Markov Decision Process

Markov Property: In probability theory and statistics, the term Markov Property refers to the memoryless property of a stochastic — or randomly determined — process.

Markov Chain: A Markov Chain is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event.

Expanding on the Markov Property

To deepen our understanding of the Markov Property, we can view it as follows:

$$P(X(t+1)=j|X(0)=i0,X(1)=i1,\ldots,X(t)=i)=P(X(t+1)=j|X(t)=i)$$

Put in words, the formula represents a situation in which the state of *X* at time *t+1* only depends on one preceding state of *X* at time *t*, and is independent of past states *X(t−1), …, X(1)*.

Now let's shed more light on this with a simple example.

In string "easy", according to Markov Property, we have:

$$\begin{aligned} P(X=easy) &= P(x0=e)P(x1=a \mid x0=e)P(x2=s \mid x0=e, x1=a)P(x3=y \mid x0=e, x1=a, x2=s) \\ &= P(x0=e)P(x1=a \mid x0=e)P(x2=s|x1=a)P(x3=y|x2=s) \end{aligned}$$

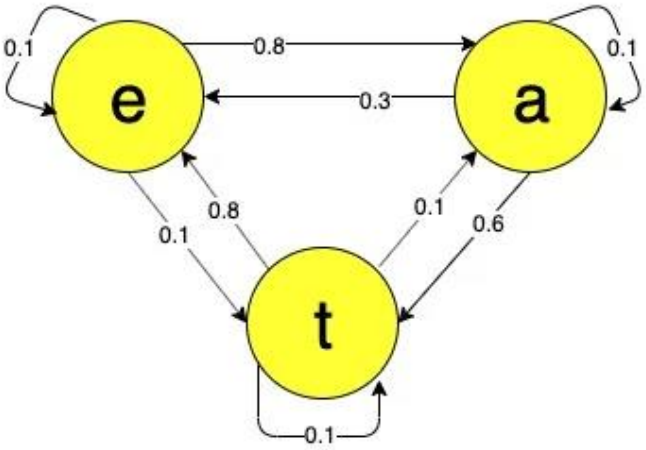
- *P(x3=y | x0=e, x1=a, x2=s)* represents the probability that y appears at time 3 when *e* appears at time 0, *a* appears at time 1 and *s* appears at time 2
- *P(x3=y|x2=s)* represents the probability that y appears at time 3 when *s* appears at time 2

So, in the above equation, Markov Property makes the *P(easy)* easier to compute with the assumption that *y only depends on the previous neighbor state s and is independent of e and a.* It means that when *y* in *"easy"* is generated, we only care about the transition probability from *s* to *y* instead of the transition probability from *eas* to *y*.

Of course, we know it may not work like this in the real world, but the hypothesis is useful nonetheless. It helps us make complicated situations computable and most of the time it works quite well.

Understanding the Markov Chain

When we put the Markov Property to work in a random process, we call it a Markov Chain.



Here is the formulated definition of a Markov Chain:

Markov chain is a tuple (S, P), where

- S is a set of states
- P(*s, s'*) is the state transition probability. the probability that state *s* at time *t* to state *s'* at time *t+1*

Using Figure 1 above, we can demonstrate how a Markov Chain can generate words.

Assume we start separately from state *e*, *a*, and *t,* with the respective probability of 40%, 30%, and 30%. According to Markov Property, a string can be generated letter by letter — taking into consideration only the letter immediately before it.

For example, we have a 40% probability of starting with *e* at time 0. Then we move from state *e* to state *a* at time 1 to get *ea*. To arrive at the word *eat*, we move directly from state *a* to state *t* at time 2, without regard for the earlier state *e*.

$$P(eat) = P(x0=e)P(x1=a|x0=e)P(x2=t|x1=a) = 0.4 \times 0.8 \times 0.6 = 0.192$$

$$P(tea) = P(x0=t)P(x1=e|x0=t)P(x2=a|x1=e) = 0.3 \times 0.8 \times 0.8 = 0.192$$

$$P(aet) = P(x0=a)P(x1=e|x0=a)P(x2=t|x1=e) = 0.3 \times 0.3 \times 0.1 = 0.009$$

With the above computations, we can see that this Markov Chain gives *eat* and *tea* an equally high score, while *aet* gets the lowest score. The formula indicates that *eat* and *tea* are more like words, while *aet* appears not to be one at all.

Defining The Markov Decision Process (MDP)

Recall our discussion of the **Markov Chain**, which works with *S,* a set of states, and *P,* the probability of transitioning from one to the next. It also uses the **Markov Property**, meaning each state depends only on the one immediately prior to it.

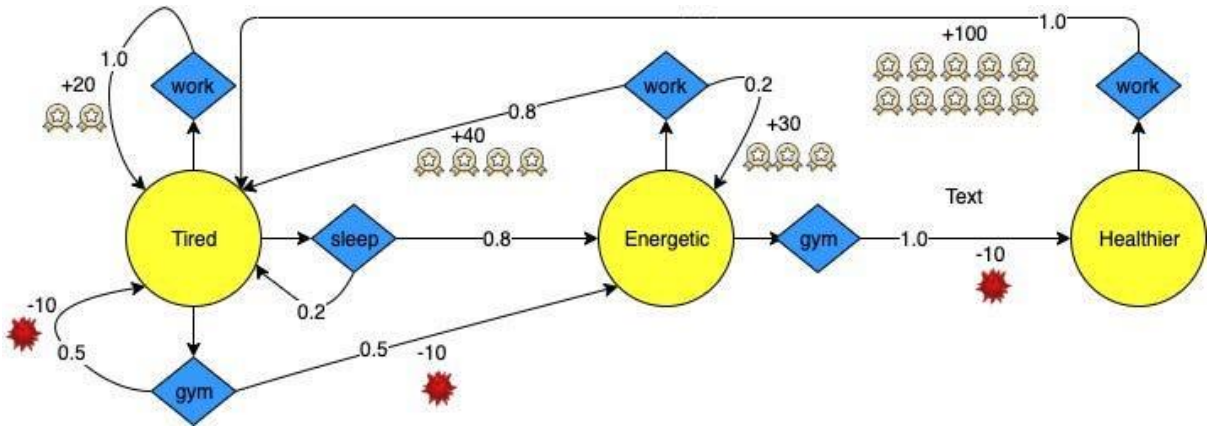


Figure 2: An example of the Markov decision process

Now, the **Markov Decision Process** differs from the Markov Chain in that it brings *actions* into play. This means the next state is related not only to the current state itself but also to the actions taken in the current state. Moreover, in MDP, some actions that correspond to a state can return rewards.

In fact, the aim of MDP is to train an agent to find a policy that will return the maximum cumulative rewards from taking a series of actions in one or more states.

Here's a formulated definition, which is what you'll probably get if you google Markov Decision Process:

An MDP is a 5-tuple (S, A, P, R, γ), where

- S is a set of states
- A is a set of actions
- P(*s, a, s'*) is the probability that action *a* in state *s* at time *t* will lead to state *s'* at time *t+1*
- R(*s, a, s'*) is the immediate reward received after a transition from state *s* to *s'*, due to action *a*
- γ is the discounted factor which is used to generate a discounted reward.

Now, let's apply this framework to Figure 2 above for a more concrete understanding of these abstract notes:

- S: Tired, Energetic, Healthier (in the yellow circles)
- A: work, sleep, gym (in the blue quadrangles)
- P: the probability of transiting from one state to another when an action is chosen
- R: the reward, given immediately when an action is taken
- γ : a discounted factor to compute discounted reward (explained later)

▶ MDP in Action: Learning with Adam

We can make this even easier to grasp with a story, using Adam as our example. As we know, this hard-working young man wants to make as much money as he can. Using the framework defined above, we can help him do just that.

▶ When Adam's state is *Tired*, he can choose one of three actions: (1) continue working, (2) go to the gym, and (3) get some sleep.

If he chooses to work, he remains in the *Tired* state with the certainty of getting a +20 reward. if he chooses to sleep, he has 80% of moving to the next state, *Energetic*, and a 20% chance of staying *Tired*.

If he doesn't want to sleep, he may go to the gym and do a workout. This gives him a 50% chance of entering the *Energetic* state and a 50% chance of staying *Tired*. However, he needs to pay for the gym, so this choice results in a -10 reward.

▶ When Adam becomes *Energetic*, he can go back to work and be more efficient. From there, he has an 80% chance of getting *Tired* again (with a +40 reward), and a 20% chance of staying *Energetic* (with a +30 reward).

Sometimes, when he is *Energetic*, he wants to do a workout. When he exercises in this state, he has a good time and gets 100% getting *Healthier*. Of course, he needs to pay for it with a -10 reward.

▶ Once he arrives at the state *Healthier*, there is only one thing on his mind: earn more money by doing more work. Because he is in such a good state, he works at peak efficiency, earns a +100 reward, and keeps working until he gets tired again.

With the above information, we can train an agent aimed at helping Adam find the best policy to maximize his rewards over time. This agent will undertake a Markov Decision Process.

However, before we can do that, we need to know how to compute the cumulative reward when an action is taken in one state. That is to say, we must be able to estimate the state value.

Don't worry! This will only take a minute to cover.

Discounted Reward

Reinforcement Learning is a multi-decision process. Unlike the "one instance, one prediction" model of supervised learning, an RL agent's target is to maximize the cumulative rewards of a series of decisions — not simply the immediate reward from one decision.

It requires the agent to look into the future while simultaneously collecting current rewards.

In Adam's example above, future rewards are as important as current rewards. But in the CartPole game we discussed here, surviving in the present is more important than anything else.

Because future rewards can be valued differently depending on the scenario, we need a mechanism to discount the importance of future rewards at different time steps.

γ : a discounted rate or discounted factor

The above symbol for a discounted rate or factor is the key to this mechanism. The rewards computed by it are referred to as *discounted rewards.*

Discount rate: $\gamma = 0.9$
Sequential rewards from time *t*: -10, -20, 70
Discounted reward at time *t*: $-10 - \gamma \times (-20) + \gamma^2 \times 70 = 28.7$.

Consider the information above. If the discount rate is close to 0, future rewards won't count for much in comparison to immediate rewards. In contrast, if the discount rate is close to 1, rewards that are far in the future will be almost as important as immediate rewards.

In short, discounted reward how we estimate the value of a state.

- **Dynamic Programming (DP):** introduced in our discussion of MDP
- **Monte-Carlo (MC) learning:** to adapt when information is lacking
- **The simplest Temporal Difference learning, TD(0):** a combination of DP and MC

MDP and Dynamic Programming

- **dynamic programming:** breaking a large problem down into incremental steps so optimal solutions to sub-problems can be found at any given stage
- **model:** a mathematical representation of a real-world process
- **Bellman Optimality Equation:** gives us the means to estimate the optimal value of each state

Now it's important to note that MDP only works with a *known* model, in which all five tuples (shown below) are evident.

An MDP is a 5-tuple (S, A, P, R, γ), where

- S is a set of states
- A is a set of actions
- P(*s, a, s'*) is the probability that action *a* in state *s* at time *t* will lead to state *s'* at time *t+1*
- R(*s, a, s'*) is the immediate reward received after a transition from state *s* to *s'*, due to action *a*
- γ is the discounted factor which is used to generate a discounted reward.

We will enter into solving an MDP problem when part of the model is *unknown*.

In this case, our agent must learn from the environment by interacting with it and collecting experiences, or *samples*. In doing so, the agent carries out strategy evaluation and iteration and can obtain the optimal strategy.

Since the theory to support this approach comes from the Monte-Carlo method, let's start by discussing Monte Carlo learning.

Monte-Carlo Learning

Entire problem can be transformed into a Markov Decision Process (MDP), which makes decisions with the five tuples < s, P, a, R, γ > above.

When we know all five, it's easy to calculate an optimal strategy to get the maximum reward. However, in the real world, we almost never have all of this information at the same time.

For example, the state transition probability (*P)* is difficult to know and, without it, we can't use the Bellman Equation below to solve V and Q values.

$$V^*(s) = max_a \sum_{s'} P(s, a, s') \left[R(s, a, s') + \gamma . V(s') \right])$$

$$Q_{k+1}(s, a) \leftarrow \sum P(s, a, s) \left[R(s, a, s') + \gamma . max_{a'} Q_k(s', a') \right]) \text{ for all } (s, a)$$

But what if we have to solve a problem without knowing *P*? How can we transform it into a Markov Decision Process?

First, consider that although we don't know what the state transition probability *P* is, we do know objectively that it exists. Therefore, we simply have to find it.

To do so, we can have our agent run trials, constantly collecting samples, getting rewards, and thereby evaluating the value function. This is exactly how the Monte-Carlo method works: try many times, and the final estimated *V* value will be very close to the real *V* value.

Monte-Carlo Evaluation

As mentioned, the Monte-Carlo method involves letting an agent learn from the environment by interacting with it and collecting samples. This is equivalent to sampling from the probability distribution *P(s, a, s')* and *R(s, a).*

However, Monte-Carlo (MC) estimation is only for trial-based learning. In other words, an MDP without the *P* tuple can learn by trial-and-error, through many repetitions.

In this learning process, each "try" is called an episode, and all episodes must terminate. That is, the final state of the MDP should be reached. Values for each state are updated only based on final reward Gt, not on estimations of neighbor states — as occurs in the Bellman Optimality Equation.

MC learns from complete episodes and is therefore only suitable for what we call *episodic MDP*.

Here is our updated state value formula:

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

In which:

- *V(St)* is the state value that we are going to estimate, which can be initialized randomly or with a certain strategy.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

- Gt is calculated above, *T* is the terminate time.
- is a parameter like learning rate. It can influence the convergence.

Various Methods to Get V(St)

Consider this: If the state *s* appears twice in an episode at time *t + 1* and time *t + 2* respectively, do we use one or both when calculating the value of the state *s*? And how often do we updateV(St)? Our answers to these questions will leads us to different approaches:

- **First-Visit Monte-Carlo Policy Evaluation**

With policy (could be a random policy just as we used in previous articles) for each episode, **only the first time** that the agent arrives at S counts:

The first time state s appears: $N(S) \leftarrow N(S) + 1$
Total rewards update: $S(S) \leftarrow S(S) + G_t$
State s value: $V(s) = S(s)/N(s)$
When $N(s) \rightarrow \infty$, $V(s) \rightarrow v_\pi(s)$ so try as many episodes as you can to get as close as possible to the real state *s* value.

- **EverEvery-Visit Monte-Carlo Policy Evaluation**

With policy (could be a random policy just as we used in previous articles) for each episode, **every time** that the agent arrives at S counts:

The first time state s appears: $N(S) \leftarrow N(S) + 1$
Total rewards update: $S(S) \leftarrow S(S) + G_t$
State s value: $V(s) = S(s)/N(s)$
When $N(s) \rightarrow \infty$, $V(s) \rightarrow v_\pi(s)$ so try as many episodes as you can to get as close as possible to the real state *s* value.

- **Incremental Monte-Carlo Updates**

For each state *St* in the episode, there is a reward *Gt*, and for every time St appears, the average value of the state, *V(St)* is calculated by the following formula:

$$N(S) \leftarrow N(S) + 1$$
$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(s_t)} [G_t - V(S_t)]$$

Temporal-Difference Learning

The Monte-Carlo reinforcement learning algorithm overcomes the difficulty of strategy estimation caused by an unknown model. However, a disadvantage is that the strategy can only be updated after the whole episode.

In other words, the Monte Carlo method does not make full use of the MDP learning task structure. Luckily, that's where the more efficient Temporal-Difference (TD) method comes in, making full use of the MDP structure.

Temporal-Difference Learning: A Combination of Deep Programming and Monte Carlo

As we know, the Monte Carlo method requires waiting until the end of the episode to determine V(St). The Temporal-Difference or TD method, on the other hand, only needs to wait until the next time step.

That is, at time *t + 1*, the TD method uses the observed reward Rt+1and immediately forms a **TD target** *R(t+1)+V(St+1)*, updating V(St) with **TD error** (which we'll define below).

Having addressed the shortcomings of Monte Carlo, we're ready to further discuss Temporal-Difference learning. The famous Q-learning algorithm falls within the TD method, but let's start with the simplest one, called *TD (0)*.

TD (0)

In Monte-Carlo, Gt is an actual return from the complete episode. Now, if we replace Gt with an estimated return *R(t+1)+V(St+1)*, this is what *TD(0)* would look like:

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

Where:

- *R(t+1)+V(St+1)* is called **TD target value**
- *R(t+1)+V(St+1)- V(St)* is called **TD error**.

MC uses accurate return Gt to update value, while TD uses the Bellman Optimality Equation to estimate value, and then updates the estimated value with the target value.

Temporal-Difference Learning: TD(λ)

We mentioned that if we replace *Gt* in the MC-updated formula with an estimated return *Rt+1+V(St+1),* we can get *TD(0)*:

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

Where:

- *Rt+1+V(St+1)* is called **TD target value**
- *Rt+1+V(St+1)- V(St)* is called **TD error**

Now, we replace the **TD target value** with **Gt(),** we can have TD(λ). **Gt()**is generated as below:

$$\begin{aligned} G_t^{(1)} &= R_{t+1} + \gamma V(S_{t+1}) \\ G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\ G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^n V(S_{t+n}) \\ G_t^{(\lambda)} &= (1-\lambda) G_t^{(1)} + (1-\lambda)\lambda\, G_t^{(2)} + \ldots + (1-\lambda)\lambda^{n-1} G_t^{(n)} \\ &\approx [(1-\lambda) + (1-\lambda)\lambda + \ldots + (1-\lambda)\lambda^{n-1}] V(S_t) = V(S_t) \end{aligned}$$

So the *TD(λ)* formula is:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(\lambda)} - V(S_t)]$$

In which:

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

As discussed, Q-learning is a combination of Monte Carlo (MC) and Temporal Difference (TD) learning. With MC and *TD(0)* covered and *TD(λ)* now under our belts, we're finally ready to pull out the big guns!

Q-Learning

Q-Value formula:

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

From the above, we can see that Q-learning is directly derived from *TD(0)*. For each updated step, Q-learning adopts a greedy method: maxaQ (St+1, a).

This is the main difference between Q-learning and another TD-based method called Sarsa, which I won't explain in this series. But as an RL learner, you should know that Q-learning is not the only method based on TD.

An Example of How Q-Learning Works

Let's try to understand this better with an example:

You're having dinner with friends at an Italian restaurant and, because you've been here once or twice before, they want you to order. From experience, you know that the Margherita pizza and pasta Bolognese are delicious. So if you have to order ten dishes, experience might tell you to order five of each. But what about everything else on the menu?

In this scenario, you are like our "agent", tasked with finding the best combination of ten dishes. Imagine this becomes a weekly dinner; you'd probably start bringing a notebook to record information about each dish. In Q-learning, the agent collects **Q-values** in a **Q-table**. For the restaurant menu, you could think of these values as a score for each dish.

Now let's say your party is back at the restaurant for the third time. You've got a bit of information in your notebook now but you certainly haven't explored the whole menu yet. How do you decide how many dishes to order from your notes — which you know are good, and how many new ones to try?

This is where **ε-greedy** comes into play.

The ε-greedy Exploration Policy

In the above example, what happened in the restaurant is like our MDP (Markov Decision Process) and you, as our "agent" can only succeed in finding the best combination of dishes for your party if you explore it thoroughly enough.

So it is with Q-Learning: it can work only if the agent explores the MDP thoroughly enough. Of course, this would take an extremely long time. Can you imagine how many times you'd have to go back to the restaurant to try every dish on the menu in every combination?

This is why Q-learning uses the **ε-greedy policy**, which is ε degree "greedy" for the highest Q values and *1* — ε degree "greedy" for random exploration.

In the initial stages of training an agent, a random exploration environment (i.e. trying new things on the menu) is often better than a fixed behavior mode (i.e. ordering what you already know is good) because this is when the agent accumulates experience and fills up the Q-table.

Thus, it's common to start with a high value for ε, such as 1.0. This means the agent will spend 100% of its time exploring (e.g. using a random policy) instead of referring to the Q-table.

From there, the value of ε can be gradually decreased, making the agent more greedy for Q-values. For example, if we drop ε to 0.9, it means the agent will spend 90% of its time choosing the best strategy based on Q-table, and 10% of its time exploring the unknown.

The advantage of the *ε*-greedy policy, compared to a completely greedy one, is that it always keeps testing unknown regions of the MDP. Even when the target policy seems optimal, the algorithm never stops exploring: it just keeps getting better and better.

There are various functions for exploration, and many defined exploration policies can be found online. Do note that not all exploration policies are expected to work for both discrete and continuous action spaces.