Sample-based Learning Methods

Miguel A. Saavedra-Ruiz

February 2020

1 Monte Carlo methods for prediction and control

What is Monte Carlo?

The term **Monte Carlo** is often used more broadly for any **estimation method** that relies on repeated **random sampling**. In RL Monte Carlo methods allow us to **estimate values** directly from experience, from sequences of states, actions and rewards. Learning from experience is striking because the agent can accurately **estimate a value function without prior knowledge of the environment dynamics**.

To use a pure Dynamic Programming approach, the agent needs to know the environments **transition probabilities**.

- In some problems we do not know the environment transition probabilities
- The computation can be error-prone and tedious

For example, imagine rolling a dice. With the help of Monte Carlo Methods, we can estimate values by averaging over a large number of random samples Fig. 1.

Sum of Faces	Probability	Samples
12	?	39 38 49 36 33 47 37 49 41 40 44 44
13	?	35 43 47 28 46 52 47 45 43 43 42 43
14	?	40 32 29 48 43 49 48 39 42 35 44 48
...		36 32 41 49 34 54 37 42 37 38 42 50
71	?	39 37 45 49 44 34 38 50 35 36 42 45
72	?	Average = 41.57

Figure 1: Averaging a dice with Monte Carlo

In reinforcement learning we want to learn a value function. **Value functions represent expected returns**. So a Monte Carlo method for learning a value function would first **observe multiple returns from the same state**. **Then, it average those observed returns to estimate the expected return from that state**. As the number of samples increases, the **average** tends to get closer and **closer** to the **expected return**. These returns can only be observed at the end of an episode. So we will focus on Monte Carlo methods for **episodic tasks**.

Recall that $v_\pi(s) \doteq \mathbb{E}_\pi [G_t | S_t = s]$ so in a Monte Carlo method obtaining the value function would look like Fig. 2.

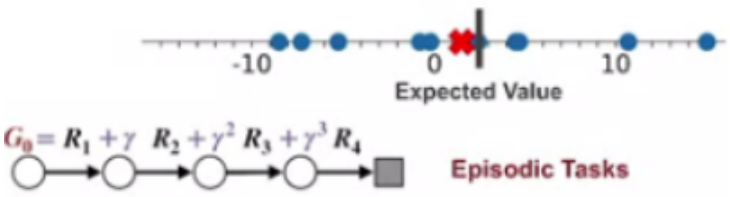


Figure 2: Expected value with Monte Carlo

Monte Carlo methods consider policies instead of arms. The value state S under a given policy is **estimated using the average return** sampled by following that policy from S to termination.

The pseudo-code of the Monte Carlo prediction can be seen below in Fig. 3

MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Append G to $Returns(S_t)$
        $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 3: Expected value with Monte Carlo

Suppose the discount factor γ is 0.5 and imagine an episode ending at time step five where the reward sequences is 3, 4, 7, 1, and 2. Let's find each return G_0 to G_5, starting by writing down the equation for each return. Notice that each return is included in the equation for the previous time steps return. That means we can avoid duplicating computations by starting at G_5 and working our way backwards. Let's do that now. The episode ends at t equal to five so $G_5 = 0$ by definition Fig. 4.

Recall that $G_t = R_{t+1} + \gamma G_{t+1}$

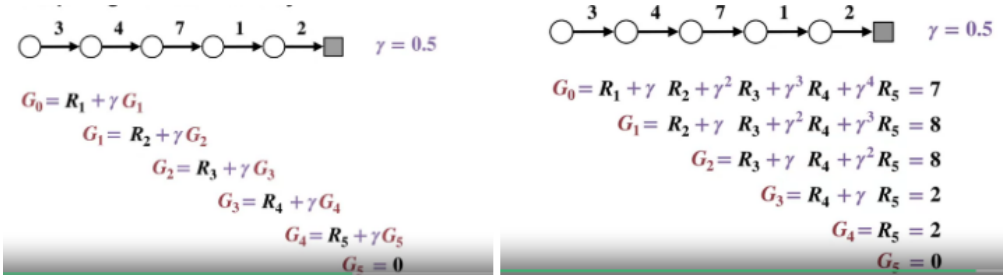


Figure 4: Computing returns efficiently

Using Monte Carlo for Prediction

Let's discuss the specifics of the Monte Carlo prediction algorithm using an example with a card game, blackjack Fig 5.

Figure 5: Blackjack example

Problem Formulation

- Undiscounted MDP where each game of blackjack corresponds to an episode
- -1 for a loss, 0 for a draw, and 1 for a win
- Hit or Stick
- States (200 in total):
  - Whether the player has a usable ace (Tes or No)
  - The sum of the player's cards (12 - 21)
  - The card the dealer shows (Ace - 10)
- Cards are dealt from a deck with replacement
- Policy: Stops requesting cards when the player's sum is 20 or 21

Now, let's look at the states starting from the end of the episode and working backwards. In the last non-terminal state, the card sum was 20 with no usable ace and the dealer had a visible 10. Let's call this state A. We add plus one to the list of returns for state A and set the value of state A equal to the average of the list. Stepping back to the second last state, state B, the agent shows 13 with no usable ace and the dealer has a visible 10. Again, we add plus one to the list of returns now for state B and set the value of state B equal to the average of the list. We've now processed every state in the first episode and completed the Monte Carlo updates for that first episode Fig 6.

The three axes of the plot Fig. 7 are the card the dealer's showing, the agents sum, and the value of that state. The plot with usable aces is much more rough than the plot with no usable ace. That's because most blackjack games do not have a usable ace so the usable ace values are estimated with far fewer samples. Both plots have a similar shape. Looking at the values going across the plot, it looks like the card the dealer's showing doesn't impact the value function very much. If we look at the agent's sum however, the value function is much higher in states where the agent has a 20 or 21.

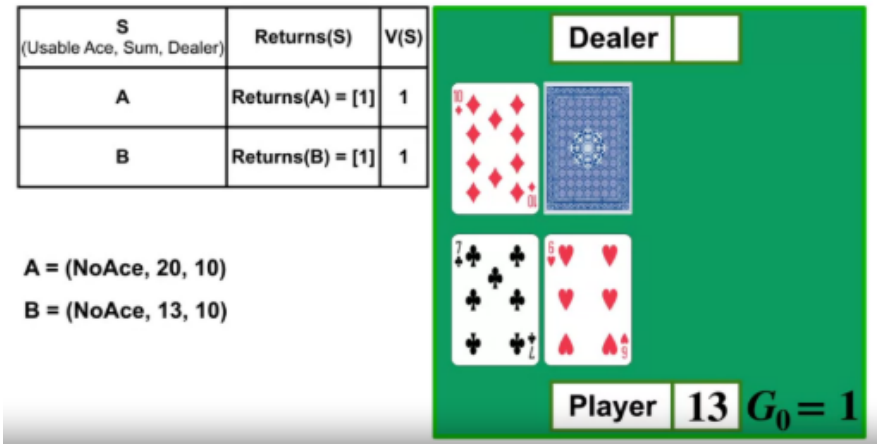


Figure 6: Blackjack episode example

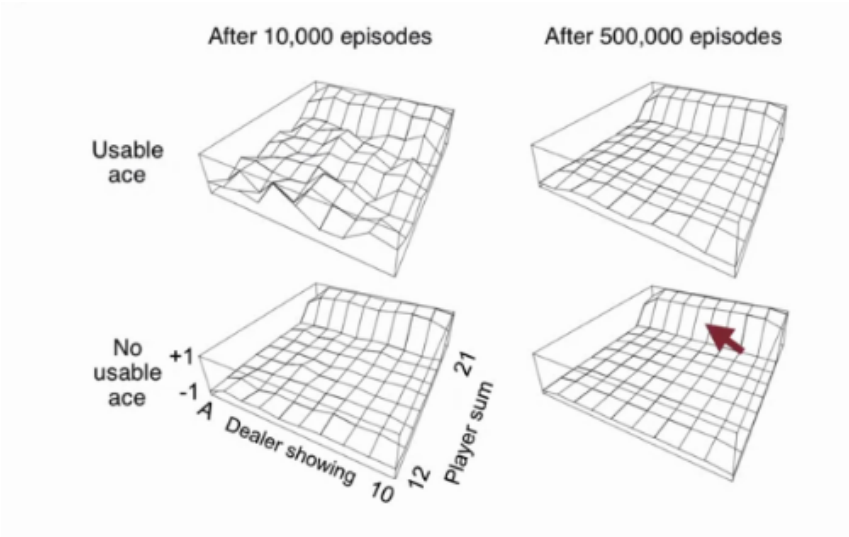


Figure 7: Approximate state-value functions

Using Monte Carlo for Action Values

So far, we've talked about using Monte Carlo to learn state-value functions for a fixed policy. This time, we'll focus on learning **action values**.

Recall that $q_\pi(s, a) \doteq \mathbb{E}_\pi [G_t | S_t = s, A_t = a]$. One way to maintain exploration is called **exploring start**s. In exploring starts, we must guarantee that **episodes start in every state-action pair**. Afterwards, the agent simply follows its policy. Let's see how exploring starts work with a policy shown in red Fig . 8. Exploring starts will **randomly sample a state and action at the start of an episode** shown in blue. In this case, the blue action is not the same as the red one. After the initial action, the agent will follow the red policy until the episode ends.

We must be able to **set the start state in this way to evaluate a deterministic policy**, like we can in this grid world. This may not always be possible. Other exploration strategies like epsilon-greedy can be used to evaluate stochastic policies.

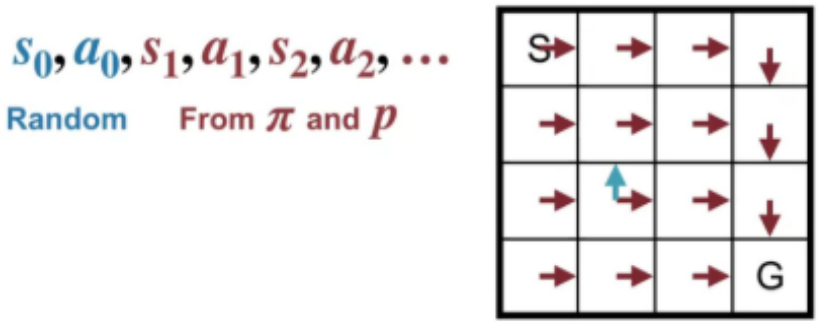


Figure 8: Exploring starts in a grid world

Using Monte Carlo methods for generalized policy iteration

General policy iteration (GPI) includes a policy evaluation and a policy improvement step. GPI algorithms produce sequences of policies that are at least as good as the policies before them. For the **policy improvement** step, we can make the policy greedy with respect to the agent's current action value estimates. For the policy **evaluation step**, we will use a Monte Carlo method to estimate the action values.

$$\pi_0 \to \pi_1 \to \pi_2 \to \ldots$$

Improvement:

$$\pi_{k+1}(s) \doteq \operatorname*{argmax}_a q_{\pi k}(s, a)$$

Evaluation: Monte Carlo Prediction Fig. 9.

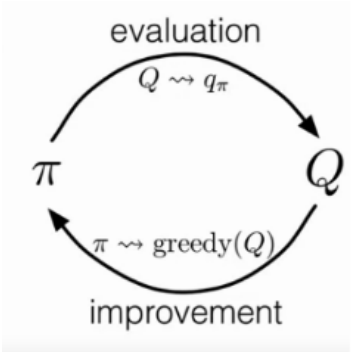


Figure 9: Monte Carlo Generalized Policy iteration

Monte Carlo control methods combined policy improvement and policy evaluation on an episode-by-episode basis.

This is the generalized policy iteration algorithm, Monte Carlo with exploring starts Fig. 10.

Solving the Blackjack Example

Exploring starts requires that episodes begin with a random state and action. Luckily for us, our version of blackjack naturally starts in random states. Then, all we have to do is to randomly select the first action

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

Initialize:

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

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

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

Loop forever (for each episode):

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

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

$G \leftarrow 0$

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

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

Append G to $Returns(S_t, A_t)$

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

$\pi(S_t) \leftarrow \text{argmax}_a \, Q(S_t, a)$

Figure 10: Monte Carlo ES (Exploring Starts)

in each episode. That is the agent ignores what it thinks is the best action in the first state and randomly chooses to hit or stick the Fig. 11 shows the optimal policy for the blackjack example.

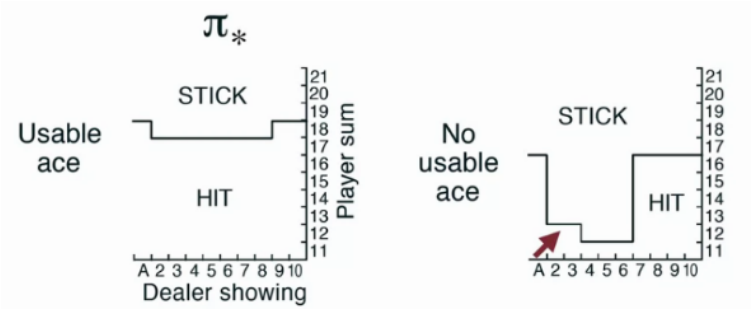


Figure 11: Monte Carlo ES (Exploring Starts)

Epsilon-soft policies

Remember Epsilon greedy exploration. We discussed this simple but effective method in the Bandit lectures we can use it with Monte Carlo to. As a quick recap **Epsilon greedy policies are stochastic policies**. They usually take the greedy action, but occasionally take a random action. Epsilon greedy policies are a subset of a larger class of policies called **Epsilon soft policies**. Epsilon soft policies take each action with probability at least **Epsilon over the number of actions**. For example, both policies shown on the Fig. 12 are valid Epsilon soft policies. The uniform random policy is another notable Epsilon soft policy.

Epsilon soft policies Force the agent to continually explore that means we can drop the exploring starts requirement from the Monte Carlo control algorithm. Epsilon soft policy assigns a nonzero probability to each action in every state because of this Epsilon soft agents continue to visit all state action pairs indefinitely.

Epsilon soft policies are always stochastic. Deterministic policies Fig. 13 specify a single action to take in each state whereas stochastic policies instead specify the probability of taking action in each state in epsilon Soft policies. All actions have a probability of at least Epsilon over the number of actions. They will eventually try all the actions.

The Epsilon greedy policy has more arrows because every action has some small probability of being selected

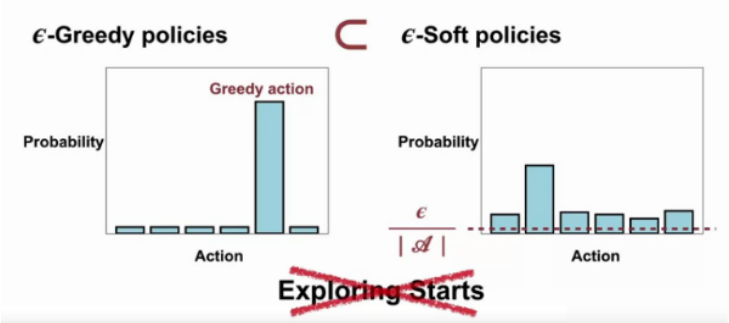


Figure 12: ϵ-Greedy exploration

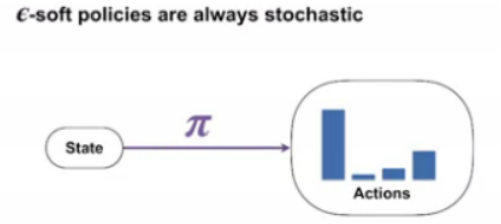


Figure 13: ϵ-Soft policies

accordingly to Fig. 14. The agent will probably follow a slightly different trajectory every episode. After enough episodes, it will have taken every action at least once in every state this difference extends the solutions that we find by exploring with Epsilon soft policies.

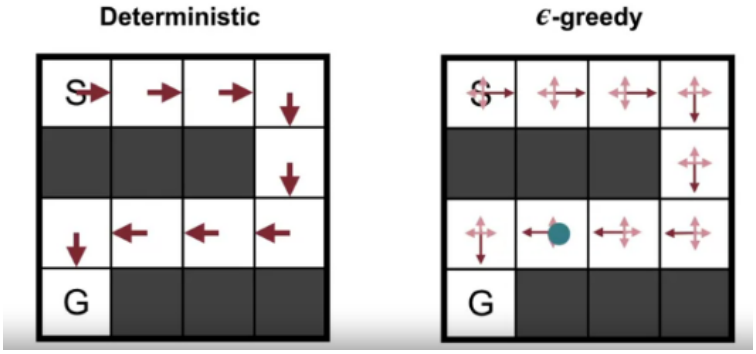


Figure 14: ϵ-greedy policies and deterministic policies

If our policy always gives at least Epsilon probability to each action, it's impossible to converge to a deterministic optimal policy. Exploring starts can be used to find the optimal policy, nevertheless, soft policies can only be used to find the optimal Epsilon soft policy. That is the policy with the highest value in each state out of all the Epsilon soft policies. This policy performs worse than the optimal policy in general. However, it often performs reasonably well and allows us to get rid of exploring starts Fig. 15.

The next Pseudo-code shows an Monte Carlo for ϵ-soft policies Fig. 16.

Why does off-policy learning matter?

The disadvantage of Epsilon soft policies is that they are sub-optimal for both acting and learning Epsilon soft policies are neither optimal policies for obtaining reward nor are the optimal for exploring to find the

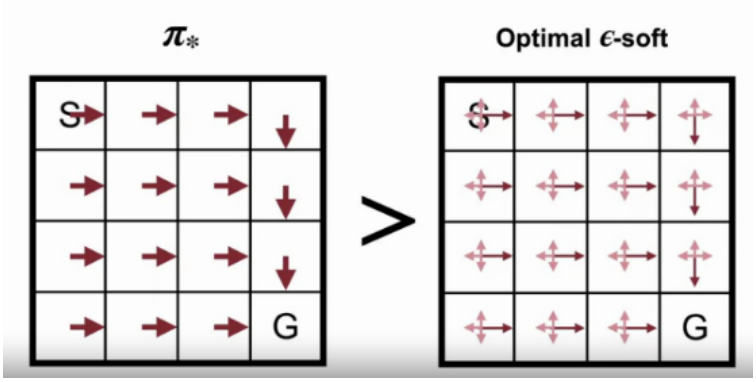


Figure 15: ϵ-soft policies might not be optimal

MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

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

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

Repeat forever (for each episode):

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

$G \leftarrow 0$

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

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

Append G to $Returns(S_t, A_t)$

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

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

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

Figure 16: MC control (for ϵ-soft policies), estimates $\pi \approx \pi_*$

best actions.

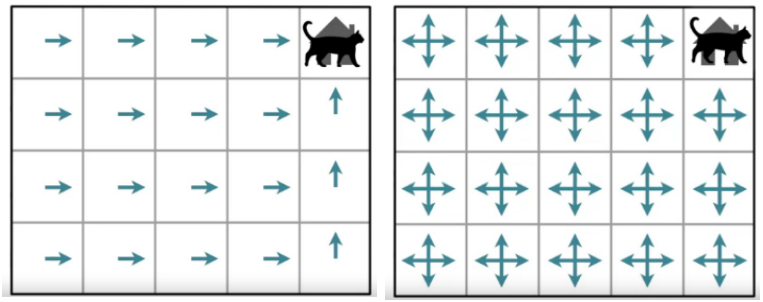
- **On-Policy**: Improve and evaluate the policy being used to select actions
- **Off-Policy**: Improve and evaluate different policy from the one used to select actions.

Target policy $\pi(a|s)$ Fig. 17(a).

- Learn values for this policy
- For example the optimal policy

Behavior policy $b(a|s)$ Fig. 17(b).

- Select actions from this policy
- Generally an exploration policy



(a) (b)

Figure 17: (a) Target policy, (b) Begavior policy.

So why are we coupling the behavior from the target policy? **Because it provides another strategy for continual exploration**. If our agent behaves according to the **Target policy** it might only experience a small number of states. If our aging can behave according to a policy that favors exploration. It can experience a much larger number of states.

One key rule of off policy learning is that the behavior policy must cover the target policy. In other words, if the target policy says the probability of selecting an action a given State S is greater than zero then the behavior policy must say the probability of selecting that action in that state is greater than 0. But there's also an intuitive reason that we show here consider this state with the behavior policy always goes up but the target policy goes to the right agent cannot learn the correct action value for that state because a never observed samples of what would happen Fig. 18.

$$\pi(a|s) > 0 \text{ where } b(a|s) > 0$$

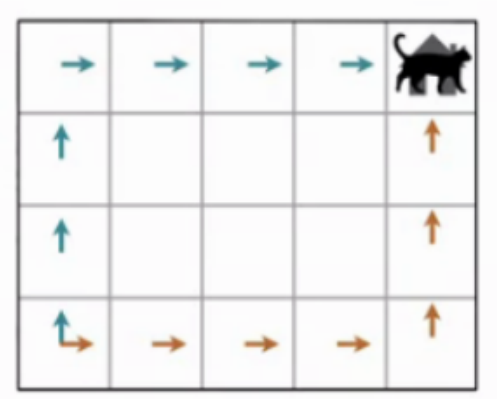


Figure 18: Exploring with ϵ-soft policies

The key points to take away from today are that off policy learning is another way to obtain continual exploration. The policy that we are learning is called the target policy and the policy that we are choosing actions from is the behavior policy.

Importance Sampling

We have some random variable x that's being sampled from a probability distribution b. We want to estimate the expected value of x but with respect to the target distribution π. Because x is drawn from b, we cannot

simply use the sample average to compute the expectation under π. This sample average will give us the expected value under b instead.

$$\textbf{Sample} : x \approx b$$

$$\textbf{Estimate} : \mathbb{E}_\pi[X]$$

Let's start with the definition of the **expected value**. We sum over all possible outcomes x multiplied by its probability according to π. Next, we can multiply by $b(x)$ divided by $b(x)$ because this term is equal to 1. $b(x)$ is **the probability of observed outcome** x under b. Shifting around the numerator, we end up with a ratio $\frac{\pi}{b(x)}$. This ratio is very important to us and is called the **importance sampling ratio** $\rho(x$.

$$\mathbb{E}_\pi[X] \doteq \sum_{x \in X} x\pi(x) = \sum_{x \in X} x\pi(x)\frac{b(x)}{b(x)}$$

$$\mathbb{E}_\pi[X] = \sum_{x \in X} x\frac{\pi(x)}{b(x)}b(x) = \sum_{x \in X} x\rho(x)b(x)$$

If we treat $x * \rho(x)$ as a new random variable, times the probability of observing $b(x)$, we can then rewrite this sum as an expectation under b. Notice that our expectation is now under b instead of being under π.

$$\mathbb{E}_\pi[X] = \sum_{x \in X} x\rho(x)b(x) = \mathbb{E}_b\left[X\rho(x)\right]$$

We know how to use importance sampling to correct the expectation, but how do we use it to estimate the expectation from data? It's actually very simple. We just need to compute a **weighted sample average** with the **importance sampling ratio as the weightings**. Note that these samples X_i are drawn from b, not π.

Recall: $\mathbb{E}[X] \approx \frac{1}{n}\sum_{i=n}^{n} x_i$

$$\mathbb{E}_b\left[X\rho(x)\right] = \sum_{x \in X} x\rho(x)b(x) \approx \frac{1}{n}\sum_{i=n}^{n} x_i\rho(x_i)$$

$$x_i \sim b$$

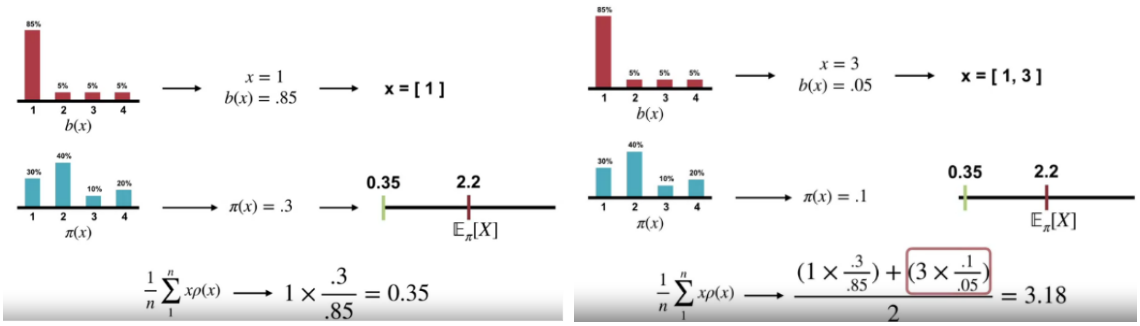
$$\mathbb{E}_\pi[X] \approx \frac{1}{n}\sum_{i=n}^{n} x_i\rho(x_i)$$

An example of importance Ratio can be seen in Fig. 19

Off-Policy Monte Carlo Prediction

If we simply average the returns, we saw from state s under behavior b, we will not get the right answer. We have to correct each return in the average. This is just what important sampling is for. All we have to do is figure out the value of ρ for each of the sampled returns.

Off-policy Monte Carlo



(a) (b)

Figure 19: (a) Importance ratio example 1, (b) Importance ratio example 2.

$$V_\pi(s) \doteq \mathbb{E}_\pi\left[G_t | S_t = s\right] \approx \text{ average } \left(\rho_0 \text{Returns}[0], \rho_1 \text{Returns}[1], \rho_2 \text{Returns}[2]\right)$$

ρ, here, is the probability of the trajectory under π divided by the probability of the trajectory under b. This ρ corrects the distribution over entire trajectories, and so corrects the distribution over returns.

$$\rho = \frac{\mathbb{P}(\text{trajectory under } \pi)}{\mathbb{P}(\text{trajectory under } b)}$$

$$V_\pi(s) = \mathbb{E}_b\left[\rho G_t | S_t = s\right]$$

Let's consider the probability distribution over trajectories. We read this probability as,given that the agent is in some state as t, what is the probability that it takes action A_t then ends up in state S_{t+1}, then it takes action A_{t+1} and ends up in S_{t+2}, and so on, until termination at time T. All of the actions are sampled according to behavior b.

$$P(A_t, S_{t+1}, A_{t+1}, \ldots, / S_t | S_t, A_{t:T})$$

Because of the Markov property, we can break this probability distribution into smaller chunks. The first chunk is the probability that the agents likes action A_t in state S_t times the probability that the environment transitions into state S_{t+1}.

$$b(A_t|S_t)p(S_{t+1}|S_t, A_t)b(A_{t+1}|S_{t+1})p(S_{t+2}|S_{t+1}, A_{t+1}) \ldots, p(S_t|S_{T-1}, A_{T-1})$$

We can rewrite this list of product probabilities using the product notation. Now, we've defined the probability of a trajectory under b. Remember where we are going. We would like to define ρ using the probability of the trajectory under π and the probability of the trajectory under b.

$$\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)$$

We would like to define ρ using the probability of the trajectory under π and the probability of the trajectory under b. Let's plug these probabilities into our definition of ρ. As we saw before, we can take these probabilities and multiply them by the importance sampling ratio.

$$\mathbb{P}(\text{Trajectory under b}) = \prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)$$

$$\rho_{t:T-1} \doteq \frac{\mathbb{P}(\text{trajectory under } \pi)}{\mathbb{P}(\text{trajectory under } b)}$$

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

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

The agent observes many returns, each according to the behavior policy b. We can estimate V_π using these returns by correcting each return with ρ.

$$\mathbb{E}_b \left[\rho_{t:T-1} G_t | S_t = s\right] = V_\pi(s)$$

In Fig. 20 is shown the algorithms for on-policy Monte Carlo algorithm:

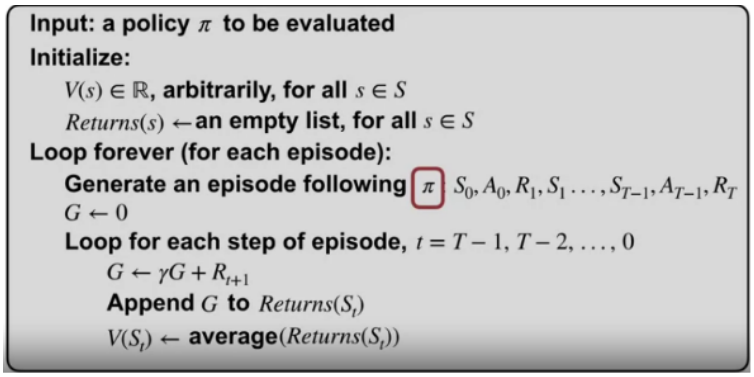


Figure 20: On-policy Monte Carlo

In Fig. 21 is the algorithm for off-policy Monte Carlo: The return is corrected by a new term W, which is the **accumulated product of important sampling ratios** $\prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$ on each time step of the episode.

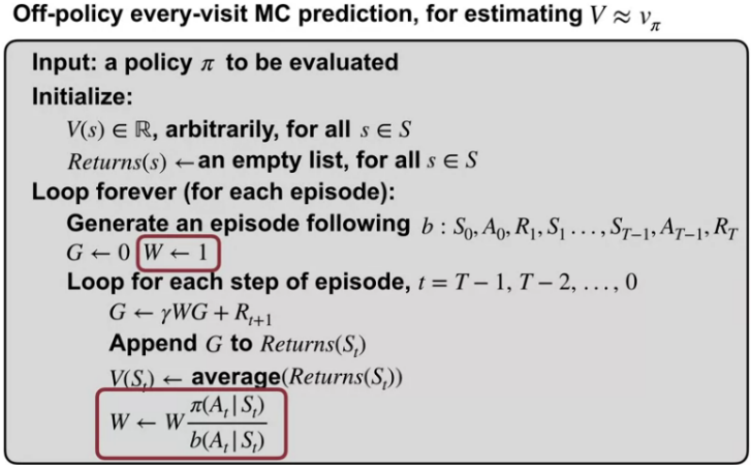


Figure 21: Off-policy Monte Carlo

We can compute ρ from t to $T-1$ incrementally. To see why, let's write out the product at each time step. Recall that the Monte Carlo algorithm loops over time steps backwards. So on the first step of the algorithm, W is set to ρ on the last time step.

On the next time step, W is the second last ρ times the last ρ, and so on. Each time step adds one additional term to the product and reuses all previous terms We can compute this recursively without having to store all past values of ρ.

Computing $\rho_{t:T-1}$ incrementally

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

$$= \underbrace{\rho_t \rho_{t+1} \rho_{t+2} \cdots \rho_{T-2} \rho_{T-1}}$$

$$W_1 \rightarrow \rho_{T-1}$$
$$W_2 \rightarrow \rho_{T-1}\rho_{T-2}$$
$$W_3 \rightarrow \rho_{T-1}\rho_{T-2}\rho_{T-3}$$
$$W_{t+1} \rightarrow W_t \rho_t$$

2 Temporal Difference Learning Methods for Prediction

What is Temporal Difference (TD) learning?

Let's discuss a small modification to our Monte Carlo policy evaluation method. We can use this formula to incrementally update our estimated value. Notice that this uses a constant step size α like our bandit learning algorithms. That means this algorithm can form a **Monte Carlo estimate without saving lists of returns**. To compute the return, we have to take samples of full trajectories. **This means we don't learn during the episode**, but we want to be able to **learn incrementally** before the end of the episode. We must come up with a new update target to achieve this goal.

Recall:

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

$$v(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right]$$

Recall the definition of the discounted return. We saw a while back that this can be written recursively like so. The value of a state at time t is the expected return at time t. We can replace the return inside this expectation with our recursive definition. We can further split up this equation because the linearity of expectation. We then get the expectation of the return on the next step, which is just the value of the next state. Now we have written the value function recursively as well.

Bootstrapping

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots$$

$$G_t = R_{t+1} + \gamma G_{t+1}$$

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

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

We can split more in terms of G_{t+1}

$$v_\pi(s) = R_{t+1} + \gamma v_\pi(S_{t+1})$$

The terms inside the brackets resemble an error, which we call the **TD error**. We will often see the TD error denoted by δ_t.

$$G_t \approx R_{t+1} + \gamma v_\pi(S_{t+1})$$

$$v(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right]$$

$$v(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$

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

The target of our update might seem a little strange at first. TD updates the value of one state towards its own estimate of the value in the next state. As the estimated value for the next state improves, so does our target $R_{t+1} + \gamma V(S_{t+1})$.

Think of time $t+1$ as the current time step and time t as the previous time step. So we simply store the state from the previous time step in order to make our TD updates. We see a stream of experience: state, action, reward, next state and so on. From the state of time t, we can take an action, and observe the next state at time $t+1$. Only then can we update the value of the previous state.

1-Step TD

$$v(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$

$$S_t \leftarrow S_{t+1}$$

Figure 22: Udating TD

In the Fig. 23 We can see the Pseudo-code for Tabular TD(0) for estimating V_π

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 23: Tabulat TD(0)

The advantages of temporal difference learning

We will look to an example where a person leaves his office and want to reach his home. TD will be used to estimate the time reaching home Fig. 24.

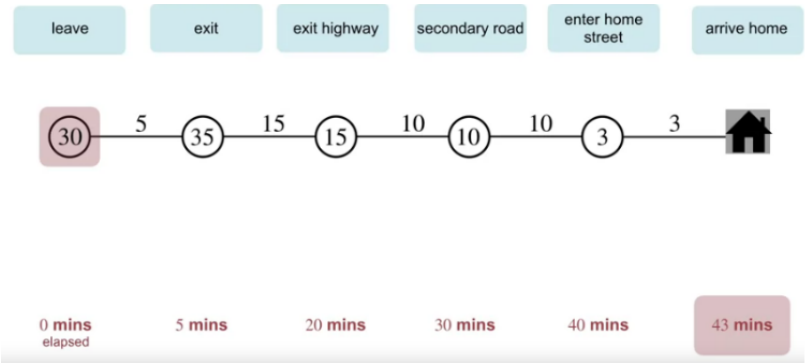


Figure 24: Example MC

Let's first look at a Monte Carlo method. In particular, we will use the constant alpha Monte Carlo method with $\alpha = 1$ and $\gamma = 1$. **In Monte Carlo methods, you update your estimates towards the return which is only available at the end of the episode**. We can only update the estimates for each of the states once we arrive home. We can start from when we leave the office. From the office, it took you a total of 43 minutes to get home. Therefore, the return at time zero is 43 Fig. 25.

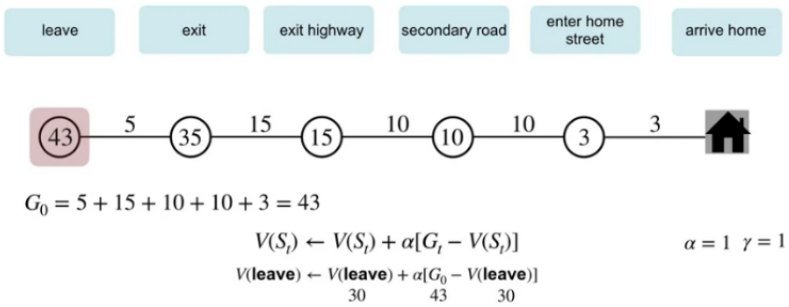


Figure 25: Calculating G_0

As $\alpha = 1$, we move our estimate completely towards the actual return. From when you exit the parking lot, it took you 38 minutes to get home, and we update our estimate accordingly. We update the estimates for the remaining states in the same way Fig. 26.

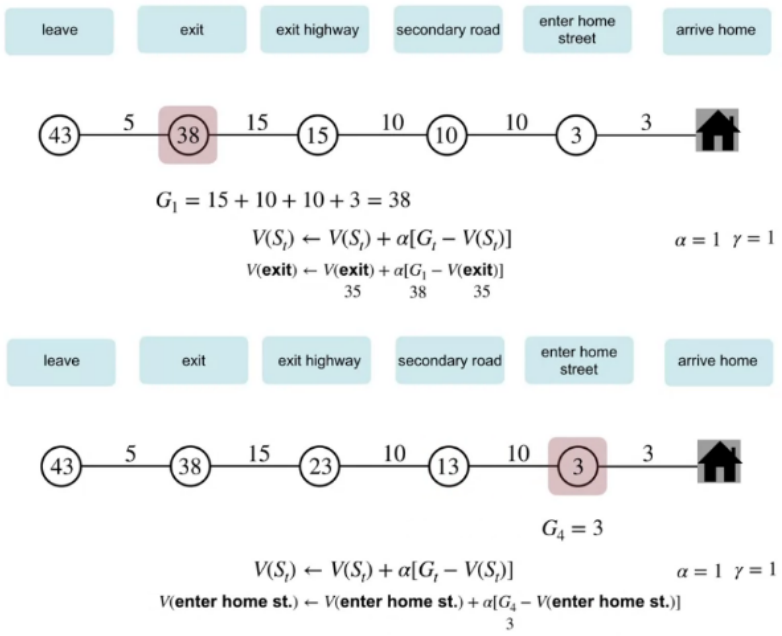


Figure 26: Updating stages MC example

Let's rewind the clock and look at how we would make updates **with TD** Fig. 27.

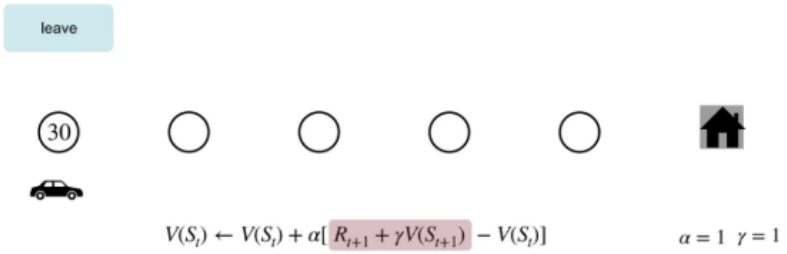


Figure 27: Same example with TD

From the office, it took you five minutes to exit the parking lot from where you made a prediction of 35 minutes Fig. 28. From the exit state, you can update the estimate for office. Your current estimate from the office is 30 minutes. You get a reward of five. The value of the next state is 35, so we update our estimate to 40 minutes.

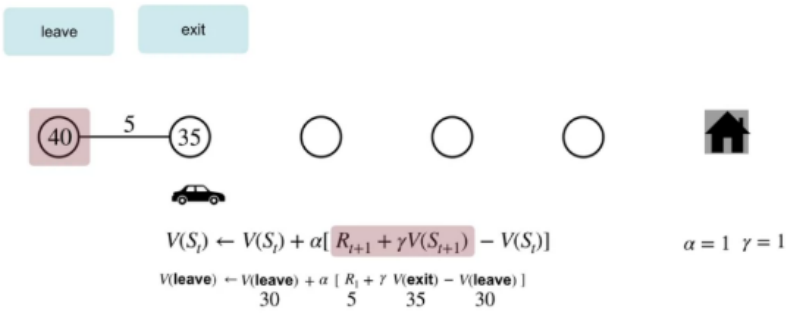


Figure 28: Same example with TD, calculating leave

Let's consider the next state. Your current estimate is 35 minutes. It took you 15 minutes from the parking lot to exit the highway Fig. 29. The estimated time to get home from when you exit the highway is also 15 minutes. So we reduce the estimated time to go from when we exit the parking lot to 30 minutes.

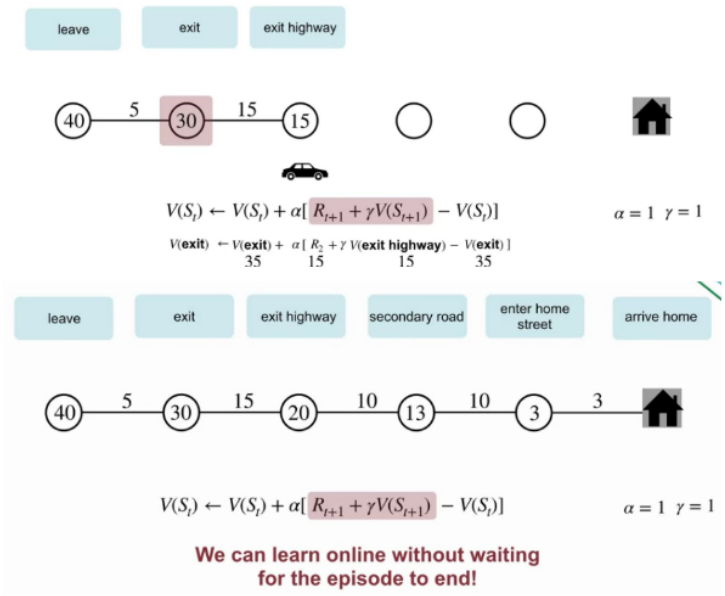


Figure 29: Same example with TD, Updating all

Remember that the terminal state $V(S_T) = 0$

Comparing TD and Monte Carlo

The next comparison is based on Fig. 30.

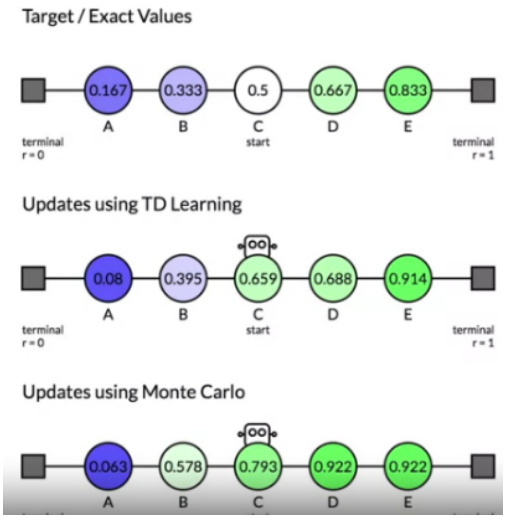


Figure 30: TD vs Monte Carlo

The x-axis represents each state in the NDP. On the y-axis, we have the estimated value. Here are the true values. Lets also plot the initial value estimates. Recall, we initialize them to 0.5. The red curve shows the values learned after the first episode. By the end of the first episode, TD only updated the value of a single state as we discussed. The estimates after a 100 episodes are about as good as they're ever going to get. Remember, we're using a constant step size of $\alpha = 0.5$ and $\gamma = 1.0$. means the values will fluctuate

in response to the outcomes of the most recent episodes. If you use a smaller learning rate or better yet a decaying learning rate, we might get a better estimate Fig. 31.

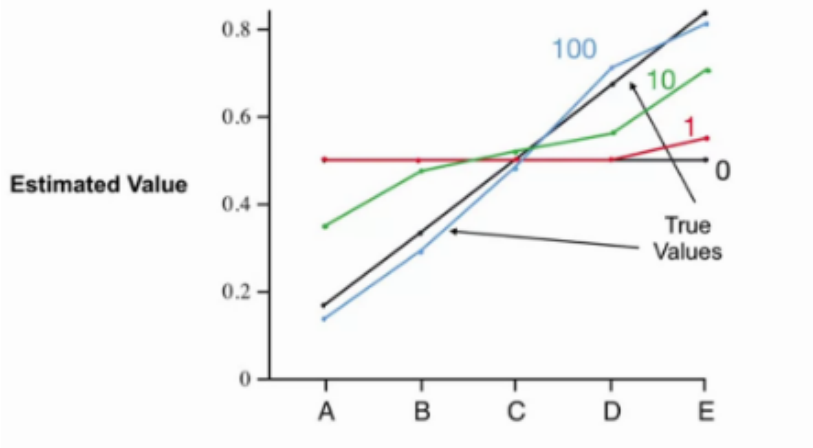


Figure 31: TD performance after multiple steps

Let's compare the performance of TD and Monte Carlo on our example problem. Here, the x-axis represents the number of episodes. The y-axis represents the root mean squared error between the value function and the learned estimates. The red learning curves represent the performance of Monte Carlo for several values of α. Each curve is averaged over a 100 independent runs Fig. 32.

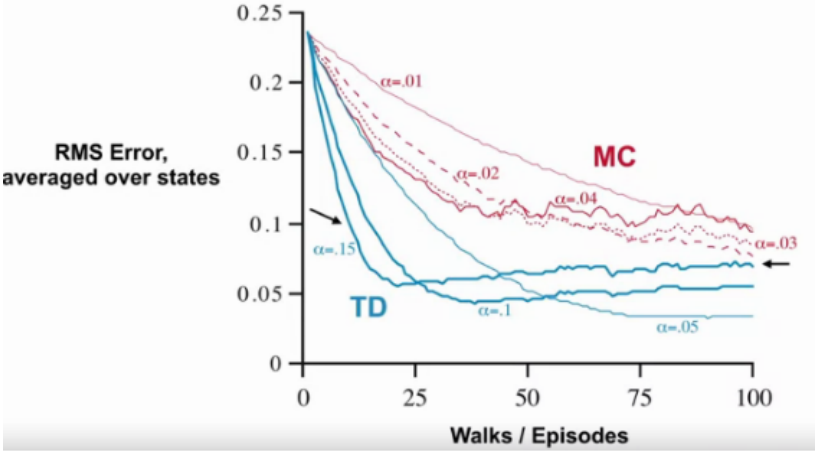


Figure 32: TD performance vs MC performance

3 Temporal Difference Learning Methods for Control

Sarsa: GPI with TD

To better remember GPI with Monte Carlo, imagine a mouse in a four-state corridor with cheese at the end. The mouse starts out knowing nothing and follows a random policy. Eventually, the mouse will stumble into the cheese just by moving randomly. At that point, the mouse updates its action values. Then it improves its policy by greedy with respect to its action values Fig. 33.

To use TD within (Generalized policy iteration) **GPI**, we need to learn an action value function. So we'll need to look at slightly different version of TD than you've seen in the past. Instead of looking at transitions

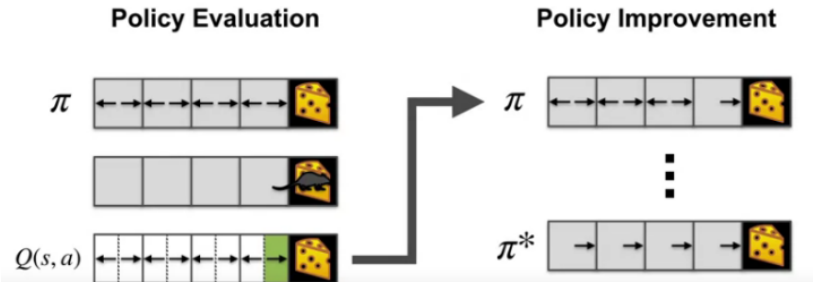


Figure 33: Mouse example

from state to state and learn the value of each state, **let's look at transitions from state action pair to state action pair and learn the value of each pair** Fig. 34.

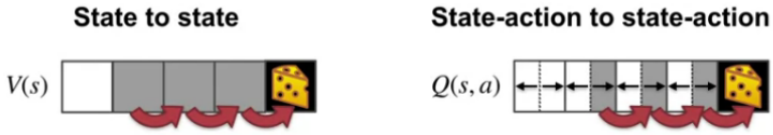


Figure 34: From state-values to action-values

The Sarsa Algorithm

$$S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$$

The agent chooses an action, in the initial state to create the first state action pair. Next, it takes that action in the current state and observes the reward R_{t+1} and next state S_{t+1}. **In Sarsa**, the agent **needs to know its next state action pair before updating its value estimates**. That means it has to commit to its next action before the update. Since our agent is learning action values for a specific policy, it uses that policy to sample the next action. Here's the full update equation.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

Recall that

$$v(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

The Fig. 35 shows the Salsa Algorithm working with the mouse example with the state-action pair.

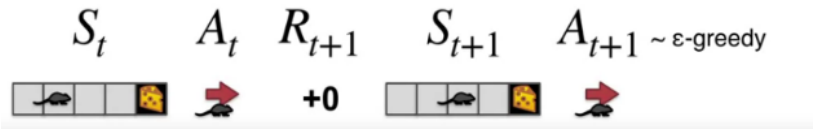


Figure 35: The Sarsa Algorithm in Mouse example

This time, we'll improve the policy every time step rather than after an episode or after convergence. In Fig. 36 is shown the Pseudo-code for the Sarsa algorithm.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 36: The Sarsa Algorithm

What is Q-learning?

Well, the new element in **Q-learning** is the action value update. Here, the target is the reward $R_{t+1} + \gamma Q(S_{t+1}.a')$. In fact, **Sarsa is a sample-based algorithm to solve the Bellman equation for action values**. Q-learning also solves the Bellman equation using samples from the environment. But **instead of using the standard Bellman equation, Q-learning uses the Bellman's Optimality Equation for action values**. The optimality equations enable Q-learning to **directly learn** Q_* instead of switching between policy improvement and policy evaluation steps.

Sarsa

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

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

Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t))$$

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

Sarsa is sample-based version of policy iteration which uses Bellman equations for action values, that each depend on a fixed policy. **Q-learning** is a sample-based version of value iteration which **iteratively applies the Bellman optimally equation**. Applying the Bellman's Optimality Equation strictly improves the value function, unless it is already optimal. So value iteration continually improves as value function estimate, which eventually converges to the optimal solution. In Fig. 37 is the Off-policy Q-learning algorithm.

How is Q-learning off-policy?

Sarsa is an on-policy algorithm. In Sarsa, the agent bootstraps off of the value of the action it's going to take next, which is sampled from its behavior policy. **Q-learning instead, bootstraps off of the largest action value in its next state**. This is like sampling an action under an estimate of the optimal policy

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 37: Q-learning algorithm

rather than the behavior policy. Since Q-learning learns about the best action it could possibly take rather than the actions it actually takes, it is learning off-policy.

Comparison with Sarsa

Sarsa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

Where:

$$A_{t+1} \sim \pi$$

Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t))$$

Where:

$$\max_{a'} \text{ and } a' \sim \pi_* \neq \pi$$

But if Q-learning learns off-policy, why don't we see any important sampling ratios? It is because the agent is estimating action values with unknown policy. It does not need important sampling ratios to correct for the difference in action selection. The action value function represents the returns following each action in a given state. The agents target policy represents the probability of taking each action in a given state. Putting these two elements together, the agent can calculate the expected return under its target policy from any given state, in particular, the next state, S_{t+1}. Q-learning uses exactly this technique to learn off-policy. Since the agents target policies greedy, with respect to its action values, all non-maximum actions have probability 0. As a result, the expected return from that state is equal to a maximal action value from that state.

Expected Sarsa

Sarsa estimates this expectation by sampling the next data from the environment and the next action from its policy. But the agent already knows this policy, so why should it have to sample its next action? Instead,

it should just compute the expectation directly. In this case, we can take a weighted sum of the values of all possible next actions. The weights are the probability of taking each action under the agents policy The Fig. 38 Shows this.

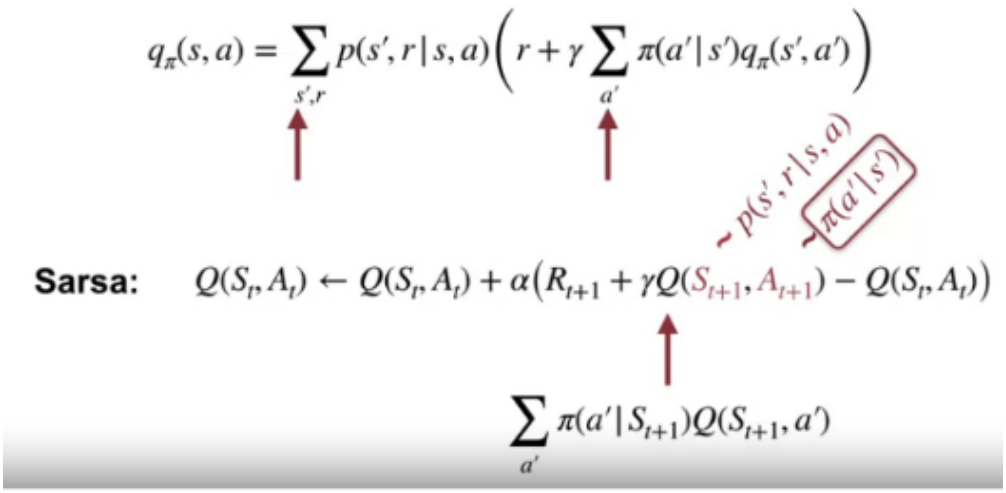


Figure 38: Expected Sarsa Equation

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

Replace $Q(S_{t+1}, A_{t+1})$ with $\sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a')$

The expected Sarsa Algorithm is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t))$$

The algorithm is nearly identical to Sarsa, except the T error uses the expected estimate of the next action value instead of a sample of the next action value. That means that on every time step, the agent has to average the next state's action values according to how likely they are under the policy. The Fig. 39 is an example of the new term of the Salsa Algorithm.

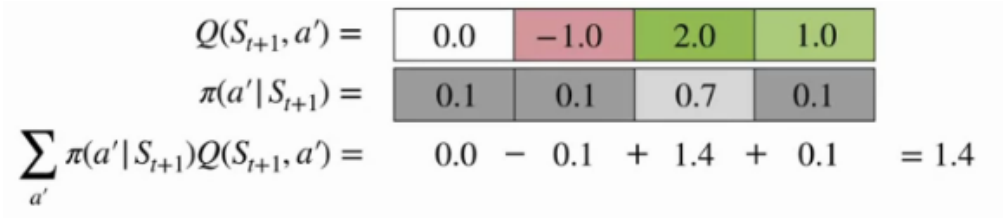


Figure 39: Expected Sarsa example

We show that the expected Sarsa algorithm explicitly computes the expectation under its policy, which is more expensive than sampling but has lower variance. In summary, the Fig. 40 summarizes all this information.

It is important to note that Expected Sarsa is **Off-policy**. **Q-learning is a special case of Expected Sarsa.**

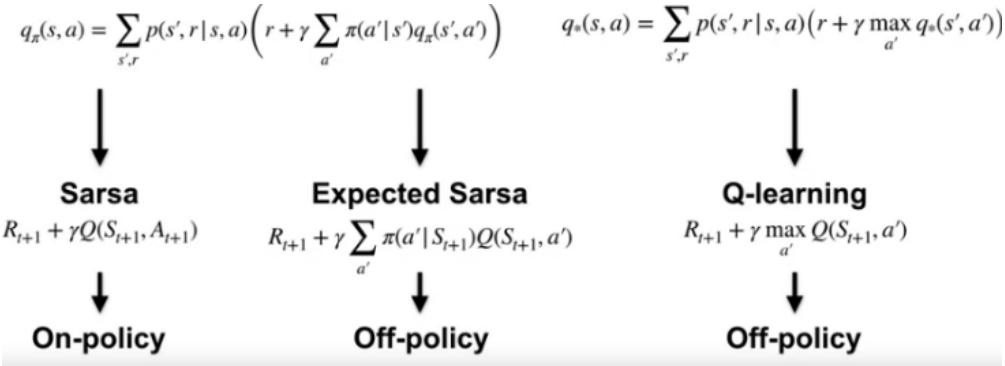


Figure 40: Summary of TD algorithms

4 Planning, Learning & Acting

What is a Model?

Models are used to store knowledge about the dynamics. When imagining different scenarios to a decision-making, this is rooted in our **knowledge about how the world works**. From a particular state in action, the **model should produce a possible next state and reward**. This allows us to see an outcome of an action without having to actually take it Fig. 41. **Models store knowledge about the transition and reward dynamics**.

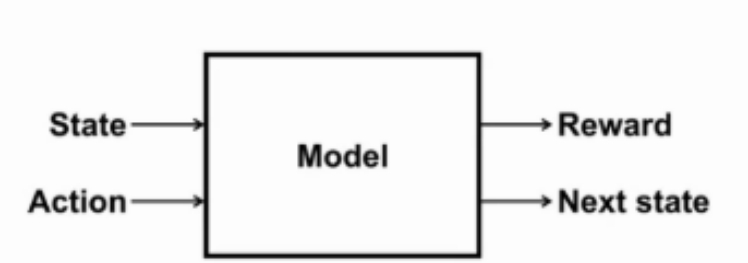


Figure 41: Model storing knowledge

A model allows for planning. **Planning refers to the process of using a model to improve a policy**. One way to plan with a model is to use simulated experience and perform **value function updates** as if those experiences happened. By improving the value estimates, we can make more informed decisions. Simulating experience improves the sample efficiency. The addition of simulated experience means the agent needs **fewer interactions with the world** to come up with the same policy Fig. 42.

Now, let's talk about the different **types of models** that may be useful to us. One is a **sample model**, which **produces an actual outcome drawn from some underlying probabilities**. For example, a sample model for flipping a coin can generate a random sequence of heads and tails. Another type of model is a **distribution model** which completely **specifies the likelihood or probability of every outcome**. In the coin flipping example, it would say that heads could occur 50 percent of the time and that tails could occur 50 percent of the time. It could also produce the odds of any sequence of heads and tails using this information Fig. 43.

Sample models can be computationally inexpensive because random outcomes can be produced according to a set of rules. For example, to flip five coins, a sample model can randomly pick zero or one independently five times. It only needs to produce a single outcome for each flip. **Distribution models** contain more information, but can be difficult to specify and can become very large. Let's go back to the example for flipping five coins. A distribution model would enumerate every possible sequence of heads and tails you

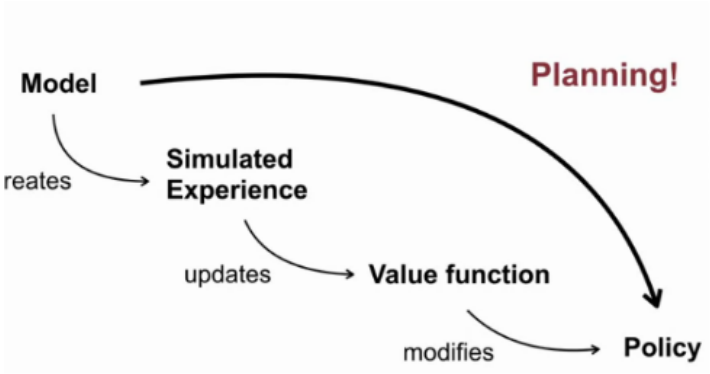


Figure 42: The use of models

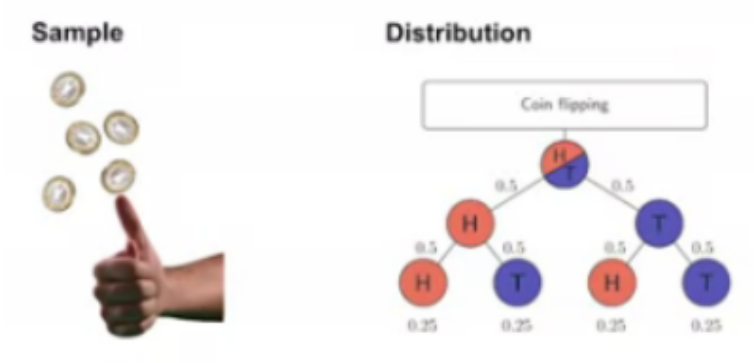


Figure 43: Types of models

could get across the five coins and assign a probability to each sequence. For this simple problem, that would consist of fully describing 32 possible outcomes.

Adventages and disadventages

- Sample models require less memory
- sample models can only approximate this expected outcome by averaging many samples together
- Distribution models can be used to compute the exact expected outcome
- Distribution models can be used to assess risk

Random Tabular Q-planning

One can leverage a model to better inform decision-making without having to interact with the world, we call this **process planning with model experience**.

Planning is a process which takes a model as input and produces an improved policy. Fig. 44

One possible approach to planning is to first sample experience from the model. This is like imagining possible scenarios in the world based on your understanding of how the world works. This generated experience can then be use to perform updates to the value function as if these interactions actually occurred in the world. Behaving greedily with respect to these improved values results in improved policy Fig. 45.

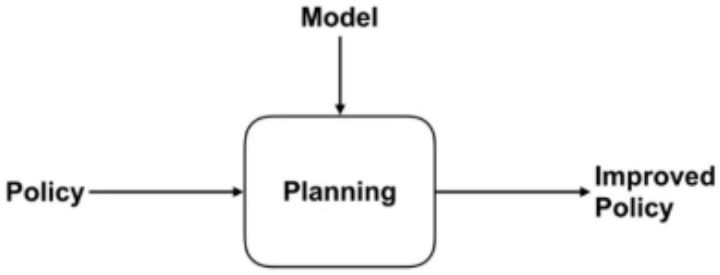


Figure 44: Planning to improve policies

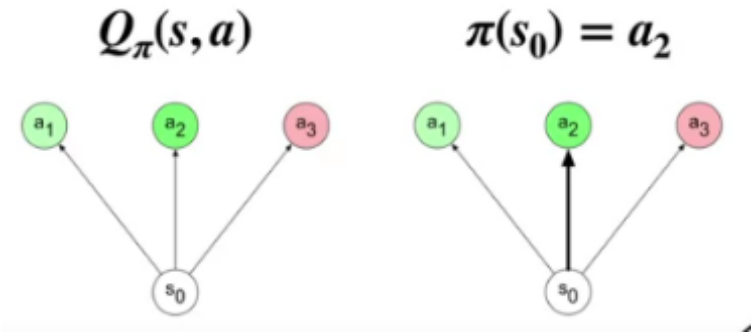


Figure 45: Planning

Recall that Q-learning uses experienced from the environment, they performs an update to improve a policy. I**n Q-planning, we use experience from the model** and perform a similar update to improve a policy Fig. 46.

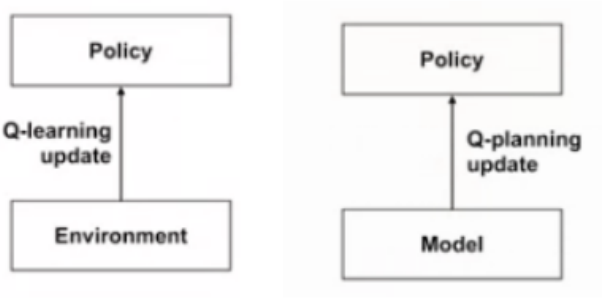


Figure 46: Connecting planning with Q-learning

Random-sample one-step tabular Q-planning illustrates this idea. This approach assumes we have a **sample model of the transition dynamics**. It also assumes that we have a strategy for sampling relevant state action pairs. One possible option is to **sample states and actions uniformly**. This algorithm first chooses a **state action pair at random** from the set of all states and actions. It then queries the sample model with this state action pair to produce a sample of the next state and reward. It then performs a Q-Learning Update on this model transition. Finally, it improves the policy by beautifying with respect to the updated action values Fig. 47.

A key point is that this planning method only uses imagined or simulated experience. **All of these updates can be done without behaving in the world or in parallel with the interaction loop** as shown in Fig. 48.

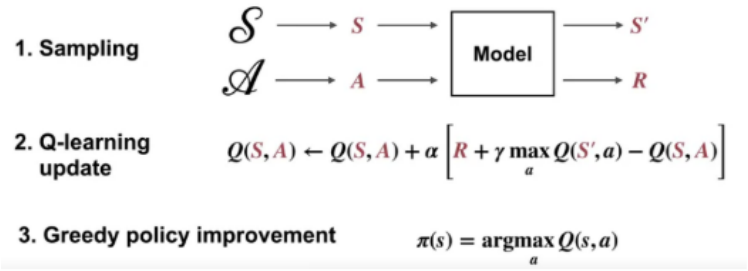


Figure 47: Random-sample one-step tabular Q-planning

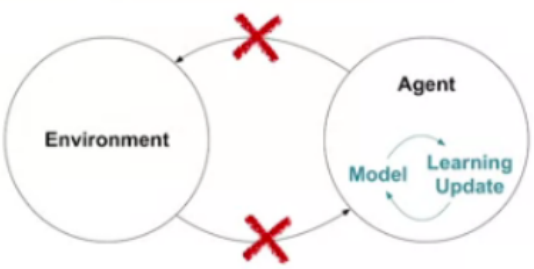


Figure 48: Planning using imagined experience

Imagine that actions can only be taken as specific time points, but Learning updates can be executed relatively fast. This results in some waiting time from after the Learning Update and when the next action is taken. We can fill in this waiting time with Planning Updates.

The pseudo-code for Random-Tabular Q-planning can be seen in Fig. 49.

Random-sample one-step tabular Q-planning

Loop forever:

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

Figure 49: Random-Tabular Q-planning

The Dyna Architecture

We've seen Q-learning which performs updates using environment experience. Q-planning on the other hand performs updates using simulated experience from the model.

We can combine these two ideas through the **Dyna architecture**. We have the usual environment and policy. They generate a stream of experience. We use this experience to perform direct RL updates. To do planning, we need a model. The model has to come from somewhere. The environment experience can be used to learn the model. This model will be used to generate model experience. In addition, we want to control how the model generates this simulated experience, what states the agent will plan from. We call this process **search control**. Planning updates are performed using the experience generated by the model

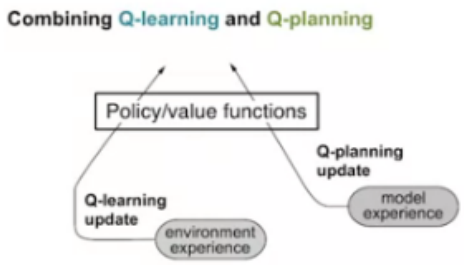


Figure 50: Combining Q-learning with Q-planning

Fig. 51.

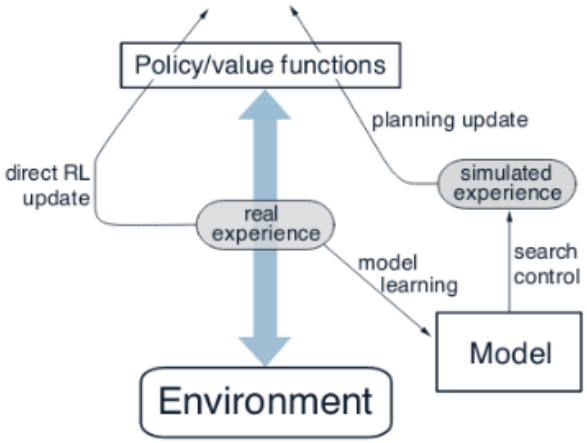


Figure 51: Dyna-Q architecture

In comparison, Q-learning require many episodes to achieve similar performance. Dyna does more computation per step, but uses limited experience more efficiently.

The Dyna Algorithm

Tabular Dyna-Q assumes deterministic transitions. For example, when this rabbit chooses to move right in state A, there's only one outcome. The rabbit transitions to state B with a reward of zero. Likewise, if the rabbit moves right in state B, it stays in state B with a reward of one. The left action in B always takes the rabbit to state A with a reward of zero. After the rabbit has experienced these transitions, the model knows what happens in these state action pairs. If a transition occurred once, it will always unfold that way in the future. The model can be certain about this.

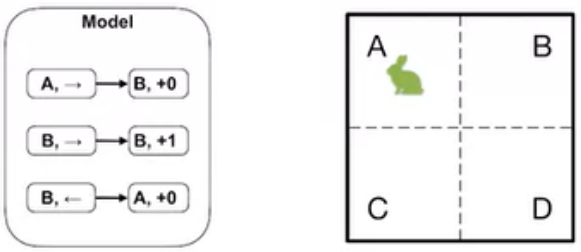


Figure 52: Rabbit example

It performs a Q-learning update with this transition, what we call direct-RL. If we were to stop here, we would get exactly the Q-learning algorithm. This is where things start to differ from model-free methods Fig. 53.

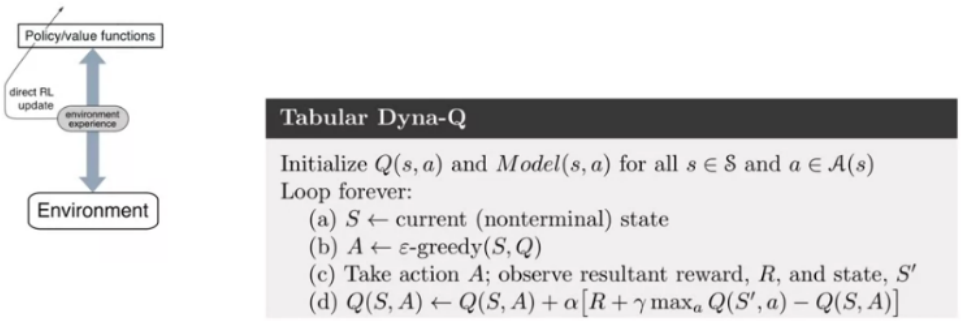


Figure 53: Dyna-Q Direct RL

Dyna-Q will take this transition and perform a **model learning step** with it. To do so, the algorithm memorizes the next state and reward for the given state action pair. This works because we assume the environment is **deterministic** Fig. 54.

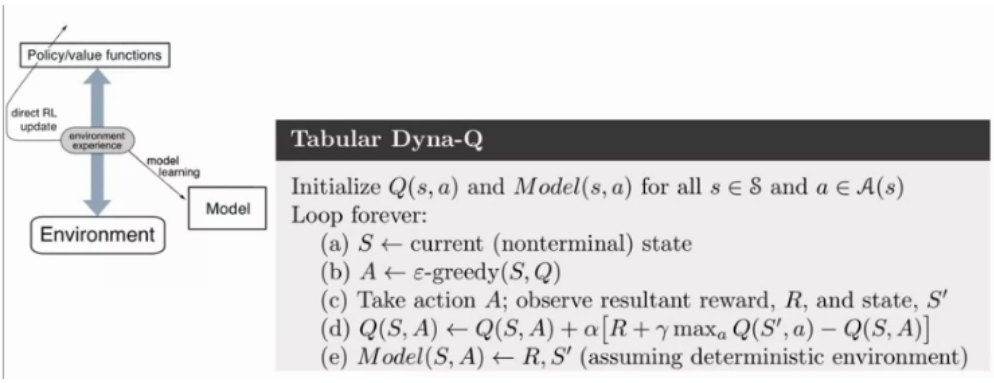


Figure 54: Dyna-Q Updating model

Dyna-Q then performs several steps of planning. Each planning step consists of three steps; **search control, model query, and value update**. In this algorithm, search controls selects a previously visited state action pair at random. It must be a state action pair the agent has seen before. Otherwise, the model would not know what happens next. We have now generated a model transition Fig. 55.

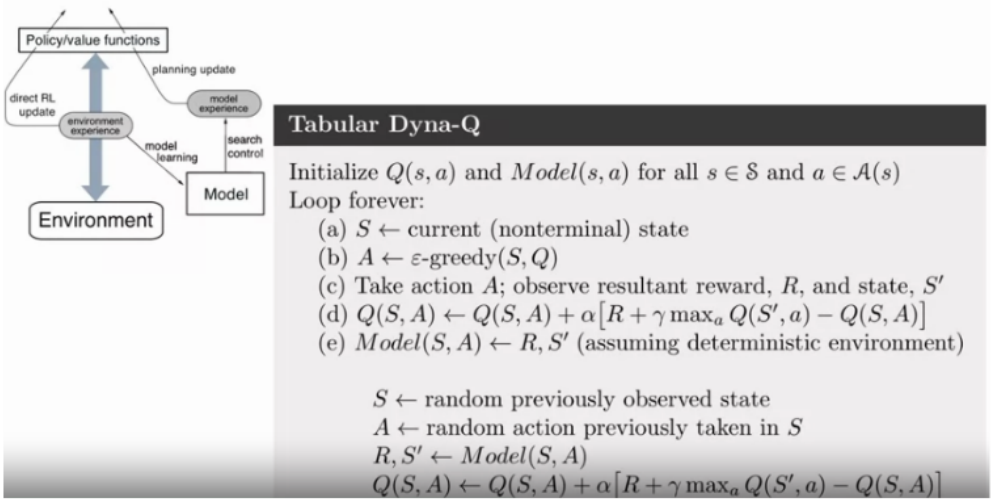


Figure 55: Dyna-Q planning

Finally, Dyna-Q performs a Q-learning update with the simulated transition. The planning step is repeated many times. The most important thing to remember is that Dyna-Q performs many planning updates for each environment transition Fig. 56.

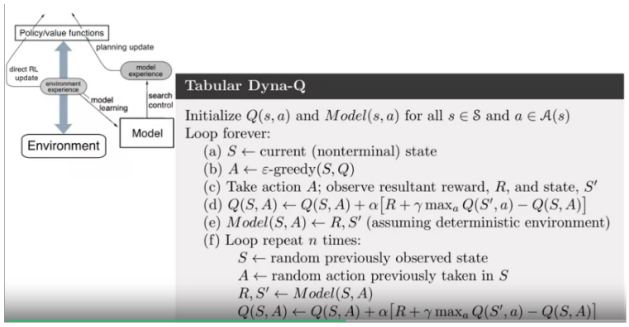


Figure 56: Dyna-Q algorithm

What if the model is inaccurate?

Models are inaccurate when transitions they store are different from transitions that happen in the environment. At the beginning of learning, the agent hasn't tried most of the actions in almost all of the states. The transitions associated with trying those actions in those states are simply missing from the model. **We call models of missing transitions incomplete models**. The model could also be inaccurate if the environment changes. Taking an action in a state could result in a different next state and reward than what the agent observed before the change. We say the model is inaccurate because what actually happens is different from what the model says Fig. 57.

The model becomes outdated when the environment changes. Planning with that model will likely make the agent's policy worse with respect to the environment.

Let's see how Dyna-Q does planning with an incomplete model. In the planning step, we must determine which state action pairs to query the model with. The model only knows the next state and reward from state action pairs it has already visited. Therefore, Dyna-Q can only do planning updates from previously visited state action pairs. Dyna-Q only plans the transitions it has already seen. So, in the first few time steps of learning, Dyna-Q might do quite a few planning updates with the same transition. However, as Dyna-

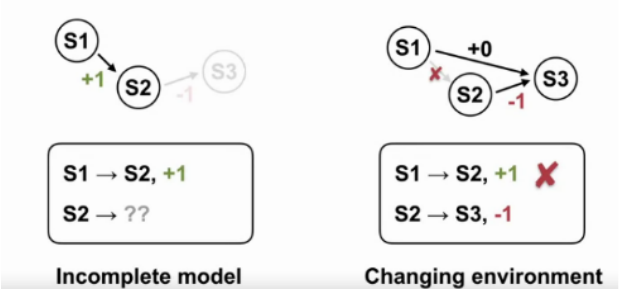


Figure 57: Innacurate models

Q visits more state action pairs in the environment, its planning updates become more evenly distributed throughout the state action space.

In-depth with changing environments

To encourage the agent to revisit its state periodically, we can add a bonus to the reward used in planning. This bonus is simply $k\sqrt{\tau}$, where r, is the reward from the model and τ is the amount of time it's been since the state action pair was last visited in the environment. τ **is not updated in the planning loop**, that would not be a real visit. k is a small constant that controls the influence of the bonus on the planning update. If $k = 0$, we would ignore the bonus completely. Adding this exploration bonus to the planning Fig. 58.

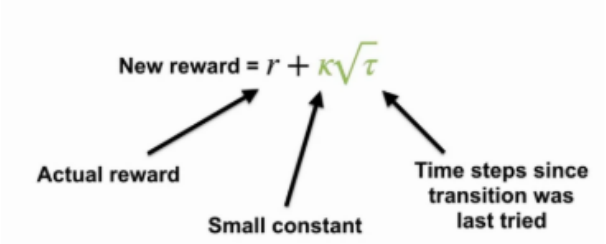


Figure 58: Bonus rewards for exploration

By artificially increasing the rewards used in planning, we increase the value of state action pairs it haven't been visited recently. Imagine a state action pair essay that has not been visited in a long time. That means the τ will be large. As τ grows, the bonus becomes bigger and bigger. Eventually, planning will change the policy to go directly to S due to the large bonus. When the agent finally visit state S, it might see a big reward, or it might be disappointed. Either way, the model will be updated to reflect the dynamics of the environment Fig. 59.

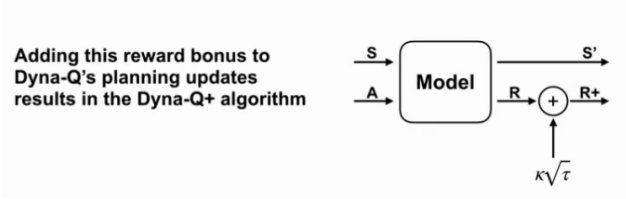


Figure 59: The Dyna-Q+ algorithm

References

[1] University of Alberta. *Sample-based Learning Methods*. 2019. URL: `https://www.coursera.org/learn/sample-based-learning-methods` (visited on 12/06/2019).

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.