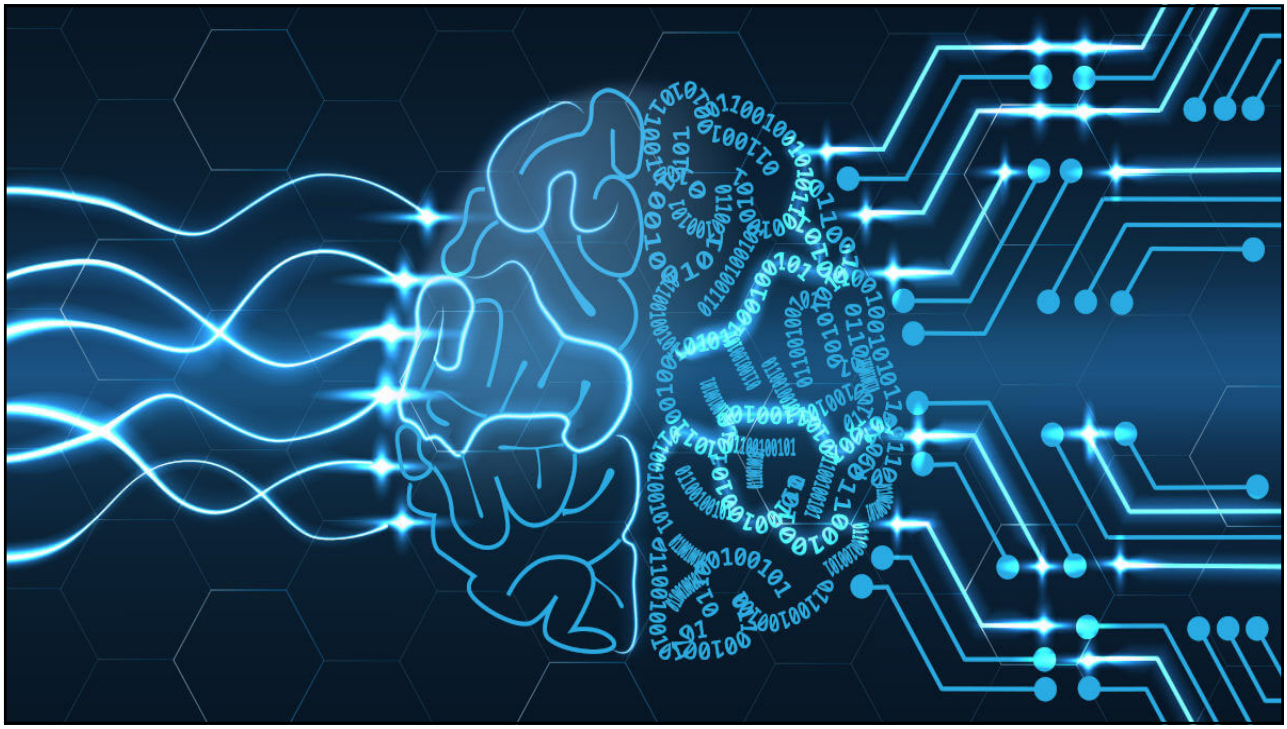
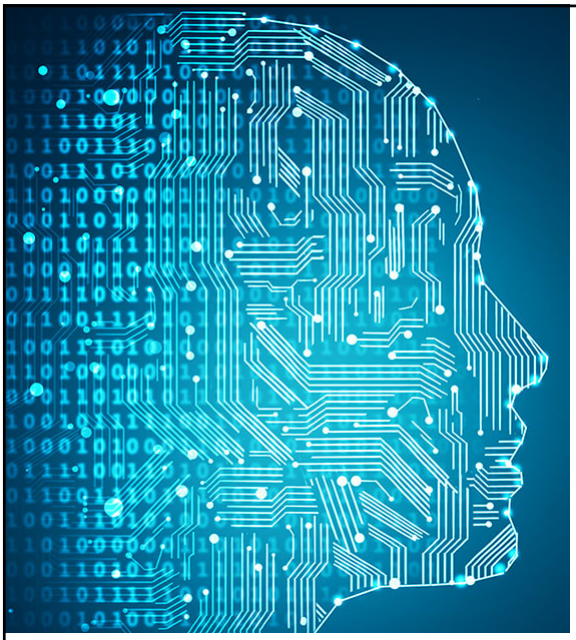


Fundamentals of Reinforcement Learning

Based on the course by the University of Alberta and Deeplizard and
Reinforcement Learning, An Introduction, Richard S., Sutton and Andrew G. Barto

Amin Jamili

2025



Reinforcement Learning

Multi-armed Bandit

Reinforcement Learning - Part 1

Reinforcement Learning

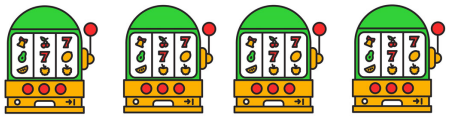
How do we know that hitting a rock is bad?

RL vs ML: The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that **evaluates the actions** taken rather than instructs by giving correct actions.

In RL we are going to train an agent in an environment to make decisions for how that agent is navigating that environment.

A k-armed Bandit Problem

You are faced repeatedly with a choice among k different options, or actions.



After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected.

Your Multi-armed Bandits objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps.

Is defined as — Reward — given

Value of action a $q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$

Expectation of — Action selected at time t

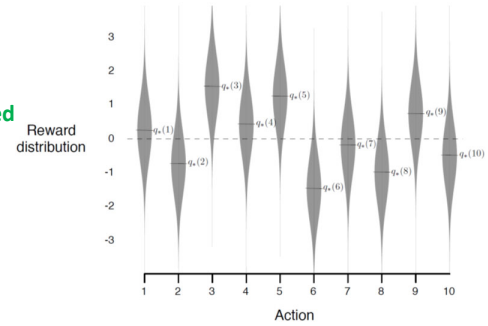
If we know the value of each action, we would always select the action with highest value.

A k-armed Bandit Problem - Estimation of $q_*(a)$

Is defined as — Reward — given

Value of action a $q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$

Expectation of — Action selected at time t



If we know the value of each action, we would always select the action with highest value.
As we don't know the action values with certainty, we are going to find the $Q_t(a)$ to be close to $q_*(a)$.

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 you select one of these actions, we say that you are **exploiting** your current knowledge of the values of the actions.

If instead you select one of the nongreedy actions, then we say you are **exploring**, because this enables you to improve your estimate of the nongreedy action's value.

Action-value Methods

We begin by looking more closely at methods for estimating the values of actions and for using the estimates to make action selection decisions, which we collectively call **action-value methods**. Recall that the true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received:

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

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

Greedy action which always **exploits** $A_t \doteq \arg\max_a Q_t(a),$

ε-greedy method: A simple alternative is to behave greedy most of the time, but every once in a while, with probability ε, select randomly. We explore with ε-greedy method.
As every action will be sampled an infinite number of times, $\boldsymbol{Q_t(a)}$ **converges to** $\boldsymbol{q_*(a)}$.

The 10-armed Testbed

To roughly assess the relative effectiveness of the greedy and ε -greedy action-value methods, we compared them numerically on a set of 2000 randomly generated k-armed bandit problems.

In the case of choosing arm # 3, it is possible to achieve a weak reward and on the other hand, by choosing arm # 2, a better solution may arises.
Therefore, selecting just a greedy action couldn't be the best approach.

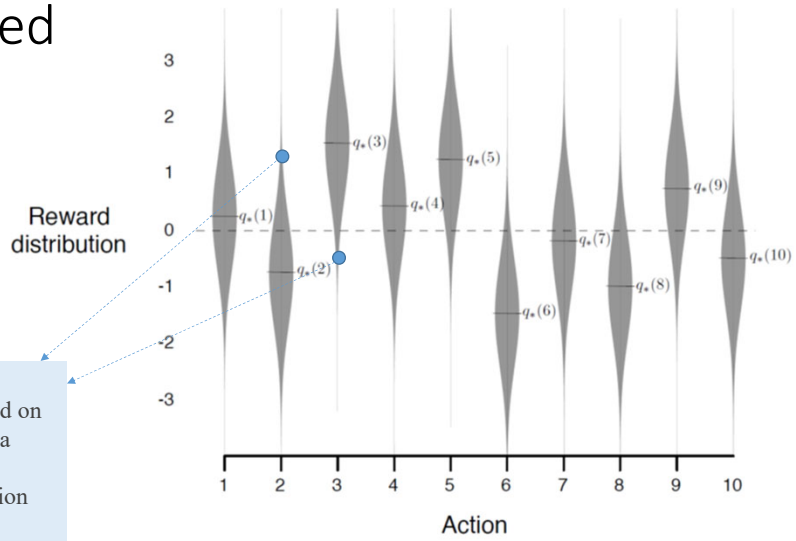
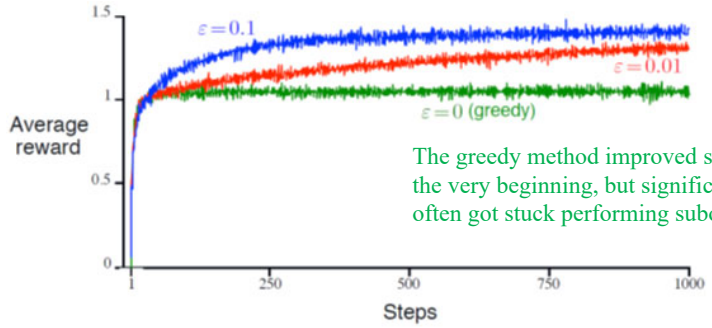


Figure 2.1: An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$ unit variance normal distribution, as suggested by these gray distributions.

Average performance of ε-greedy action-value methods on the 10-armed testbed.

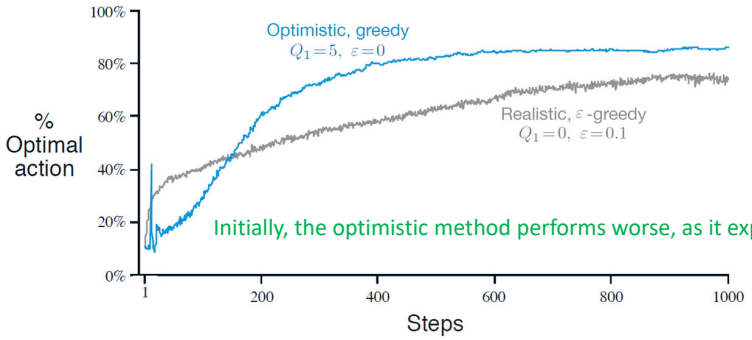
These data are averages over 2000 runs with different bandit problems.
Each Run includes applying 1000 time steps to one of the bandit problems.



The greedy method improved slightly faster than the other methods at the very beginning, but significantly worse in the long run because it often got stuck performing suboptimal actions

Optimistic Initial Values

All the methods are dependent to some extent to the initial action-value estimates.
The initial action value can be used as a way to encourage exploration.



Initially, the optimistic method performs worse, as it explores more.

It is not well suited to nonstationary problems since its drive for exploration is inherently temporary.

Upper-Confidence-Bound Action Selection

This method is an upgrade to the ε-greedy method in a way that consider preferences to those actions that are near greedy or particularly uncertain.

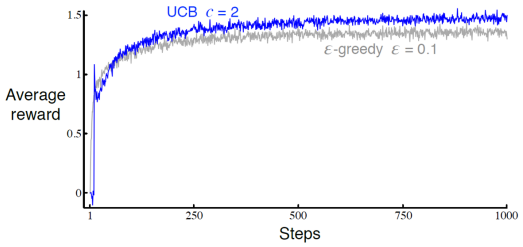
The use of ln (Natural Logarithm) means that the increase get smaller over time.

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

Denotes the number of times that action a has been selected prior to time t.

The number c>0 controls the degree of exploration.

Actions with lower value estimates or that have already been selected frequently, will be selected with decreasing frequency over time.



Incremental Implementation

Reward received after i^{th} selection of this action

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1} \quad \Rightarrow \quad Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i$$

Estimate of the action value

Conventional Computing method

To tackle the averages in a computationally efficient manner

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

Step size parameter

$$Q_{n+1} \doteq Q_n + \alpha\left[R_n - Q_n\right]$$

In non-stationary problem, It makes sense to give more weight to recent rewards than to long-past rewards

Pseudocode for a complete bandit algorithm

using incrementally computed sample averages and ε-greedy action selection

A simple bandit algorithm

Initialize, for $a = 1$ to k:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:
$$A \leftarrow \begin{cases} \text{argmax}_a\, Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

The function *bandit* (A) is assumed to take an action and return a corresponding reward.

Tracking a Nonstationary Problem

The averaging methods discussed so far are appropriate for stationary bandit problems, that is, for bandit problems in which the reward probabilities do not change over time.

As noted earlier, we often encounter reinforcement learning problems that are effectively nonstationary. In such cases it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is to use a constant step-size parameter.

step-size parameter. For example, the incremental update rule for updating an average Q_n of the $n-1$ past rewards is modified to be

$$Q_{n+1} \doteq Q_n + \alpha\Big[R_n - Q_n\Big],$$

Sometimes it is convenient to vary the step-size parameter from step to step. Let $\alpha_n(a)$ denote the step-size parameter used to process the reward received after the nth selection of action a. As we have noted, the choice $\alpha_n(a) = \frac{1}{n}$ results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of the sequence $\{\alpha_n(a)\}$. A well-known result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty.$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence.

Note that both convergence conditions are met for the sample-average case, $\alpha_n(a) = \frac{1}{n}$, but not for the case of constant step-size parameter, $\alpha_n(a) = \alpha$. In the latter case, the second condition is not met, indicating that the estimates never completely converge but continue to vary in response to the most recently received rewards.

Figure 2.3 shows the performance on the 10-armed bandit testbed of a greedy method using $Q_1(a) = +5$, for all a. For comparison, also shown is an ε-greedy method with $Q_1(a) = 0$. Initially, the optimistic method performs worse because it explores more, but eventually it performs better because its exploration decreases with time. We call this technique for encouraging exploration *optimistic initial values*. We regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration. For example, it is not well suited to nonstationary problems because its drive for exploration is inherently temporary.

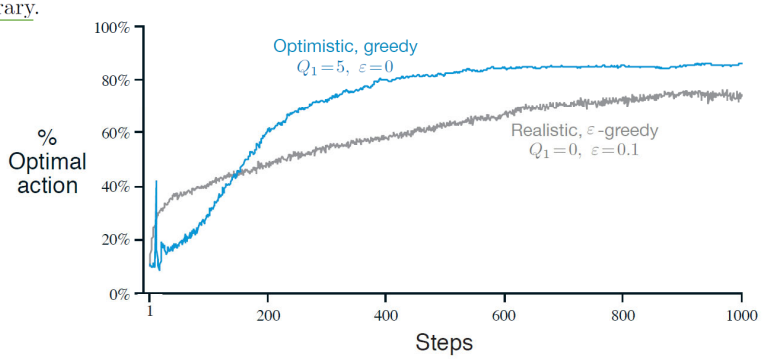
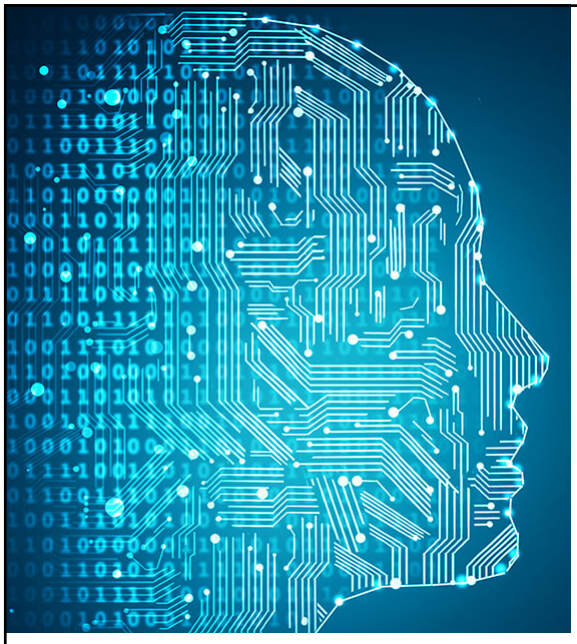


Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

If the task changes, creating a renewed need for exploration, this method cannot help. Indeed, any method that focuses on the initial conditions in any special way is unlikely to help with the general nonstationary case. The beginning of time occurs only once, and thus we should not focus on it too much. This criticism applies as well to the sample-average methods, which also treat the beginning of time as a special event, averaging all subsequent rewards with equal weights. Nevertheless, all of these methods are very simple, and one of them—or some simple combination of them—is often adequate in practice.



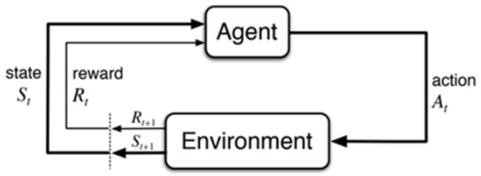
Reinforcement Learning

Finite Markov Decision Processes

Reinforcement Learning - Part 2

Markov Decision Processes

The finite Markov decision processes, or finite **MDPs**, involves evaluative feedback, as in bandits, but also an associative aspect—choosing different actions in different situations. In MDPs actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Whereas in bandit problems we estimated the value $q_*(a)$ of each action a, in MDPs we estimate the value $q_*(s, a)$ of each action a in each state s.



The learner or decision maker is called the **agent**.
The thing it interacts with, comprising everything outside the agent, is called the **environment**. Then, the agent selects **actions** and the environment responds to these actions and presents new situations to the agent which is called **states**. The environment also gives rise to **rewards**, special numerical values that the agent seeks to maximize over time through its choice of actions.

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

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

Use of a reward signal to formalize the idea of a goal

The **goals** should be formulated in terms of **reward signals**. In practice it has proved to be flexible and widely applicable.

1. To make a robot learn to walk: The reward is proportional to the robot's forward motion, on each time step .
2. To make a robot learn how to escape from a maze as quickly as possible: The reward is −1 for every time step that passes prior to escape to encourage the agent to escape ASAP.
3. To make a robot learn to find and collect empty soda cans for recycling: The reward is +1 for each can collected and zero on the other states. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it.
4. For an agent to learn to play checkers or chess, the natural rewards are +1 for winning, −1 for losing, and 0 for drawing and for all nonterminal positions.

پاداش راهی است جهت برقراری ارتباط با Agent (ربات) برای دستیابی به چیزی که می‌خواهیم و نه اینکه چگونه می خواهیم به آن برسیم.

Note on defining the reward signal

- In particular, the reward signal **is not the place to impart to the agent prior knowledge about how to achieve what we want it to do**. (Better places for imparting this kind of prior knowledge are the **initial policy or initial value function**, or in influences on these.)
- For example, 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. If achieving these sorts of subgoals were rewarded, then the agent might find a way to achieve them without achieving the real goal. For example, it might find a way to take the opponent's pieces even at the cost of losing the game. The reward signal is your way of communicating to the robot what you want it to achieve, not how you want it achieved.

Returns

We have said that the agent's goal is to maximize the cumulative reward it receives in the long run.
the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized.

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

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

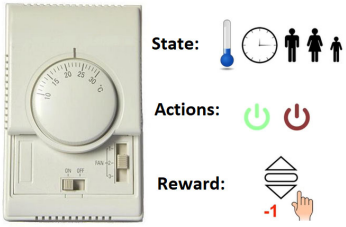
The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately.

when the agent–environment interaction breaks naturally into subsequences, which we call episodes, 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. Tasks with episodes of this kind are called episodic tasks.
On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. For example, this would be the natural way to formulate an on-going process-control task, or an application to a robot with a long life span. We call these continuing tasks.

Episodic Tasks vs Continuing Tasks

When the agent–environment interaction breaks naturally into subsequences, which we call episodes, 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. Tasks with episodes of this kind are called episodic tasks.

On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. For example, this would be the natural way to formulate an on-going process-control task like a smart thermostat, or an application to a robot with a long life span. We call these continuing tasks.

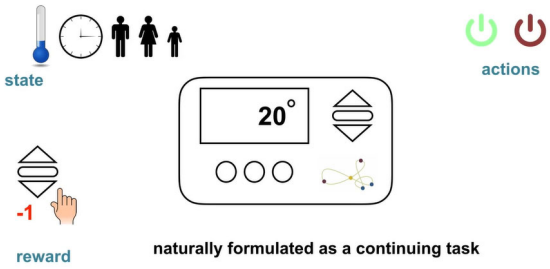


Reward: -1 every time someone has to **manually adjust** the temperature and **zero** otherwise.

A smart thermostat

Consider a smart thermostat which regulates the temperature of a building. This can be formulated as a continuing task since the thermostat never stops interacting with the environment.

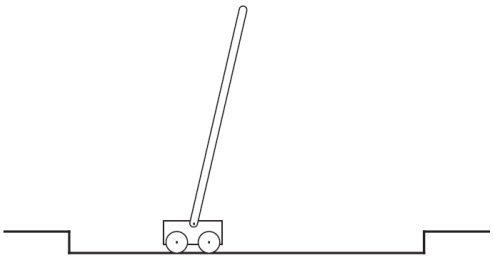
The state could be **the current temperature** along with details of the situation like the **time of day** and **the number of people in the building**.

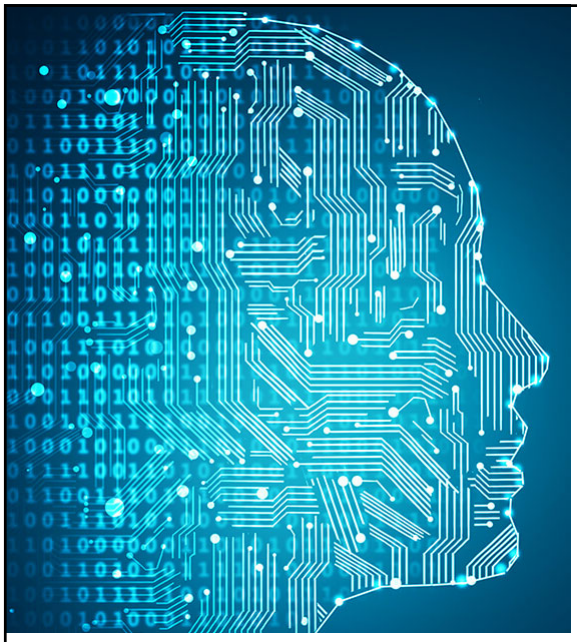


There are just **two actions**, **turn on the heater or turn it off**.
The reward to be **minus one** every time someone has to **manually adjust** the temperature and **zero** otherwise.

To avoid negative reward, the thermostat would learn to anticipate the user's preferences.

Example: Pole-Balancing
The objective in this task is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over: A failure is said to occur if the pole falls past a given angle from vertical or if the cart runs off the track. The pole is reset to vertical after each failure. This task could be treated as episodic, where the natural episodes are the repeated attempts to balance the pole. The reward in this case could be +1 for every time step on which failure did not occur, so that the return at each time would be the number of steps until failure. In this case, successful balancing forever would mean a return of infinity. Alternatively, we could treat pole-balancing as a continuing task, using discounting. In this case the reward would be −1 on each failure and zero at all other times. The return at each time would then be related to $-\gamma^K$, where K is the number of time steps before failure. In either case, the return is maximized by keeping the pole balanced for as long as possible.





Reinforcement Learning

Policies and Value Functions - 1

Reinforcement Learning - Part 3

Policy

Almost all reinforcement learning algorithms involve estimating value functions of states that estimate how good it is for the agent to be in a given state. The notion of "how good" here is defined in terms of future rewards that can be expected.
Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.

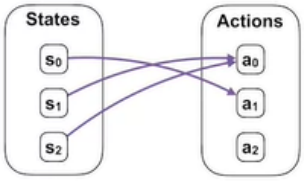
$$\pi(a|s) = P(A_t = a|S_t = s)$$

Formally, 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 π(a|s) is the probability that $A_t = a$ if $S_t = s$.

Deterministic policy notation

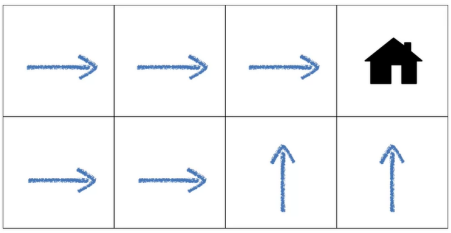
$$\pi(s) = a$$

Pi of S represents the action selected in state S by the policy Pi.



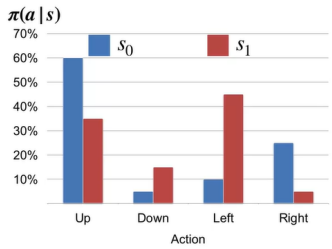
In this example, Pi selects the action A1
in state S0 and action A0 in states S1 and S2.

Example: deterministic policy



In this example an agent moves towards its house on a grid. The states correspond to the locations on the grid. The actions move the agent up, down, left, and right. The arrows describe one possible policy, which moves the agent towards its house. Each arrow tells the agent which direction to move in each state.

Stochastic policy notation



Here we show the distribution over actions for state S0 according to π. π in S1 corresponds to a completely different distribution over actions.

$$\pi(a \mid s)$$

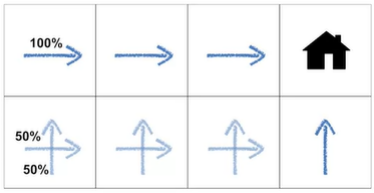
$$\sum_{a \in \mathcal{A}(s)} \pi(a \mid s) = 1$$

$$\pi(a \mid s) \geq 0$$

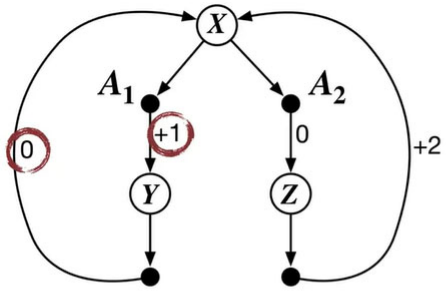
A stochastic policy is one where multiple actions may be selected with non-zero probability.
Remember that π specifies a separate distribution over actions for each state.

The sum over all action probabilities must be one for each state, and each action probability must be non-negative.

Example: stochastic policy



A stochastic policy might choose up or right with equal probability in the bottom row.
Notice the stochastic policy will take the same number of steps to reach the house as the deterministic policy we discussed before.



$\pi_1(X) = A_1 \quad \pi_2(X) = A_2$

$\gamma = 0$

$v_{\pi_1}(X) = 1$ ✓

$v_{\pi_2}(X) = 0$

$\gamma = 0.9$

$$v_{\pi_1}(X) = 1 + 0.9 * 0 + (0.9)^2 * 1 + \ldots$$

$$= \sum_{k=0}^{\infty} (0.9)^{2k} = \frac{1}{1 - 0.9^2} \approx 5.3$$

$$v_{\pi_2}(X) = 0 + 0.9 * 2 + (0.9)^2 * 0 + \ldots$$

$$= \sum_{k=0}^{\infty} (0.9)^{2k+1} * 2 = \frac{0.9}{1 - 0.9^2} * 2 \approx 9.5$$ ✓

Value function of state

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. For MDPs, we can define $v_\pi(s)$ formally by:

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

v_π : *state-value function for policy* π.

Note that the value of the terminal state, if any, is always zero.

Value functions are imperative in reinforcement learning, they allow an agent to **query the quality of its current situation** instead of waiting to observe the long-term outcome, even if the return is stochastic..
The value function summarizes all the possible futures by averaging over returns.

Action -Value function

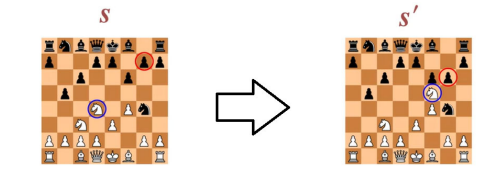
Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π :

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

We call q_π the *action-value function for policy* π.

Value function in game of chess

Consider an agent playing the game of chess which is an **episodic MDP**. The **state is given by the positions of all the pieces on the board.** In this two player game, **the opponent's move is part of the state transition**. For example, the environment moves both the agents piece and the opponent's piece.
Actions are the legal moves, and termination occurs when the game ends in either a win, loss, or draw.
The **reward is +1 for winning** and zero for all the other moves.



As the state value is equal to the expected sum of future rewards, and Since the only possible non-zero reward is +1 for winning, the **state value simply specify the probability of winning** if we follow the current policy π.

Bellman equation for state value function

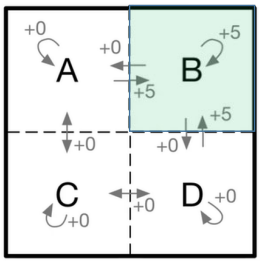
A fundamental property of value functions used throughout reinforcement learning is that they satisfy recursive relationships.

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

: The **Bellman equation**

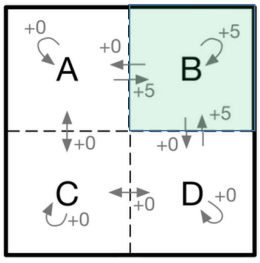
It expresses a relationship between the value of a state and the values of its successor states.

Grid word Example



States: Four states labeled A, B, C and D.
Actions: Moving up, down, left and right. (Actions which would move off the grid, keep the agent in place.)
Rewards: The reward is 0 everywhere except for any time the agent enters state B. which results a reward of +5. This includes starting in state B and hitting a wall to remain there.
Policy: Uniform random policy, which moves in every direction 25% of the time.
The discount factor gamma is 0.7.

What is the value of each of these states A, B, C and D under this policy?



The **Bellman equation:**

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right], \quad \text{for all } s \in \mathcal{S},$$

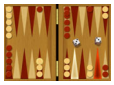
$$V_\pi(A) = \frac{1}{4}(5 + 0.7V_\pi(B)) + \frac{1}{4}0.7V_\pi(C) + \frac{1}{2}0.7V_\pi(A)$$
$$V_\pi(B) = \frac{1}{2}(5 + 0.7V_\pi(B)) + \frac{1}{4}0.7V_\pi(A) + \frac{1}{4}0.7V_\pi(D)$$
$$V_\pi(C) = \frac{1}{4}0.7V_\pi(A) + \frac{1}{4}0.7V_\pi(D) + \frac{1}{2}0.7V_\pi(C)$$
$$V_\pi(D) = \frac{1}{4}(5 + 0.7V_\pi(B)) + \frac{1}{4}0.7V_\pi(C) + \frac{1}{2}0.7V_\pi(D)$$

$$V_\pi(A) = 4.2$$
$$V_\pi(B) = 6.1$$
$$V_\pi(C) = 2.2$$
$$V_\pi(D) = 4.2$$

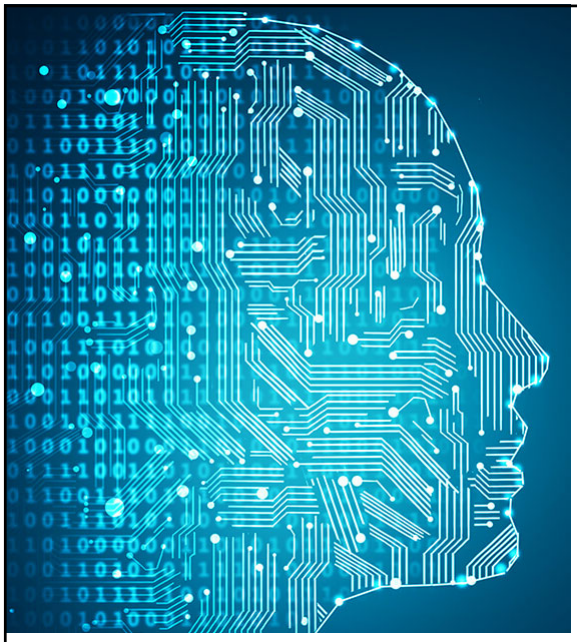
A system of 4 equations for 4 variables

The important thing to note is that the Bellman equation **reduced an unmanageable infinite sum over possible futures** to ,**a simple linear algebra problem**.

Number of states, Number of equations equals to 10^{45}



Number of states, Number of equations equals to 10^{20}



Reinforcement Learning

Policies and Value Functions - 2

Reinforcement Learning - Part 3

The Bellman Optimality equations

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

$$v_*(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

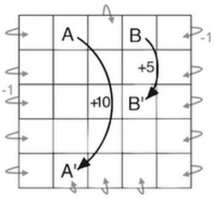
The Bellman optimality equation for the state value function

$$q_\pi(s,a) = \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\right]$$

$$q_*(s,a) = \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma \sum_{a'} \pi_*(a'|s') q_*(s',a')\right]$$

$$q_*(s,a) = \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$$

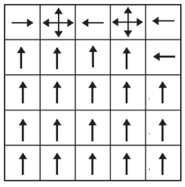
The Bellman optimality equation for action-value function



Gridworld Example: State A is followed by a reward of +10 and transition to state A′, while state B is followed by a reward of +5 and transition to state B′. The reward is zero everywhere else except for -1, for bumping into the walls. The discount factor is 0.9.

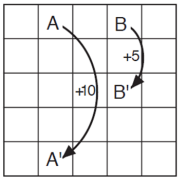
3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Value function by the **uniform random policy**



Updated policy.
Where there are multiple arrows in a cell, all of the corresponding actions are optimal.

$$\pi(s) = \underset{a}{\text{argmax}} \sum_{s'} \sum_{r} p(s', r \mid s, a)\left[r + \gamma v_{\pi}(s')\right] \text{ for all } s \in \mathcal{S}$$



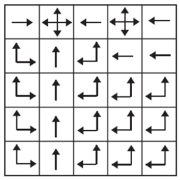
Gridworld Example: State A is followed by a reward of +10 and transition to state A′, while state B is followed by a reward of +5 and transition to state B′. The reward is zero everywhere else except for -1, for bumping into the walls. The discount factor is 0.9.

v_*

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Optimal value function

π_*



Optimal policies.
Where there are multiple arrows in a cell, all of the corresponding actions are optimal.

$$\pi(s) = \underset{a}{\text{argmax}} \sum_{s'} \sum_{r} p(s', r \mid s, a)\left[r + \gamma v_{\pi}(s')\right] \text{ for all } s \in \mathcal{S}$$

مثال

رباتی برای بازیافت زباله در اختیار داریم. وضعیت شارژ این ربات یا در حالت low و یا high قرار می‌گیرد. اقداماتی که می‌تواند اتخاذ کند در حالت high عبارت است از search و wait. در حالت low نیز search، wait و recharge.

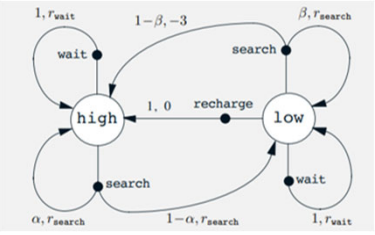
زمانی که ربات با حالت high شروع به فعالیت می‌کند، با احتمال a حالت بعدی آن نیز high است و با احتمال (1-a) low می‌باشد.

زمانی که ربات با حالت low کار خود را شروع کند، با احتمال B حالت بعدی آن نیز low است و با احتمال (1-B) باتری خالی می‌شود. پاداش‌ها عبارت‌اند از $r_{search} > r_{wait}$ و 3- که مربوط به زمانی است که باتری کاملا تخلیه و می‌بایست جایگزین شود.

$v_*(high)$ و $v_*(low)$ را محاسبه کنید.

۱۰۳

جدول و تصویر شماتیک مربوط به احتمالات حالات مختلف:



s	a	s'	$p(s'\|s,a)$	$r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-

۱۰۴

مثال ربات بازیافت

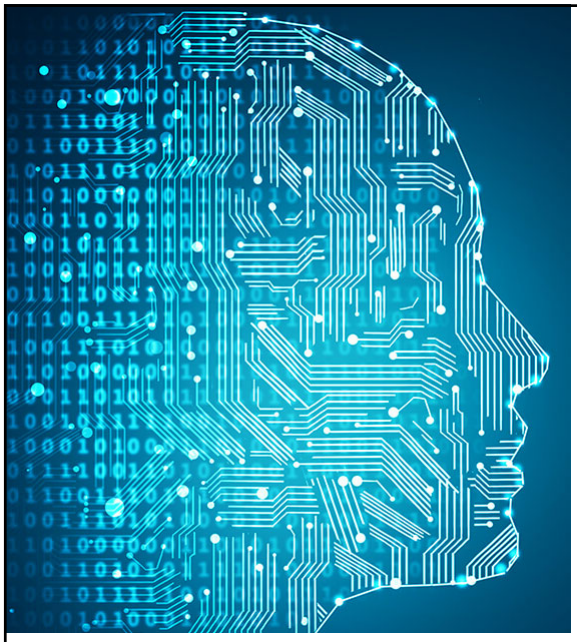
برای راحتی، stateهای high و low با نماد h و l نمایش داده شده اند و actionهای search، recharge و wait به ترتیب با s، r و w.

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right]$$ Bellman optimality equation

$$\begin{aligned}
v_*(\mathtt{h}) &= \max \left\{ \begin{array}{l} p(\mathtt{h}|\mathtt{h},\mathtt{s})[r(\mathtt{h},\mathtt{s},\mathtt{h}) + \gamma v_*(\mathtt{h})] + p(\mathtt{l}|\mathtt{h},\mathtt{s})[r(\mathtt{h},\mathtt{s},\mathtt{l}) + \gamma v_*(\mathtt{l})], \\ p(\mathtt{h}|\mathtt{h},\mathtt{w})[r(\mathtt{h},\mathtt{w},\mathtt{h}) + \gamma v_*(\mathtt{h})] + p(\mathtt{l}|\mathtt{h},\mathtt{w})[r(\mathtt{h},\mathtt{w},\mathtt{l}) + \gamma v_*(\mathtt{l})] \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} \alpha[r_\mathtt{s} + \gamma v_*(\mathtt{h})] + (1-\alpha)[r_\mathtt{s} + \gamma v_*(\mathtt{l})], \\ 1[r_\mathtt{w} + \gamma v_*(\mathtt{h})] + 0[r_\mathtt{w} + \gamma v_*(\mathtt{l})] \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} r_\mathtt{s} + \gamma[\alpha v_*(\mathtt{h}) + (1-\alpha) v_*(\mathtt{l})], \\ r_\mathtt{w} + \gamma v_*(\mathtt{h}) \end{array} \right\}.
\end{aligned}$$

$$v_*(\mathtt{l}) = \max \left\{ \begin{array}{l} \beta r_\mathtt{s} - 3(1-\beta) + \gamma[(1-\beta) v_*(\mathtt{h}) + \beta v_*(\mathtt{l})], \\ r_\mathtt{w} + \gamma v_*(\mathtt{l}), \\ \gamma v_*(\mathtt{h}) \end{array} \right\}$$

برای هر یک از مقادیر γ , β , α , r_w , r_s دقیقا یک جفت از $v_*(h)$ و $v_*(l)$ وجود دارد که هر 2 تساوی صادق باشد.



Reinforcement Learning - Part 4

Reinforcement Learning

Dynamic Programming

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]$$

Recall that

π, p, γ → Linear System Solver → v_π

In practice

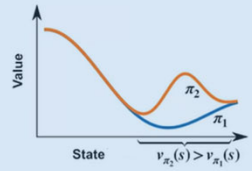
π, p, γ → Dynamic Programming → v_π

Policy Evaluation vs. Control

There exists two distinct tasks, policy evaluation and control:

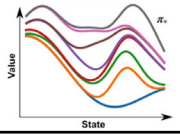
- **Policy evaluation** is the task of determining the **value function for a specific policy**. (determining v_π for a particular policy π.)
- **Control** is the task of **finding a policy to obtain as much reward as possible**. In other words, finding a policy which maximizes the value function.

Control is the ultimate goal of reinforcement learning. But the task of policy evaluation is usually a necessary first step. It's hard to improve our policy if we don't have a way to assess how good it is.



A policy π_2 is considered as good as or better than π_1 if the value under π_2 is greater than or equal to the value under π_1 in every state.
We say π_2 is strictly better than π_1 if π_2 is as good as or better than π_1 and there's at least one state where the value under π_2 is strictly greater than the value under π_1

The goal of the control task is to modify a policy to produce a new one which is strictly better.
Dynamic programming algorithms use the **Bellman equations** to define **iterative algorithms** for both policy evaluation and control.



Iterative Policy Evaluation

The Bellman equation provides a recursive expression for v_π.

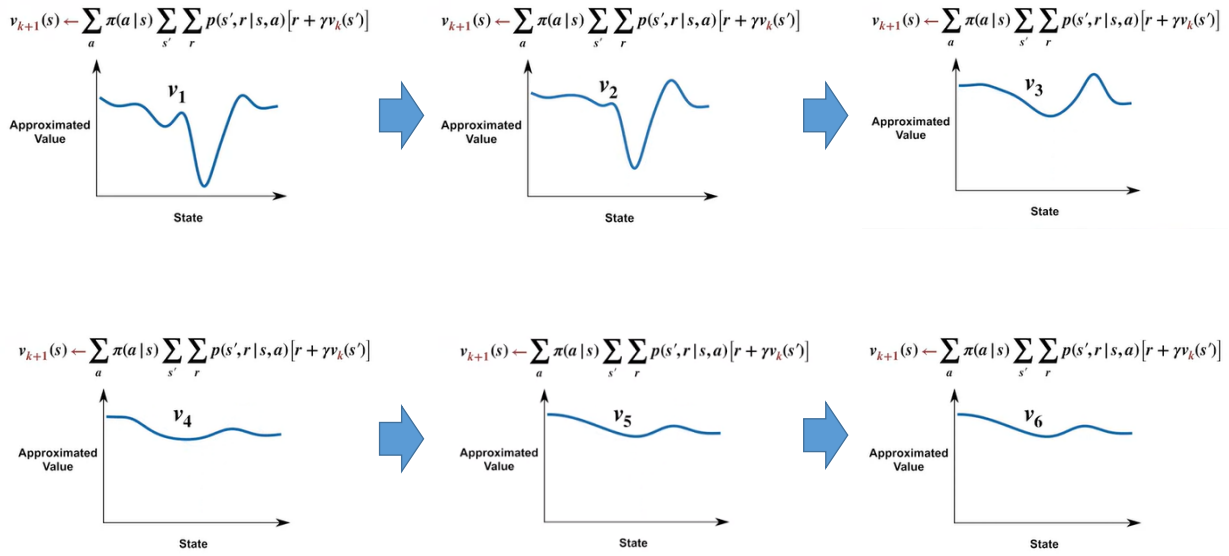
$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

The iterative policy evaluation is defined by a minor changes in Bellman equation to make an update rule.

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_k(s')\right]$$

Now, **instead of an equation** which holds for the true value function, we have a **procedure** we can apply to iteratively refine our estimate of the value function.
This will produce a sequence of better and **better approximations to the value function**.

- Each iteration applies this update rule to every state, S, in the state space, which we call a sweep.
- The iterations continue till the value function approximation doesn't change any more, that is, if v_{k+1} equals v_k for all states, which means we have found the value function.



Iterative Policy Evaluation Example (A four-by-four grid world)

An episodic MDP with the terminal state which are shaded and located in the top left and bottom right corners.

The reward is -1 on all transitions until the terminal state is reached. The value function represents the expected number of steps until termination from a given state.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

Suppose the agent follows the equiprobable random policy

Policy

۱۴۰

Iterative Policy Evaluation Pseudocode

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated — Take any policy we want to evaluate

$V \leftarrow \vec{0}, V' \leftarrow \vec{0}$ — initialize two arrays V and V prime.

Loop:

$\Delta \leftarrow 0$ — stopping parameter. The smaller the value we choose, the more accurate our final value estimate will be.

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

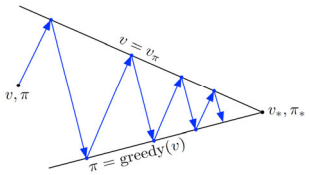
$$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$$

$\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$ — We track the largest update to this state value in a given iteration.

$V \leftarrow V'$

until $\Delta < \theta$ (a small positive number) — The outer loop continues until the change in the approximate value function becomes small.

Output $V \approx v_\pi$ — once the approximate value function stops changing, we have converged to Vπ.

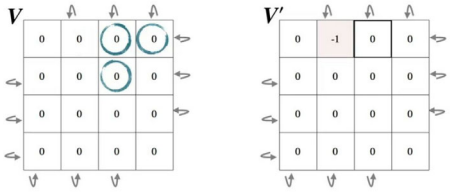


interaction between the evaluation and improvement processes

Iterative Policy Evaluation Example

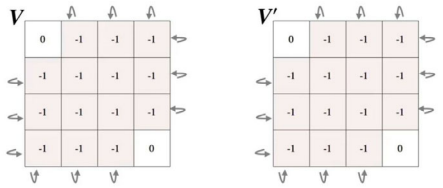
$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)[r + \gamma V(s')]$

$0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) = -1$



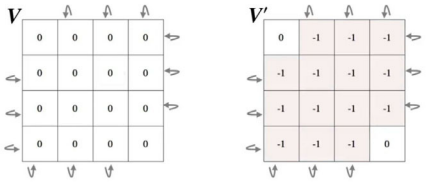
$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)[r + \gamma V(s')]$

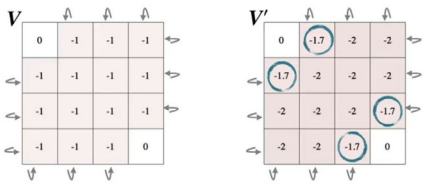
$\theta = 0.001 \quad \Delta = 1.0$



We've already completed one iteration, and the maximum change in value was 1.0. Since this is greater than 0.001, we carry on to the next iteration.

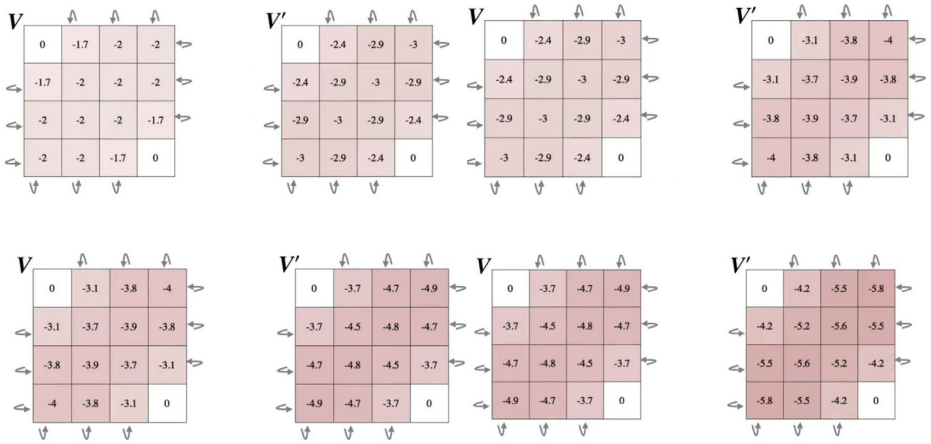
$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)[r + \gamma V(s')]$





After the second sweep, notice how the terminal state starts to influence the value of the nearest states.

Iterative Policy Evaluation Example

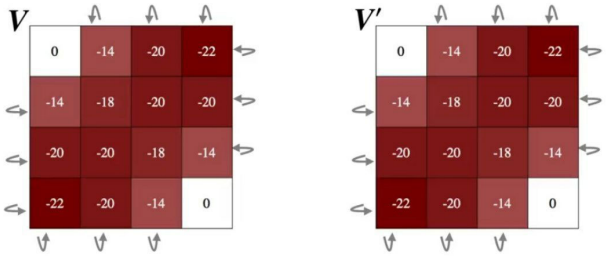


۱۴۴

Iterative Policy Evaluation Example

در انتها با توجه به شرط $\Delta < 0.001$ بررسی به پایان می‌رسد.

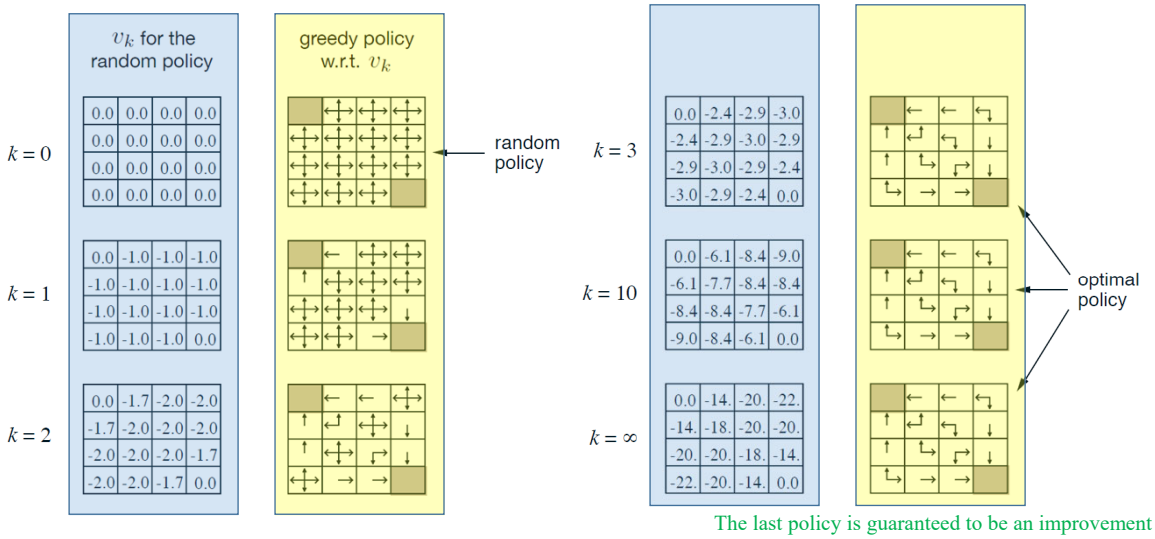
$\Delta < 0.001$



Let's keep running until our maximum delta is less than theta. Here is the result we eventually arrive at, our approximate value function has converged to the value function for the random policy, and we're done.

۱۴۵

Convergence of iterative policy evaluation



The last policy is guaranteed to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

Policy Iteration Steps

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

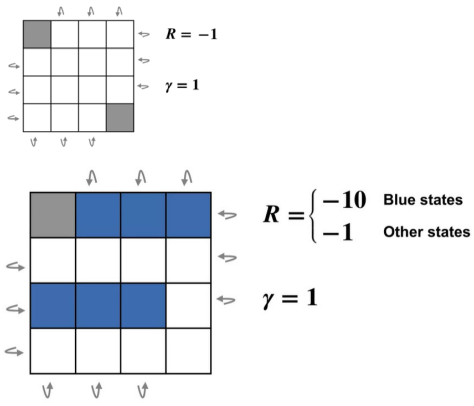
2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   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

Here's what this procedure looks like in pseudocode.
We initialize v and π in any way we like for each state s. Next, we call iterative policy evaluation to make V reflect the value of π.
This is the algorithm we learned earlier in this module.
Then, in each state, we set π to select the maximizing action under the value function.
If this procedure changes the selected action in any state, we note that the policy is still changing, and set policy stable to force.
After completing step 3, we check if the policy is stable.
If not, we carry on and evaluate the new policy.

Policy Iteration Example

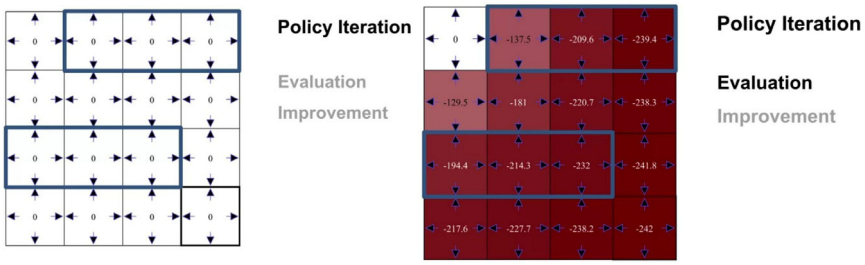
اکنون حالت کمی پیچیده تر مثال قبلی را بررسی می‌کنیم. این بار reward ورود به stateهای آبی مقدار 10- و سایر stateها 1- است.



Remember the four-by-four grid ruled example we used to demonstrate iterative policy evaluation.
Previously, we showed that by evaluating the random policy, and greedifying just once, we could find the optimal policy.
This is not a very interesting case for policy iteration.
Let's modify this problem a little bit to make the control task a bit harder. First, let's remove one of the terminal states so that there's only one way to end the episode.
Previously, each state admitted a reward of minus 1. Instead, let's add some especially bad states. These bad states are marked in blue.
Transitioning into them gives a reward of negative 10. The optimal policy should follow the winding low cost path in white to the terminal state. This additional complexity means that policy iteration takes several iterations to discover the path.

Policy Iteration Example

در ابتدا برآورد تمام stateها مقدار 0 است. اصلاح را آنقدر ادامه می‌دهیم تا شرط توقف برقرار شود.



Policy Iteration Example

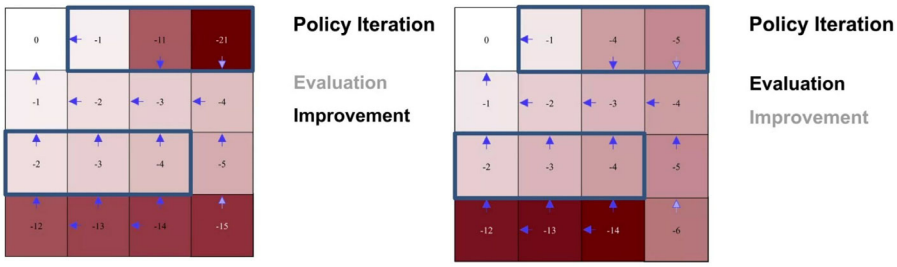
ابتدا با توجه به evaluation صورت گرفته، policy دچار تغییر می‌شود (شکل سمت چپ). اکنون policy جدید مورد ارزیابی واقع می‌شود و value هر state تعیین می‌گردد.

همین قدم‌ها آنقدر تکرار می‌شود تا به v_* دست یابیم.

۱۶۰

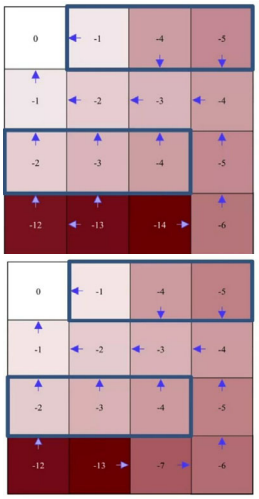
Policy Iteration Example

همین قدم‌ها آنقدر تکرار می‌شود تا به v_* دست یابیم.



۱۶۱

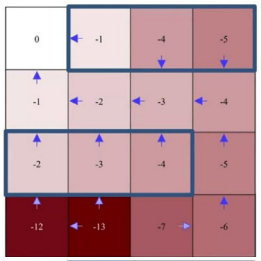
Policy Iteration Example



Policy Iteration

Evaluation

Improvement



Policy Iteration

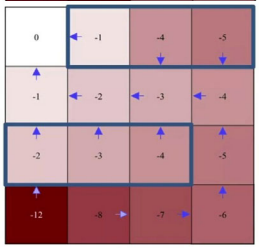
Evaluation

Improvement

Policy Iteration

Evaluation

Improvement



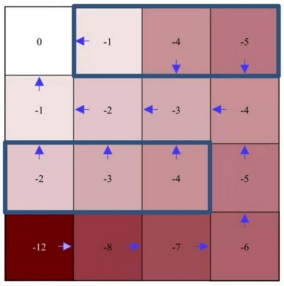
Policy Iteration

Evaluation

Improvement

۱۶۲

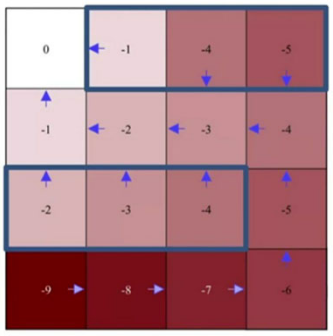
Policy Iteration Example



Policy Iteration

Evaluation

Improvement



در انتها به π_* و v_* دست پیدا می‌کنیم. همانطور که ملاحظه می‌کنید هرگز به خانه‌های آبی وارد نمی‌شویم و هر چه از terminal state فاصله می‌گیریم، state-value مقدار کوچکتری پیدا می‌کند.

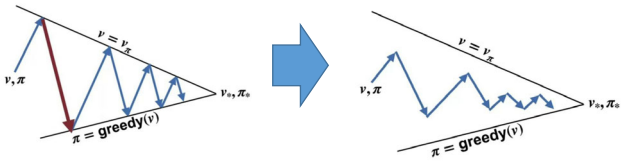
۱۶۳

Generalized Policy Iteration

یکی از ایرادات policy iteration، این است که در هر یک از تکرارها نیازمند ارزیابی policyها هستیم که خود می‌تواند کاری بسیار زمانبر باشد.

So far, we've presented policy iteration as a fairly rigid procedure.
We alternate between evaluating the current policy and greedify to improve the policy.
The framework of generalized policy iteration allows much more freedom than this while maintaining our optimality guarantees.

Policy Iteration **Generalized Policy Iteration**



Recall the dance of policy and value. The policy iteration algorithm runs each step all the way to completion.
Intuitively, we can imagine **relaxing** this. Imagine instead, we follow a trajectory like this.
Each evaluation step brings our estimate a little closer to the value of the current policy but not all the way.
Each policy improvement step makes our policy a little more greedy, but not totally greedy.
Intuitively, this process should still make progress towards the optimal policy and value function.
In fact, the theory tells us the same thing. We will use the term **generalized policy iteration** to **refer to all the ways we can interleave policy evaluation and policy improvement.**

۱۶۵

Generalized Policy Iteration

The policy iteration algorithm runs each step all the way to completion. Generally, we do not need to run **policy evaluation** to completion. **We may just perform just one sweep over all the states.**

Policy Iteration

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

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 positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   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

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, 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 $\Delta < \theta$

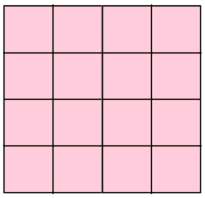
Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

Instead of choosing the action based on the policy (determined in Policy Improvement step), indeed in each sweep, we compute the best action in each state. In other words, after each complete sweep, we do directly Policy Improvement step.

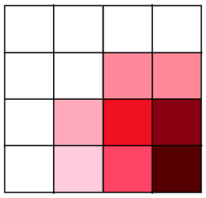
Asynchronous Dynamic Programming

Asynchronous dynamic programming algorithms **update the values of states in any order**, they do not perform systematic sweeps.
They might update a given state many times before another is updated even once.
In order to **guarantee convergence**, asynchronous algorithms must **continue to update the values of all states.**

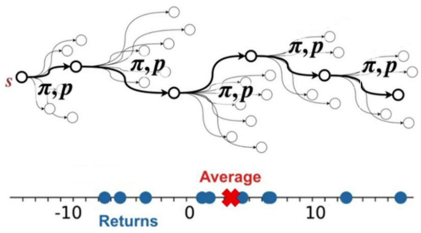
Synchronous DP



Asynchronous DP



Mote Carlo Method

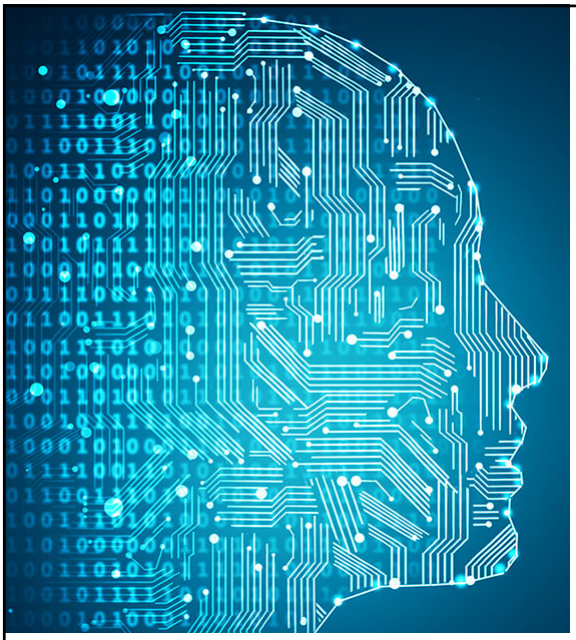


Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

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: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$

The method works by **running** simulations or **episodes** where an agent interacts with the environment until it reaches a terminal state. At the **end** of each **episode**, the **algorithm** looks back at the **states visited** and the **rewards received** to **calculate** what's known as the "**return**" — the cumulative reward starting from a specific state until the end of the episode. Monte Carlo policy evaluation **repeatedly simulates episodes**, tracking the total **rewards** that follow **each state** and then **calculating the average**. These averages give an **estimate of the state value** under the policy being followed.
By aggregating the results over many episodes, the method converges to the true value of each state when following the policy. These values are useful because they help us understand which states are more valuable and thus guide the agent toward better decision-making in the future. Over time, as the agent learns the value of different states, it can refine its policy, favouring actions that lead to higher rewards.



Reinforcement Learning

Q Learning

Reinforcement Learning - Part 5

Q learning is a reinforcement learning technique used for learning the optimal policy in a Markov decision process.

The Bellman Optimality equations

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

$$v_*(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_*(s')\right]$$

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_*(s')\right]$$

The Bellman optimality equation for the state value function

$$q_\pi(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\right]$$

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \sum_{a'} \pi_*(a'|s') q_*(s', a')\right]$$

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]$$

The Bellman optimality equation for action-value function

Optimal action-value function

Similarly, the optimal policy has an *optimal* action-value function, or *optimal* Q-function, which we denote as q_* and define as

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. In other words, q_* gives the largest expected return achievable by any policy π for each possible state-action pair.

Once we have our optimal Q function, q_*, we can determine the optimal policy by applying a reinforcement-learning algorithm to find the action that maximizes q_* for each state.
Q learning is to solve for the optimal policy in a MDP. The objective of Q learning is to find a policy that is optimal in the sense that the expected return over all successive time steps is the maximum achievable so in other words the goal of Q learning is to find the optimal policy by learning the optimal Q values for each state action pair.

Bellman optimality equation for q_*

One fundamental property of q_* is that it must satisfy the following equation.

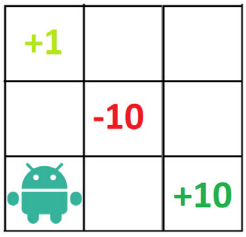
$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

This is called the *Bellman optimality equation*. It states that, for any state-action pair (s,a) at time t, the expected return from starting in state s, selecting action a and following the optimal policy thereafter (AKA *the Q-value* of this pair) is going to be the expected reward we get from taking action a in state s, which is R_{t+1}, plus the *maximum* expected discounted return that can be achieved from any possible next state-action pair (s',a').

The Q learning algorithm iteratively updates the Q values for each state action pair using the bellman equation until the Q function converges to the optimal Q function q_* this iterative approach is called value iteration.

An Example

- The agent in our environment is a **robot**.
- The actions include moving left right up or down
- The states are determined by each tile.
- The rewards are shown in the image. If the robot lands in tiles with -10 or +10 rewards, the episode ends.



Q table is to store the Q values for each state action pair.
At the start of the game the robot has absolutely **no idea of how good any given action** is from any given state, Therefore the **Q values** for each state action pair will all be **initialized to zero**.

		Actions			
		Left	Right	Up	Down
States	+1	0	0	0	0
	Empty 1	0	0	0	0
	Empty 2	0	0	0	0
	Empty 3	0	0	0	0
	-10	0	0	0	0
	Empty 4	0	0	0	0
	Empty 5	0	0	0	0
	Empty 6	0	0	0	0
	+10	0	0	0	0

REMINDER: The Bellman Optimality equations

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

$$v_*(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_*(s')\right]$$

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_*(s')\right]$$

The Bellman optimality equation for the state value function

$$q_\pi(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\right]$$

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \sum_{a'} \pi_*(a'|s') q_*(s', a')\right]$$

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]$$

The Bellman optimality equation for action-value function

REMINDER: The Bellman Optimality equations

The Bellman optimality equation for action-value function:

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]$$

$$q_*(s, a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s', a')\right]$$

Updating method of the Q values

The recursive formula to calculate the new Q-value for state-action pair:

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

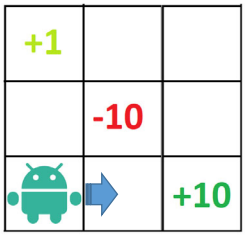
learning rate

The discounted estimate of the optimal future Q value of the next state-action pair

Exploitation

Exploration

Updating the Q Table



$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{new value}}$$

$$= (1-0.7)(0) + 0.7\left(-1 + 0.99\left(\max_{a'} q(s',a')\right)\right)$$

$$= (1-0.7)(0) + 0.7(-1 + 0.99(0))$$

$$= 0 + 0.7(-1) = -0.7$$

		Actions			
		Left	Right	Up	Down
States	+1	0	0	0	0
	Empty 1	0	0	0	0
	Empty 2	0	0	0	0
	Empty 3	0	0	0	0
	-10	0	0	0	0
	Empty 4	0	0	0	0
	Empty 5	0	-0.7	0	0
	Empty 6	0	0	0	0
	+10	0	0	0	0

Q-learning pseudocode

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

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

TD stands for Temporal Difference which means it doesn't need to continue to the end of the episode like MonteCarlo method to compute Q(S,A).

Offline (Off-Policy) Learning
Definition: The agent learns from **data generated by a different (older/exploratory) policy**. (It doesn't consider the action which is going to be selected like SARSA, but It consider the best action for the next move.)

- Separates *behavior policy* (for exploration) from *target policy* (for learning).

Online (On-Policy) Learning:
Definition: In On- Policy like SARSA Algorithm, the agent learns **while interacting with the environment** using the *current policy*.

SARSA vs. Q-Learning

In the SARSA algorithm, the Q-value is updated taking into account the action, a′, performed in the state, s′. In Q-learning, the action with the highest Q-value in the next state, s′, is used to update the Q-table.

$$\text{Q-learning}: \; Q_{t+1} = Q_t(s,a) + \alpha[r(s,a) + \gamma max_{a'} Q(s',a') - Q_t(s,a)]$$
$$\text{Sarsa} \quad : \; Q_{t+1} = Q_t(s,a) + \alpha[r(s,a) + \gamma Q(s',a') - Q_t(s,a)]$$

Both SARSA and Q-learning are useful for reinforcement learning, and each have their own uses based on agent needs. SARSA may be better as a non-deterministic algorithm and for choosing low-risk action paths in comparison to Q-learning.

Frozen Lake

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend. The surface is described using a grid like the following:

```
SFFF
FHFH
FFFH
HFFG
```

S is the agent **starting point** and it's considered safe for the agent to be here. **F** represents the **frozen surface** and is also safe **H** represents a **hole** and if our agent steps in a hole in the middle of a frozen lake, you **failed**. finally **G** represents the **goal** which is the space on the grid where the prized Frisbee is located. The agent can navigate left right up and down and the episode ends when the agent reaches the goal or falls in a hole It receives a reward of 1 if it reaches the goal and 0 otherwise.

First we're importing all the libraries

```
import numpy as np
import gym
import random
import time
from IPython.display import clear_output
```

```
env = gym.make("FrozenLake-v0")
```

To create our environment we just called gym.make and pass a string of the name of the environment we want to set up.

With this end object we can do several things:
we can query for information about the environment
we can sample states and actions
retrieve rewards and
have our agent navigate the frozen lake

```
action_space_size = env.action_space.n
state_space_size = env.observation_space.n

q_table = np.zeros((state_space_size, action_space_size))
print(q_table)
```

This section is to construct our Q table and initialize all the key values to zero for each state action pair.
Remember the number of rows in the table is equivalent to the size of the state space in the environment and the number of columns is equivalent to the size of the action space.

we can get this information using env.action.spac, end.observation.space. Then we use this information to build the Q table and fill it with zeros.

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

This part is to create and initialize all the parameters needed to implement the Q learning algorithm let's step through each of these:

```
num_episodes = 10000
max_steps_per_episode = 100

learning_rate = 0.1
discount_rate = 0.99

exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.01
exploration_decay_rate = 0.001
```

the total number of episodes we want our agent to play during training

maximum number of steps that our agent is allowed to take within a single episode
so if by the 100th step that agent hasn't reached the Frisbee or fallen through a hole then the episode will terminate with the agent receiving 0 points

alpha

gamma

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

Exploration exploitation trade-off to determine the selected action in each state.

```
rewards_all_episodes = []

# Q-learning algorithm
for episode in range(num_episodes):
    state = env.reset()
    done = False
    rewards_current_episode = 0

    for step in range(max_steps_per_episode):

        # Exploration-exploitation trade-off
        exploration_rate_threshold = random.uniform(0, 1)
        if exploration_rate_threshold > exploration_rate:
            action = np.argmax(q_table[state,:])
        else:
            action = env.action_space.sample()

        new_state, reward, done, info = env.step(action)
```

We create this list to hold all the rewards we'll get from each episode this will be so we can see how our game scores change over time.

In this block of code we'll implement the entire Q learning algorithm

This first for loop contains everything that happens within a single episode

For each episode we're going to first reset the state of the environment back to the starting state

The done variable just keeps track of whether or not our episode is finished so we initialize it to false and then it will get updated to notify us when the episode is over.

To keep track of the rewards within the current episode.

This second nested loop contains everything that happens for a single time step within each episode

Set the **exploration rate** threshold to a random number between 0 and 1 to determine whether our agent will explore or exploit the environment. It is initially set to 1.

Then our agent will **exploit** the environment and choose the action that has the highest Q value in the Q table for the current state.

The agent will **explore** the environment and sample in action randomly.

After our action is chosen we then take that action by calling step.env object and pass our action to it.
Step returns a tuple containing the **new state** the **reward** for the action we took whether or not the **action ended our episode** and some **diagnostic information** regarding our environment which may be helpful for us if we end up needing to do any debugging.

After we observe the reward we obtained from taking the action from the previous state we can then update the Q value for that state action pair in the Q table.

```
# Update Q-table for Q(s,a)
q_table[state, action] = q_table[state, action] * (1 - learning_rate) + \
    learning_rate * (reward + discount_rate * np.max(q_table[new_state, :])
```

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

This is done using the formula we introduced it earlier

```
state = new_state
rewards_current_episode += reward

if done == True:
    break
```

Next we set our current state to the new state that was returned to us once we took our last action.

Then update the rewards from our current episode by adding the reward we received for our previous action.

Then check to see if our last action ended the episode for the agent meaning did our agent step in a hole or reach the goal

So the episodes over if the action did in the episode then we jump out of this loop and move on to the next episode otherwise we transition to the next time step within this same episode

```
# Exploration rate decay
exploration_rate = min_exploration_rate + \
    (max_exploration_rate - min_exploration_rate) * np.exp(-exploration_decay_rate*episode)

rewards_all_episodes.append(rewards_current_episode)
```

Now once an episode is finished we need to update our exploration rate using exponential decay which just means that the exploration rate decreases or decays at a rate proportional to its current value we can decay the exploration rate using this formula which makes use of all the exploration rate parameters that we define last time.
Then just append the rewards from the current episode to the list of rewards from all episodes and next move on to the next episode.

```
# Calculate and print the average reward per thousand episodes
rewards_per_thosand_episodes = np.split(np.array(rewards_all_episodes),num_episodes/1000)
count = 1000
print("********Average reward per thousand episodes********\n")
for r in rewards_per_thosand_episodes:
    print(count, ": ", str(sum(r/1000)))
    count += 1000
```

After all episodes are finished we now are going to calculate the average reward per thousand episodes from our list that contains the rewards from all episodes.

```
# Print updated Q-table
print("\n\n********Q-table********\n")
print(q_table)
```

```
********Average reward per thousand episodes********

1000 :  0.16800000000000012
2000 :  0.32800000000000024
3000 :  0.46900000000000036
4000 :  0.5350000000000004
5000 :  0.6580000000000005
6000 :  0.6910000000000005
7000 :  0.6470000000000005
8000 :  0.6550000000000005
9000 :  0.6980000000000005
10000 :  0.7000000000000005
```

We can print it out and see how our rewards change over time during training from this printout we can see our average rewards per thousand episodes did indeed progress over time

Reaching the frisbee 70% of the time by the end of training isn't too shabby especially since the agent had no explicit instructions to reach the frisbee anyway it learned through reinforcement that this is the correct thing to do.

```
# Print updated Q-table
print("\n\n********Q-table********\n")
print(q_table)
```

```
********Q-table********

[[0.57804676 0.51767675 0.50499139 0.47330103]
 [0.07903519 0.16544989 0.16052137 0.45023559]
 [0.37592905 0.18333739 0.18905787 0.17227745]
 [0.01504804 0.         0.         0.        ]
 [0.59422496 0.42787803 0.43837162 0.45604075]
 [0.         0.         0.         0.        ]
 [0.1814022  0.13794979 0.31651935 0.09308381]
 [0.         0.         0.         0.        ]
 [0.43529839 0.32298132 0.36007182 0.64475741]
 [0.3369853  0.75303211 0.42246585 0.50627733]
 [0.65743421 0.48185693 0.32179817 0.35823251]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.53127669 0.63965638 0.86112718 0.53141807]
 [0.68753949 0.94078659 0.76545158 0.71566071]
 [0.         0.         0.         0.        ]]
```

lastly we can print out our updated Q table just so we can see how that's transitioned from its initial state of all zeros these are the values that were learned and updated during training and that's all there is to the Q learning implementation.

Delayed Q-learning vs. Double Q-learning vs. Q-Learning

Delayed Q-learning and double Q-learning are two extensions to Q-learning that are used throughout RL, so it's worth considering them in a simple form.

- **Delayed Q-learning** simply delays any estimate until there is a statistically significant sample of observations. Slowing update with an exponentially weighted moving average is a similar strategy. Instead of updating the **Q-value** after every action, Delayed Q-Learning waits until **a state-action pair has been visited a certain number of times** before updating its value.
- **Double Q-learning** includes two Q-tables, in essence two value estimates, to reduce bias. The final policy uses the average (or sum) of Q^A and Q^B.

$$Q_{t+1}^A = Q_t^A(s,a) + \alpha[r(s,a) + \gamma max_{a'} Q^B(s',a') - Q_t^A(s,a)]$$

$$Q_{t+1}^B = Q_t^B(s,a) + \alpha[r(s,a) + \gamma max_{a'} Q^A(s',a') - Q_t^B(s,a)]$$

In general, double Q-learning tends to be more stable than Q-learning. And delayed Q-learning is more robust against outliers, but can be problematic in environments with larger state/action spaces. I.e. you might have to wait for a long time to get the required number of samples for a particular state-action pair, which will delay further exploration.

SARSA vs. Q-Learning

SARSA is an on-policy algorithm used in reinforcement learning to train a Markov decision process model on a new policy. It's an algorithm where, in the current state (S), an action (A) is taken and the agent gets a reward (R), and ends up in the next state (S1), and takes action (A1) in S1, or in other words, the tuple S, A, R, S1, A1.
In the SARSA algorithm, the Q-value is updated taking into account the action, A1, performed in the state, S1. In Q-learning, the action with the highest Q-value in the next state, S1, is used to update the Q-table.

$$\text{Q-learning}: \; Q_{t+1} = Q_t(s,a) + \alpha[r(s,a) + \gamma max_{a'} Q(s',a') - Q_t(s,a)]$$

$$\text{Sarsa} \quad : \; Q_{t+1} = Q_t(s,a) + \alpha[r(s,a) + \gamma Q(s',a') - Q_t(s,a)]$$

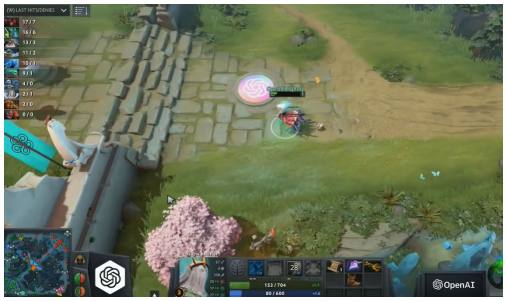
Both SARSA and Q-learning are useful for reinforcement learning, and each have their own uses based on agent needs. SARSA may be better as a non-deterministic algorithm and for choosing low-risk action paths in comparison to Q-learning.

By the frozen lake example we saw that Q learning algorithm did a pretty decent job in relatively small state spaces. But its **performance will drop off** considerably when we work in more **complex** and **sophisticated environments**.

Our environment was relatively simple with only 16 states and 4 actions giving us a total state action space of 64 Q values to update in the Q table.

A video game where a player has a large environment to roam around, **each state** in the environment would be represented by **a set of pixels** and the agent may be able to **take several actions** from each state.

In a large state space like this, it becomes computationally inefficient and perhaps infeasible due to the computational resources and time.

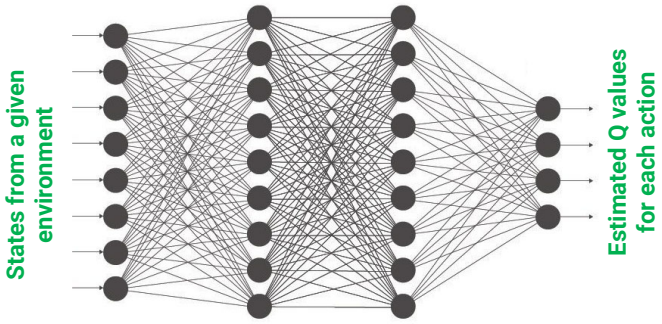


Rather than using **value iteration** to directly compute Q values and find the optimal Q function we instead use a function approximator to **estimate the optimal Q function**.

At approximating functions artificial **neural networks** do a pretty good job.
We make use of a deep neural network to estimate the Q values for each state-action pair in a given environment and in turn the network will approximate the optimal Q function.
The act of **combining Q learning with a deep neural network** is called **deep Q learning** and a deep neural network that approximates the Q function is called a **deep Q network, or DQN.**

Deep Q Learning (DQN)

The DQN algorithm, combining Q-Learning with Deep Neural Networks.



The **loss** from the network is calculated by comparing the **outputted Q values to the target Q values.** we use a function approximator to **estimate the optimal Q function**.

The objective is to minimize the loss to compute **the weights within the network updated via stochastic gradient descent and back propagation**

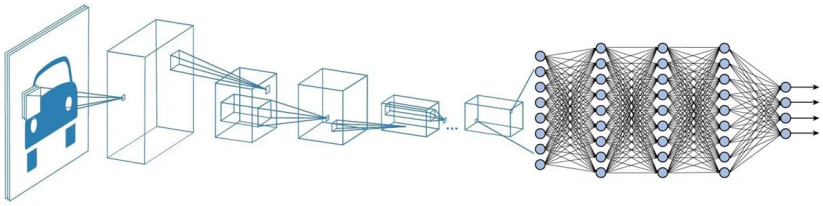
So we discussed earlier that the network would accept states from the environment as input. Thinking of the game frozen lake that we played last time we could easily represent the states of this environment using a simple coordinate system from the **grid** of the environment.

If we're in a more complex environment though like a **video game** for example then we'll use **images as our input**. Specifically we'll use frames that capture sates from the environment as the input to the network.

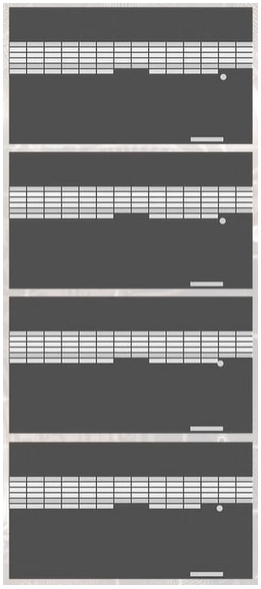
The **standard pre-processing** done on the frames usually involves converting the RGB data into grayscale data since the color in the image is probably usually not going to affect the state of the environment additionally will typically see some cropping and scaling as well to both cut out unimportant information from the frame and shrink the size of the image.

In real life examples of RL like Automated Robots, Self driving cars, gaming or **video games, which involve a** complex environment**, we may** use more than one **frame, or image to describe the** important aspects of the **states, so** the neural network can distinguish all important elements like the speed of the objects, their possible directions and so on.

In this case some, **standard pre-processing should be** done on the frames. These include converting the RGB data into grayscale data and making some cropping and scaling to shrink the size of the image.
Moreover, in order to process the image **we may use the Convolutional Neural network** and define some models like below which contains **some convolutional layers** followed **by some fully connected layers**.



Now actually rather than having a single frame represent a single input we usually use a stack of **a few consecutive frames** to represent a single input. Therefore, we'd grab say four consecutive frames from a video game. We then do all the pre-processing on each of these four frames including the greyscale conversion, the cropping, scaling. Then we'd take the pre-processed frames and stack them on top of each other in the order of which they occurred in the game. We do this because a single frame usually isn't going to be enough for our network or even for our human brains to fully understand the state of the environment. For example by just looking at this single frame from the Atari game, we can't tell if the ball is coming down to the paddle or going up to hit the block. We also don't have any indication about the **speed of the ball** or **which direction the paddle is moving in**. If we look at our four consecutive frames though then we have a much better idea about the current state of the environment because we now do indeed have information about all these things that we didn't know with just a single frame.



Experience replay technique

that is utilized during the training process of DQN

We store the **agent's experiences** at **each time step in a data set** called the **replay memory**.
At time t, the agent's experience is defined as the following tuple:

Experience replay

At time t, the agent's experience e_t is defined as this tuple:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

In practice the replay memory is set to some **finite size** limit **N**. This **replay memory** data set is what will **randomly be sampling** from to train the network to break the correlation between consecutive samples of experiences.
The **experience replay** is the act of **gaining experience** and sampling from the **replay memory** that stores these experiences.

DQN algorithm (before training the neural network)

1. Initialize replay memory capacity.
2. Initialize the policy network with random weights.
3. Clone the policy network and call it the target network
3. For each episode:
    - 3.1. Initialize the starting state.
    - 3.2. For each time step:
        - 3.2.1. Select an action (Via exploration of **random action** or exploitation of **greedy action**)
        - 3.2.2. Execute selected action in an emulator.
        - 3.2.3. Observe reward and next state.
        - 3.2.4. Store experience in replay memory.

DQN algorithm (training the neural network)

3.2.5. Sample random batch from replay memory.

3.2.6. Preprocess states from batch.

3.2.7. Pass batch of preprocessed states to policy network.

3.2.8. considering the input state data, we implement the forward propagation, then output an estimated Q value for each possible action from the given input state.

3.2.9. Calculate loss between output Q-values and the corresponding optimal Q value or Target Q-values.

Requires a pass to the target network for the next state

$$loss = q_*(s,a) - q(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]$$

Recall that $\acute{s}$ and $\acute{a}$ are the state and action that occur in the following time step.

3.2.10. Gradient descent updates weights in the policy network to minimize loss.

After x time steps, weights in the target network are updated to the weights in the policy network.

DQN algorithm (training the neural network)

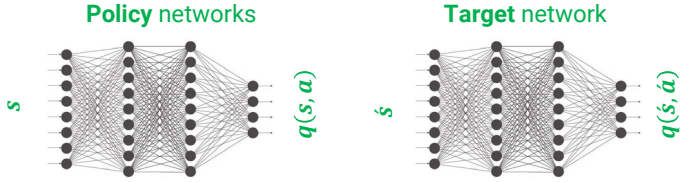
3.2.9. Calculate loss between output Q-values and the corresponding optimal Q value or Target Q-values.

Requires a pass to the target network for the next state

$$loss = q_*(s,a) - q(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]$$

Recall that $\acute{s}$ and $\acute{a}$ are the state and action that occur in the following time step. To compute $\max_{\acute{a}} q_*(\acute{s}, \acute{a})$, we pass **the next state, $\acute{s}$,** to the **target** network. By this method we do **two forward passes before we do any gradient updates**.

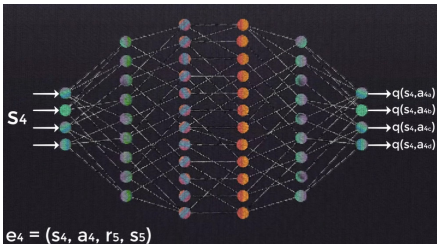
Note: we **update the weights** in the target network based on the policy networks **every certain amount of time steps**. This **certain amount of time steps** can be looked at as another **hyper parameter** that we'll have to test and **tune**.

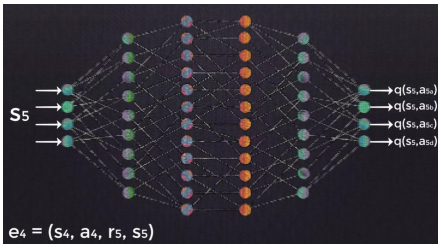


Note that for our specific example, we're working with experience tuple $e_4 = (s_4, a_4, r_5, s_5)$. So our initial state that we passed to the network was s_4. the general $\acute{s}$ and $\acute{a}$ notation then would take on s_5 and a_5 as the next state and action.

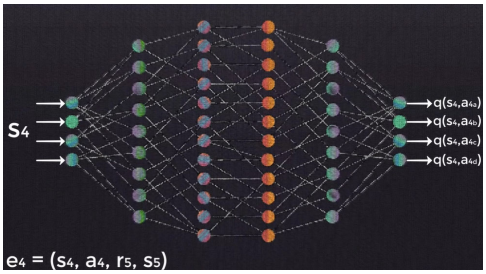
Once we find the value of this max term, we can then plug it in to calculate $q_*(s,a) = E\left[R_{t+1} + \gamma \max_{\acute{a}} q_*(\acute{s},\acute{a})\right]$ **for the original state input** passed through the policy network. As stated previously, this term enables us to compute the loss between the Q value given by the policy network for the state action pair from our original experience tuple and target optimal key value for this same state action pair.

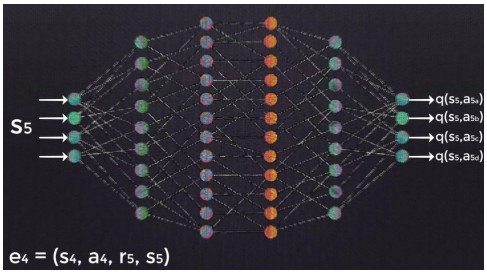
.





So to quickly touch base note that we **first forward** pass the state from our original experience tuple (say s_4) to the network and **got the Q value for the action** from our experience tuple as output. we then **pass the next state,** s_5, contained in our experience tuple to the network to **find the max Q value among the next actions** that can be taken from that state. This **second** step was just done to aid us in **calculating the loss** for our original state action pair.
This may seem a bit **odd doing two forward passes before we even do any type of gradient update** but let it sink in for a minute and see if the idea clicks.





Our Objective is to get the Q-value to approximate the target Q value. Remember we don't know ahead of time what our target Q value is. So we attempt to approximate it with the network. As it has mentioned, the **second pass uses the same exact weights in the network** as the first pass. Therefore, the targets are calculated using the same weight. **So as our Q values will be updated with each iteration to move closer to the target key values but the target Q values will also be moving in the same direction.** This makes the optimization appear to be **chasing its own tail** which introduces instability. As our Q values move closer and closer to their targets the targets continue to move further and further because we're using the same network with the same weights to calculate both of these values.

$$q(s,a) \Rightarrow q_*(s,a)$$

How DQN Works (Quick Recap)

DQN is a **value-based** method that **approximates the Q-function using a neural network**. It learns the best action to take in a given state by estimating Q-values and selecting the action with the highest Q-value (i.e., using an ε-greedy policy).

However, DQN has some limitations:

1. **Struggles with Continuous Action Spaces** – Since DQN works by selecting the action with the highest Q-value, it's inefficient for environments with infinite or large action spaces (e.g., robotics control).
2. **Limited Exploration** – ε-greedy exploration may not always be optimal.
3. **Indirect Policy Learning** – The policy is derived from the Q-values rather than being optimized directly.

What is the Policy Gradient Method?

Unlike DQN, which is **value-based**, the **policy gradient method is policy-based**. Instead of learning a Q-function, **it directly learns a policy $\pi_\theta(a|s)$**, which is a **probability distribution over actions** given a state.

Key Idea:

Instead of selecting the action with the highest estimated *Q*-value, we directly **parameterize** the policy using a neural network and **adjust the parameters θ** using **gradient ascent** to maximize expected rewards.

Comparison: DQN vs. Policy Gradient

Feature	DQN	Policy Gradient
Type	Value-based	Policy-based
Action Space	Discrete	Discrete & Continuous
Exploration	ε-greedy	Stochastic policy
Stability	More stable	Less stable, high variance
Convergence	Faster	Slower but sometimes more optimal

Policy Gradient Variants

There are improved versions of the basic policy gradient method:

1.REINFORCE (Monte Carlo Policy Gradient) – Uses the full trajectory return to estimate the gradient.

2.Actor-Critic – Combines value-based and policy-based methods by using a critic network to estimate the value function and an actor network to learn the policy.

3.PPO (Proximal Policy Optimization) – A more stable and efficient variant of policy gradient methods.

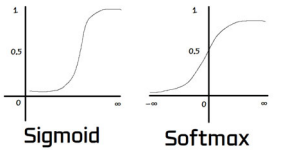
Detailed Policy Gradient Algorithm Explanation

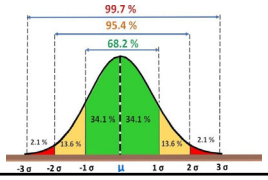
Step 1: Initialize the Policy Network

- We define a policy $\pi_\theta(a|s)$ using a **neural network** with parameters θ.
- The network takes a **state** s as input and outputs a **probability distribution** over actions.

💡 **Example**:

- In a **discrete action space**, the output is a **softmax distribution** over possible actions.
- In a **continuous action space**, the output could be the **mean and variance** of a Gaussian distribution.





Detailed Policy Gradient Algorithm Explanation

Step 2: Collect Trajectories

- Start from the **initial state** s_0.
- **For each time step** t:
  1. Use the policy π_θ to **sample an action** a_t (not the max action like DQN, but a sampled action from the probability distribution).
  2. Execute action a_t in the environment.
  3. Observe the **new state** s_{t+1} and **reward** r_t.
  4. Store (s_t, a_t, r_t) in memory.
- Repeat until the **episode ends** (or max steps reached).

For example, in a car racing video game, in each episode, we start the game (Initial state) until it is over (Final State)

Detailed Policy Gradient Algorithm Explanation

Step 3: Compute the Returns

Once we have collected an episode, we calculate the **total reward** for each time step.

using **Monte Carlo estimation**. $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$

where:

- γ (discount factor) controls how much future rewards matter.
- Higher γ means long-term rewards are prioritized.

💡 **Example**:
If we collected rewards $r_0 = 1$, $r_1 = 2$, $r_2 = 3$ with $\gamma = 0.9$:

$$R_0 = 1 + 0.9(2 + 0.9(3)) = 5.43$$

$$R_1 = 2 + 0.9(3) = 4.7$$

$$R_2 = 3$$

How we can improve Step 3.

1. Why Monte Carlo Estimates Are Inaccurate

The key problem is that Monte Carlo **only uses actual sampled trajectories**, meaning:

1. **High Variance**: If the agent takes a lucky/unlucky path, the reward estimate fluctuates wildly.
2. **Delayed Updates**: The agent must complete a full episode before making updates.
3. **No Bootstrapping**: It doesn't reuse past experience efficiently.

Because of this, updates to the policy are **slow and unstable**.

2. How Can We Improve It?

Since Monte Carlo is **not exact**, we can **reduce variance** and improve accuracy by introducing **bootstrapping**. There are two major ways to do this:

Method 1: Use a Baseline (Variance Reduction)

Instead of using R_t directly, we subtract a **baseline function** $V(s)$:

$$G_t = R_t - V(s_t)$$

- $V(s)$ is an estimate of the **expected return** from state s.
- This reduces variance because it removes unnecessary fluctuations.

🔹 **Actor-Critic methods** use this idea, where:

- The **Actor** updates the policy using policy gradient.
- The **Critic** learns $V(s)$ (a value function) to reduce variance.

Method 2: Use Temporal Difference (TD) Learning

Instead of waiting for the episode to finish, we estimate the return **on-the-fly** using **TD bootstrapping**:

$$R_t \approx r_t + \gamma V(s_{t+1})$$

- This means we **don't need to wait for the full episode**.
- TD-based estimates are **lower variance**, though they introduce some bias.

🔹 **Advantage Actor-Critic (A2C, PPO, etc.)** methods use this approach.

Detailed Policy Gradient Algorithm Explanation

Step 4: Compute the Policy Gradient Estimate

We define the **objective function** as the expected cumulative reward:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

where:

- $\tau = (s_0, a_0, s_1, a_1, \dots)$ is a trajectory (a sequence of states and actions),
- $R(\tau)$ is the total reward of a trajectory,
- $\pi_\theta(a|s)$ is the policy parameterized by θ.

To optimize $J(\theta)$, we compute its **gradient** using the **policy gradient theorem**:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} R_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

For each time step:

1. Compute $\log \pi_\theta(a_t|s_t)$ using the policy network.
2. Multiply it by the return R_t.
3. Compute the gradient $\nabla_\theta \log \pi_\theta(a_t|s_t)$.

Case 1: Discrete Action Spaces (Softmax Policy)

In discrete action settings, we often use a **Softmax policy**:

$$\pi_\theta(a \mid s) = \frac{\exp(f_\theta(s,a))}{\sum_{a'} \exp(f_\theta(s,a'))}$$

where $f_\theta(s,a)$ is a function of state-action pair, often modeled by a neural network.

The log of this function is:

$$\log \pi_\theta(a \mid s) = f_\theta(s,a) - \log \sum_{a'} \exp(f_\theta(s,a'))$$

Taking the gradient:

$$\nabla_\theta \log \pi_\theta(a \mid s) = \nabla_\theta f_\theta(s,a) - \sum_{a'} \pi_\theta(a' \mid s) \nabla_\theta f_\theta(s,a')$$

Case 2: Continuous Action Spaces (Gaussian Policy)

For continuous actions, we usually assume a **Gaussian (Normal) policy**:

$$\pi_\theta(a \mid s) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(a-\mu_\theta(s))^2}{2\sigma^2}\right)$$

where:

- $\mu_\theta(s)$ is the mean (output by the Actor network).
- σ^2 is the variance (sometimes learned, sometimes fixed).

Taking the log:

$$\log \pi_\theta(a \mid s) = -\frac{(a-\mu_\theta(s))^2}{2\sigma^2} - \log \sqrt{2\pi\sigma^2}$$

Taking the gradient:

$$\nabla_\theta \log \pi_\theta(a \mid s) = \frac{(a-\mu_\theta(s))}{\sigma^2} \nabla_\theta \mu_\theta(s)$$

Derivation of the Policy Gradient Formula

We want to maximize the **expected return** (total reward), defined as:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

where $R(\tau)$ is the total reward of a trajectory τ generated using policy $\pi_\theta(a|s)$.

Step 1: Express the Expectation as a Sum Over Trajectories

The expectation over trajectories means we are summing over all possible trajectories τ weighted by their probability:

$$J(\theta) = \sum_{\tau} P_\theta(\tau) R(\tau)$$

where:

- $P_\theta(\tau)$ is the probability of a trajectory occurring under policy π_θ.
- $R(\tau)$ is the total return of that trajectory.

The probability of a trajectory is the product of probabilities of states and actions:

$$P_\theta(\tau) = P(s_0) \prod_{t=0}^{T} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t)$$

where:

- $P(s_0)$ is the probability of the initial state.
- $P(s_{t+1}|s_t, a_t)$ is the transition probability.
- $\pi_\theta(a_t|s_t)$ is the policy that we control.

Step 2: Compute the Gradient

We take the gradient of $J(\theta)$:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{\tau} P_\theta(\tau) R(\tau)$$

Since $R(\tau)$ is independent of θ, we move the gradient inside:

$$\nabla_\theta J(\theta) = \sum_{\tau} R(\tau) \nabla_\theta P_\theta(\tau)$$

Using the **log trick** (also known as the likelihood ratio trick):

$$\nabla_\theta P_\theta(\tau) = P_\theta(\tau) \nabla_\theta \log P_\theta(\tau)$$

we get:

$$\nabla_\theta J(\theta) = \sum_{\tau} P_\theta(\tau) R(\tau) \nabla_\theta \log P_\theta(\tau)$$

Since:

$$\log P_\theta(\tau) = \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)$$

Since the environment transition probabilities $P(s_{t+1} \mid s_t, a_t)$ are **independent of** θ, we only care about the policy:

$$P_\theta(\tau) \propto \prod_{t=0}^{T} \pi_\theta(a_t \mid s_t)$$

we obtain:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} R(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

Detailed Policy Gradient Algorithm Explanation

Step 5: Update Policy Parameters

Finally, we **update** the policy parameters θ using **gradient ascent**:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} R_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

where α is the learning rate.

This update ensures:

- Actions that led to **higher returns** become **more probable**.
- Actions that led to **lower returns** become **less probable**.

Note: Backpropagation computes **$\nabla\theta \log \pi\theta(a|s)$** in policy gradients.

Detailed Policy Gradient Algorithm Explanation

Step 6: Repeat Until Convergence

- Continue collecting episodes, computing gradients, and updating the policy.
- Over time, the policy improves, leading to higher expected rewards.

Policy Gradient Algorithm Explanation in a nutshell

How Policy Gradient Works (Step-by-Step)

1. **Initialize** a neural network with parameters θ to represent $\pi_\theta(a|s)$.
2. **Collect trajectories** by interacting with the environment using π_θ.
3. **Compute the rewards** associated with each action in the trajectories.
4. **Compute the policy gradient** using the policy gradient theorem.
5. **Update** the policy parameters θ using **gradient ascent**:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

6. **Repeat** until convergence.

The Objective function is the Max of expected value of cumulative rewards (Return)

Key differences between DQN (Deep Q-Network) and Policy Gradient

1. Core Difference: Value-Based vs. Policy-Based

The biggest difference between DQN and Policy Gradient is how they **represent and update** the agent's behavior:

Method	How It Works
DQN (Deep Q-Network)	**Value-Based**: Learns the Q-value function $Q(s, a)$ and picks actions greedily based on the highest Q-value.
Policy Gradient	**Policy-Based**: Learns a policy (\pi_\theta(a

Key differences between DQN (Deep Q-Network) and Policy Gradient

2. How Actions Are Selected

Feature	DQN	Policy Gradient
Action Selection	Selects the action with the highest Q-value $\max_a Q(s,a)$ (or uses ε-greedy for exploration).	Samples an action a from the policy distribution (\pi_\theta(a

💡 **Example:**

- **DQN**: If $Q(s,a_1) = 5$ and $Q(s,a_2) = 3$, it picks a_1 (greedy).
- **Policy Gradient**: If $\pi_\theta(a_1|s) = 0.7$ and $\pi_\theta(a_2|s) = 0.3$, it samples a_1 70% of the time and a_2 30% of the time.

🚀 **Key takeaway**:

- **DQN is deterministic** (picks the best action).
- **Policy Gradient is stochastic** (samples from probabilities).

Key differences between DQN (Deep Q-Network) and Policy Gradient

3. Action Space Compatibility

Feature	DQN	Policy Gradient
Discrete Actions	✅ Works well	✅ Works well
Continuous Actions	❌ Not efficient	✅ Works well

💡 **Why?**

DQN **selects** actions based on a Q-value table, which works well for a **finite number of discrete actions**. But in **continuous action spaces** (e.g., robotics, self-driving cars), DQN requires **function approximation** (e.g., DDPG), making it inefficient.

Policy Gradient **directly models the action probability distribution**, making it **naturally suited for continuous actions**.

🚀 **Key takeaway**:

- **DQN is better for discrete action spaces.**
- **Policy Gradient is better for continuous action spaces.**

Key differences between DQN (Deep Q-Network) and Policy Gradient

4. Objective Function & Learning Process

DQN: Bellman Equation (Q-Learning)

DQN **estimates the Q-value function** using the Bellman equation:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

It **minimizes the Temporal Difference (TD) error**:

$$\mathcal{L}(\theta) = \left(r + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)\right)^2$$

✅ Uses **experience replay** to stabilize learning.
✅ Uses **target networks** to reduce instability.

Policy Gradient: Maximizing Expected Reward

Policy Gradient **directly maximizes the expected return**:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

It updates the policy using the gradient:

$$\nabla_\theta J(\theta) = \sum_t R_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

❌ **No replay buffer** (learns from on-policy data).
❌ **Higher variance** but **better exploration**.

🚀 **Key takeaway**:

- **DQN learns a value function** and selects the best action.
- **Policy Gradient directly optimizes the policy** without needing a Q-function.

key differences between DQN (Deep Q-Network) and Policy Gradient

5. Exploration vs. Exploitation

Feature	DQN	Policy Gradient
Exploration	Uses **ε-greedy** (randomly explores with probability ϵ).	Naturally **stochastic** (explores by sampling from a learned probability distribution).

💡 **Example**:

- **DQN**: With $\epsilon = 0.1$, 90% of the time it picks the best action, 10% of the time it picks a random action.
- **Policy Gradient**: If the policy learns $\pi_\theta(a_1|s) = 0.8$ and $\pi_\theta(a_2|s) = 0.2$, it naturally explores by sometimes choosing a_2.

🚀 **Key takeaway**:

- **DQN needs manual exploration tuning** (ϵ-greedy).
- **Policy Gradient naturally explores via sampling.**

key differences between DQN (Deep Q-Network) and Policy Gradient

6. Stability & Sample Efficiency

Feature	DQN	Policy Gradient
Stability	More stable (uses experience replay and target networks).	Less stable (higher variance in updates).
Sample Efficiency	More efficient (reuses old experiences via replay buffer).	Less efficient (on-policy learning, needs many samples).

💡 **Why?**

- **DQN stores and reuses past experiences**, making it **sample-efficient**.
- **Policy Gradient only learns from fresh data** (on-policy), meaning **it needs more samples to converge**.

🚀 **Key takeaway**:

- **DQN is more sample-efficient** (thanks to replay buffers).
- **Policy Gradient is more data-hungry** but allows for direct policy optimization.

key differences between DQN (Deep Q-Network) and Policy Gradient

7. Variants & Extensions

Method	Variants
DQN	Double DQN, Dueling DQN, Rainbow DQN
Policy Gradient	REINFORCE, Actor-Critic (A2C, A3C, PPO, TRPO)

Policy Gradient is often combined with **value-based methods** to create **Actor-Critic algorithms**, which reduce variance and improve stability.

key differences between DQN (Deep Q-Network) and Policy Gradient

Final Comparison Table

Feature	DQN	Policy Gradient
Type	Value-Based	Policy-Based
Output	Q-values $Q(s, a)$	Probability distribution (\pi_\theta(a
Action Selection	Greedy / ε-greedy	Sampling from policy
Best for	Discrete action spaces	Continuous action spaces
Exploration	ε-greedy	Stochastic policy
Sample Efficiency	✅ High (Replay buffer)	❌ Low (On-policy)
Stability	✅ Stable (uses experience replay)	❌ High variance
Updates	Uses Bellman equation	Uses Policy Gradient theorem
Variants	Double DQN, Dueling DQN	REINFORCE, Actor-Critic (PPO, A3C)

🚀 **When to use DQN vs. Policy Gradient?**

- Use **DQN** for **discrete action spaces** where stability and efficiency matter.
- Use **Policy Gradient** for **continuous action spaces** or when you need a stochastic policy (e.g., robotics, self-driving cars).

DQN in a nutshell

DQN Algorithm (Deep Q-Network)

DQN is a **value-based** method that learns a **Q-value function** and selects actions greedily.

Step 1: Initialize the Q-Network

- Initialize a **neural network** $Q(s, a; \theta)$ with parameters θ.
- Initialize a **target network** $Q'(s, a; \theta^-)$ (copy of the main network but updated less frequently).
- Initialize an **experience replay buffer** to store transitions.

Step 2: Start Interaction with the Environment

- Start in **state** s_0.

Step 3: Choose an Action Using ϵ-Greedy Policy

- With probability ϵ, select a **random action** (exploration).
- Otherwise, select the **best action** based on the current Q-network:

$$a_t = \arg\max_a Q(s_t, a; \theta)$$

Step 4: Execute Action and Store Experience

- Execute **action** a_t, observe **reward** r_t and **next state** s_{t+1}.
- Store the experience (s_t, a_t, r_t, s_{t+1}) in the **replay buffer**.

Step 5: Sample a Mini-Batch from Replay Buffer

- Sample a random batch of past experiences (s, a, r, s').

Step 6: Compute Target Q-Value Using Bellman Equation

- Compute the **target Q-value** for each experience:

$$y = r + \gamma \max_{a'} Q'(s', a'; \theta^-)$$

where $Q'(s', a')$ is from the **target network**.

Step 7: Compute Loss and Update Q-Network

- Compute the **loss** using Mean Squared Error (MSE):

$$\mathcal{L}(\theta) = (y - Q(s, a; \theta))^2$$

- Update the **Q-network parameters** θ using **gradient descent**.

Step 8: Update the Target Network (Periodically)

- Every N steps, update the **target network**:

$$\theta^- \leftarrow \theta$$

Step 9: Repeat Until Convergence

- Keep interacting with the environment, updating Q-values, and improving the policy.

Policy Gradient in a nutshell

Policy Gradient Algorithm (REINFORCE)

Policy Gradient is a **policy-based** method that directly learns a **stochastic policy** $\pi_\theta(a|s)$.

Step 1: Initialize the Policy Network

- Initialize a **neural network** $\pi_\theta(a|s)$ with parameters θ.
- The output is a **probability distribution** over actions.

Step 2: Start Interaction with the Environment

- Start in **state** s_0.

Step 3: Sample an Action from the Policy

- Instead of selecting the action with the highest value (like in DQN), we **sample** an action from the policy distribution:

$$a_t \sim \pi_\theta(a|s_t)$$

This allows **stochastic exploration**.

Step 4: Execute Action and Observe Reward

- Execute **action** a_t, observe **reward** r_t and **next state** s_{t+1}.
- Store (s_t, a_t, r_t) in memory.

Step 5: Compute the Return for Each Time Step

- Compute the total **discounted return** from each time step onward:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

Step 6: Compute the Policy Gradient

- Compute the gradient of the policy using:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} R_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

This tells us how to adjust θ to **increase the probability of good actions**.

Step 7: Update Policy Parameters

- Update the **policy network** parameters θ using gradient ascent:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} R_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Unlike DQN, **we don't use target networks or experience replay**.

Step 8: Repeat Until Convergence

- Continue collecting episodes and updating the policy.

Side-by-Side Comparison of Algorithm Steps

Step	DQN (Value-Based)	Policy Gradient (Policy-Based)
1. Initialize	Q-network $Q(s, a; \theta)$, target network, replay buffer	Policy network (\pi_\theta(a
2. Select Action	ϵ-greedy (greedy action with probability $1 - \epsilon$, random action with probability ϵ)	Sample from (\pi_\theta(a
3. Execute Action	Observe r_t, s_{t+1}, store in buffer	Observe r_t, s_{t+1}, store in memory
4. Learn from Experience	Sample mini-batch from replay buffer	Use full episode (on-policy learning)
5. Compute Update	Compute target Q-value using Bellman equation	Compute return R_t and policy gradient
6. Update Model	Update Q-network using gradient descent	Update policy using gradient ascent
7. Additional Techniques	Uses target network and experience replay	No replay buffer, but can use baselines (e.g., Advantage Actor-Critic)
8. Repeat	Continue updating Q-values	Continue updating policy probabilities

Step 5: Compute the Return R_t in Policy Gradient

1. Why Do We Need Monte Carlo?

- Unlike **DQN**, which estimates **Q-values using the Bellman equation**, Policy Gradient methods rely on actual **sampled rewards**.
- This means we do **not** need to know future actions in advance—we just run the episode, observe the rewards, and then compute R_t using **Monte Carlo estimation**.
- Since we sum over actual episode rewards, **we do not bootstrap** (like in Q-learning).

2. Monte Carlo Estimation for Policy Gradient

- In Monte Carlo methods, we let the **agent play a full episode** before making any updates.
- Once the episode ends, we compute the returns R_t based on the rewards received.

How It Works:

1. **Run an episode**: Start from an initial state and let the policy $\pi(a|s)$ generate actions until termination.
2. **Collect rewards**: Store rewards $r_0, r_1, r_2, ..., r_T$.
3. **Compute returns R_t after the episode ends** using:
$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t} r_T$$
4. **Use R_t to update policy parameters**.

Since we wait until the **end of the episode** to compute R_t, this method is called **Monte Carlo Policy Gradient** (**REINFORCE**).

1. Why Monte Carlo Estimates Are Inaccurate

The key problem is that Monte Carlo **only uses actual sampled trajectories**, meaning:

1. **High Variance**: If the agent takes a lucky/unlucky path, the reward estimate fluctuates wildly.
2. **Delayed Updates**: The agent must complete a full episode before making updates.
3. **No Bootstrapping**: It doesn't reuse past experience efficiently.

Because of this, updates to the policy are **slow and unstable**.

2. How Can We Improve It?

Since Monte Carlo is **not exact**, we can **reduce variance** and improve accuracy by introducing **bootstrapping**. There are two major ways to do this:

Method 1: Use a Baseline (Variance Reduction)

Instead of using R_t directly, we subtract a **baseline function** $V(s)$:

$$G_t = R_t - V(s_t)$$

- $V(s)$ is an estimate of the **expected return** from state s.
- This reduces variance because it removes unnecessary fluctuations.

🔹 **Actor-Critic methods** use this idea, where:

- The **Actor** updates the policy using policy gradient.
- The **Critic** learns $V(s)$ (a value function) to reduce variance.

Method 2: Use Temporal Difference (TD) Learning

Instead of waiting for the episode to finish, we estimate the return **on-the-fly** using **TD bootstrapping**:

$$R_t \approx r_t + \gamma V(s_{t+1})$$

- This means we **don't need to wait for the full episode**.
- TD-based estimates are **lower variance**, though they introduce some bias.

🔹 **Advantage Actor-Critic (A2C, PPO, etc.)** methods use this approach.

3. Comparing Monte Carlo and TD

Method	Accuracy	Variance	Sample Efficiency	When to Use?
Monte Carlo (MC)	High bias, inexact	High variance	Inefficient	When episodes are short & rewards are sparse
Temporal Difference (TD)	Biased but more stable	Lower variance	More efficient	Works for continuous tasks & large environments
Actor-Critic (TD+MC)	Balanced	Reduced variance	More efficient	When variance is a problem

✅ **Monte Carlo is not exact**, but by using **TD bootstrapping** + **a baseline**, we can improve accuracy while keeping policy gradient methods stable.

Actor-Critic method

Why Policy Gradient Needs Improvement?

The basic **Policy Gradient (REINFORCE)** algorithm uses **Monte Carlo estimation** to compute the **return R_t**. However, this has several issues:
1.High Variance: Returns fluctuate because they depend entirely on sampled episodes.
2.Delayed Updates: We must wait for the **entire episode** to finish before updating the policy.
3.Inefficient Learning: It does not reuse past experiences and does not use bootstrapping.
To fix these problems, we introduce the **Actor-Critic** method.

Actor-Critic Algorithm

2. What is Actor-Critic?

Actor-Critic is a hybrid method that combines **Policy Gradient (Actor)** with **Value Function Estimation (Critic)**.
•**Actor**: The policy $\pi_\theta(a|s)$ that selects actions.
•**Critic**: A learned value function $V_w(s)$ that **estimates how good a state is**.
How Does This Help?
•The **Critic (Value Function $V_w(s)$) replaces the Monte Carlo estimate R_t** with a more stable, **bootstrapped estimate**.
•This reduces **variance** while still allowing policy updates

Actor-Critic Algorithm

Actor-Critic runs continuously, unlike Monte Carlo REINFORCE, which waits for the episode to finish.
Let's break down each step.

Step 1: Initialize

- Initialize **policy network (Actor)** with parameters θ.
- Initialize **value function (Critic)** with parameters w.
- Choose a discount factor γ.

Step 2: Run the Policy to Collect Experience

- Start in **state** s_t.
- Select an **action** a_t **using policy** $\pi_\theta(a_t|s_t)$.
- Execute a_t, observe **reward** r_t and **new state** s_{t+1}.

Actor-Critic Algorithm

Step 3: Compute Temporal Difference (TD) Error

Instead of waiting for the full return R_t, we estimate it using **TD bootstrapping**:

$$\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t)$$

where:

- $V_w(s_t)$ is the Critic's estimate of **how good state s_t is**.
- δ_t (TD error) tells us **if the value estimate was too high or too low**.

🔷 **Key Improvement**:

- Unlike Monte Carlo, which waits for R_t, we update after every step.
- **Bootstrapping with $V_w(s)$** makes learning faster.

Bootstrapping means we use **an estimate to update another estimate**, instead of waiting for the final reward.

🔷 **Monte Carlo (No Bootstrapping)**:

- Uses **full episode return** R_t to update values.
- High variance but no bias.

🔷 **Bootstrapping (TD Learning)**:

- Uses **partial return estimate** $r_t + \gamma V(s_{t+1})$.
- Lower variance but introduces some bias.

🔷 **Why is Bootstrapping Useful?**

- We **don't have to wait until the end of the episode** to update the value function.
- This allows **faster learning** and works well in continuous tasks.

Step 4: Update Critic (Value Function)

We update the Critic by minimizing the **Mean Squared Error (MSE)** between the estimated and actual return:

Gradient Descent: $$w \leftarrow w + \alpha_c \delta_t \nabla_w V_w(s_t)$$

- α_c is the learning rate for the Critic.
- The Critic learns to **predict better state values** over time.

The Critic is updated to **reduce the TD error** by minimizing the squared difference:

$$L(w) = (r_t + \gamma V_w(s_{t+1}) - V_w(s_t))^2$$

🔷 **Steps to Update the Critic:**

1. **Compute TD error**:
$$\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t)$$
2. **Compute gradient of loss**:
$$\nabla_w L(w) = \delta_t \nabla_w V_w(s_t)$$
3. **Update the Critic's parameters**:
$$w \leftarrow w + \alpha_c \delta_t \nabla_w V_w(s_t)$$

🔷 **Key Idea:**

- If the **TD error is large**, the update is **strong**.
- If the **TD error is small**, the Critic's estimate is already good, so updates are **small**.

The Critic is **similar to DQN**, but with some key differences.

🔷 **Similarities to DQN:**

- The Critic learns a **value function** to estimate how good a state is.
- It updates using **Temporal Difference (TD) learning**, just like DQN.
- It reduces variance by using bootstrapping instead of full Monte Carlo returns.

🔷 **Differences from DQN:**

Feature	DQN (Value-Based)	Critic (Actor-Critic)
Outputs	Estimates **Q-value** $Q(s,a)$	Estimates **Value** $V(s)$ or $Q(s,a)$
Action Selection	Uses $\max Q(s,a)$ for action selection	Does not select actions (Actor does that)
Policy Type	Off-policy (uses experience replay)	On-policy (updates with current policy)
Learning	Uses Bellman equation	Uses TD bootstrapping

🔷 **Key Difference**:

- **DQN learns $Q(s,a)$ (action-value function)**, which helps **select actions** directly.
- **Critic learns $V(s)$ (state-value function) or $Q(s,a)$**, but **does not select actions**—it only helps train the Actor.

The Critic does much more than just compute R_t.

🔹 **Monte Carlo (REINFORCE):**

- Computes R_t after the episode ends and uses it to update the policy.
- High variance, delayed updates.

🔹 **Critic in Actor-Critic:**

- **Continuously estimates** $V(s)$ and provides **instant feedback** to the Actor.
- Uses **TD learning (bootstrapping)** for updates.
- Reduces variance and allows faster learning.

🔹 **Key Difference:**

- In **Monte Carlo**, R_t is the full reward sum from that episode.
- In **Actor-Critic**, the Critic **estimates** $V(s)$ and gives updates **at every step**.

✅ **The Critic is not just a replacement for R_t, it is a learned function that makes training more efficient!**

The Critic in **Deep Actor-Critic methods** typically uses a **Neural Network** to approximate $V(s)$ or $Q(s, a)$.

How Does It Work?

- The Critic takes **state** s_t (or **state-action pair** (s_t, a_t) in some cases) as input.
- It **outputs an estimate** of $V(s_t)$ or $Q(s_t, a_t)$.
- The Critic network is trained using **gradient descent** to minimize the TD error.

Critic's Neural Network Training

$$L(w) = (r_t + \gamma V_w(s_{t+1}) - V_w(s_t))^2$$

- w are the weights of the **Critic's neural network**.
- It learns to predict **better estimates of state values** over time.

Actor-Critic Algorithm

Step 5: Update Actor (Policy)

We update the Actor using the policy gradient formula:

$$\theta \leftarrow \theta + \alpha_a \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

🔹 **Key Improvement**:

- Instead of using noisy Monte Carlo R_t, we use **TD error** δ_t, which is more stable.
- **TD error tells the Actor whether the action was better or worse than expected.**

TD error (δ_t) measures **how surprised the Critic is** about the reward received.

$$\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t)$$

🔹 **Intuition:**

- If δ_t is **high**, the Critic was **wrong about its estimate** of $V(s_t)$.
- If δ_t is **low**, the Critic's estimate was **accurate**.

🔹 **What Happens If TD Error is High?**

- The Critic learns **faster** because its value function was incorrect.
- The Actor gets a **stronger policy update** since the policy needs to change.
- If TD error is **too high**, training may become **unstable**.

🔹 **How to Handle High TD Error?**

- Use a **baseline** (Advantage function) to stabilize learning:

$$A_t = \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

- Clip gradient updates (used in PPO).
- Use **TD(λ) (eligibility traces)** to balance bias and variance.

How Does the Critic Give Feedback to the Actor?

The Critic's role is to **guide the Actor** by providing an estimate of whether an action was **better or worse than expected**.

How It Works (Step-by-Step)

1. The **Critic evaluates the state** s_t and estimates $V(s_t)$.
2. The **Actor selects an action** a_t **based on policy** $\pi(a_t|s_t)$.
3. The environment **returns a reward** r_t **and next state** s_{t+1}.
4. The **Critic computes TD error** δ_t (or Advantage function A_t).
5. The **Actor updates its policy using** δ_t:

$$\theta \leftarrow \theta + \alpha_a A_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Key Takeaway

- The Critic tells the Actor **how good an action was**.
- The Actor **adjusts its policy** to take better actions in the future.

✅ **The Critic acts like a "teacher" that gives feedback to the Actor, helping it learn the best actions over time.**

Actor-Critic Algorithm

Step 6: Repeat Until Convergence

- Keep running the policy.
- Keep improving the Critic's value function.
- The Actor learns **better actions** based on the Critic's feedback.

Why Actor-Critic is Better Than REINFORCE?

Aspect	REINFORCE (Monte Carlo PG)	Actor-Critic
Reward Estimation	Uses full episode return R_t (high variance)	Uses TD bootstrapped estimate (lower variance)
Update Frequency	Only updates after full episode	Updates every time step
Efficiency	Slow learning, inefficient	Faster learning, more stable
Bias-Variance Tradeoff	High variance, no bias	Lower variance, some bias

✅ **Actor-Critic learns faster and with lower variance compared to Monte Carlo Policy Gradient!**

Deep Deterministic Policy Gradient (DDPG)

DDPG is a **variant of Actor-Critic that learns $Q(s, a)$ instead of $V(s)$.**

Key Features of DDPG

- **Off-Policy**: Uses a **replay buffer** like DQN.
- **Deterministic Policy**: Instead of sampling $\pi(a|s)$, it directly **outputs the best action**.
- **Uses Two Networks**:
  1. **Actor** $\pi(s)$: Outputs the best action.
  2. **Critic** $Q(s, a)$: Estimates Q-values.

Algorithm Steps

1. **Initialize Actor-Critic networks** $Q_w(s, a)$ and $\pi_\theta(s)$.
2. **Store experiences in a replay buffer**.
3. **Train Critic** using TD target:
$$L(w) = (r_t + \gamma Q_w(s_{t+1}, \pi(s_{t+1})) - Q_w(s_t, a_t))^2$$
4. **Train Actor** using policy gradient:
$$\nabla_\theta J \approx \mathbb{E}\left[\nabla_a Q(s, a)|_{a=\pi(s)} \nabla_\theta \pi_\theta(s)\right]$$
5. **Use target networks to stabilize training**.

✅ **DDPG is powerful for continuous control problems, like robotics!**

PPO (Proximal Policy Optimization) vs. Actor-Critic

PPO improves Actor-Critic by **making policy updates safer** and more stable.

Problems with Standard Actor-Critic

- If we update the policy too aggressively, training becomes **unstable**.
- If we update too conservatively, learning is **slow**.

How PPO Fixes This?

1. **Clipped Policy Updates**
   - PPO **limits how much the policy changes** at each update.
   - Instead of using a plain policy gradient, it uses:
   $$L(\theta) = \min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)$$
   where $r_t(\theta)$ is the probability ratio between the new and old policy.
2. **Advantages Over Standard Actor-Critic**
   - More **stable** than vanilla policy gradients.
   - **Faster** than Trust Region Policy Optimization (TRPO).
   - Works well for **continuous and discrete action spaces**.

Summary of Key Differences

Algorithm	Key Idea	Main Benefit
Actor-Critic	Uses **Critic to evaluate actions**	Reduces variance
DDPG	Uses **Critic to learn** $Q(s, a)$	Works for **continuous action spaces**
PPO	Uses **clipped policy updates**	More **stable** and **efficient**

Paper Title: Reinforcement Learning for the Traveling Salesman Problem: Performance Comparison of Three Algorithms.
Authors: Jiaying Wang, Chenglong Xiao, Shanshan Wang, and Yaqi Ruan
Journal: The Journal of Engineering
Date: 2023

1. **Problem Modeling**:
   - **State**: The current city visited by the agent (salesman) and the set of already visited cities.
   - **Action**: The next city to visit, chosen from the unvisited cities.
   - **Reward**: Defined as the negative distance between cities (e.g., $R_1 = 1/d_{ij}$, $R_2 = -d_{ij}$, or $R_3 = -d_{ij}^2$)
2. **Algorithms**:
   - **Q-Learning**: An off-policy RL algorithm that updates Q-values using the maximum future reward (Bellman equation). It explores the environment and learns the optimal policy independently of the agent's actions.
   - **Sarsa**: An on-policy algorithm that updates Q-values based on the current policy, making it more conservative but potentially slower to converge.
   - **Double Q-Learning**: An extension of Q-Learning that uses two Q-tables to reduce overestimation bias, often yielding more accurate results.

3. **Exploration-Exploitation**:
   - The ε-greedy strategy balances exploration (random actions) and exploitation (best-known actions) by decaying ε over time. Four decay strategies were tested (linear, concave, convex, and step-wise).
4. **Optimization**:
   - The agent learns to select actions (city sequences) that maximize cumulative rewards (minimize total distance). The Q-table is iteratively updated using experiences (state-action-reward-next state).
5. **Results**:
   - Double Q-Learning often achieved solutions closest to the optimal, outperforming Q-Learning and Sarsa, especially for larger TSP instances. The reward function $R_1 = 1/d_{ij}$ and linear ε-decay (ε_1) were most effective.

Algorithm 1: Q-learning Algorithm

Set the parameters: α, γ and ε
Initialize the matrix $Q(s, a)$
Observe the state s
repeat
    Take action a using the ε-greedy method
    Receive immediate reward $r(s, a)$,Observe the new state s'
    Update $Q(s, a)$ with Eq. (1)
    $s = s'$
until *the stopping criterion is satisfied*;

Algorithm 2: Sarsa

Set the parameters: α, γ and ε
Initialize the matrix $Q(s, a)$
Observe the state s
Chooses the action a using the ε-greedy method
repeat
    Take the action a
    Receive immediate reward $r(s, a)$,Observe the new state s'
    Choose the new action a using ε-greedy method
    Update $Q(s, a)$ with Eq. (2)
    $s = s', a = a'$
until *the stopping criterion is satisfied*;

Algorithm 3: Double Q-Learning

Set the parameters: α, γ and ε
Initialize the matrix $Q^A(s, a) = 0$, $Q^B(s, a) = 0$
Observe the state s
repeat
    Take action a(e.g.ε-greedy) based on Q^A, Q^B
    Receive immediate reward $r(s, a)$,Observe the new state s'
    Choose (e.g.random)either UPDATE(A) or UPDATE(B)
    if *UPDATE(A)* **then**
        Update Q^A with Eq. (3)
    if *UPDATE(B)* **then**
        Update Q^B with Eq. (4)
    $s = s'$
until *end*;

Scalability Issues

- **Problem**: The method relies on Q-tables, which become infeasible for large TSP instances due to exponential growth in state-action space.
- **Modification**: Replace tabular methods with **Deep Reinforcement Learning (DRL)**

- **Problem**: The reward function in RL for TSP should align with the actual objective (minimizing total tour length), not just individual step-wise distances.
- **Modification**:
  - **Reward = Negative total tour length** (only given at the end of an episode).
  - **Pros**: Directly optimizes the true objective.
  - **Cons**: Sparse rewards make learning slower (requires many episodes).