The three horsemen of Classical Reinforcement Learning

From: Vizuara AI from Vizuara’s Substack (vizuara@substack.com)
To: yogeshkulkarni@yahoo.com
Date: Friday, August 8, 2025 at 04:10 PM GMT+5:30

“Hi, team Vizuara here. As always, thanks for letting us into your inbox. If you’re a new subscriber, welcome!”

Every issue of our newsletter is a deeply researched article on Artificial Intelligence and Machine Learning.

Our team consists of PhDs, professors and industry experts.

No fluff. In-depth AI content.

The three horsemen of Classical Reinforcement Learning

All about Dynamic Programming, Monte-Carlo and Temporal Difference Methods

VIZUARA AI
AUG 8

♡ 💬 ⇪ ⟳ READ IN APP ↗

Let us start by looking at value functions.

Value Functions

The value of a state is defined as the expected return starting in a state and following a policy thereafter.

This is also called the **state value function.**

Simply put, it is the expected value of all the cumulative rewards the agent is expected to receive from a given state.

Let's do a bit of mathematics.

The state value function is mathematically defined as follows:

$$v(s) = E(G(t)|S_t = s)$$

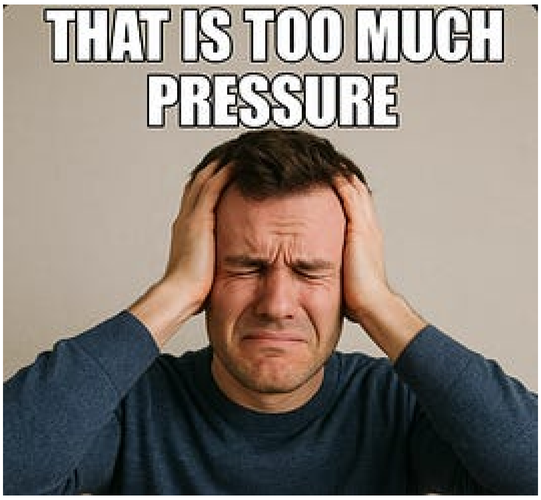
Here, G(t) is the total return received from the state s.

Remember we had discussed about the return in the previous lecture.

The return is given as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots + \ldots$$

The value function is the single most important concept that we will keep coming back to in this lecture series.



Now, let us look at another value function which is the action value function.

The action value function is slightly different than the state value function.

It signifies the expected return starting in a state, taking an action, and following a policy thereafter.

The action value function is also denoted by Q, or Q (s,a), meaning that the action value is calculated for a state and an action taken from that state.

The action value function is mathematically defined as follows:

$$Q(s,a) = E(G(t)|S_t = s, A_t = a)$$

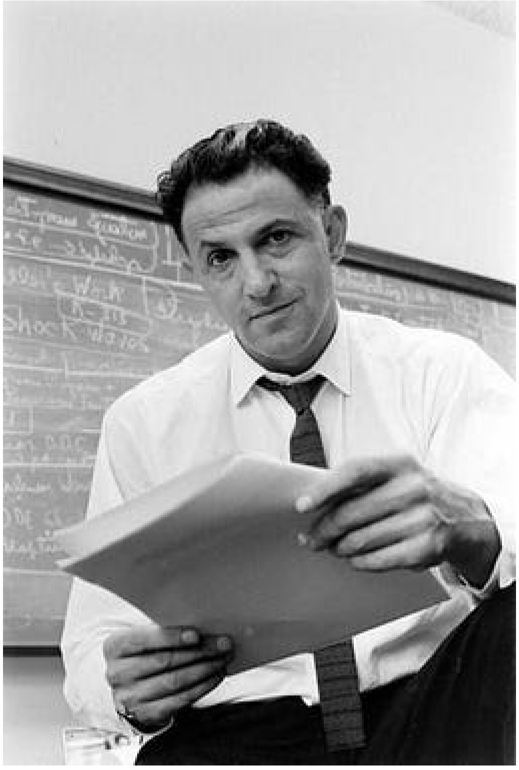
Another question that might be coming to your mind is, "How exactly are these value functions calculated?"

The main difficulty is that you cannot predict what the expected rewards are in the future.

For example, if you are playing the game of chess, the value of a move is the addition of all the cumulative rewards received in the future. But you cannot know these rewards unless you play the game and know whether you won or lost.

Let us come to the main character in our story. The person who is responsible for the entire progression of the field of reinforcement learning.

Mr Richard Bellman.



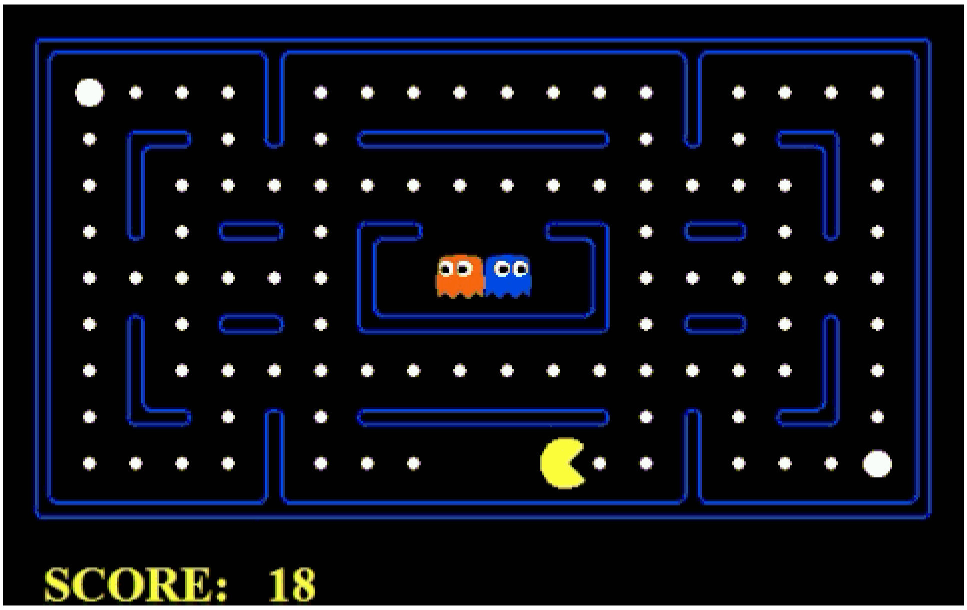
The Bellman Equation

Richard was a brilliant and highly creative person. He was known for his clear and simple insights, and he was a few years critic of unnecessary complex mathematics.

The Bellman equation is actually very simple.

It states that the value of being in a status is the expected reward for acting according to the policy π plus the expected value of wherever you end up next.

Imagine that you are playing the game "Pac-Man" with a routine strategy.



The routine that you follow is:

1. If a ghost is nearby, you always move away
2. If there is a pellet, you always move towards it
3. You never chase fruit

This routine is the policy π.

Now the question that you ask is,

> "If I start in this specific maze location with this policy, how many points will I score on average?"

So, you are asking: What is the value of every state?

The Bellman equation says that:

The value is given by the expected reward you receive now after choosing a specific action (e.g: pellet = +10), and the expected future value from wherever that move leads you.

So, it is like a recursive equation. The value of this state depends on the immediate reward plus the value of the next state.

Why was this groundbreaking?

Before Bellman equation, people struggled to solve long-term planning problems efficiently.

People thought, "How can you solve long-term planning problems?"

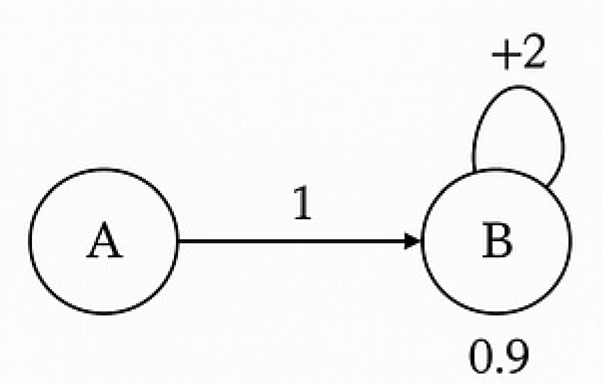
Aren't they too complex?

The Bellman equation showed that you can look at it one step at a time. Even if you can't comprehend the totality of the problem-solving, solving the problem one step at a time in a recursive manner can help you solve for the value of states.

This the main reason why this equation is considered to be groundbreaking.

Believe it or not, this will be at the heart of all the modern deep learning RL-Algorithms that we'll be looking at subsequently in this lecture series.

Simple Example:



Let us look at a simple Markov decision process to understand Bellman equation properly.

Let us say we have two states A and B.

For each state, there is only one action. From state A, you can go to state B. And from state B, you remain in state B.

The rewards are given as follows:

A→B, Reward = +1,

B→B, Reward = +2.

The discount factor is gamma = 0.9.

Now the question is that, we have to use the Bellman equation to compute the values of states A and B.

The Bellman equation states that the value of a state is equal to the immediate reward plus the expected value of the next state.

In practice, people also add a discount factor, gamma, to the expected value of the next state.

First, we will write down the equation for the value of state of B. For state B, after taking the action, you remain in state B and get an immediate reward of 2.

So, the Bellman equation can be written as:

$$V(B) = 2 + 0.9V(B)$$

Here, 0.9 is the discount factor.

Solving this equation, we get the value of state B as 20.

For state A, after taking the action, you go to state B and get a reward of +1.

So, the Bellman equation can be written as:

$$V(A) = 1 + 0.9V(B)$$

After substituting the value of state B, we get the value of state A as 19.

Finally, the values of straight A and B can be written as follows:

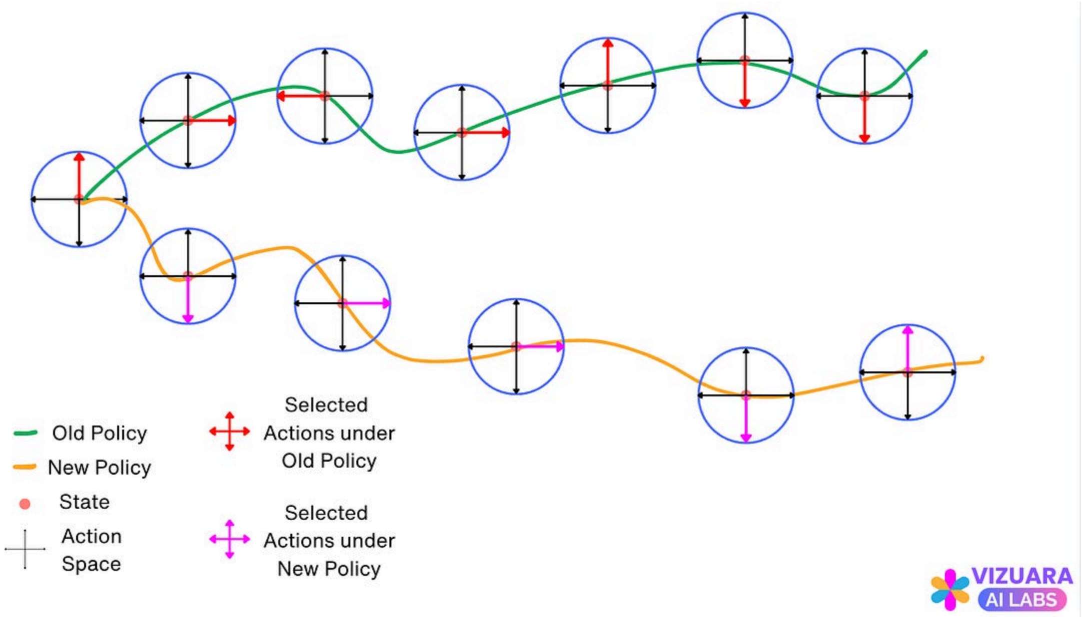
$$V(A) = 19, V(B) = 20$$

In this example, the policy was already given to us. In practical examples, one of the main goals of reinforcement learning is to find an optimal policy.

Can we look at the Bellman equation and understand how we can use it to find an optimal policy?

The reinforcement learning agent can follow different trajectories based on the actions it selects at every state.

Have a look at the two trajectories which are shown below for two policies (named Old and New):


Old Policy
New Policy
State
Action Space
Selected Actions under Old Policy
Selected Actions under New Policy
VIZUARA AI LABS

In this diagram, at every state, the agent has four possible actions, but depending on the policy the agent has chosen, we get completely different trajectories.

Hence, to find the optimal policy, choosing the right actions at every state is very important. The actions should be chosen such that our total cumulative reward in the episode is maximized.

But how do we choose actions that are the right actions in every state?

We can simply evaluate the value function for every action taken, and then choose the action which leads to the maximum value function.

So, for every state, we choose actions which maximize the summation of the immediate reward and the expected value of the next state

This is also called as the Bellman optimality equation.

Bellman Optimality Equation:

The Bellman Optimality equation is given as follows:

$$V^*(s) = max_a(\text{Reward} + \gamma * \text{Next State Value})$$

This equation states that for every state, we will find the action which has the maximum value of the addition of the immediate reward and the discounted value of the next state.

All of this might seem a little bit complex. Let us look at a practical example to understand the Bellman optimality equation better.

Recycling Robot solved using the Bellman Optimality Equation:

The recycling robot problem is very interesting. In this problem, a robot can be in one of two states: High or Low. The state indicates the battery level of the robot.

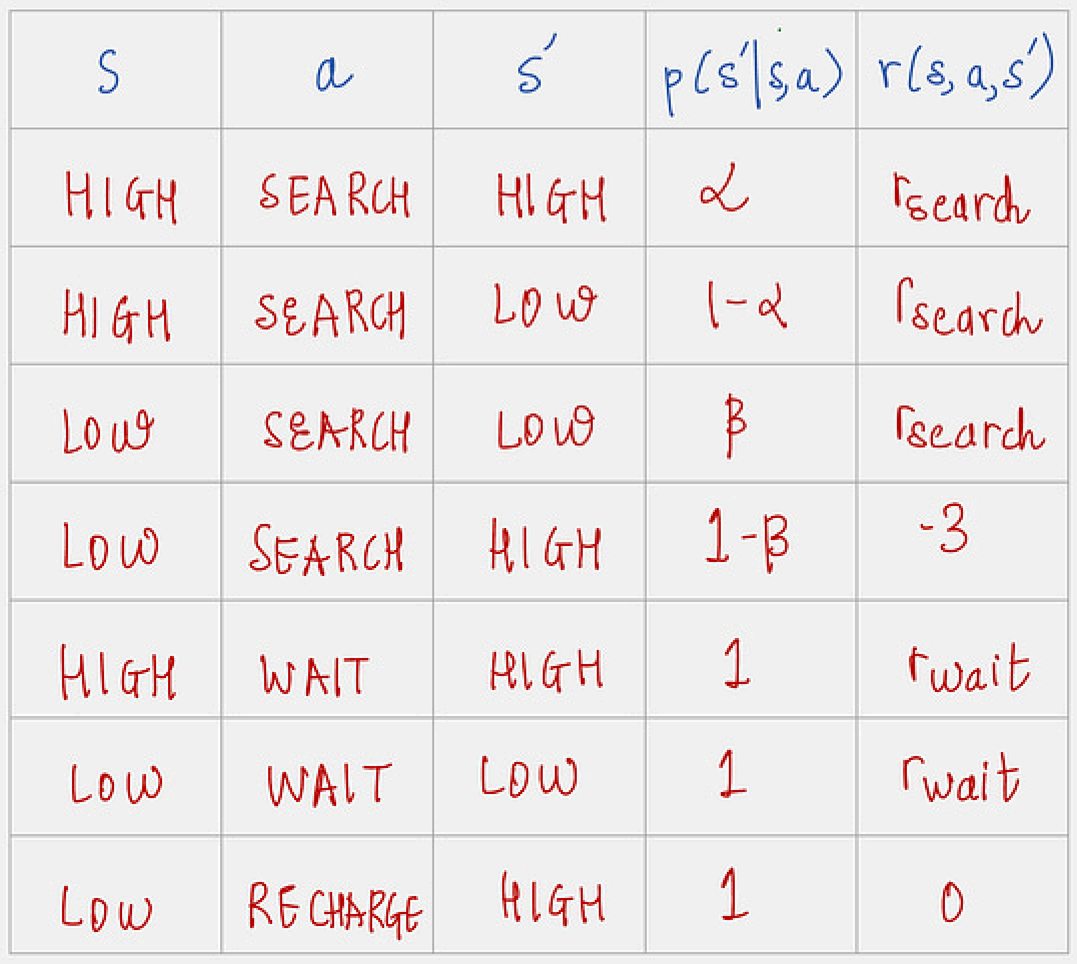
For each state, the robot has options of:

1. Searching
2. Waiting
3. Recharging

These are the actions in our agent-environment interface.

We first construct a table which lists down all possible states, actions, and the next state. We also write down the probability of transitioning from one state to the next state.

The table looks as follows:



s	a	s'	$p(s' \mid s,a)$	$r(s,a,s')$
HIGH	SEARCH	HIGH	α	r_{search}
HIGH	SEARCH	LOW	$1-\alpha$	r_{search}
LOW	SEARCH	LOW	β	r_{search}
LOW	SEARCH	HIGH	$1-\beta$	-3
HIGH	WAIT	HIGH	1	r_{wait}
LOW	WAIT	LOW	1	r_{wait}
LOW	RECHARGE	HIGH	1	0

You can see by a careful examination that this table does make sense. For example, if the robot is in a high state, and the action taken is to wait, then the next state is going to be high with the probability of 1.

Similarly, if the robot is in a low state and the action taken is to wait, then the next state is going to be low with a productivity of 1.

Now we want to understand what is the optimal action which the robot should take when it is in a high state as well as when it is in the low state.

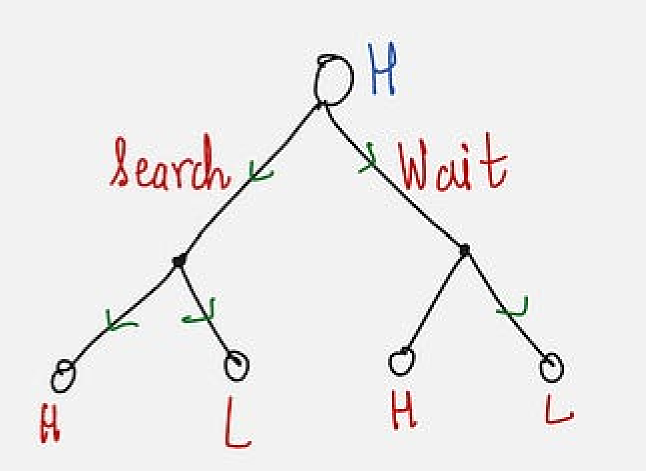
First, let us understand the optimal action in the high state.

If the robot is in a high state, there are two possible actions:

1. To search
2. To wait

We can create a diagram to understand what happens after each of these actions is taken. This diagram is called as the backup diagram.

The backup diagram for the high state is given below:



Remember that according to the Bellman optimality equation, we have to take the maximum of both these paths: the search path and the wait path.

The value of the search path is given by the addition of the two subsequent paths reaching the high and low states.

This can be written as:

$$\text{Term}_1 = \alpha[r(Search, H) + \gamma V^*(H)] + (1-\alpha)[r(Search, L) + \gamma V^*(L)]$$

Here, alpha is the probability of choosing the search path.

Similarly, the value of the wait path is given as:

$$\text{Term}_2 = r(Wait, H) + \gamma V^*(H)$$

This appears simplified because the robot cannot go from a high-state to low-state after waiting. So the second term is zero.

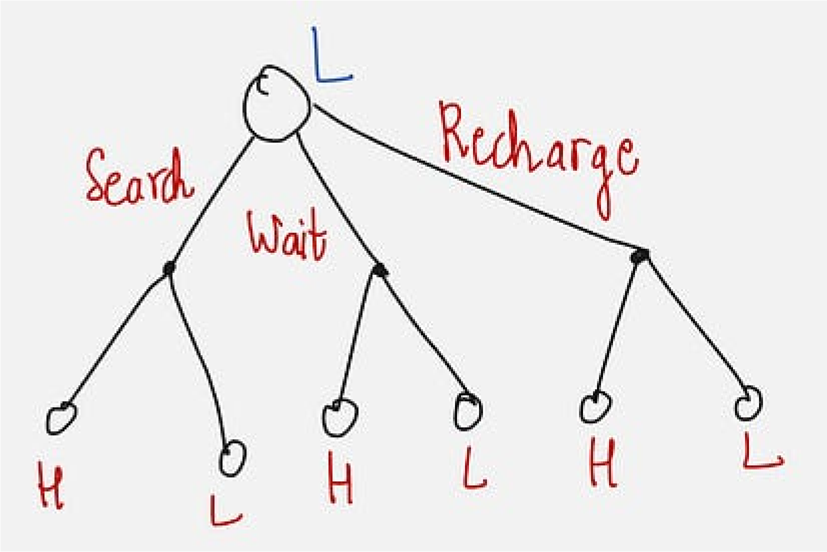
Now, according to the Bellman Optimality Equation, when the robot is in a high state, it should choose an action which has the maximum value.

If Term 1>Term2, robot will search.

If Term 2>Term 1, robot will wait.

Similarly, we can create the backup diagram for the low state.

The backup diagram for the low state is given below:



Remember that according to the Bellman optimality equation, we have to take the maximum of the three paths: the search path, the wait path and the recharge path.

The value of the search path is given by the addition of the two subsequent paths reaching the high and low states.

This can be written as:

$$\text{Term}_1 = (1-\beta)[r(Search, H) + \gamma V^*(H)] + \beta[r(Search, L) + \gamma V^*(L)]$$

Similarly, the value of the wait path is given as:

$$\text{Term}_2 = r(Wait, L) + \gamma V^*(L)$$

The value of the recharge path is given as:

$$\text{Term}_3 = r(Recharge, H) + \gamma V^*(H)$$

Now, according to the Bellman Optimality Equation, when the robot is in a low state, it should choose an action which has the maximum value.

If Term 1>Term2 and Term3, robot will search.

If Term 2>Term 1 and Term 3, robot will wait.

If Term 3>Term 1 and Term 2, robot will recharge.

You might have noticed that these equations are difficult to solve because in all these terms we have the value function which we do not know.

This is the main issue of having recursive equations because the variable that you are trying to solve appears both on the left-hand side and right-hand side of the equation.

So, how do we solve these equations?

This naturally brings us to the following topic.

The three horsemen of Classical Reinforcement Learning

First, we will start with **Dynamic Programming.**

Dynamic programming is a collection of algorithms that help us compute optimal policies for the reinforcement learning problem using the Bellman equation.

Let us discuss the steps involved in dynamic programming algorithms:

Step 1: Policy Evaluation

Policy evaluation means that if a policy is given, then how can I calculate the state value function for that policy?

This is also called as the prediction problem.

The Bellman equation tells us that the value function for a given state is given as follows:

$$V(s) = r + \gamma V(s^{'})$$

Here, s is the current state, s' is the next state and r is the reward collected while transitioning from state s to state s'. Note that this equation is simplified, and we are assuming that the probability of transition from straight s to s' is 1.

In dynamic programming, what we do is that we replace the above equation by the following:

$$V_{k+1}(s) = r + \gamma V_k(s^{'})$$

The way we solve this equation is by creating different iterations of the value function.

Let us initialize the first iteration as follows:

$$V_0(s) = 0$$

We have initialized the first iteration to be zero for all the states.

The next iteration can be written as follows:

$$V_1(s) = r$$

The next iteration can be written as follows:

$$V_2(s) = r + \gamma r$$

Similarly, we keep on going until the value function converges.

This is at the heart of dynamic programming.

For the dynamic programming algorithm to work, we need the following:

(1) The model of the environment requires the entire transition probabilities from each state to the subsequent states. That is, we need a proper backup diagram for all the states.

(2) Dynamic programming at every iteration requires all the values of all the states to be changed simultaneously. This is very computationally expensive.

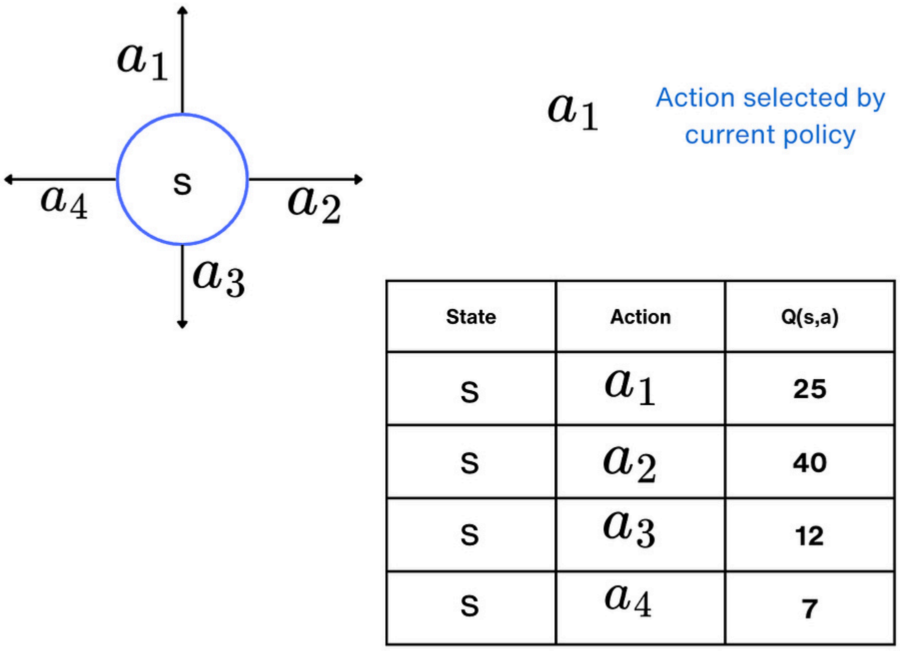
These are the two major drawbacks of dynamic programming.

Now that we have learned to predict the value function for a given policy, let us try to understand how we can improve the policy to reach the optimal policy.

Step 2: Policy Improvement

Policy improvement is straightforward since it very closely follows the Bellman optimality equation.

Consider the following case:



State	Action	Q(s,a)
s	a_1	25
s	a_2	40
s	a_3	12
s	a_4	7

Let us say that we are in the state ‘s’ and our current policy chooses action a1.

Now, looking at the action values for all the possible actions, we know that this action is not optimal, since the action a2 produces the highest action value.

This means that the policy has scope for improvement.

We change our current policy by telling our agent that when you reach a state, take action a2. We do a similar step for all the states the environment.

This is called policy improvement.

Notice that, the table in the above figure is obtained from the previous step of policy evaluation.

But how do we reach the optimal policy?

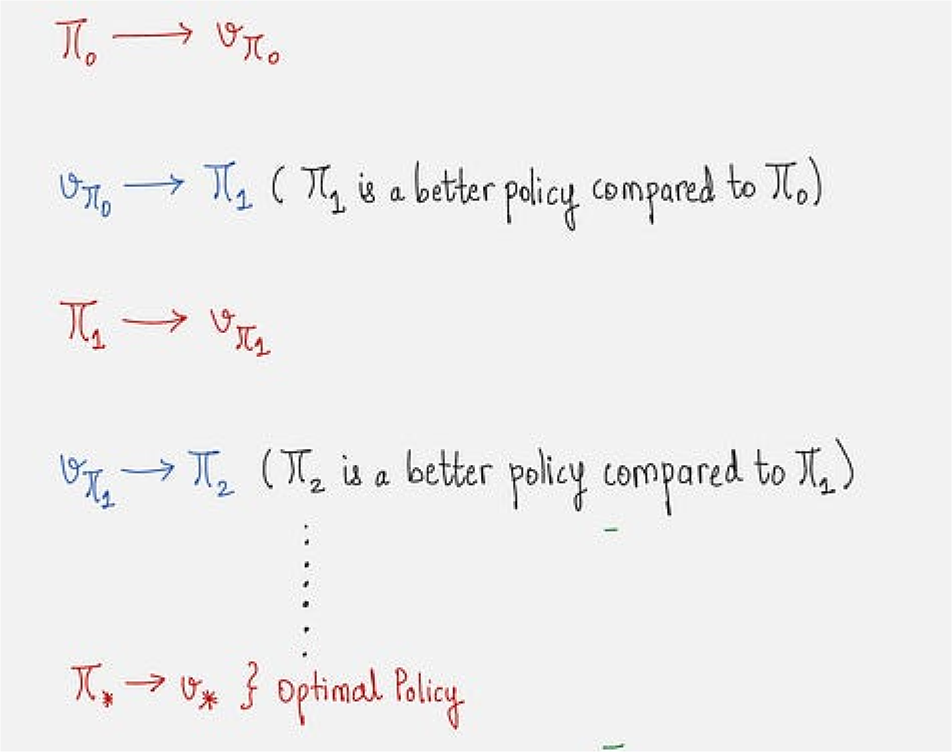
This brings us to step 3.

Step 3: Policy Iteration

First, we use the policy to calculate the value function. Then we use the value function to update the policy.

We continue the process till we reach a stage, where, for all the states, we have selected actions which are producing the best action values for that state.

The process looks like this:



$\pi_0 \longrightarrow v_{\pi_0}$

$v_{\pi_0} \longrightarrow \pi_1$ (π_1 is a better policy compared to π_0)

$\pi_1 \longrightarrow v_{\pi_1}$

$v_{\pi_1} \longrightarrow \pi_2$ (π_2 is a better policy compared to π_1)

$\vdots$

$\pi_* \rightarrow v_* \}$ Optimal Policy

The main issue with dynamic programming is that we need an actual model of the environment, including the transition probabilities for each state.

In most of the cases, this is not available with us.

For example, in the chess problem, we do not know the probability of transitioning from each state to every other state. It is very difficult to calculate.

Hence we need other methods.

This brings us to the second horseman of classical reinforcement planning.

Monte-Carlo Methods:

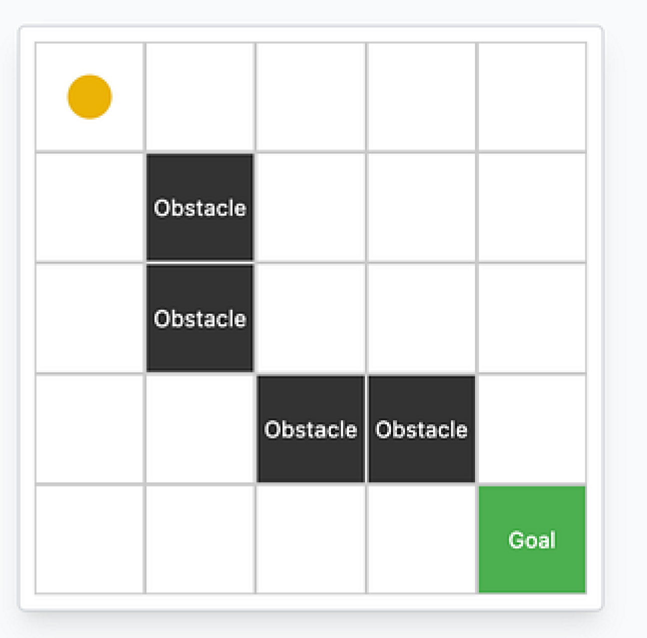
Dynamic programming methods require a complete model of the environment. This is not possible to obtain in most cases.

In contrast, Monte Carlo methods require only experience and do not require any prior knowledge about the dynamics of the environment.

Imagine our agent is a rover that we have sent to Mars. We cannot use dynamic programming to calculate the optimal policies for the rover because we do not know the model of the Martian environment.

In such cases, the Monte Carlo methods are very useful.

Let us understand about these methods with an example.

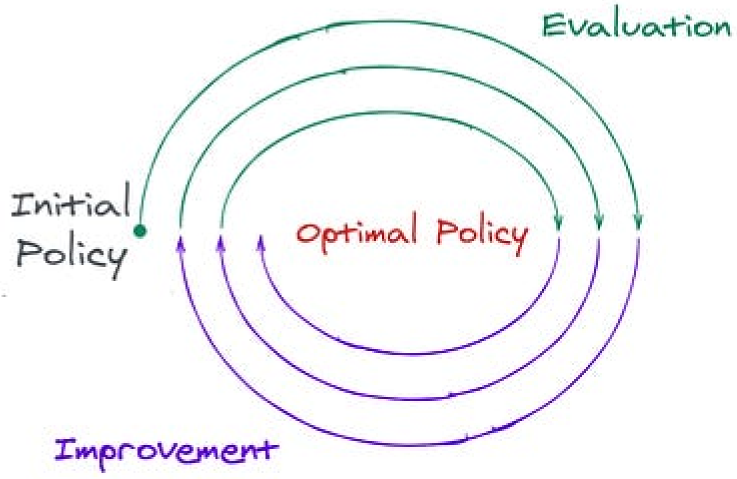


In the above figure, we are the yellow dot, and our goal is to reach the tile marked as ‘Goal’, by avoiding the obstacles.

To calculate the optimal value functions associated with the optimal policy, let us look at both the methods of dynamic programming and Monte Carlo.

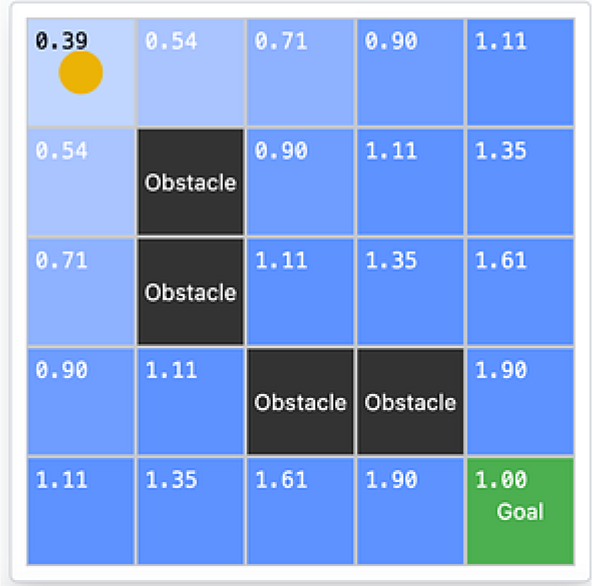
Method 1: Dynamic Programming

In dynamic programming, we will do the following:



This would require the model of the environment, that is the probability of transitioning from its current state to the next state, for all possible states in the grid.

Finally, we would obtain the optimal value functions for each state, which might look something like this:



0.39 ●	0.54	0.71	0.90	1.11
0.54	Obstacle	0.90	1.11	1.35
0.71	Obstacle	1.11	1.35	1.61
0.90	1.11	Obstacle	Obstacle	1.90
1.11	1.35	1.61	1.90	1.00 Goal

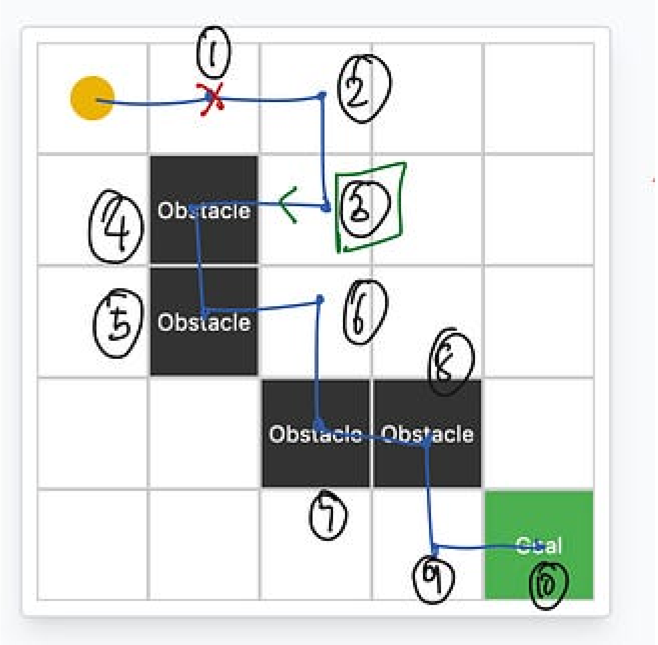
Method 2: Monte-Carlo

Now let us assume that we do not know anything about the environment.

But we can collect a lot of experiences by taking a number of different paths.

Let us look at some sample episodes:

Episode 1:



In the Monte-Carlo method, what we do is, after an episode is completed, we update the values of all the states by calculating the return received from that state.

For example, let us look at the state marked as 1.

The return for this state is given as:

$G_1^1 = R_2 + \gamma R_3 + \gamma^2 R_4 + \gamma^3 R_5 + \gamma^4 R_6 + \gamma^5 R_7 + \gamma^6 R_8 + \gamma^7 R_9 + \gamma^8 R_{10}$

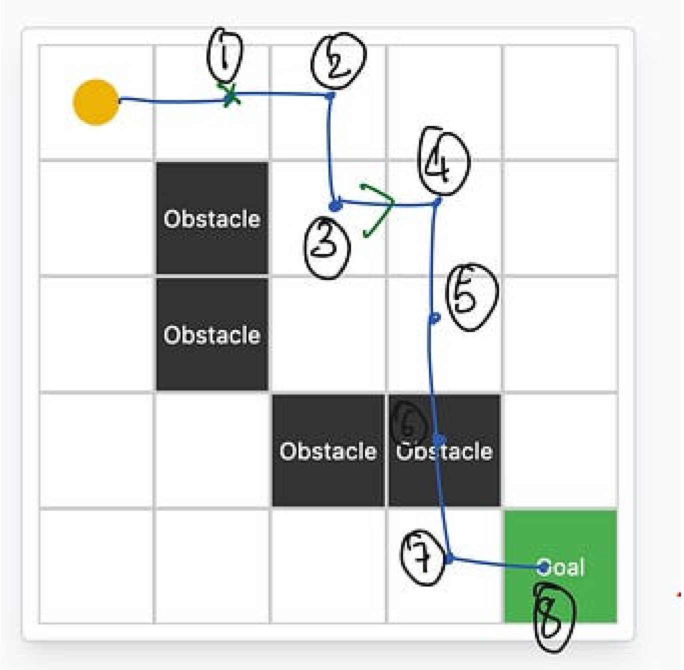
Next we set the value of the state equal to this return:

$$V_1 = G_1^1$$

Similarly, we calculate the return for all the states and update their values.

Now, we carry out another episode.

Episode 2:



Now we get another set of returns for the states.

For example, for state 1, we can get the second return as follows:

$G_1^2 = R_2 + \gamma R_3 + \gamma^2 R_4 + \gamma^3 R_5 + \gamma^4 R_6 + \gamma^5 R_7 + \gamma^6 R_8$

Next we set the value of the state as the average of returns from episodes 1 and 2.

$$V_1 = \frac{G_1^1 + G_1^2}{2}$$

Similarly, we calculate the return for all other states and update their values by finding the average of both the episodes.

Now you can get an idea of what would happen if we keep on increasing the episodes. For example, the value of state 1 for episode 100 would look as follows:

$$V_1 = \frac{G_1^1 + G_1^2 + G_1^3 + ... + G_1^{100}}{100}$$

The beautiful property of Monte Carlo methods is that as the number of episodes increase, the approximate value functions of the states will approach the true value functions of the states.

The algorithm for the Monte Carlo prediction problem looks as follows:

```
Initialize a Empty list of Returns called R.

Generate an Episode using the Policy

For each state in the episode do the following:

Calculate the expected return (G) following the first
visit to s

Append G to Returns list (R)

Calculate the average of the Returns List
```

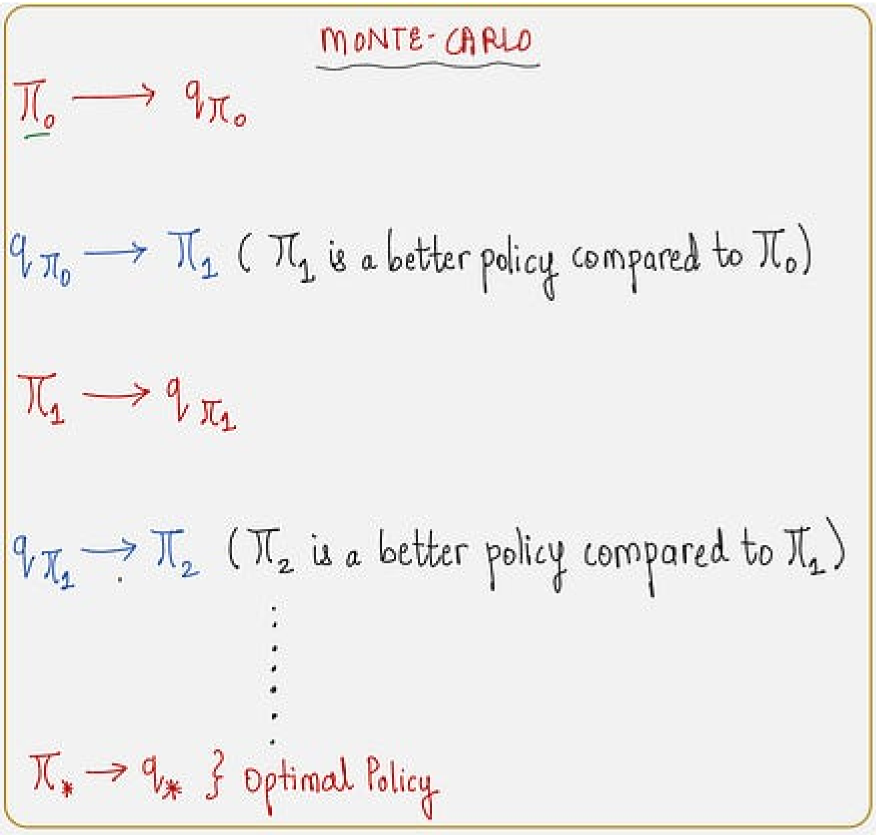
Using the Monte Carlo method, we have learnt to calculate the value of each state. To find out which action to select at every state, we need the action values.

This is done in the exact same way as the algorithm described above. The only difference is that, we are now talking about visits to a state-action pair rather than to a state.

Once we have solved the prediction problem, the objective of the control problem is to find the policy which is optimal.

Here, we use the same approach of policy iteration that we used in dynamic programming. The only difference is that instead of a state value function, we use an action value function.

The policy iteration approach looks as follows:



MONTE-CARLO

$\pi_0 \longrightarrow q_{\pi_0}$

$q_{\pi_0} \longrightarrow \pi_1$ (π_1 is a better policy compared to π_0)

$\pi_1 \longrightarrow q_{\pi_1}$

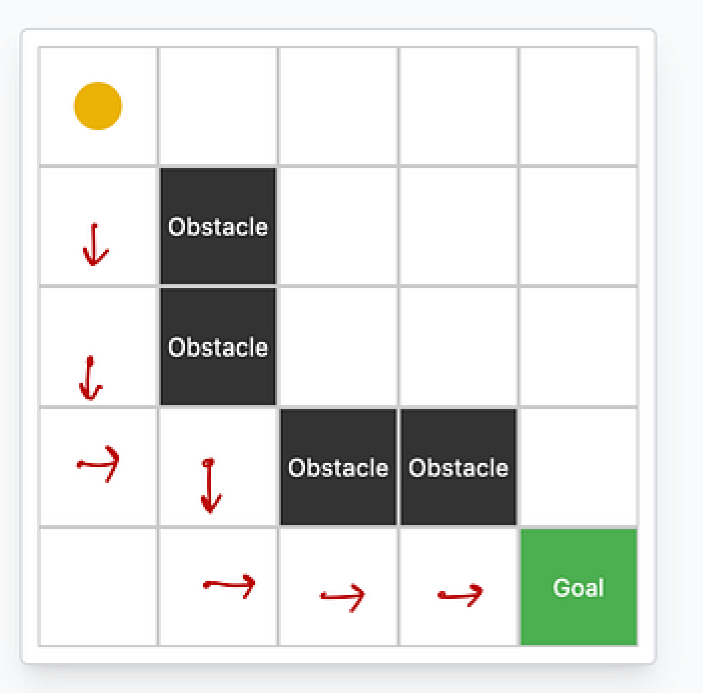
$q_{\pi_1} \longrightarrow \pi_2$ (π_2 is a better policy compared to π_1)

⋮

$\pi_* \rightarrow q_*$ } Optimal Policy

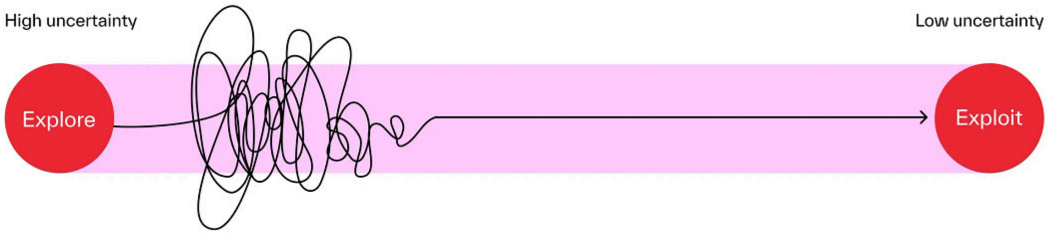
A common issue with the Monte Carlo algorithm is the following:

What if we never explore some actions?



What if the actions shown in the above figure are never explored?

This brings us to the trade-off between exploration and exploitation.



Exploitation uses the agent's current knowledge to maximize immediate rewards by choosing the best known action. Exploration involves trying out new actions to discover potentially better strategies or rewards.

We need to design the policy which has a combination of both exploitation and exploration.

Our current policy is called a greedy policy because it only chooses the best possible action at every state.

We will adopt a new strategy which says that every once in a while, with a small probability ε, select randomly among all actions.

For example, if ε is equal to 0.1, out of 100 moves, 10 moves will be exploratory random moves.

This will make sure that all moves are tried, and the estimated value of the actions will converge to the true values.

This is exactly how we solve the Monte Carlo control problem.

This policy is called the as ε-greedy policy.

Monte Carlo methods do not require a model of the environment, which gives them an edge over dynamic programming. However, to make even a single update, we have to wait till the end of an episode. This is inconvenient.

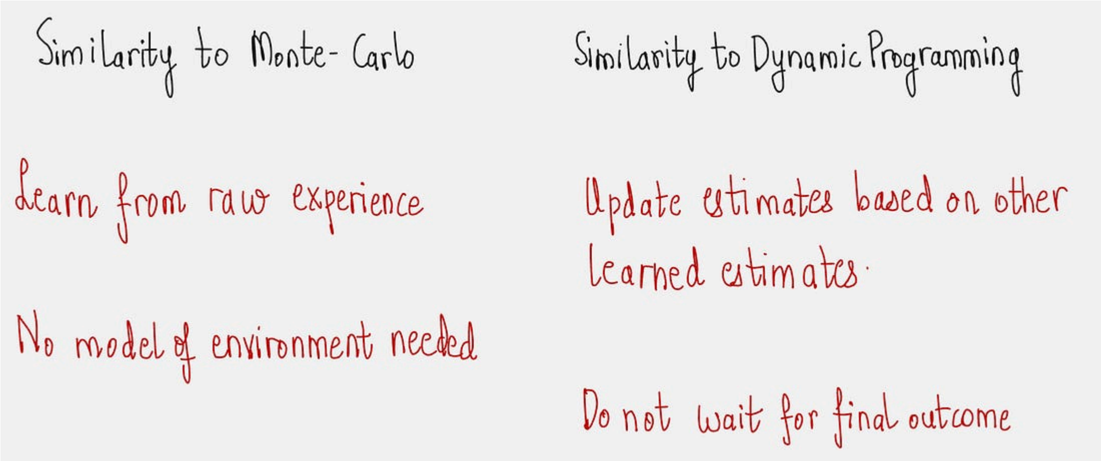
This brings us to the third horseman of classical reinforcement learning: Temporal Difference Methods.

Temporal Difference Methods

These methods are a combination of Monte Carlo ideas and dynamic programming ideas.

They are the best of both worlds.

First, let us look at a schematic which illustrates how temporal difference methods are similar to both the methods we looked at before. Then we will understand these similarities in detail.



Similarity to Monte-Carlo	Similarity to Dynamic Programming
Learn from raw experience	Update estimates based on other learned estimates.
No model of environment needed	Do not wait for final outcome

Let us dive a bit into some mathematics.

HOLD ON TO YOUR SEATBELTS, BECAUSE THIS IS GO TO BE A BUMPY MATHEMATICAL RIDE

Mathematical Intuition:

Let us say that we want to use the Monte Carlo method to estimate the value of a state.

Let us say that we have completed 100 episodes, and the value function of a state is given as follows:

$$V_{100}(s) = \frac{G_0+G_2+G_3+\ldots+G_{99}}{100}$$

Now we collect a 101st episode. The value function will be updated as follows:

$$V_{101}(s) = \frac{G_0+G_2+G_3+\ldots+G_{99}+G_{100}}{101}$$

The updated value function can be written in terms of the previous value function as follows:

$$V_{101}(s) = V_{100}(s) + \frac{1}{101}\left[G_{100} - V_{100}(s)\right]$$

This can also be written as follows:

$$\text{New Estimate} = \text{Old Estimate} + \alpha[\text{Return} - \text{Old Estimate}]$$

Here, α is called as the step-size parameter.

The Return is only known to us after the episode is completed.

From the above expression, we can clearly see that, for Monte Carlo updates, we need to wait till the end of the episode before finding the new estimate.

In temporal difference methods, we ask the question: "Can we replace the return by a quantity, which can be estimated without waiting till the end of the episode”

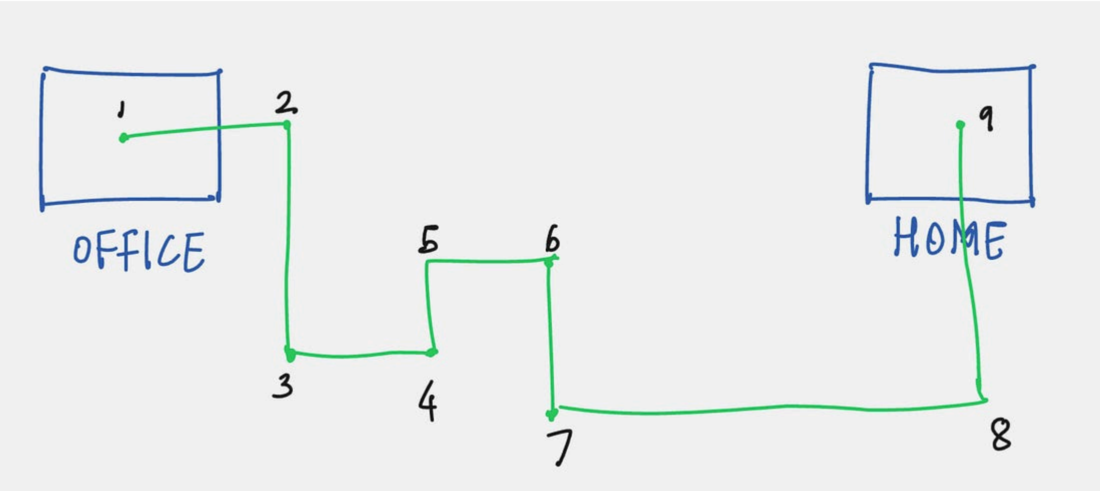
In the Temporal Difference method, the return is replaced by the addition of the immediate reward and the value of the next state. This is given by the following expression:

$$R_{100} + \gamma V(s')$$

So, our updated value function can be written in terms of the previous value function as follows:

$$V_{101}(s) = V_{100}(s) + \frac{1}{101}[R_{100} + \gamma V(s') - V_{100}(s)]$$

Let us understand the difference between Monte Carlo and temporal difference methods using a very **practical example.**



Let us assume that we are going home from our office.

In between, there are a number of junctions.

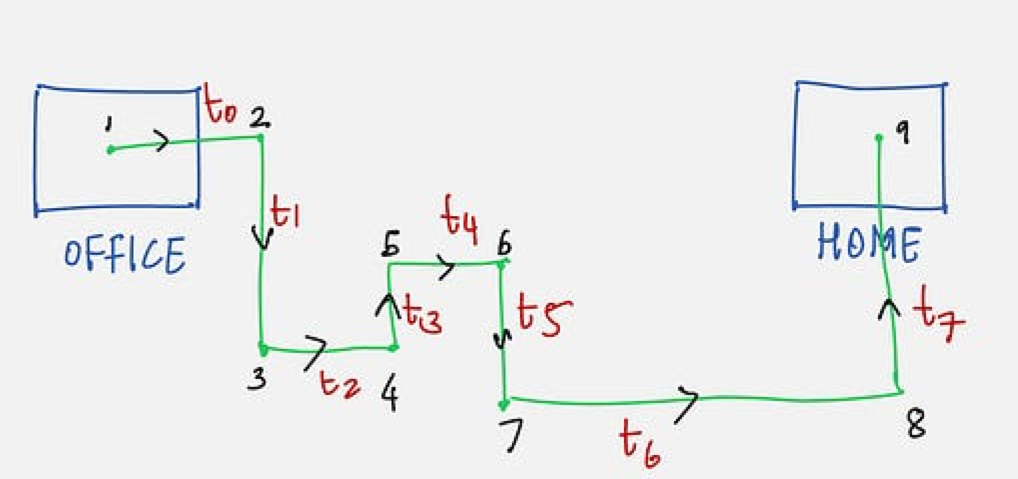
We are interested to find out how much time it takes from each junction to reach home.

First, let us formulate this problem as a reinforcement learning problem with an agent-environment interface.

States: Junctions

Actions: Fixed. We are not learning anything here

Rewards: Time taken between the junctions. Look at the diagram below.



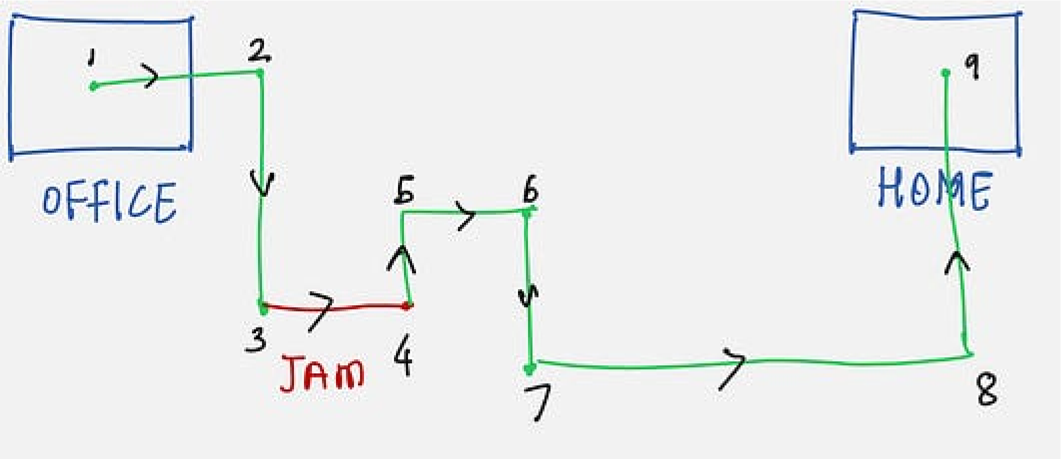
This formulation ensures that the value of each state is the total time taken to reach from each junction to home, which is what we want to estimate.

First, let us see how the values are calculated using Monte Carlo methods.

Let us focus on junction 3. Using the Monte Carlo estimation formula, which we looked at before, the value of junction 3 will be given as:

$$V_3 = V_3 + \alpha[(t_2 + t_3 + \ldots + t_7) - V_3]$$

Now let us say that there is a big traffic jam between junction 3 and junction 4.



When will this information be used to update the value of junction 3 and increase its value?

Only after the episode is completed, only when we reach home on that day.

Why should we wait for the entire episode to be completed to update the value?

This is where temporal difference learning comes in.

According to TD Approach, learning happens immediately.

The Temporal Difference update rule for state ‘3’ is given by:

$$V_3 = V_3 + \alpha(t_2 + \gamma V_4 - V_3)$$

If there is a traffic jam between junctions 3 and 4, the value t2 will be high, and this will immediately update the value of state 3.

This example nicely illustrates the difference between Temporal Difference and Monte Carlo Methods.

Now that we have developed a brief understanding of temporal difference methods, let us look at the two widely used temporal difference algorithms which will be used to control our reinforcement learning agent:

SARSA:

Based on the previous discussion, we know that the Temporal Difference update rule is given by the following equation:

$$V(s) = V(s) + \alpha[R + \gamma V(s') - V(s)]$$

Now we can simply replace the state value function with the action value function.

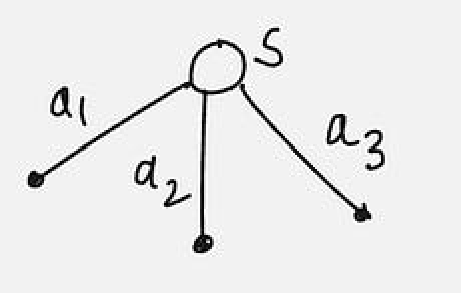
$$Q(s,a) = Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)]$$

Now you can probably understand where the name SARSA comes from. All the initials are included in the above formula:

$$s, a, r, s', a'$$

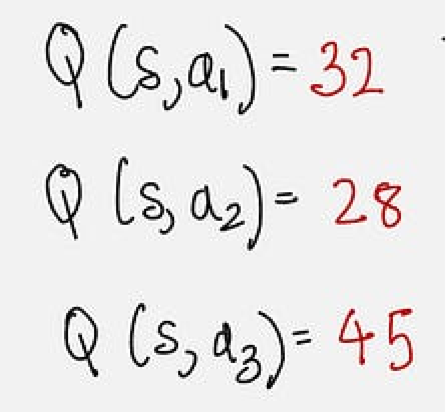
Now, to solve the control problem, we again use the method of policy iteration as we did for Dynamic Programming and Monte-Carlo methods.

Let us revisit this briefly.



Let us say that we are in state S. Our policy tells us to choose action a2.

However, our current estimate of the action value function gives us the following values of the actions.



$Q(s,a_1) = 32$

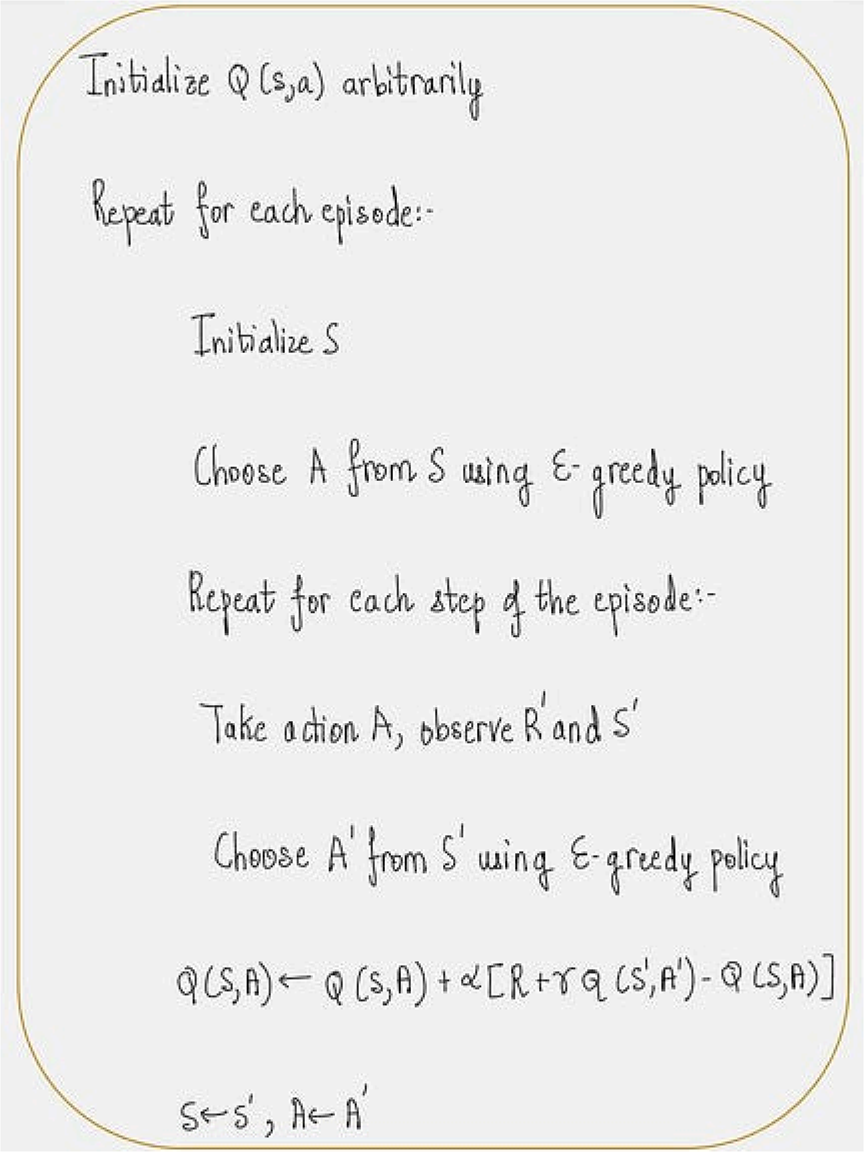
$Q(s,a_2) = 28$

$Q(s,a_3) = 45$

Clearly, our policy is not optimal; we should choose a3 and not a2 as suggested by our current policy.

However, this would be a greedy policy. Our agent might not explore other policies.

Hence, same as we did for the Monte Carlo Method, we will use an Epsilon Greedy Policy to balance exploration and exploitation.

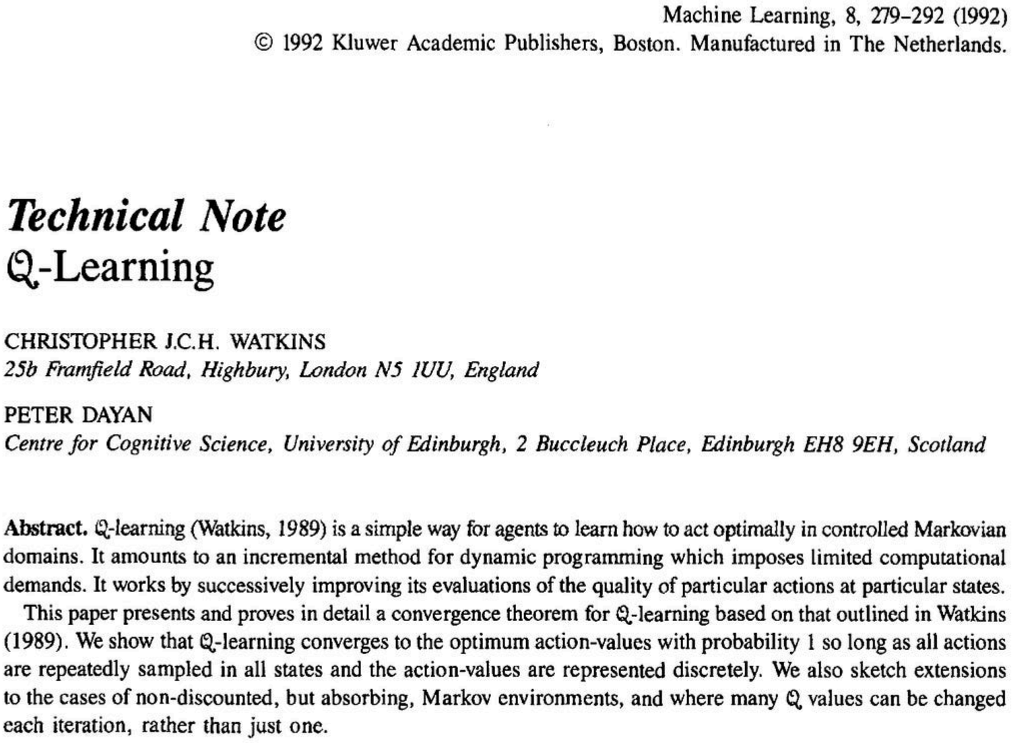


```
Initialize $Q(s,a)$ arbitrarily
Repeat for each episode:-
        Initialize $S$
        Choose $A$ from $S$ using $\epsilon$- greedy policy
        Repeat for each step of the episode:-
          Take action $A$, observe $R'$ and $S'$
          Choose $A'$ from $S'$ using $\epsilon$-greedy policy
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
        $S \leftarrow S'$, $A \leftarrow A'$
```

Now, we look at the second temporal difference algorithm which is even more popular in SARSA and it is called Q-learning.

Q-Learning

The Q-learning algorithm was developed by Chris Watkins in 1992.



Machine Learning, 8, 279–292 (1992)
© 1992 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Technical Note
Q-Learning

CHRISTOPHER J.C.H. WATKINS
25b Framfield Road, Highbury, London N5 1UU, England

PETER DAYAN
Centre for Cognitive Science, University of Edinburgh, 2 Buccleuch Place, Edinburgh EH8 9EH, Scotland

Abstract. Q-learning (Watkins, 1989) is a simple way for agents to learn how to act optimally in controlled Markovian domains. It amounts to an incremental method for dynamic programming which imposes limited computational demands. It works by successively improving its evaluations of the quality of particular actions at particular states.

This paper presents and proves in detail a convergence theorem for Q-learning based on that outlined in Watkins (1989). We show that Q-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely. We also sketch extensions to the cases of non-discounted, but absorbing, Markov environments, and where many Q values can be changed each iteration, rather than just one.

This algorithm went under the radar when it was developed. However, it was resurrected in the year 2013, when DeepMind released a paper on using Q-learning to play Atari games.

Let us understand Q-learning.

First, let us take an example. Let us say that we are trying to ride a bicycle.

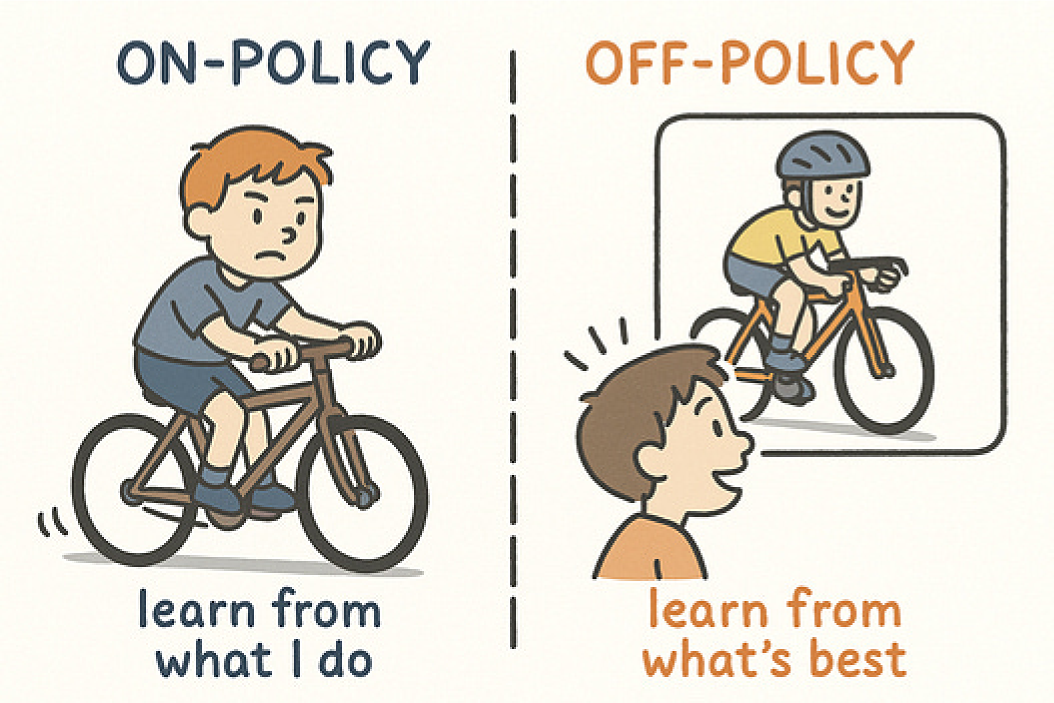
The way you ride now is your current policy. You are trying to get better at your style of riding. You take notes about what happens when you follow your current style.

You are learning the same policy which you are following. This is called on-policy learning.

Now imagine that you are still trying to ride a bike, but you are watching a video of a professional cyclist.

You still ride around and try things, but when you update your understanding, you ask, "What would have happened if I had balanced perfectly like the expert?"

So you are behaving one way (riding around) but learning about another way (balancing perfectly like the expert). This is called off-policy learning.



Q-learning is an off-policy temporal difference algorithm. It is one of the most important breakthroughs in reinforcement learning.

It is mathematically written with a small change from SARSA:

$$Q(s,a) = Q(s,a) + \alpha[R + \gamma max_a[Q(s',a')] - Q(s,a)]$$

You can see that here we are choosing the next state by selecting an action which gives the maximum Q-value. This is a greedy policy.

However, our actual policy is epsilon-greedy.

Our agent behaves according to the epsilon-greedy policy, but we are trying to learn from the greedy policy. This is why Q-learning is an example of off-policy learning.

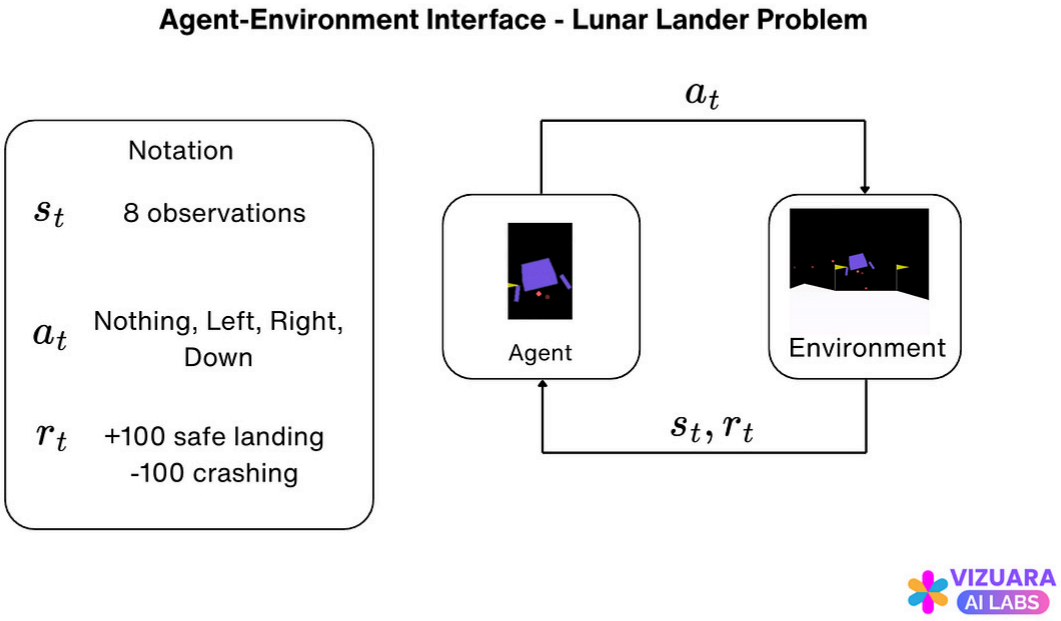
Behavior Policy: Policy which agent uses to interact with the environment.

Target Policy: Greedy policy, always take the action the highest Q-value.

To strengthen our understanding of the three horsemen of classical reinforcement learning, let us take an example using the OpenAI Gymnasium where we will actually simulate the lunar lander using both Monte Carlo and temporal difference methods.

Controlling our Lunar Lander using MC and TD Methods:

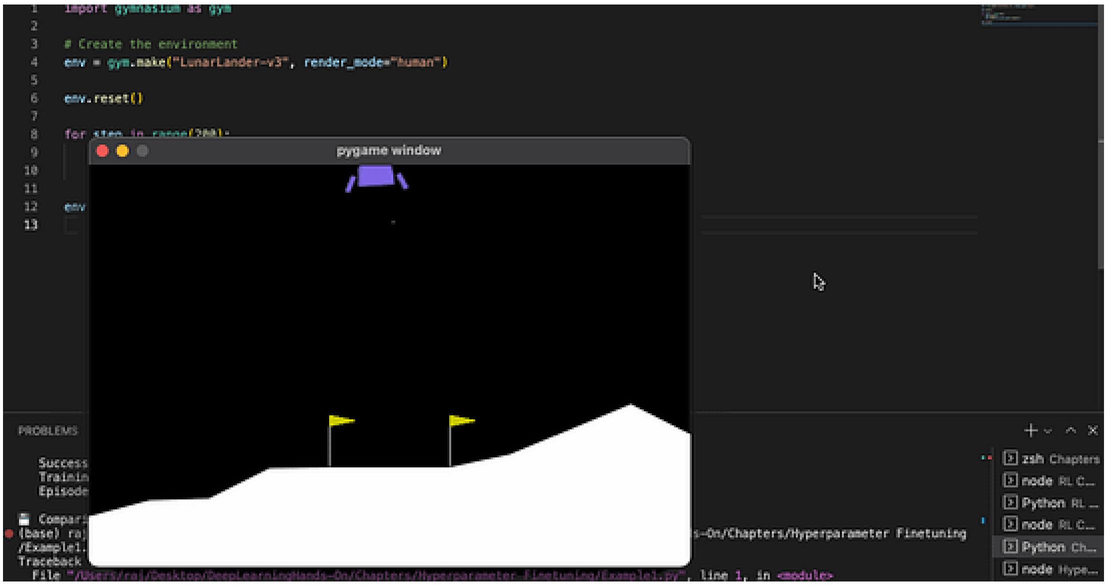
Remember from our past lecture that the agent-environment interface for the lunar lander environment looked like this:


Agent-Environment Interface - Lunar Lander Problem
Notation
s_t 8 observations
a_t Nothing, Left, Right, Down
r_t +100 safe landing -100 crashing
a_t
Agent
Environment
s_t, r_t
VIZUARA AI LABS

And we looked at a code to control our lander with random actions:

```
import gymnasium as gym

# Create the environment
env = gym.make("LunarLander-v3", render_mode="human")

env.reset()

for step in range(200):
    env.render()
    env.step(env.action_space.sample())

env.close()
```

But in this approach, we are doing our actions randomly. Our agent does not perform well.



```
import gymnasium as gym

# Create the environment
env = gym.make("LunarLander-v3", render_mode="human")

env.reset()

for step in range(200):



env
```

```
  Success
  Trainin
  Episode

 Compari
(base) raj                                    s-On/Chapters/Hyperparameter Finetuning
/Example1.
Traceback
  File "/Users/raj/Desktop/DeepLearningHands-On/Chapters/Hyperparameter Finetuning/Example1.py", line 1, in <module>
```

This was because we were missing one of the key elements of reinforcement learning problems, which was the policy.

Now we will look at two policies.

Policy #1: Monte Carlo Method

The key piece of code in the Monte Carlo method looks as follows:

```
# Calculate discounted returns (working backwards)
    G = 0
    for t in range(episode_length - 1, -1, -1):
        G = GAMMA * G + episode_rewards[t]

        state_t = episode_states[t]
        action_t = episode_actions[t]

        # First-visit Monte Carlo: only update if this is
first occurrence of state-action pair
        if not any(episode_states[i] == state_t and
episode_actions[i] == action_t for i in range(t)):
            returns_sum[state_t, action_t] += G
            returns_count[state_t, action_t] += 1

            # Update Q-value with average return
            Q[state_t, action_t] = returns_sum[state_t,
action_t] / returns_count[state_t, action_t]
```

In this code, ‘G’ holds the return for the current episode, and ‘returns_sum’ holds the summation of returns for each state-action pair across episodes.

Example Scenario:

Episode 1:

- State 42, Action 1 → Return G = 150

Episode 5:

- State 42, Action 1 → Return G = 200

Episode 12:

- State 42, Action 1 → Return G = 100

What We Want:

The Q-value for (State 42, Action 1) should be the **average** of all returns we've seen:

```
Q[42, 1] = (150 + 200 + 100) / 3 = 150
```

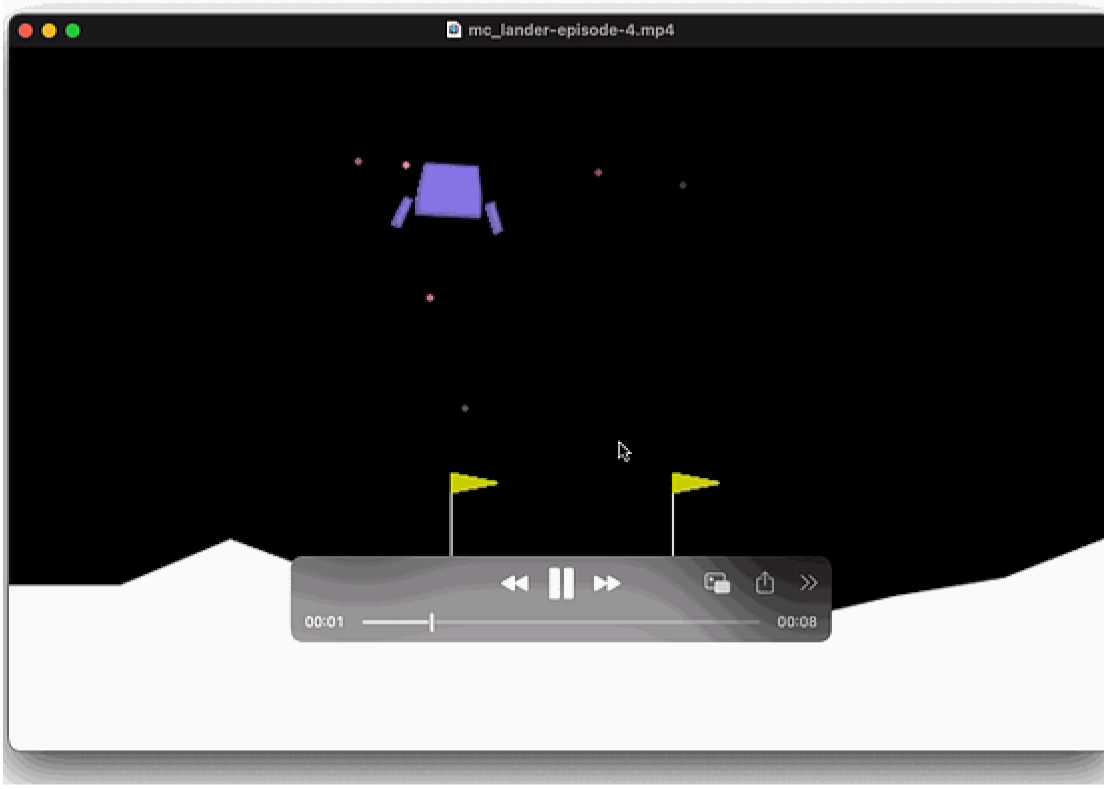
How the arrays in the code work?

```
returns_sum[42, 1] = 150 + 200 + 100 = 450
returns_count[42, 1] = 3
Q[42, 1] = returns_sum[42, 1] / returns_count[42, 1] =
450/3 = 150
```

You will also see the following line in the above code:

```
if not any(episode_states[i] == state_t and
episode_actions[i] == action_t for i in range(t)):
```

This is an example of First Visit Monte Carlo implementation. It means that if there is no earlier occurrence of this state-action pair, then only update the Q-value.



This is how our lunar lander performs. Not bad right? It is definitely better than the random lunar lander.

This is how the average rewards increase during the training, which means that our agent is actually learning:

Now let us try using the Temporal Difference approach.

Policy #2: Temporal Difference Method - Q Learning

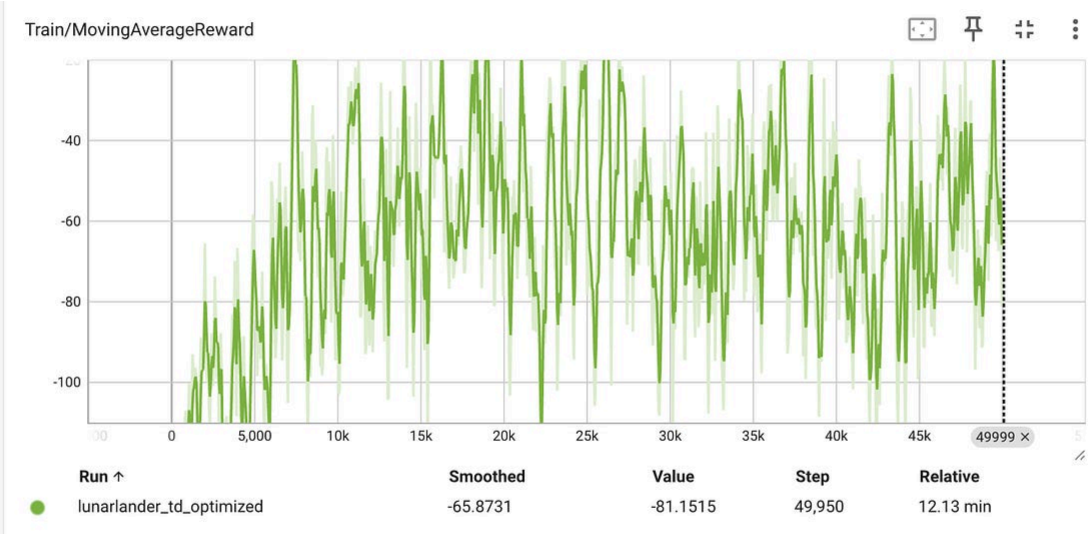
The key piece of code in the temporal difference method looks as follows:

```
    while not done and steps < 1000:  # Add step limit to
prevent infinite episodes
        action = choose_action(state, EPSILON)
        next_state, reward, done, truncated, _ =
env.step(action)
        next_state_discrete = discretize(next_state)

        # Vectorized Q-learning update
        if not (done or truncated):
            td_target = reward + GAMMA *
np.max(Q[next_state_discrete])
        else:
            td_target = reward

        Q[state, action] += ALPHA * (td_target - Q[state,
action])

        total_reward += reward
        state = next_state_discrete
        steps += 1
```

Here you can see that we are using the exact same Q-Learning formula which we had seen before. Also, we are not waiting till the end of the episode to make an update.

We are taking a step in the environment and immediately making an update.

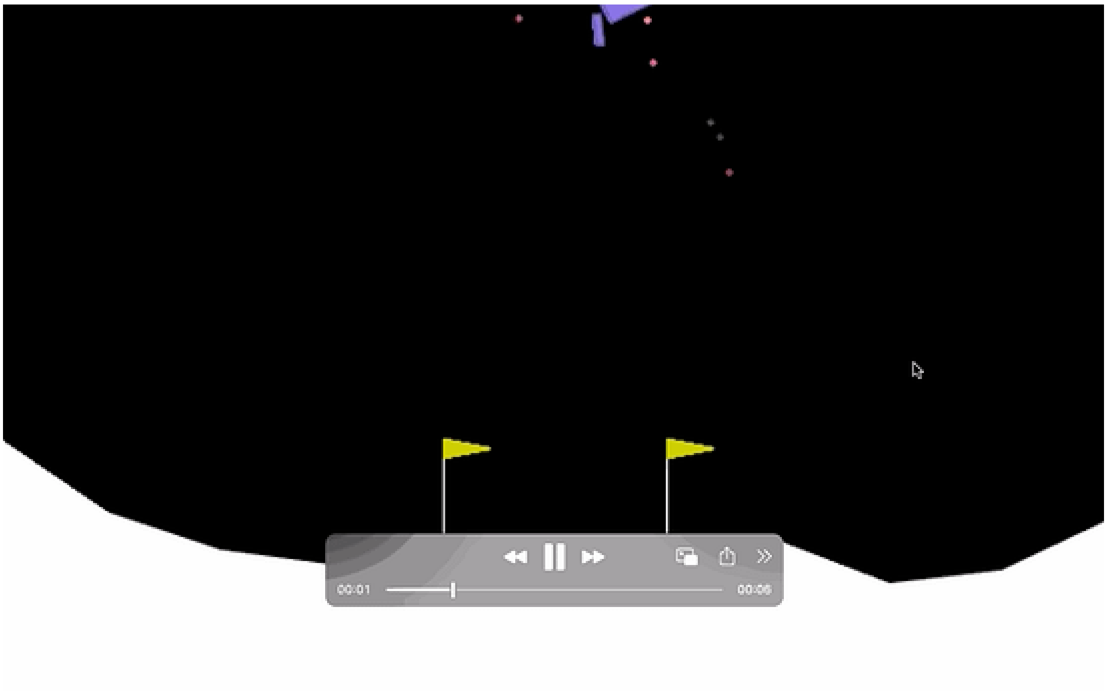
This is how the average rewards look like when training using the Q-learning method.



Run ↑	Smoothed	Value	Step	Relative
● lunarlander_td_optimized	-65.8731	-81.1515	49,950	12.13 min

You can see that there is not a clear trend, but overall, we seem to be doing well, and the agent seems to be achieving more rewards.

This is how our rover performs:



Again, not bad!

This example shows the power of methods like Q-learning.

However, if you look at this example closely, you will realize that we have maintained the Q-values in an array.

Can you imagine what will happen if we have an environment which has a huge number of states?

For example, the game of chess has 10^46 states. It will be impossible for us to tabulate all these values in an array.

So, what is the practical solution?

Can we learn a function to represent the Q-values for state-action pairs based on a limited amount of information and then extrapolate from there to unseen data?

Does this remind you of something?

We will cover this in detail in the next lecture.

548 Market Street PMB 72296, San Francisco, CA 94104