Reinforcement Learning

Reinforcement Learning (LR) is a computational goal-directed approach to learning from interaction.

Index

1. History

The history of RL has three threads:

One thread concern learning by trial and error and started in the psychology of animal learning. The second thread concerns the problem of optimal control and its solution using value functions and dynamic programing. A third thread concerns about temporal difference (TD) methods. All three threads converged into RL around the late 1980s.

1.1.1. Trial and Error

Edward Thorndike was probably the first one introducing the idea of trial-and-error learning. All the concepts are gathered under the ***"Law of Effect"*** summarized in the effects of reinforcing event on the tendency to select actions.

Law of effects gather two crucial aspects.

Selectional: trying alternatives and selecting among them by comparing their consequences.

Associative: the alternatives found are associated with particular situations.

Natural evolution is selectional but not associative. Supervised learning is associative but not selectional. Another way to say it would be: The Law of Effect is an elementary way of **combining search and memory**.

1.1.2. Optimal control

This term, that came in in 1950s, is very important to explain as it stablishes fundamental basis on RL. However, as it was understood at the beginning, there was no learning on it. It describes the problem of designing a controller to minimize a measure of a dynamical system's behavior over time.

One approach was developed by Richard Bellman in 1950s. This approach uses the concepts of a dynamical system's state and a value function, to define a functional equation called **Bellman equation**. The class of methods for solving optimal control problem by using this equation came to be known as **dynamic programming** (DP).

*Bellman introduced the discrete stochastic version of the optimal control problem knowns as **Markovian Decision Processes** (MDP), and Ron Howard (1960) devised the **Policy Iteration Method** from MDPs.*

DP is considered the only feasible way of solving MDPs. It suffers from what Bellman called ***"the curse of dimensionality"*** – the computational requirements grow exponentially with the number of state variables.

*DP has been extensively developed since then, including extensions to **Partially Observable MPDs** (POMPDs) – Lovejoy, **Approximation Methods** – Rust, and **Asynchronous Methods** – Whittle.*

1.1.3. Temporal Difference

These methods are distinctive in being driven by the difference between temporally successive estimates of the same quantity.

Claude Shannon's in the 1950 suggested that a computer could be programmed to use an evaluation function to play chess, and that it might be able to improve its play by modifying this function on-line.

Sutton and Barto extended Klopf's ideas on animal learning, describing learning rules driven by changes in temporally successive predictions, and developed a psychological model of **classical conditioning based on temporal-difference learning**. The method developed by them for using TD in trial-and-error learning is known as **actor-critic architecture**.

Ian Witten in 1977 published the first paper regarding temporal-difference on a method that now is called TD(0). A key step made by Sutton separating TD learning from control and treating it as a general prediction method lead to the introduction of TD(λ) algorithm.

Finally, the TD and optimal control threads were fully merged in 1989 with Chris Watkins's development of **Q-Learning.**

2. Introduction

RL is learning what to do by mapping situations (or sates) to actions so as to maximize a numerical reward signal. The agent must discover what actions yield the most reword by trying them.

The agent must be able to sense, to some extent and some accuracy, the state of the environment, and must be able to take actions that affect that state.

One of the challenges of RL is the trade-off between exploration and exploitation. The agent must try a variety of actions and *progressively* favor those that appear to be the best.

2.1. Elements of RL

Beyond the agent and the environment, there are 4 main sub elements.

2.1.1. Policy

The policy defines the learning agent's way of behaving at a given state. It essentially **maps perceived states to actions** to be taken. **Stimulus-Response Rules.**

How does a policy look like? They could be simply tables, up to very complicated search algorithms.

2.1.2. Reward function

The reward function defines the goal in a RL problem. It essentially **maps perceived states to the reward**, a single number. This reward indicates the desirability of that state. The reward function defines what are good and bad events for the agent.

2.1.3. Value function

However, RL sole objective is to maximize the reward in the long run. Reward function only takes care of what is happening in an immediate sense.

The value of a state is how much reward an agent can expect to accumulate from that state on the future. Note that without rewards there are no values, and the only purpose of estimating values is to achieve more reward.

Unfortunately, while rewards are basically given by the environment, values need to be estimated based on sequences of observations by the agent.

Methods for efficiently estimate values is the most important component of RL.

2.1.4. Model

The model of the environment simply mimics the behavior of it. **Maps states and actions into a new state**.

2.1.5. Task

The task is an instance of the RL problem. It is what it needs to be done so the RL problem can be considered achieved. There are two types of tasks in RL:

- **Episodic:** tasks that have a well-defined starting and ending point. We refere to a complete sequence of interaction as an **episode**. Therefore, these tasks have a **terminal state**.
- **Continuing:** tasks that continuo forever without end.

2.2. Example on Tic-Tac-Toe

RL approach to play Tic-Tac-Toe.

We first set up a table (the Value function) of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability to win from that state, which is the *state's value*.

We change the values of the states in which we are while we are playing.

Our goal is to make them more accurate estimates of the probabilities of winning. To do it, we back up the value of the state after each *"greedy"* move to the state before the move.

In the Figure 1 the arrows will represent this back up action. Once we know how good/bad our current state is, we update how good was the previous one, before taking the last action.

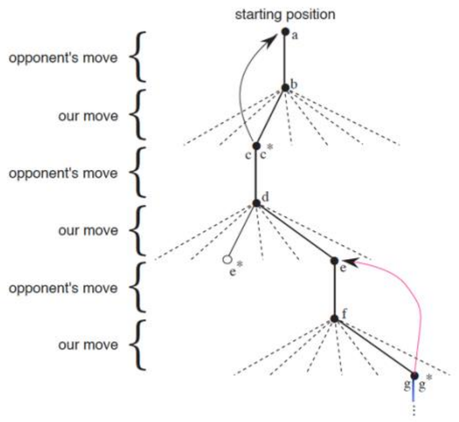


Figure 1.Tic-Tac-Toe Game Tree

Mathematically, it can be represented as in Eq 1:

$$V(s) \leftarrow V(s) + \alpha[V(s') - V(s)] \qquad \text{Eq 1}$$

In Eq 1, we try to adjust the current state (s) value to be close to the value of the next state (s') after applying an action, by moving its value $V(s)$a fraction (α) towards the new value $V(s')$. This fraction is called the *step-size parameter* and influences the rate of learning. This update rule is an example of **Temporal Difference**.

The key point of the method is that, after enough iterations and with a suitable value of α that will be decreased with time, it will **converge to an optimal policy** for playing the game.

Gerry Tesauro combined this algorithm with an artificial neural network to learn to play backgammon, which has an outrageously larger number of possible states than tic-tac-toe. The neural network provides the algorithm the ability to generalize from its experience, so that in new states it selects moves based on information saved from similar states faced in the past.

How well a reinforcement learning agent system can work (with such large state sets) is intimately tied to how appropriately it can generalize from past experience.

3. Exploration and Exploitation

The need to balance exploration and exploitation is a distinctive challenge in RL. This chapter gathers simple balancing methods for the n-armed bandit problem.

3.1. Action-Value Methods

3.1.1. Sample-Average Method

If by time step t, the action a has been chosen $K_a$times prior to t, the its value is estimated to be:

$$Q_t(a) = \frac{R_1 + \cdots + R_{K_a}}{K_a} \tag*{Eq 2}$$

Being $q_*(a)$ the true value of action a (the mean reward received when that action is selected), by the law of large numbers, $Q_t(a) \to q_*(a)$ when $K_a \to \infty$. The method is named sample-average then after being each estimate an average of the sample of relevant rewards.

The simples action selection rule is to select the one with the highest estimated action value.

$$Q_t(A_t^*) = \max_a Q_t(a) \tag*{Eq 3}$$

3.1.2. ε-greedy Methods

Previous method always exploits current knowledge to maximize immediate reward but spends no time in sampling other possibilities.

An alternative to this is to do exploration every time but, once in a while with a small probability ε, give every action the same probability to be pick regardless the value estimates. These methods ensure that after infinite replies, all $Q_t(a) \to q_*(a)$.

3.2. Softmax Action Selection

The difference is that ε-greedy methods when exploring gave equal probability to each action. In task were worst actions are very bad, this technique is unsatisfactory. Using the softmax function of Eq 4 we could give probabilities to the actions in accordance to their estimated value.

$$P(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{i=1}^{n} e^{Q_t(i)/\tau}} \tag*{Eq 4}$$

τ is a parameter called *temperature*. High temperatures cause the action to be all nearly equiprobable. Low temperatures cause a greater difference, that depend in their estimated values.

3.3. Incremental Implementation

The obvious implementation for Eq 2 is to keep track of the reward obtained every time for action a. This is very inefficient in terms of memory and computational requirements, as it grows exponentially with the size of the state's space.

However, we can make use of an incremental update to compute averages to solve this issue as in Eq 5:

$$Q_{k+1} = \frac{1}{k}\sum_{i=1}^{k} R_i = Q_k + \frac{1}{k}\sum_{i=1}^{k}[R_k - Q_k]$$
$$NewEstimate = \ OldEstimate + StepSize\ [Target - OldEstimate] \qquad \text{Eq 5}$$

4. The RL Problem

This problem defines the field of RL: any method that is suited to solving this problem is considered to be a RL method.

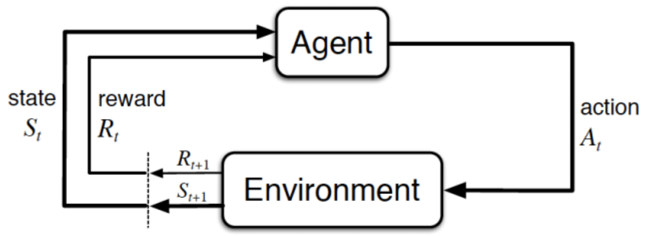


Figure 2. Agent-Environment interaction in RL

We need to describe the RL problem in the broadest possible way. The both learner and the decision maker is called the *agent.* The thing it interacts with, comprising everything outside the agent, is called the *environment.* These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. The environment also give rise to *rewards*, special numerical values that the agent tries to maximize.

At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy. It is denoted with π. Then, $\pi(a|s)$ is the probability of action at time t and state s being a: $A_t = a$ if $A_t = a$.

RL methods specify how an agent changes its policy based on experience.

4.1. The Markov Property

State representations can be highly processed versions of original sensations, or they can be complex structures built up over time from the sequence of sensations. Furthermore, there will be hidden state information in the environment that could be useful for the agent if it knew it.

In many RL cases, we don't look at problems where we need previous information to make a decision. Instead, we make our decision only based on our current state. A state signal that succeeds in retaining all relevant information is said to be Markov, or to have the Markov property.

A good example is to look at the current state of a board game. We don't need the previous information to take a decision to maximize the reward, we could use all the information contained at that particular screenshot of the game.

Mathematically, the Markov property can be defined as:

$$\boldsymbol{P}\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \ldots, R_t, S_t, A_t\} \rightarrow \boldsymbol{P}\{R_{t+1} = r, S_{t+1} = s' | R_t, S_t\} \quad \text{Eq 6}$$

If an environment has Markov property, by iterating the equation (right hand) it is possible to predict all future states and expected rewards from knowledge only of the current state.

4.2. Markov Decision Processes (MDP)

A RL learning task that satisfies the Markov property. If the state and action spaces are finite, the process is called finite MDP, and they stablish the basis to understand 90% of modern RL.

A particular MDP is defined by its state, action and the dynamics of the environment. Given those, the probability of each possible next steps is defined in terms of it **transition probabilities**:

$$p(s' \mid s, a) = \boldsymbol{P}\{S_{t+1} = s' \mid S_t = s,\ A_t = a\} \tag{Eq 7}$$

Similarly, the expected value of the next reward can be defined as:

$$r(s, a, s') = \mathbb{E}[R_{t+1} \mid S_t = s,\ A_t = a,\ S_{t+1} = s'] \tag{Eq 8}$$

4.3. Value Functions (state-value and action-value)

Almost all RL algorithms involve estimating value functions, which means estimate **how good is an action in terms of expected return**. Since the rewards depend on what actions will be taken, **value functions are defined with respect to particular policies.**

Being the value of a state, the expected return following π for an MPD is defined by the ***state-value function for policy π***

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+q} \mid S_t = s\right] \tag{Eq 9}$$

Similarly, we can call the ***action-value function for policy π***, as the expected return starting from s, taking the action a, and thereafter following policy π; such as:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s,\ A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s,\ A_t = a\right] \tag{Eq 10}$$

These two values are estimated from experience.

If an agent follows policy π and maintains an average for every state encounter, the average of its returns will converge to $v_\pi(s)$ as the number of time steps approaches infinity. If separate averages are track for each action taken in a state, these averages will converge to $q_\pi(s, a)$.

*We call estimation these methods as **Monte Carlo** methods because they involve averaging over many random samples of actual returns.*

VISUALIZING STATE-VALUE FUNCTIONS

To give sense to all these concepts, let's put these terms into a visual example. Let's say we have an agent at the top left corner of a 3x3 gridworld, and its goal is to reach the bottom right corner, as represented in . . Each step made is a reward of -1, expect if the agent bumps into a mountain that the reward is -3, because a mountain is an obstacle that will make the process of reaching the goal to take longer. The reward for reaching the goal is +5.

To calculate the state-value function for the first state, we need first to have a given policy. Let's forget about math here, we just care the concepts. So, let's assume a policy that gives the behavior shown by the arrows in Figure 4.

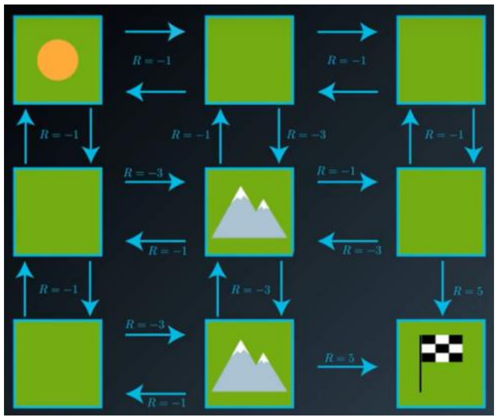


Figure 3. Grid world example [1]

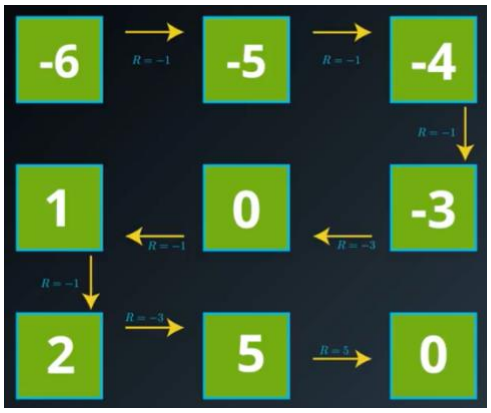
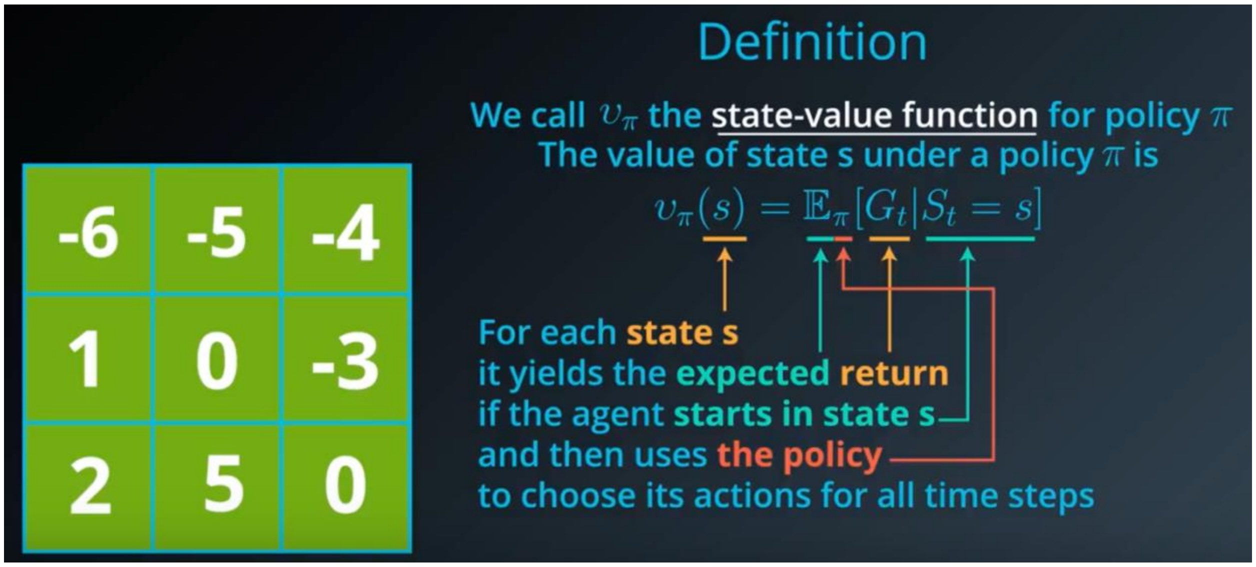


Figure 4. Calculate state-value of a state [1]

We need to calculate the accumulative reward from being in that state and follow the policy until the goal state, where the episode ends.

We could calculate the value of -6 for the first given state. Also, we could calculate the same way we did, for each of the states that make up the path to the goal state.

We finally come up with the given definition in the previous correspondent paragraph, visualized in Figure 5.



Definition

We call v_π the state-value function for policy π

The value of state s under a policy π is

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

For each state s
it yields the expected return
if the agent starts in state s
and then uses the policy
to choose its actions for all time steps

Figure 5.Visual State-value function definition [1]

VISUALIZING THE BELLMAN EQUATION

We can easily realize that we have done a lot of redundant computations to calculate the state value function of all the states that made up the path for that particular given policy. We could simplify the computation by taking advantage of the recursivity of the calculations made. To do this, as shown in Figure 6, we could calculate the value of the previous state, by using the value of the one-step-ahead state and the imitate reward to reach that state[1].

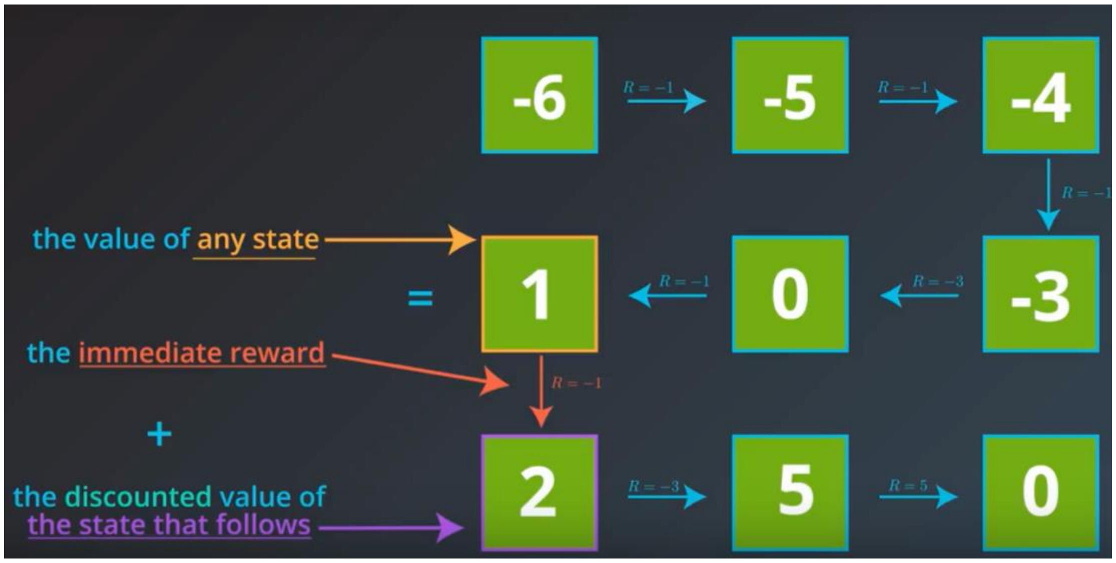
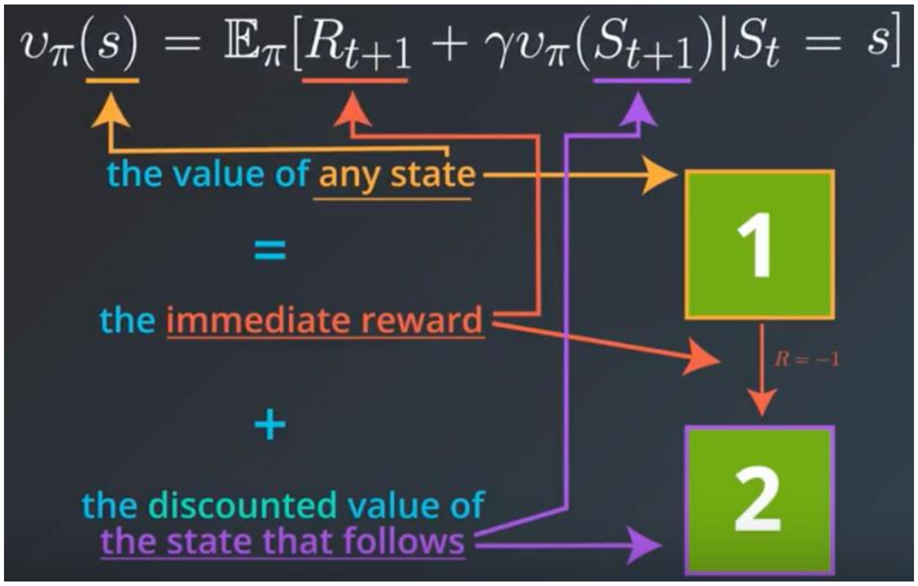


Figure 6. Bellman visualization – recursivity [1]

We could rearrange the terms of the calculations to come up with a simplified notation of what just happened as we do in Figure 7, to reach what is known as the **Bellman Expectation Equation**, broadly used in RL and adopted in the common notation on the RL community, :



$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

Figure 7. Bellman visualization – formulation [1]

[1] Note that the discount factor value on the given example has been set to 1 for simplicity

This is because, in general, with more complicated worlds, the immediate reward and next states cannot be known with certainty. We could define then the **Bellman Expectation Equation**:

The value of any state is the expected value of it if the agent stats in state s and the uses the policy Pi to choose its actions for all future time steps (or states) given the immediate reward toward the one-step-ahead state and the discounted value of the states that follow.

See Figure 8.

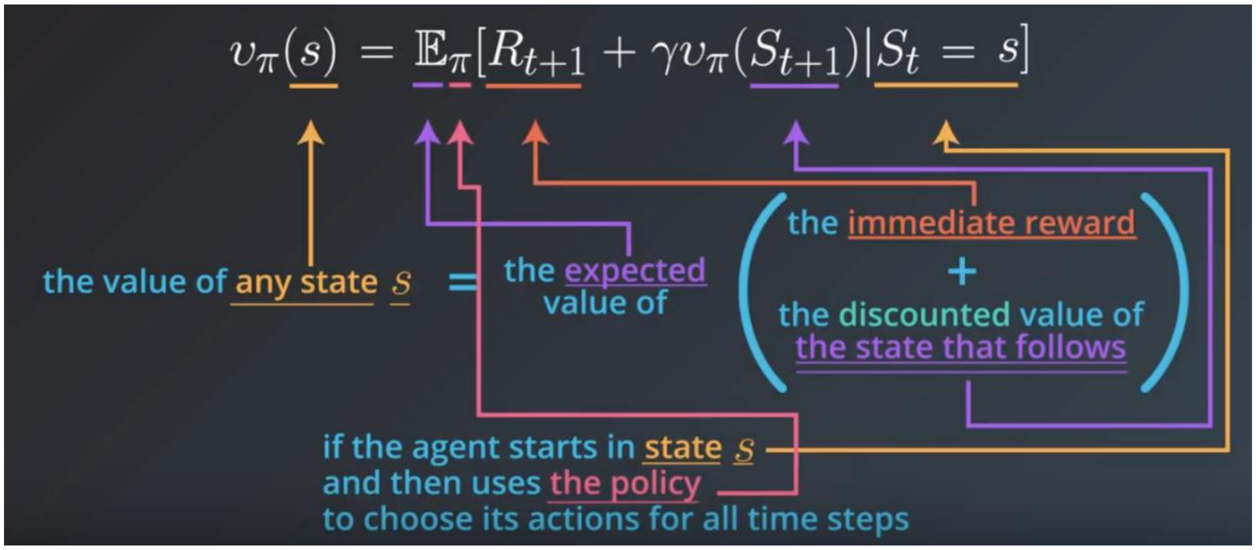

$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$
the value of any state s =
the expected value of
the immediate reward
+
the discounted value of the state that follows
if the agent starts in state s
and then uses the policy
to choose its actions for all time steps

Figure 8. Bellman visualization - definition [1]

VISUALIZING ACTION-VALUE FUNCTION

We have seen previously the other possibility to represent the expected return for a given state given a policy. In the case of the action-value action, we append the action that is chosen between the possible actions at the state to which the function is being applied. Therefore, as shown in FIG, the could summarize the function as the value of taking the action a in state s and then continue under policy π.

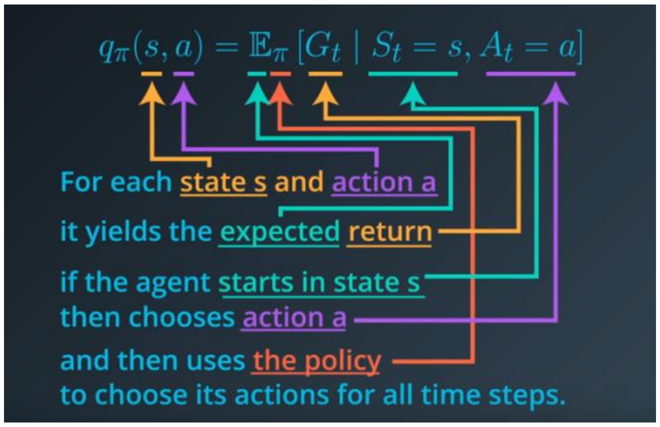

$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$
For each state s and action a
it yields the expected return
if the agent starts in state s
then chooses action a
and then uses the policy
to choose its actions for all time steps.

Figure 9. Action-value function [1]

4.3.1. Why value functions? – Recursion

A fundamental property of value functions used in RL and DP is that **they allow to recursively express the state-value** of a state and its possible successor through the **Bellman Equation for $v_\pi(s)$**.

$v_\pi(s) = \sum_a \pi(a\|s) \sum_{s'} p(s' \| s, a)[r(s, a, s') + \gamma v_\pi(s')]$	Eq 11

In Figure 10 we could see how for (a) a given state there is a set of options to pick. Bellman equation averages over all the possibilities, weighted by their probability of occurring and $v_\pi(s)$ is the unique solution to it.

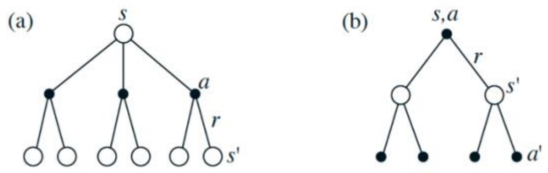


Figure 10.Backup diagrams for (a) $v_\pi(s)$ and (b) $q_\pi(s,a)$.

The backup operations transfer value information from successors back to a state (or state-action pair).

4.4. Optimal Value Functions

For finite MDPs, we can define an **optimal policy π_*** as the policy which maximizes the value function over all the states. One policy is better than another one, if every state has a higher value function, as shown in Figure 11 for the gridworld example.

$v_\pi(s) \geq v'_\pi(s)\ \forall s \in S$	Eq 12

Optimal policies share the shame *optimal state-value function* and *optimal action-value function.*

$v_*(s) = \max_\pi v_\pi(s)\ \forall s \in S$ $q_*(s,a) = \max_\pi q_\pi(s,a)\ \forall s \in S, \forall a \in A(s)$	Eq 13

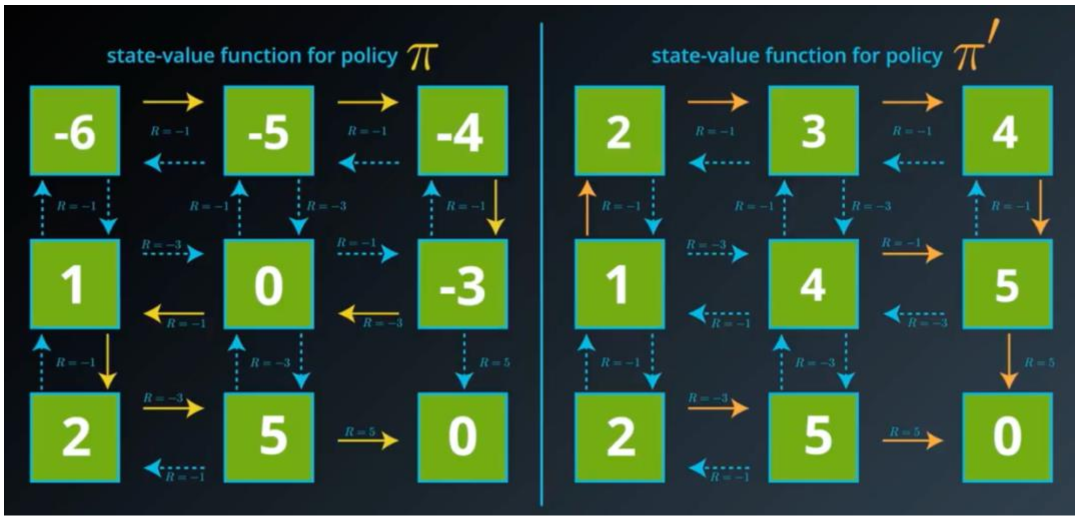

state-value function for policy π
-6
-5
-4
1
0
-3
2
5
0
state-value function for policy π'
2
3
4
1
4
5
2
5
0

Figure 11. Optimal policies – comparison between two policies

Tabular Action-Value Methods

This king of methods as we have already discussed are limited to cases where the state and action spaces are small enough for the approximate action-value function to be represented as an array, or table. Methods for these cases normally find the optimal value function and optimal policy; whereas approximate methods don't find them but can hold way more complicated scenarios.

A brief summary of the methods before entering into details is:

Method	Advantages	Drawbacks
Dynamic Programming (DP)	Well-developed mathematically	Require a complete and accurate model of the environment.
Monte Carlo Methods (MC)	Simple and don't require a model	Not suitable for step-by-step incremental computation
Temporal Difference (TD)	Don't require model and fully incremental	Highly complex to analyze
MC + TD = Eligibility Trace		
DP + MC + TD	Complete unified solution to the tabular RL problem	

The name tabular methods come from the fact that the information of the policies can be (and is) stored in a tabular shape like the one show in Table 1 for the following toy example. Imagine as shown in Figure 12.2x2 grid world toy example, a robot in a 2x2 grid world stating in the bottom-left corner which wants to go to the bottom-right corner and there is a wall between them two.

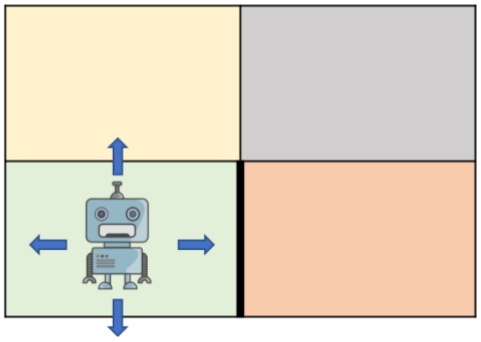


Figure 12.2x2 grid world toy example

If we don't count the terminal state, the number of possible states to visit by the agent in the world is 3. Also, the total possible actions for the agent at each state is 4, shown by the arrows. This will lead to the Table 1, where we could counter the cumulative reward for that state after taking that action, and the use that information to calculate the corresponding state-value or action-value functions.

Table 1. Tabular methods representation

		Actions			
States		a1	a2	a3	a4
	s1	+7	+6	+5	+6
	s2	+8	+7	+8	+9
	s3	+10	+9	+9	+9

5. The prediction problem

We could divide the behavior of these methods as, differentiating between **the prediction problem and the control problem.**

It is essential to have in mind the difference implications of both problems. RL can be summarized is combining iteratively both approach until finding the best policy, the best way of behaving at every possible incoming situation.

*The prediction will simply tell us what the value function expectation at a given state is if we follow a certain policy. We will call this as the **policy evaluation**. On the other hand, the control actually makes use of the policy evaluation to update the policy while searching for the optimal one, what we call the **policy improvement**.*

Note 2

Dynamic programming won't solve the RL problem!

They are a good fundamental introduction to how we estimate value functions and policies and how we can update the policy searching for the best one; BUT, **they are meant to solve the planning problem**.

This means that they are **actually given the MDP**, and they just perform different algorithms (covered in 5.1) to find the optimal way of behaving and the get best results for that MDP.

On the other hand, MC and TD (covered in 5.2 and 5.3 respectively) methods are gather under the name of **model free methods**, since they do not have prior information about the environment and the agent must go there and find out by itself through iteration.

Model-free methods try to figure out the value function of an unknown MDP given a policy.

5.1. Dynamic Programming

The term DP refers to:

Collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov Decision Process

Background:

DP algorithms are not useful to solve any RL problem. However, they set the theoretical base to everything else. In fact, we will see how **every other model are attempts to improve DP in terms of less computation and without assuming a perfect model** of the environment.

Goal:

Find how DP can be used to compute the value functions to organize and structure the search for good policies.

Methodology:

We can obtain optimal policies once we have found the optimal value and action functions which satisfies the Bellman optimality equations.

We will turn Bellman equations EQX EQY into assignments, that is, into update rules for improving approximations of the desired value functions.

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s,\ A_t = a] \\ &= \max_a \sum_{s'} p(s' \mid s, a)[r(s,a,s') + \gamma v_*(s')\,]\,; \quad \forall s \in S \end{aligned}$$

Eq 14

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\mid S_t = s,\ A_t = a\right] \\ &= \sum_{s'} p(s' \mid s, a)\left[r(s,a,s') + \gamma \max_{a'} q_*(s', a')\;\right]; \ \forall s \in S, \forall a \\ &\in A(s) \end{aligned}$$

Eq 15

5.1.1. Policy Evaluation

The policy evaluation attempts to compute the state-value function for an arbitrary policy.
Problem: *evaluate a given policy*
Solution: *iterative application of Bellman expectation backup*

Being the state-value function defined in Eq11, if the environment's dynamics are completely known we will have S linear equations with S unknowns. This equation is solvable, but we prefer iterative solutions to it, making use of Eq 11 as an update rule.

Iterative Policy Evaluation

Given the sequence $v_0,\ v_1, \dots$ and setting the initial approximation v_0 arbitrarily (except that the terminal state, if any, must be equal to 0), we use the update rule as:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s' \mid s, a)[r(s,a,s') + \gamma v_k(s')\,] \end{aligned} \qquad \text{Eq 16}$$

The solution to it is clearly that $v_k = v_\pi$ as v_π converges to it when $k \to \infty$.

Note that each v is a vector with the value of every state until the solution given a policy.

Applying the same operations to every state:

*Iterative policy evaluation replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. This operation is called **full backup**.*

Each iteration backs up the value of every state once to produce the new approximate value function v_{k+1}.

Implementation

The natural way to approach this would be to have two arrays, one for $v_k(s)$ and other for $v_{k+1}(s)$. New values can be computed from the old values.

However, we could use one single array and **update the values in place** – each new backed up value overwriting the old one – and use the new values on the right-hand side of the Eq 16. This way in fact converges faster and is the version normally used in the context of DP algorithms.

```
Input $\pi$, the policy to be evaluated
Initialize an array $v(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $temp \leftarrow v(s)$
        $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
        $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $v \approx v_\pi$
```

Figure 13. Iterative Policy Evaluation Pseudocode

Let's see how actually the iterative algorithms evaluates the easiest policy – which is the random policy that assigns the same probability to every action – on a grid world where we go to arrive to one of the shaded corners.

At the beginning every action has the same probability to be picked at every state, since this is how we defined $\pi(a|s)$.

On the next step, the successor of the states close to the corners can tell the previous states that taken that action from their state is good. Therefore, they update the policy for that state to pick that action. The -1.7 (which actually is -1.75) comes from having 3 possible steps where it will get -2, and 1 step (moving left) where it will get just -1.

Iteratively using the same update rule until the optimal policy is found by convergence. We still have to see how the policy is improved at every iteration in next subsection.

The cells with multiple arrows represent states which several actions achieve the maximum in Eq 19.

What we have seen here?

Just by evaluating the value function (left panels) we are able to figure out better policies (panel right). So, any value function can be used to determine a better value function.

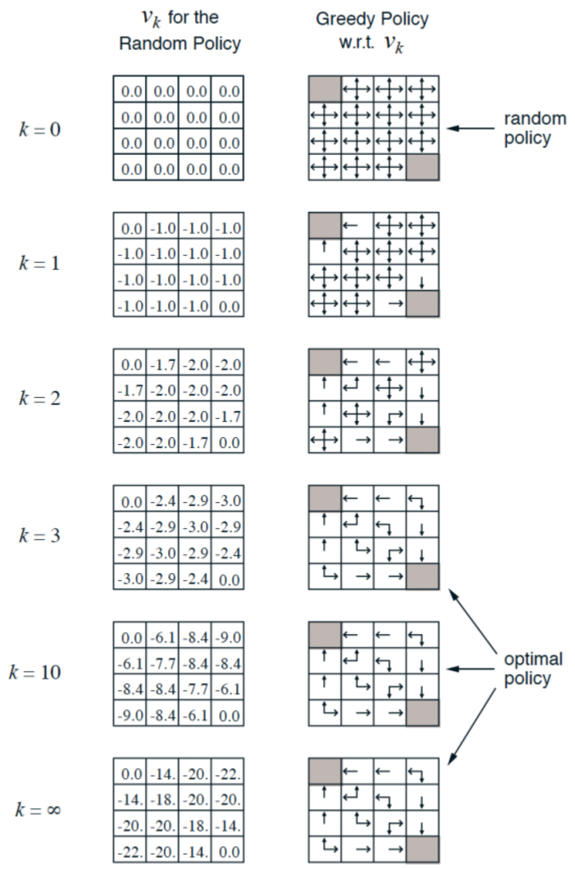


Figure 14. Convergence of iterative policy evaluation

We intuitively know as ourselves, **how do we make our policy better?**

5.1.2. Policy Improvement

Based on what we have seen in Figure 14, we ask ourselves the question:

Would it be better to change the current policy?

What happens if we choose some action $a \neq \pi(s)$?

Well we have seen a way to answer this question, by considering selecting action a in s and thereafter following the existing policy π. The value of this behaving is our Eq 10 – action-value function for policy π.

If it is greater to select a once in s and thereafter follow π than it would be to just follow π all the time, then we would expect to be better to still select a every time s is encountered, and the new policy would in fact be a better one overall.

We can express this conclusion – that receives the name of **policy improvement theorem** – as:

$$if\colon q_\pi\big(s,\pi'(s)\big) > v_\pi(s);\;\; then\; \rightarrow \pi' \geq \pi \;\; or\; v_{\pi'}(s) \geq v_\pi(s) \qquad \text{Eq 17}$$

The idea behind the proof this policy improvement theorem is easy to demonstrate, simply by keep expanding the q_π side and reapplying Eq 16 until we get $v_{\pi'}(s)$.

$$\begin{aligned} v_\pi(s) < q_\pi\big(s,\pi'(s)\big) &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma q_\pi\Big(S_{t+1,}\, \pi'\big(S_{t+1,}\big)\Big) \mid S_t = s\Big] \\ &= \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_t = s]\big] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+4}) \mid S_t = s] \ldots = v_{\pi'} \end{aligned} \qquad \text{Eq 18}$$

Given a policy and its value function, we can evaluate a change in the policy at a single state to a particular action. It is a natural extension to consider changes at all states to all possible actions! To do it, we need to select at each state the action that appears best according to $q_\pi(s,a)$

In other words, we consider the new greedy policy π'given by:

$$\begin{aligned} \pi'(s) &= arg\max_a q_\pi(s,a) = arg\mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s,\, A_t = a] \\ &= arg\max_a \sum\nolimits_{s'} p(s' \mid s,a)[r(s,a,s') + \gamma v_\pi(s')\,] \end{aligned} \qquad \text{Eq 19}$$

The greedy policy takes the action that looks best in the short term according to v_π. We are now in a position to give the full definition for this process.

Policy improvement: *the process of making a new policy that improves on an original policy, by making it greedy w.r.t. the value function of the original policy.*

In summary, we are making a given policy better by:

- Evaluate the policy $v_\pi(s)$
- Improve the policy by acting greedily w.r.t. the value function $\pi' = greedy(v_\pi)$

This process is summarized under the name of a policy iteration process.

5.1.3. Policy Iteration

In an iterative mode, one a policy π has been improved using v_π to yield a better policy π', we can recursively compute $v_{\pi'}$ to yield from it an even better policy π''.

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

Figure 15. Policy Iteration

We can obtain a sequence of monotonically improving policies and value functions like in Figure 15 where E stands for evaluation and I for improvement. And, because of a finite MDP has a finite number of policies, this algorithm converges to an optimal policy in a finite number of iterations.

This way of finding the optimal policy is called **policy iteration**, and the complete algorithm is provided in Figure 16.

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

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $temp \leftarrow v(s)$
           $v(s) \leftarrow \sum_{s'} p(s'|s,\pi(s))\Big[r(s,\pi(s),s') + \gamma v(s')\Big]$
           $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$  (a small positive number)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $temp \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s,a)\Big[r(s,a,s') + \gamma v(s')\Big]$
       If $temp \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $v$ and $\pi$; else go to 2
```

Figure 16. Policy Iteration

We have seen how we act greedy on one step ahead and then check the value functions given the previous policy. The immediate question to it is:

Why not update policy every iteration?

This will be, stop after k=1, update the policy, and repeat. This is equivalent to what is known as Value Iteration.

5.1.4. Value Iteration

Any optimal policy can be subdivided into two components:

- An optimal first action, A*
- Followed by an optimal policy from successor state, s’

The intuition then can be understood as starting with the final reward and work backwards.

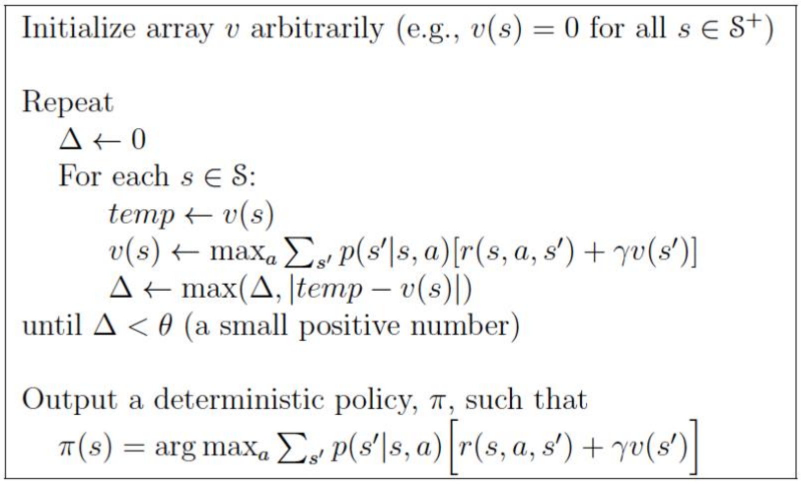
One drawback of policy iteration is that each of its iterations involves policy evaluation. There must be ways to possibly truncate the policy evaluation without losing the convergence guarantees of policy iteration.

Value iteration *consists on truncating the policy iteration by stopping policy evaluation after just one sweep – one backup of each state.*
Problem: *find the optimal policy*
Solution: *iterative application of Bellman expectation backup*

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s,\ A_t = a] \\ &= \max_a \sum_{s'} p(s' \mid s, a)[r(s,a,s') + \gamma v_k(s')\,]\,; \quad \forall s \in S \end{aligned} \qquad \text{Eq 20}$$

To terminate the algorithm, we stop once the value function changes by only small amount in a sweep.

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement.



```
Initialize array $v$ arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
      $temp \leftarrow v(s)$
      $v(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
      $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
  $\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\Big[r(s,a,s') + \gamma v(s')\Big]$
```

Figure 17. Value Iteration

We again have a sequence of value function arrays at each iteration $v_0,\ v_1, \ldots, v_*$ but this time they are not constructed after any policy. They are just a result of intermediate steps of the algorithm, and they could even don’t correspond to any real policy. However, we are only interested in v_*, which indeed correspond to a real, optimal policy.

5.1.5. Generalized Policy Iteration

Summarizing all these methods, we could say that:

Policy iteration consists of two simultaneous interacting processes:

- *One making the value function consistent with the current policy – policy evaluation*
- *One making the policy greedy with respect to the current value function – policy improvement*

We use the term generalized policy iteration (GPI) to refer to the idea of letting policy evaluation and improvement processes interact independent of the granularity and other details of the two processes.

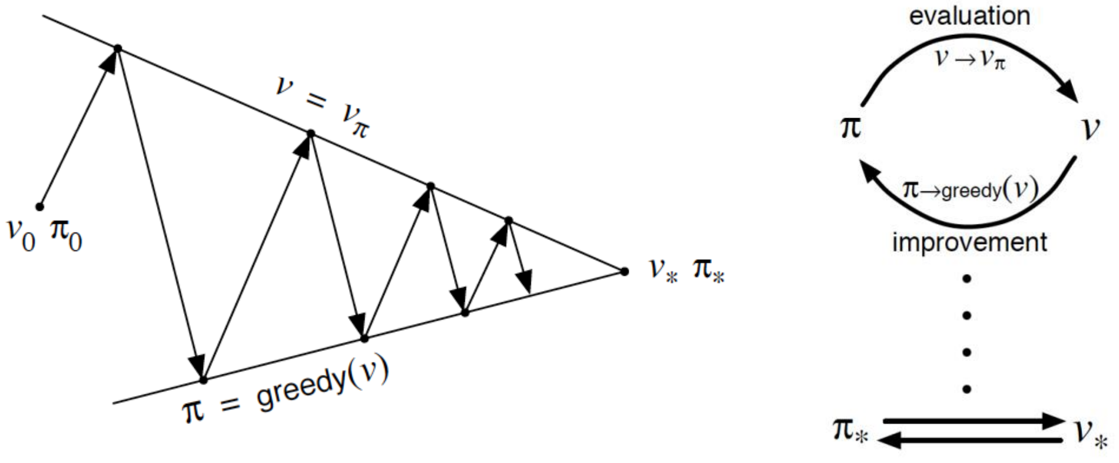


Figure 18. Policy iteration scheme

The idea on the left-hand side of Figure 18 is that we have an initial policy and we calculate the value function to evaluate the initial policy. Once we have it, we act greedy with respect to that policy to give us a new policy. Once we have the new policy, we evaluate the new policy, and so on. This is guaranteed to converge to the optimal value function and therefore to the optimal policy.

5.2. Monte Carlo Methods for Prediction

As we introduce at the beginning on section 5, MC methods are model-free.

How does MC work?

Monte Carlo methods are based on direct experience with the environment. To do so, they run a (big) number of simulations at every state, to determine what is the average output for all the simulations run from that state.

Intuitively, we can think about the first limitation of MC. The episodes **(each simulation) has to end**, so we can have an actual return for that particular simulation. Furthermore, we indeed need to give some information from the model to be able to run simulations. However, **we only need the transition probabilities**, not the entire probability distribution of all possible transitions as for DP.

We could summarize then by saying:

Monte Carlo methods solves the RL problem by averaging sample returns. To ensure well-defined returns, MC methods are only applicable for episodic tasks. It is only upon the completion of an episode that value estimates and policies are changes.

We are also in a good position to see one major drawback of MC methods: we need to wait until the episode ends to be able to make any update – **MC methods don't bootstrap**.

5.2.1. Monte Carlo Policy Evaluation – Prediction

Goal: we want to estimate the value of a state s under the policy π, $v_\pi(s)$.

When we run a simulation from, let's say, the initial state, at some point we could encounter the state s. **Each occurrence of state s in an episode is called a visit to s**. If there are cycles in the states space, it is possible that we encounter the same state more than once in an episode. The way we handle this situation distinguish between **first-visit MC** vs **every-visit MC** methods**. How are they different?**

- First-visit only averages the returns that follow the first visit to that state
- Every-visit averages the returns that follow every visit to that state in a set of episodes

More visually, if we recap Table 1, one particular cell was defined the value of taking the corresponding action (column) at the corresponding state (row). If we apply first-visit, we would include in that cell the reward obtains after starting on this state the first time we encountered it. Whereas if we apply every-visit, we will include in that cell the average return of the rewards obtains starting from that state every time we encountered it. [2]

Does it matter if in our sampling we don't cover all the states?

It does not. We are interested in evaluating a given policy. If we run a big number of simulations, we can be sure that we have visited the states susceptible to being reached following that policy. Therefore, we are considering the states that we are interested on.

[2] Check the example of MC on the Blackjack game here

5.3. Temporal Difference for Prediction

As we introduce at the beginning on section 5, MC methods are model-free. We could divide the behavior of these methods as we did for DP and MC again, differentiating between the prediction and the control.

How does TD work?

TD gathers advantages of previous DP and MC:

- Like MC, TD can learn directly from experience without knowing the model dynamics.
- Like DP, TD updates estimates based in part on other learned estimates – bootstrap.

How is TD better than MC?

Whereas MC need to wait until the end of the episode to determine the value of G_t, TD methods need wait only until the next time step.

TD defines a Target at the next time step and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest method is knowns as TD(0):

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \qquad \text{Eq XX}$$

We could think of the target of a MC as G_t itself, whereas TD build the target as $R_{t+1} + \gamma V(S_{t+1})$. This is represented in Figure 19 (left MC and right TD). In it, at every point in time (state) we made an estimate on how much time we thought it will take us to arrive home from work.

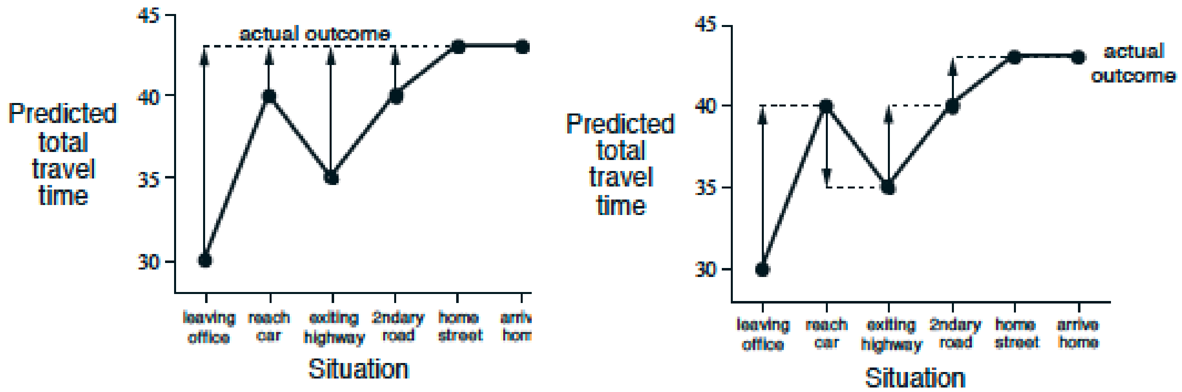


Figure 19. Difference on target definition between MC and TD

In the MC case, the error for every state is the difference between the estimate we did at that state and the real time it took. However, in TD the error for every state is the difference between the estimate at that state and the estimate at the next time step.

Each error is proportional to the change over time of the prediction, that is, to the ***temporal differences*** *in predictions.*

One natural question that comes up now is: if TD does not wait until the final outcome to learn, how does it actually learn to find the optimal value? Does it converge to the optimal policy? Or reframing the question, **does TD and MC converge to the same answer?** Yes, it does.

For any given policy π, TD has been proven to converge to v_π in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation condition.

5.4. Monte Carlo vs Temporal Difference

Monte Carlo

- High variance – Zero bias (even with function approximation)
- Not very sensitive to initial state
- Very simple

Temporal Difference

- Usually more efficient than MC
- TD(0) converges to $v_\pi(s)$ (not always with function approximators)
- More sensitive to the initial state (since we are bootstrapping)

The next question to our previous one is evident: if they converge to the same solution (therefore, they learn the same), **which methods learns faster?**

Actually, no one has been able to prove mathematically that one converges faster than the other. Empirically, however, TD methods usually converge faster than constant-α MC methods. Let's try both on a simple example to explore their behaviors.

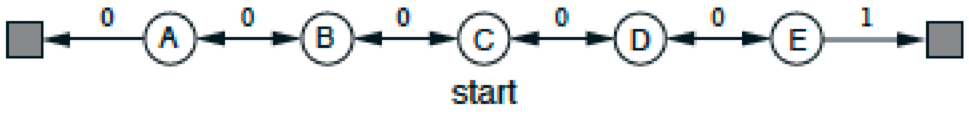


Figure 20. MDP for Random Walks

Figure 20 represents a MP for a random walk, where there is 50% chance of going left or right at every step. The reward of ending right is 1, 0 otherwise. Then, the probabilities from state A to E respectively are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}$. Figure 21 (left) shows the evolution of the value function after 10 and 100 episodes following TD(0). The final estimate is as closed as to the real values. The right picture shows the evolution of the root mean squared error between the true and the learned value function for MC and TD with different step-sizes.

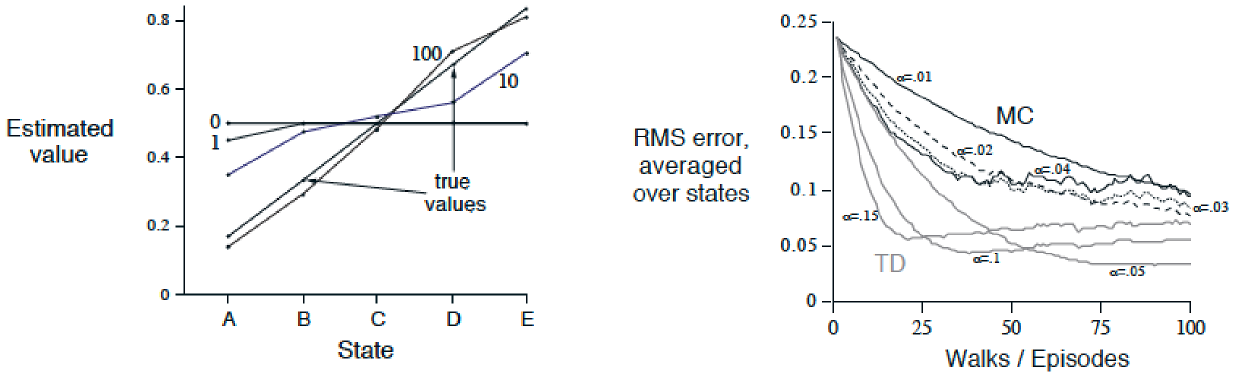


Figure 21. MC vs TD on a Random Walk

We can see how TD method converge faster to the best solution.

5.4.1. Batch learning

To have another point of view, let's suppose that there is a limited data (amount of experience), say 10 episodes. **A common approach with incremental learning methods is to present the previous experience repeatedly until the method converges**.

Given an approximate V, the increments are computed every time step at which a nonterminal state is visited, BUT the value function is changed only once, by the sum of all increments.

Under batch updating, TD and MC converges to different solutions.

E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
A,0, B,0	B,1	B,1	B,1	B,1	B,1	B,1	B,1	B,1	B,0

In this example, we have two states A and B and those are the observations of each episode. **How will MC and TD approach this scenario?**

In the case of MC is trivial. B,1 $V(A) = 0$. $V(B) = \frac{6}{8}$. MC simply averages the rewards for each state by the number of visits to those states. However, TD realizes that 100% of the observed transitions from A went to B. Therefore, it first models the MDP as shown in Figure 22, before computing the estimate, which in this case gives $V(A) = \frac{6}{8}$

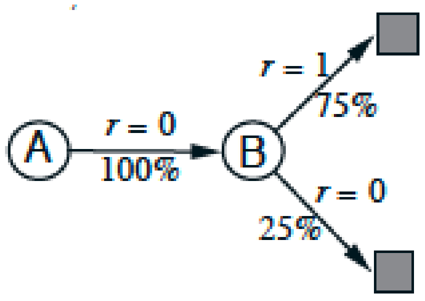


Figure 22. MDP - Batch Updating example

What does this observation mean?

It means that, even the MC answer will lead to the minimal squared error on the training data, if the process is Markov, we expect TD to produce lower errors on future data.

MC converges to solution with minimum mean-squared error to find the best fit to the observed returns. MC does not exploit Markov property and therefore is usually more efficient in non-Markov environments

TD (0) converges to solution of max likelihood Markov model that best fits the data. TD exploits Markov property, being usually more efficient in Markov environments

5.5. Unified view of RL

We have seen so far, the three main approaches on the recent scope of RL. We could put them into a single picture as:

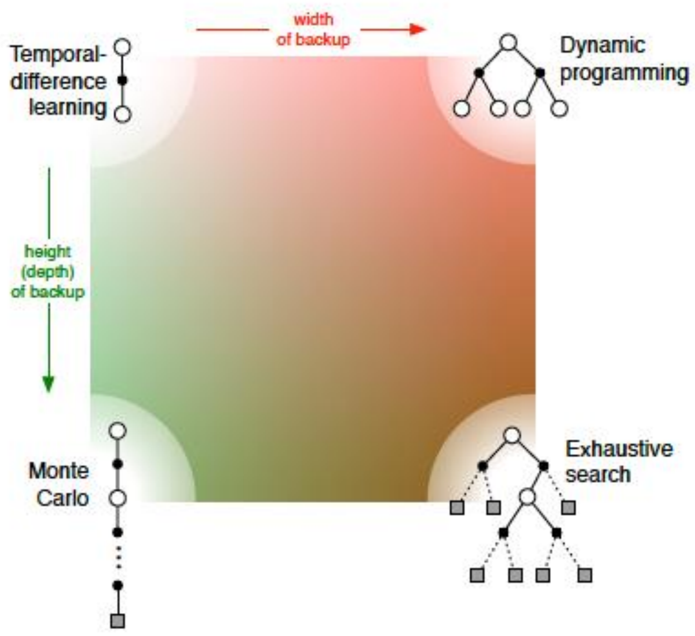


Figure 23. Unified view of Reinforcement Learning

On the right-hand side, we have a dimension of exploration where the space covered is exhaustive. Exhaustive search will simply cover all states space after the optimal solution. DP will look at all the possible outcomes of every possible action after 1-time step. On the left-hand side, we have a dimension that focuses in only one action out of the possible ones. We can summarize saying that the horizontal dimension corresponds to the depth of the backups.

On the top side, we have a dimension that only look at 1-time step ahead, whereas on the down side the methods go as deep as necessary until they reach a terminal state. We can summarize saying that the depth dimension is whether they are sample backups of full backups.

This picture comprises the 4 corner cases on RL. The next step is to combine TD with MC in what it's called after $TD(\lambda)$, where λ determines the part of the spectrum of the vertical axes where we are between MC and TD, represented in Figure 24.

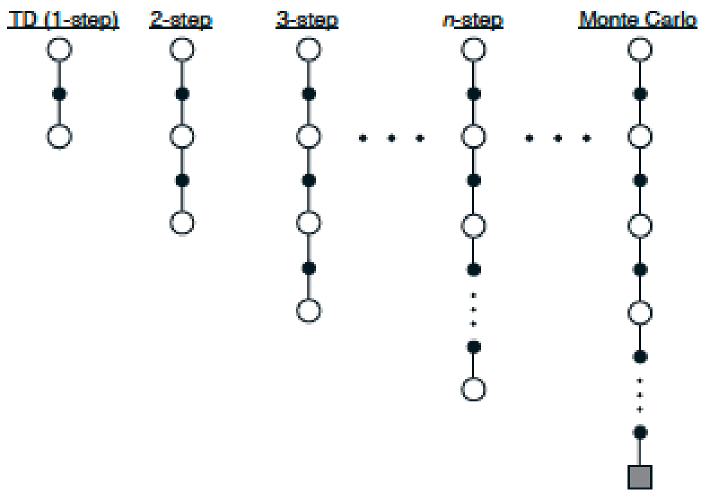


Figure 24. Spectrum of $TD(\lambda)$

The idea is that we could actually visit beyond 1-time step horizon in TD learning and take the information from several states to update the estimate of the current state. To make Figure 24 suitable for averaging and computationally efficient, a geometrical average is performed, by weighting the value of each state as shown in Figure 25 (left). We can decide until which time step ahead (T) truncate the λ and the total sum of the weights will add up to 1.

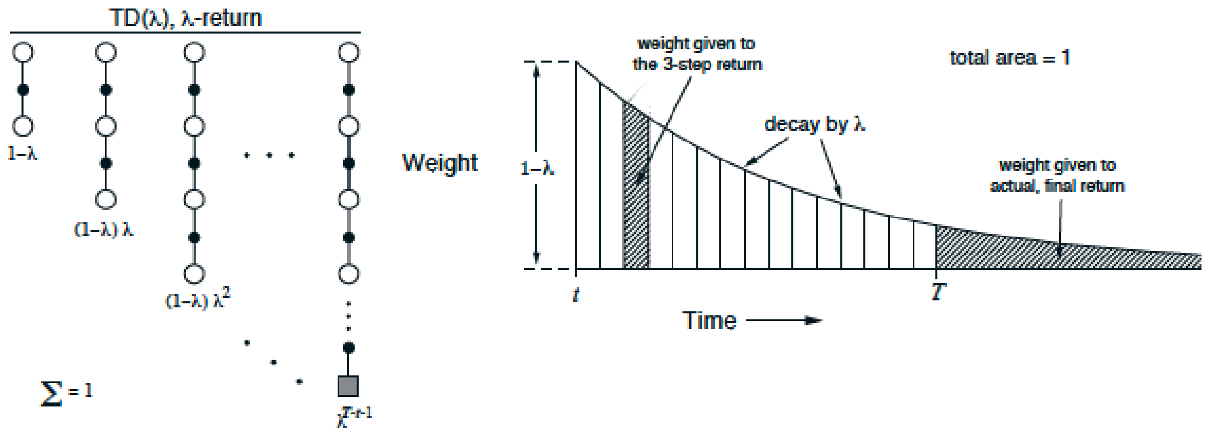


Figure 25. $TD(\lambda)$ geometric averaging

The return at time step t at a given state will be then defined by:

$$G_t{}^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} G_t{}^{(n)} + \lambda^{T-t-1} G_t \qquad \text{Eq XX}$$

What we have covered is what it's called the theoretical or **forward view** of a learning algorithm, where for each state visited we look forward in time to all future rewards and decide how best to combine them.

The λ-return algorithm, is the basis for the forward view of **eligibility traces** as used in $TD(\lambda)$ method.

5.6. Eligibility Traces

5.7. Planning and learning with Tabular Methods

6. The control problem

The goal of the control problem is actually since we imagine RL before knowing anything about it. How can I just put an agent, robot… in an environment, so it starts interacting with it and learning from the experience to update its own policy and behave in a way to obtain the maximum possible return (reward).

What is again, the difference with the prediction problem we saw in last section for model-free methods?

*The prediction problem **estimates** the value function of an unknown MDP*

*The control problem **optimizes** the value function of an unknown MDP*

The points covered on the **model-free control problem** will follow:

- On-Policy MC
- On-Policy TD
- Off-Policy Learning

So then, **what is the difference between on-policy and off-policy learning?**

On-policy learns on the job:

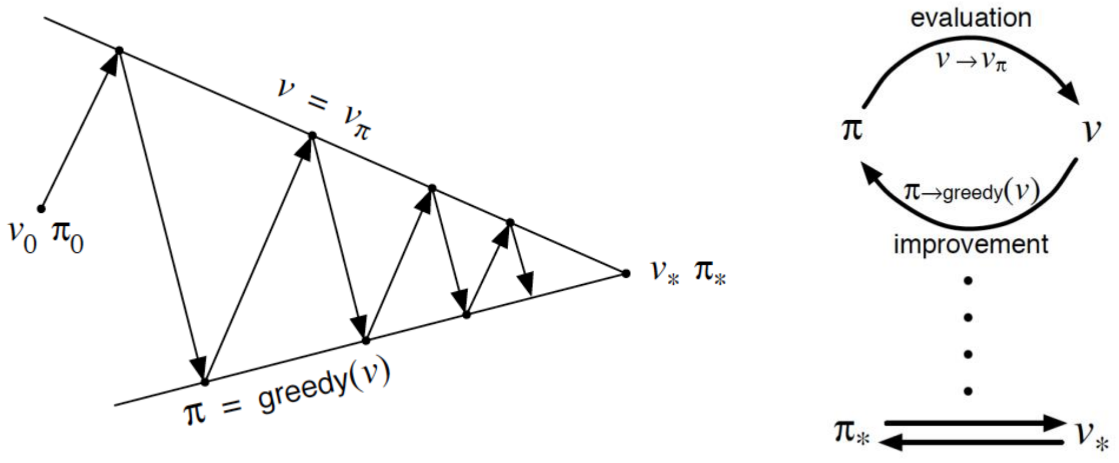
Learn about policy π, from experience sampled from that π.

Off-policy learns by looking over someone's shoulder:

Learn about policy π, from experience sampled from μ

For any method, Monte Carlo or Temporal Difference, we will approach the control problem followed the theory introduced in Dynamic Programming on Generalize Policy Iteration (GPI) and Figure 18.

Here is just a reminder to have the picture clear in our minds from here on.



6.1. Monte Carlo Methods for Control

Can we simply plug in MC in the GPI framework?

The policy evaluation is feasible, since MC will simply run several simulations and average the returns to determine which is the state value function for that particular state.

However, GPI don't hold MC greedy policy over $V(s)$, since it would require a model of MDP.

$$\pi'(s) \ = arg \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \mathcal{R}_{ss'}^a V(s') \qquad \text{Eq XX}$$

If you only have the state-value function and you want to act greedy you need to know the MDP. You only know the value of the states; therefore, you will have to 'imagine' by unrolling one step ahead and see what the value of the states are. But we don't know those dynamics, we are trying to be model free.

The alternative is to use action-value functions covered in section 3.1, Q, that allows us to behave model free. If we have the Qs, and we want to do greedy policy improvement, we just need to maximize over the q values.

The Q values are telling us, from a state s, how good is to perform each of the possible actions

So, we just pick the action that maximizes our Q values:

$$\pi'(s) \ = arg \max_{a \in \mathcal{A}} Q(s,a) \qquad \text{Eq XX}$$

We have them reframed out GPI to work for Action-Value Functions:

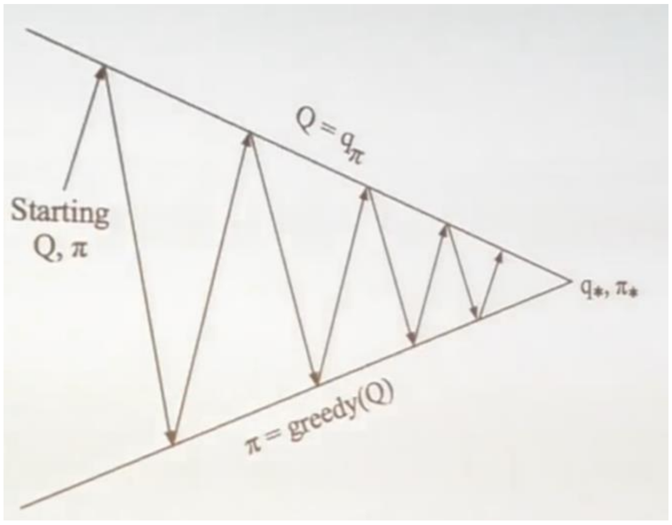


Figure 26. GPI with Action-Value Functions

We are just substituting the state value evaluation for the new pair of state-action value evaluation to determine how good is our current policy.

There is something actually not good in the picture. In DP – where we used this picture initially – we were covering every possible state in every sweep. However, Monte Carlo only covers those states that are reachable by the given policy. Therefore,

---

*We can't improve the policy greedily in model-free since we can't know if the unexplored states are better that the ones covered by the policy being evaluated – **exploration problem**.*

To overcome this issue, as we saw in section 3.1.2 we act ε-greedily, given the chance to explore some states regardless the actions to make to get there are not the best at first sight.

Acting ε-greedily is demonstrated that the new policy will give at least the same state-value function for any particular state.

$$\begin{aligned} q_\pi(s,\pi'(s)) &= \sum_{a\in\mathcal{A}} \pi'(a|s)\cdot q_\pi(s,a) = \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\varepsilon) q_\pi(s,a) \\ &\geq \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\varepsilon)\sum_{a\in\mathcal{A}} \frac{\pi(a|s)-\frac{\varepsilon}{m}}{1-\varepsilon} q_\pi(s,a) \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)\cdot q_\pi(s,a) = V_\pi(s) \end{aligned}$$

Eq XX

Therefore, we can be sure that we are improving (or keeping as good as it was) our policy.

$$V_{\pi'}(s) \geq V_\pi(s)$$

Eq XX

References

[1] Udacidy, "Reinforcement Learning Nanodegree," [Online]. Available: https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893.