, epsilon, gamma = .4, .1, .9
num_states = env.observation_space.n
num_actions = env.action_space.n
num_iterations = 1000
# Initialize
Q = np.zeros([num_states, num_actions])

# Loop for each episode
for iter in range(num_iterations):
  S = env.reset() # Initialize S
  # Loop for each step of episode
  done = False
  while not done:
    # Epsilon-greedy XX
    if np.random.rand() < epsilon: A = env.action_space.sample() # take a random action
    else: A = np.argmax(Q[S, :])
    S_prime, R, done, info = env.step(int(A))
    Q[S, A] += alpha * (R + gamma * np.max(Q[S_prime, :]) - Q[S, A])
    S = S_prime
```

Expected Sarsa (1/2)

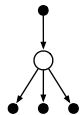
- We can consider an algorithm uses the expected value whereas Q-learning uses the maximum over next state-action pairs to update current policy:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned} \tag{56}$$

  and it is called *expected Sarsa* algorithm.



Q-learning



Expected Sarsa

- Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1}.
  - Given the same amount of experience, expected Sarsa generally perform slightly better than Sarsa or Q-learning.

Expected Sarsa (2/2)

- In this expected Sarsa algorithm implementation, we use ε-greedy policy, however if the policy is the greedy policy then this algorithm is exactly same with Q-learning.

```
# Let us implement expected Sarsa algorithm.
# Define hyperparameters
alpha, epsilon, gamma = .4, .1, .9
num_states, num_actions = env.observation_space.n, env.action_space.n
num_iterations = 1000
# Initialize
Q = np.zeros([num_states, num_actions])

# Loop for each episode
for iter in range(num_iterations):
  S = env.reset() # Initialize S
  # Loop for each step of episode
  done = False
  while not done:
    # Epsilon-greedy XX
    if np.random.rand() < epsilon: A = env.action_space.sample() # take a random action
    else: A = np.argmax(Q[S, :])
    # Calculate Sarsa moving distance in expectation
    tmp = 0
    for a in range(Q.shape[-1]):
      if a == A: tmp += (1-epsilon)*Q[S_prime, a]
      else: tmp += epsilon*(1/(num_actions-1))*Q[S_prime, a]
    S_prime, R, done, info = env.step(int(A))
    Q[S, A] += alpha * (R + gamma * tmp - Q[S, A])
    S = S_prime
```

Maximization Bias and Double Learning (1/3)

- We have a maximization operation for all the control algorithms that we have discussed, *e.g.*, Q-learning, ε-greedy policy, *etc.*
- In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias.
  - For example, when the true values $q(s,a)$ are all zero but its estimated value $Q(s,a)$ are uncertain thus distributed some above and some below zero, then the maximum of $q(s,a)$ is zero, but the maximum of $Q(s,a)$ is positive, called *maximization bias*.
- One way to view the problem is that using the same samples (plays) both to determine the maximizing action and to estimate its value.
- We can use one estimate, Q_1 to determine the maximizing action $A^* = \arg\max_a Q_1(a)$, and the other, Q_2 to provide the estimate of its value, $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$, and this is the idea of *double learning*.

Maximization Bias and Double Learning (2/3)

- The double learning idea extends naturally to algorithms for full MDPs:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big], \quad (57)$$

  or have same update with Q_1 and Q_2 switched with 0.5 probability.

- There are double versions of Q-learning, Sarsa, and expected Sarsa, and for example, double Q-learning algorithm to estimate $Q_1 \approx Q_2 \approx q_*$ is;

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$
Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
        Take action A, observe R, S'
        With 0.5 probability:
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha(R + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A))$
        else:
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha(R + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A))$
    until S is terminal

Maximization Bias and Double Learning (3/3)

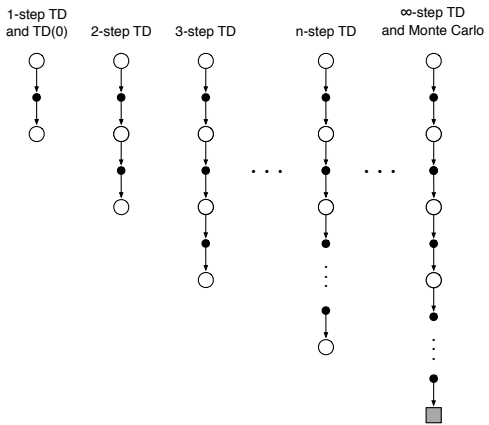
- The double Q-learning is also implemented in Google Colab.

```
# Let us implement Double Q-learning algorithm.
# Define hyperparameters
alpha, epsilon, gamma = .4, .1, .9
num_states, num_actions = env.observation_space.n, env.action_space.n
num_iterations = 1000
# Initialize
Q1, Q2 = np.zeros([num_states, num_actions]), np.zeros([num_states, num_actions])

# Loop for each episode
for iter in range(num_iterations):
  S = env.reset() # Initialize S
  # Loop for each step of episode
  done = False
  while not done:
    # Epsilon-greedy XX
    if np.random.rand() < epsilon: A = env.action_space.sample() # take a random action
    else:
      tmp = Q1+Q2
      A = np.argmax(tmp[S, :])
    S_prime, R, done, info = env.step(int(A))
    # Flip a coin
    if np.random.rand() < 0.5:
      Q1[S, A] += alpha * (R + gamma * Q2[S_prime, np.argmax(Q1[S_prime, :])] - Q1[S, A])
    else:
      Q2[S, A] += alpha * (R + gamma * Q1[S_prime, np.argmax(Q2[S_prime, :])] - Q2[S, A])
    S = S_prime
```

Tabular Solution Methods: n-step Bootstrapping

n-step Bootstrapping

- We can unify the MC methods and the one-step TD methods.
- n-step methods span a spectrum with MC methods at one end and one-step TD methods at the other.



n-step TD Prediction (1/2)

- For MC updates, the estimate of $v_\pi(S_t)$ is updated in the direction of the complete return, *i.e.*, the *target* of the update:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

where T is the last time step of the episode.

- Whereas in MC updates the target is the return, in one-step updates (*e.g.*, $\mathrm{TD}(0)$) the target is the *one-step return*:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}).$$

- Then we can consider the target for an arbitrary n-step update, *n-step return*:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \tag{58}$$

for all n, t such that $n \geq 1$ and $0 \leq t \leq T - n$.

- The natural state-value learning algorithm for using n-step returns is thus:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)], \qquad 0 \leq t \leq T. \tag{59}$$

n-step TD Prediction (2/2)

- For estimating $V \approx v_\pi$, the n-step TD prediction algorithm is;

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily
All store and access operations (for S_t and R_t) can take their index mod $n + 1$
Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \cdots$:
        If $t < T$, then:
            Take an action according to $\pi(\cdot|S_t)$
            Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
            If S_{t+1} is terminal, then $T \leftarrow t + 1$
        $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
        If $\tau \geq 0$:
            $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
            If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$
            $V(S_\tau) \leftarrow V(S_\tau) + \alpha[G - V(S_\tau)]$
    Until $\tau = T - 1$

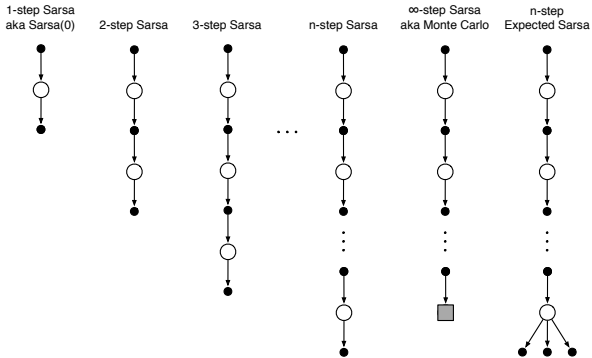
n-step Sarsa (1/4)

- We can redefine n-step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \tag{60}$$

for all $n \geq 1$ and $0 \leq t \leq T - n$, then the natural algorithm is:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \qquad 0 \leq t \leq T. \tag{61}$$



n-step Sarsa (2/4)

- For estimating $Q \approx q_*$ or q_π, the n-step Sarsa (control) algorithm is;

Initialize $Q(s,a)$ arbitrarily, $\forall\ s \in \mathcal{S}$, $a \in \mathcal{A}$

Initialize π to be ε-greedy with respect to Q, or to a fixed given policy

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n

All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:

Initialize and store $S_0 \neq$ terminal
Select and store an action $A_0 \sim \pi(\cdot|S_0)$
$T \leftarrow \infty$
Loop for $t = 0, 1, 2, \cdots$:
Loop A
Until $\tau = T - 1$

- Loop A

If $t < T$, then:

Take action A_t
Observe and store R_{t+1} and S_{t+1}
If S_{t+1} is terminal, then: $T \leftarrow t+1$
else: Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$

If $\tau \geq 0$:

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$
$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$
If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy w.r.t. Q

Machine Intelligence Lab

n-step Sarsa (3/4)

- Obviously, this n-step update idea can be applied to the expected Sarsa algorithm.
- We can also redefine the n-step return as:

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}), \qquad t+n < T, \tag{62}$$

where $\bar{V}_t(s)$ is the *expected approximate value* of state s, using the estimated action values at time t, under the target policy:

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) Q_t(s,a), \qquad \text{for all } s \in \mathcal{S}. \tag{63}$$

- If s is terminal, then its expected approximate value is defined to be 0.

n-step Sarsa (4/4)

- The n-step Sarsa is implemented in Google Colab[6].

```
# Let us implement n-step Sarsa algorithm.
num_states, num_actions = env.observation_space.n,
      env.action_space.n
num_iterations = 1000
alpha, epsilon, gamma, n = .4, .1, .9, 4
# Initialize
Q, pi = np.zeros([num_states, num_actions]),
      np.zeros([num_states])
# Initialize policy with epsilon-greedy XX
for i in range(pi.shape[0]):
  if np.random.rand() < epsilon: pi[i] =
        env.action_space.sample() # take a random action
  else: pi[i] = np.argmax(Q[i, :])

# Loop for each episode
for iter in range(num_iterations):
  S_t, A_t, R_t = [], [], []
  S = env.reset() # Initialize S
  S_t.append(S) # Store S
  A = pi[S]
  A_t.append(int(A)) # Store A
  T = 1e+10 # infinity
```

```
for t in range(0, int(1e+10)):
  if t < T:
    S, R, done, _ = env.step(int(A)) # Take action
    R_t.append(R)
    S_t.append(S)
    if done == True: T = t + 1
    else:
      A = pi[S]
      A_t.append(int(A))
  tau = t - n + 1
  if not tau < 0: G = 0
    for i in range(tau+1, min(tau+n, T)): G += np.power(gamma,
          i-tau-1) * R_t[i]
    if (tau+n) < T: G += np.power(gamma, n) * Q[S_t[tau+n],
          A_t[tau+n]]
    Q[S_t[tau], A_t[tau]] += alpha * (G - Q[S_t[tau],
          A_t[tau]])
    # Learning policy with epsilon-greedy XX
    if np.random.rand() < epsilon: pi[S_t[tau]] =
          env.action_space.sample()
    else: pi[S_t[tau]] = np.argmax(Q[S_t[tau], :])
  if tau == T - 1: break
```

[6]This code was implemented by Ko.

n-step Off-policy Learning (1/2)

- To make a simple off-policy version of n-step TD, the update for time t (actually made at time $t+n$) can simply be weighted by $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\rho_{t:t+n-1}[G_{t:t+n} - V_{t+n-1}(S_t)], \qquad 0 \le t < T, \tag{64}$$

  where $\rho_{t:t+n-1}$, called the *importance sampling ratio*, is the relative probability under the two policies of taking the n actions from A_t to A_{t+n-1} (*cf.*, Eq. (42)):

$$\rho_{t:h} \doteq \prod_{k=t}^{\min(h,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{65}$$

- Then, the n-step Sarsa update can be completely replaced by a simple off-policy form:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha\rho_{t+1:t+n}[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \qquad 0 \le t < t. \tag{66}$$

n-step Off-policy Learning (2/2)

- For estimating $Q \approx q_*$ or q_π, the off-policy n-step Sarsa algorithm is;

Input: an arbitrary behavior policy b

Initialize $Q(s,a)$ arbitrarily, $\forall\ s \in \mathcal{S}$, $a \in \mathcal{A}$

Initialize π to be greedy w.r.t. Q, or as a fixed given policy

Algorithm parameters: $\alpha \in (0,1]$, $\varepsilon > 0$, n

All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:

Initialize and store $S_0 \neq$ terminal

Select and store an action $A_0 \sim b(\cdot|S_0)$

$T \leftarrow \infty$

Loop for $t = 0, 1, 2, \cdots$:

Loop A

Until $\tau = T - 1$

- Loop A

If $t < T$, then:

Take action A_t

Observe and store R_{t+1} and S_{t+1}

If S_{t+1} is terminal, then: $T \leftarrow t+1$

else: Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$

If $\tau \geq 0$:

$\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1,T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$

If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$

$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho[G - Q(S_\tau, A_\tau)]$

If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy w.r.t. Q

Machine Intelligence Lab

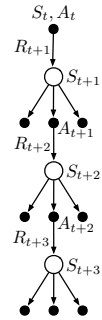
Off-policy Learning without Importance Sampling (1/2)

- Without importance sampling, the off-policy learning can be expanded to n-step bootstrapping, using the *tree-backup algorithm*.
- In the tree-backup update, the target includes all these things plus the estimated values of the dangling action nodes hanging off the sides, at all levels.
- The redefined n-step return for the tree-backup algorithm is:

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}, \qquad t < T-1,\ n \geq 2 \quad (67)$$

- This target is then used with the usual action-value update rule from n-step Sarsa:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \qquad 0 \leq t < T. \quad (68)$$



the 3-step tree-backup update

Off-policy Learning without Importance Sampling (2/2)

- For estimating $Q \approx q_*$ or q_π, the off-policy n-step tree backup algorithm is;

Initialize $Q(s,a)$, π

Algorithm parameters: $\alpha \in (0,1]$, $\varepsilon > 0$, n

All store and access operations can take their index mod $n+1$

Loop for each episode:

Initialize and store $S_0 \neq$ terminal
Choose and store an action A_0 arbitrarily
$T \leftarrow \infty$
Loop for $t = 0, 1, 2, \cdots$:
Loop A
Until $\tau = T - 1$

- Loop A

If $t < T$, then:

Take action A_t
Observe and store R_{t+1} and S_{t+1}
If S_{t+1} is terminal, then: $T \leftarrow t+1$
else: Choose and store an action A_{t+1} arbitrarily

If $\tau \geq 0$:

If $T+1 \geq t$: $G \leftarrow R_T$
else: $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
Loop for $k = \min(t, T-1)$ down through $\tau + 1$:

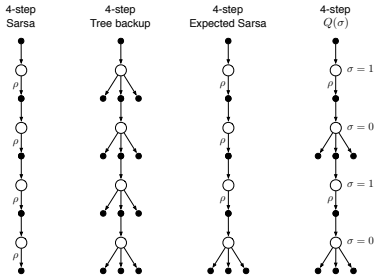
$$G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$$

$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$
If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy w.r.t. Q

A Unifying Algorithm: n-step $\mathrm{Q}(\sigma)$ (1/2)

- Let $\sigma_t \in [0, 1]$ denote the degree of sampling on step t.
  - $\sigma = 1$: full sampling and $\sigma = 0$: a pure expectation without sampling
  - The random variable σ_t might be set as a function of the state, action, or state-action pair at time t.
- We can develop a new algorithm, n-step $\mathrm{Q}(\sigma)$ with a new target:

$$G_{t:h} \doteq R_{t+1} + \gamma\Big(\sigma_{t+1}\rho_{t+1} + (1 - \sigma_{t+1})\pi(A_{t+1}|S_{t+1})\Big)\Big(G_{t+1:h} - Q_{h-1}(S_{t+1}, A + t + 1)\Big) + \gamma\bar{V}_{h1}(S_{t+1}), \qquad t < h \leq T. \tag{69}$$



A Unifying Algorithm: n-step $\mathrm{Q}(\sigma)$ (2/2)

- For estimating $Q \approx q_*$ or q_π, the off-policy n-step $\mathrm{Q}(\sigma)$ algorithm is;

Input: an arbitrary behavior policy b

Initialize $Q(s,a)$, π

Algorithm parameters: $\alpha \in (0,1]$, $\varepsilon > 0$, n

All store and access operations can take their index mod $n+1$

Loop for each episode:

Initialize and store $S_0 \neq$ terminal
Choose and store an action $A_0 \sim b(\cdot|S_0)$
$T \leftarrow \infty$
Loop for $t = 0, 1, 2, \cdots$:
Loop A
Until $\tau = T - 1$

- Loop A

If $t < T$, then:

Take action A_t; Observe and store R_{t+1} and S_{t+1}
If S_{t+1} is terminal, then: $T \leftarrow t+1$
else:
Choose and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
Select and store σ_{t+1}
Store $\frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$ as ρ_{t+1}

$\tau \leftarrow t - n + 1$

If $\tau \geq 0$:

$G \leftarrow 0$
Loop for $k = \min(t, T-1)$ down through $\tau + 1$:
if $k = T$: $G \leftarrow R_T$
else: $\bar{V} \leftarrow \sum_a \pi(a|S_k) Q(S_k, a)$; $G \leftarrow$ Eq. (69)
$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$
If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy w.r.t. Q

Tabular Solution Methods:
Planning and Learning with Tabular Methods

Planning and Learning with Tabular Methods

- We can consider a unified view of RL methods that require:
  - a model of the environment, such as DP and heuristic search (*i.e.*, *model-based*),
  - and methods that can be used without a model, such as MC and TD methods (*i.e.*, *model-free*).
- Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*.
- Although there are real differences between these two kinds of methods, there are also great similarities.
  - Both kinds of methods compute value functions.
  - All the methods are based on looking ahead to future events, computing a backed-up value, and then using it as an update target for an approximate value function.

Models and Planning (1/2)

- A *model* of the environment: anything that an agent can use to predict how the environment will respond to its actions
  - *Distribution models*: models that produce a description of all possibilities and their probabilities
  - *Sample models*: models that produce just one of the possibilities, sampled according to the probabilities
- Models can be used to mimic or *simulate experience*.
- We use the planning as a computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment[7].

$$\text{model} \xrightarrow{\text{planning}} \text{policy}$$

[7] There are two distinct approaches to planning, *state-space planning* and *plan-space planning*.

Models and Planning (2/2)

- There are two basic ideas for state-space planning:
  - **(1)** all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy,
  - **(2)** they compute value functions by updates or backup operations applied to simulated experience.

$$\text{model} \longrightarrow \begin{matrix}\text{simulated} \\ \text{experience}\end{matrix} \xrightarrow{\text{backups}} \text{values} \longrightarrow \text{policy}$$

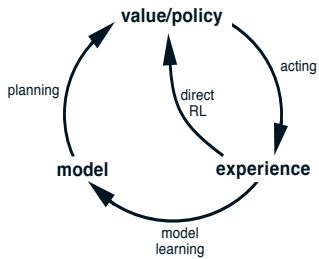
- For instance, a random-sample one-step tabular Q-planning algorithm is;

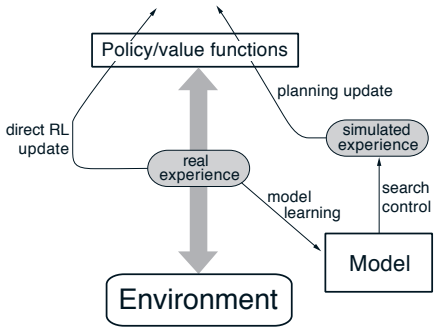
Loop forever:

Select a state, $S \in \mathcal{S}$, and an action $A \in \mathcal{A}$, at random
Send S, A to a sample model, and obtain
a sample next reward, R, and a sample next state, S'
Apply one-step tabular Q-learning to S, A, R, S':
$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

Dyna: Integrated Planning, Acting, and Learning (1/3)

- Dyna-Q: a simple architecture integrating the major functions needed in an online planning agent
- Within a planning agent, there are at least two roles for real experience:
  - it can be used to improve the model (to make it more accurately match the real environment, *i.e.*, *model learning*),
  - it can be used to directly improve the value function and policy using the kind of RL methods (*i.e.*, *direct RL*).





Dyna: Integrated Planning, Acting, and Learning (2/3)

- The tabular Dyna-Q algorithm is;

  Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$

  Loop forever:

  $S \leftarrow$ current (nonterminal) state

  $A \leftarrow \varepsilon$-greedy(S,Q)

  Take action A; observe resultant reward, R, and state, S'

  a $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

  b $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)

  c Loop repeat n times:

  $S \leftarrow$ random previously observed state

  $A \leftarrow$ random action previously taken in S

  $R, S' \leftarrow Model(S,A)$

  $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

- The lines **a**, **b**, and **c** respectively denote direct RL, model-learning, and planning.
  - If **b** and **c** are omitted, the remaining algorithm will be one-step tabular Q-learning.

Dyna: Integrated Planning, Acting, and Learning (3/3)

- We also implement the Dyna-Q algorithm[8].

```
num_states, num_actions = env.observation_space.n, env.action_space.n
num_iterations = 1000
alpha, epsilon, gamma, n = .4, .1, .9, 5
Q, model = np.zeros([num_states, num_actions]), np.zeros([num_states, num_actions, 2])

for iter in range(num_iterations):
  S = env.reset() # Initialize S
  trajectory = np.zeros([num_states, num_actions])
  done = False
  while not done:
    env.s = S
    if np.random.rand() < epsilon: A = env.action_space.sample() # take a random action
    else: A = np.argmax(Q[S, :])
    S_prime, R, done, info = env.step(int(A))
    Q[S, A] += alpha * (R + gamma * np.max(Q[S_prime, :]) - Q[S, A]) # Direct RL
    model[S, A, 0], model[S, A, 1] = R, int(S_prime) # Model Learning (Assuming deterministic env.)
    trajectory[S, A] = 1
    current = S_prime
    for i in range(n): # if n=0, this algorithm do not have planning.
      S = np.random.choice(np.nonzero(np.sum(trajectory, -1))[0]) # random previously observed state
      A = int(np.random.choice(trajectory[S, :])) # random action previously taken in S
      R, S_prime = model[S, A, 0], int(model[S, A, 1])
      Q[S, A] += alpha * (R + gamma * np.max(Q[S_prime, :])-Q[S, A])
    S = current
```

[8]This code was implemented by Ko.

Prioritized Sweeping

- It is natural to prioritize the updates according to a measure of their urgency, and perform them in order of priority.
  - This is the idea behind prioritized sweeping.
- Prioritized sweeping for a deterministic environment algorithm is;

Initialize $Q(s,a)$, $Model(s,a)$, for all s, a, and $PQueue$ to empty

Loop forever:

$S \leftarrow$ current (nonterminal) state
$A \leftarrow policy(S,Q)$
Take action A; observe R, S'
$Model(S,A) \leftarrow R, S'$
$P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$
If $P > \theta$, then insert S, A into $PQueue$ with priority P
Loop repeat n times, while $PQueue$ is not empty:

Loop A

- Loop A

$S, A \leftarrow first(PQueue)$
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow$
$Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:
$\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
$P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S}, \bar{A})|$
if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

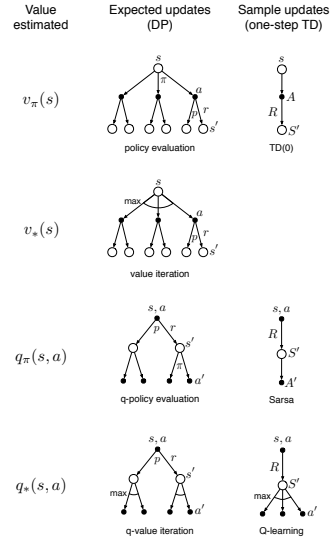
Expected vs. Sample Updates

- For value function updates, these updates vary primarily along three binary dimensions.
  - The first two dimensions are whether update state values or action values.
- The other dimension is whether the updates are *expected* updates or *sample* updates.
- The expected update for a state-action pair, s, a, is:

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r|s,a)\Big[r + \gamma \max_{a'} Q(s',a')\Big]. \tag{70}$$

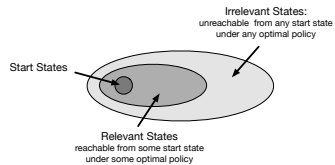
- The corresponding sample update for s, a, given a sample next state and reward, S' and R (from the model) is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[R + \gamma \max_{a'} Q(S',a') - Q(s,a)\Big]. \tag{71}$$



Trajectory Sampling

- The classic distributing updates approach from DP is to perform sweeps through the entire state (or state-action) space, updating each state (or state-action pair) once per sweep.
  - This is problematic on large tasks!
- Another approach is to sample from the state or state-action space according to some distribution.
  - One is to distribute updates according to the on-policy distribution, *i.e.*, according to the distribution observed when following the current policy.
- Sample state transitions and rewards are given by the model, and sample actions are given by the current policy.
  - One simulates explicit individual trajectories and performs updates at the state or state-action pairs encountered along the way, and we call this way of generating experience and updates *trajectory sampling*.
- *Real-time dynamic programming* (RTDP) is an on-policy trajectory-sampling version of the value iteration algorithm of DP.



Machine Intelligence Lab

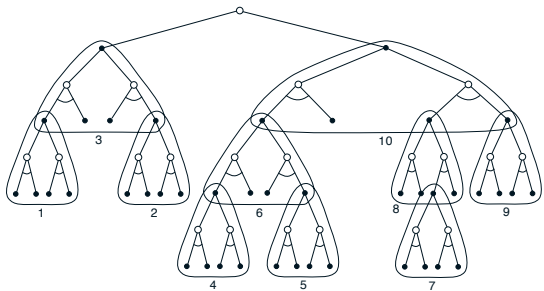
Planning at Decision Time

- *Background planning*: using simulated experience to gradually improve a policy or value function
  - typified by DP and Dyna
  - Before an action is selected for any current state S_t, planning has played a part in improving the table entries, or the mathematical expression, needed to select the action for many states, including S_t.

- *Decision-time planning*: using simulated experience to select an action for the current state
  - Decision-time planning is to begin and complete it after encountering each new state S_t, as a computation whose output is the selection of a single action A_t; on the next step planning begins anew with S_{t+1} to produce A_{t+1}, and so on.

Heuristic Search

- The classical state-space planning methods in artificial intelligence are decision-time planning methods collectively known as *heuristic search*.
- Heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step.
- The point of searching deeper than one step is to obtain better action selections.
  - If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies.



Machine Intelligence Lab

Rollout Algorithms

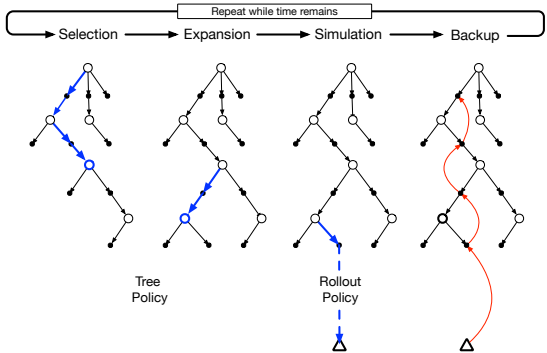
- *Rollout algorithms* are decision-time planning algorithms based on MC control applied to simulated trajectories that all begin at the current environment state.
- Unlike the MC control algorithms, the goal of a rollout algorithm is not to estimate q_* or q_π for given π.
  - The algorithm produces MC estimates of action values only for each current state and for a given policy usually called the *rollout policy*.
- As decision-time planning algorithms, rollout algorithms make immediate use of these action-value estimates, then discard them.
- For the rollout algorithms, there is no need to sample outcomes for every state-action pair, and there is no need to approximate a function over either the state space or the state-action space.

Monte Carlo Tree Search (1/2)

- *Monte Carlo Tree Search* (MCTS) is a recent and strikingly successful example of decision-time planning.
- MCTS is executed after encountering each new state to select the agent's action for that state; it is executed again to select the action for the next state, and so on.
- A basic version of MCTS consists of the following four steps:
  1. *Selection* Starting at the root node, a tree policy based on the action values attached to the edges of the tree traverses the tree to select a leaf node.
  2. *Expansion* On some iterations, the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.
  3. *Simulation* From the selected node, or from one of its newly-added child nodes, simulation of a complete episode is run with actions selected by the rollout policy.
  4. *Backup* The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS.

Monte Carlo Tree Search (2/2)

- By incrementally expanding the tree, MCTS effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state-action pairs visited in the initial segments of high-yielding sample trajectories.
- MCTS avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.



Machine Intelligence Lab

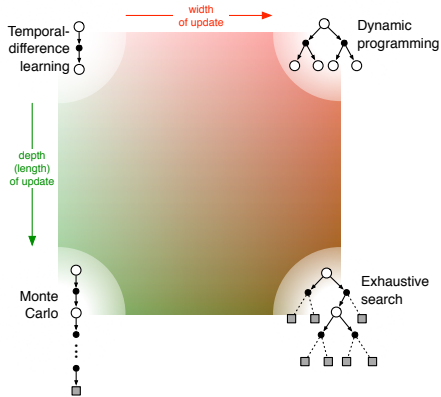
Summary of Tabular Solution Methods: Terms

- Agent; an active-decision maker, learner
- Environment; everything outside the agent
  - Trajectory; $S_t, A_t, R_{t+1}, \cdots, S_{T-1}, A_{T-1}, R_T$
  - Return; G_t
  - Transition; $p(s', r|s, a)$
  - Policy; $\pi(a|s)$
    - On-policy and off-policy
- State-value function v
  - Evaluation (prediction)
- Action-value function q
  - Improvement (control)
- Dynamic Programming, Monte Carlo Methods, and Temporal-Difference Learning
  - Planning; fully-known environment
    - Model-based
  - Learning
    - Model-free and model-based

Machine Intelligence Lab

Summary of Tabular Solution Methods: Dimensions

- The two important dimensions: the depth and width of the updates



- A third dimension is the binary distinction between on-policy and off-policy methods.

Machine Intelligence Lab

Remaining Topics

- Tabular Methods:
  - 1st week:
    - Introduction
    - Multi-armed Bandits
    - Finite Markov Decision Processes
    - Dynamic Programming
  - 2nd week:
    - Monte Carlo Methods
    - Temporal-Difference Learning
  - 3rd week:
    - n-step Bootstrapping
    - Planning and Learning with Tabular Methods

- Approximate Solution Methods:
  - 4th week:
    - On-policy Prediction with Approximation
    - On-policy Control with Approximation
  - 5th week:
    - Off-policy Methods with Approximation
    - Eligibility Traces
    - Policy Gradient Methods

- Looking Deeper:
  - 6th week:
    - Applications and Case Studies
    - Frontiers

References

- R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT press, 2018.
  - Online free available, Click here to download.
- C. Szepesvári, *Algorithms for Reinforcement Learning*, Morgan & Claypool Publishers, 2010.
  - Online free available, Click here to donwload.
- A. Juliani, *Simple Reinforcement Learning with Tensorflow*, 2017.
  - 번역본: 강화학습 첫걸음, 한빛미디어
  - A MEDIUM blog written by author
- 이웅원 외, *파이썬과 케라스로 배우는 강화학습*, 위키북스, 2017.
  - A gitbook written in Korean
- K. Arulkumaran *et al.*, "Deep Reinforcement Learning: A Breif Survey," *IEEE Signal Processing Magazine*, 2017.
  - Click here to download.
- Reinforcement Learning course by David Silver
  - Video lectures and lecture notes are online available., Click here to go to course.

Thank You!

(Q & A)

wjko@korea.ac.kr