Notes

Subject: Artificial Intelligence

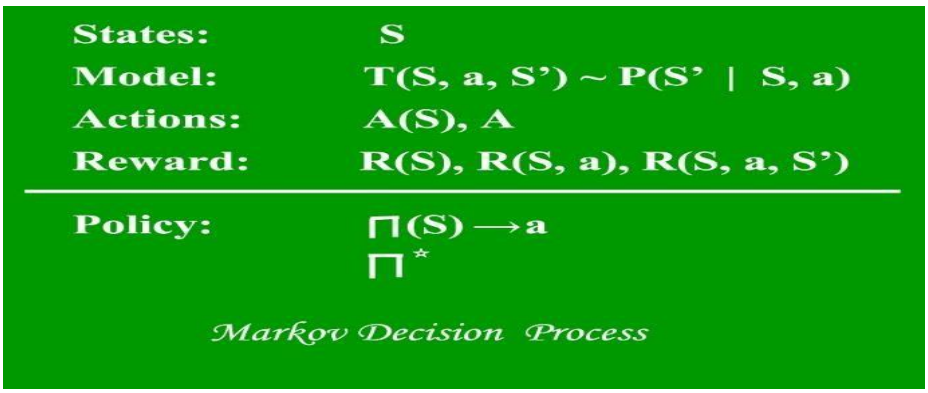
Unit- 4 Markov Decision Process

Markov Decision Process

Reinforcement Learning is a type of Machine Learning. It allows machines and software agents to automatically determine the ideal behavior within a specific context, in order to maximize its performance. Simple reward feedback is required for the agent to learn its behaviour; this is known as the reinforcement signal.
There are many different algorithms that tackle this issue. As a matter of fact, Reinforcement Learning is defined by a specific type of problem, and all its solutions are classed as Reinforcement Learning algorithms. In the problem, an agent is supposed to decide the best action to select based on his current state. When this step is repeated, the problem is known as a **Markov Decision Process**.

A **Markov Decision Process (MDP)** model contains:

- A set of possible world states S.
- A set of Models.
- A set of possible actions A.
- A real valued reward function R(s,a).
- A policy the solution of **Markov Decision Process**.



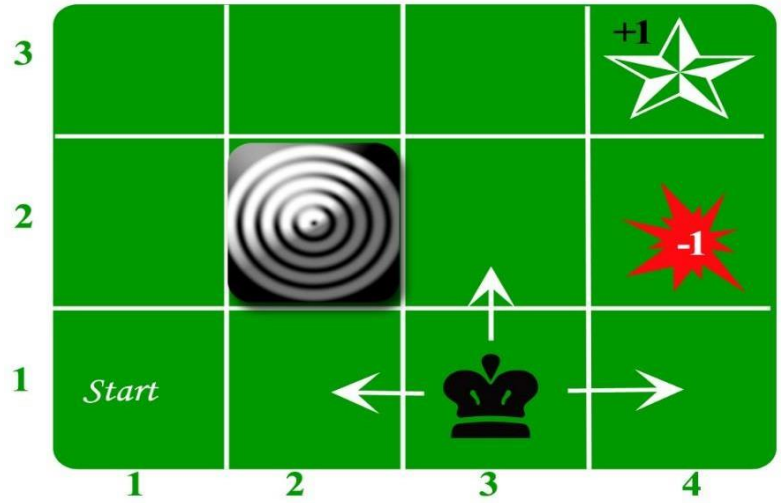
What is a State?
A **State** is a set of tokens that represent every state that the agent can be in.

What is a Model?
A **Model** (sometimes called Transition Model) gives an action's effect in a state. In particular, T(S, a, S') defines a transition T where being in state S and taking an action 'a' takes us to state S' (S and S' may be same). For stochastic actions (noisy, non-deterministic) we also define a probability P(S'|S,a) which represents the probability of reaching a state S' if action 'a' is taken in state S. Note Markov property states that the effects of an action taken in a state depend only on that state and not on the prior history.

What are Actions?
An **Action** A is set of all possible actions. A(s) defines the set of actions that can be taken being in state S.

What is a Reward?
A **Reward** is a real-valued reward function. R(s) indicates the reward for simply being in the state S. R(S,a) indicates the reward for being in a state S and taking an action 'a'. R(S,a,S') indicates the reward for being in a state S, taking an action 'a' and ending up in a state S'.

What is a Policy?
A **Policy** is a solution to the Markov Decision Process. A policy is a mapping from S to a. It indicates the action 'a' to be taken while in state S.
Let us take the example of a grid world:



An agent lives in the grid. The above example is a 3*4 grid. The grid has a START state (grid no 1,1). The purpose of the agent is to wander around the grid to finally reach the Blue Diamond (grid no 4,3). Under all circumstances, the agent should avoid the Fire grid (orange color, grid no 4,2). Also, the grid no 2,2 is a blocked grid, it acts like a wall hence the agent cannot enter it.
The agent can take any one of these actions: **UP, DOWN, LEFT, RIGHT**
Walls block the agent path, i.e., if there is a wall in the direction the agent would have taken, the agent stays in the same place. So, for example, if the agent says LEFT in the START grid he would stay put in the START grid.
First Aim: To find the shortest sequence getting from START to the Diamond. Two such sequences can be found:

- **RIGHT RIGHT UP UP RIGHT**
- **UP UP RIGHT RIGHT RIGHT**

Let us take the second one (UP UP RIGHT RIGHT RIGHT) for the subsequent discussion.
The move is now noisy. 80% of the time the intended action works correctly. 20% of the time the action agent takes causes it to move at right angles. For example, if the agent says UP the probability of going UP is 0.8 whereas the probability of going LEFT is 0.1 and probability of going RIGHT is 0.1 (since LEFT and RIGHT is right angles to UP).
The agent receives rewards each time step: -

- Small reward each step (can be negative when can also be term as punishment, in the above example entering the Fire can have a reward of -1).
- Big rewards come at the end (good or bad).

- The goal is to Maximize sum of rewards.

Markov Property

A state S_t is *Markov* if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$$

Markov Property

Assume that a Robot was seated on a chair, it stood up and put its right foot forward. So currently, it is standing with its right foot forward (this is its current state).

Now, according to the Markov Property, the current state of the Robot depends only on its immediate previous state (or the previous timestep) i.e., the state it was in when it stood up. And evidently, it doesn't depend on the state where it was sitting on the chair. Similarly, its next state depends only on its current state.

Formally, ***for a* state *S_t to be Markov****, the probability of the next state* ***S_(t+1)*** *being* **s'** *should only be dependent on the current state* ***S_t* = s_*t*,** *and not on the rest of the past states* $S_1 = s_1$ **,** $S_2 = s_2$ **, …**

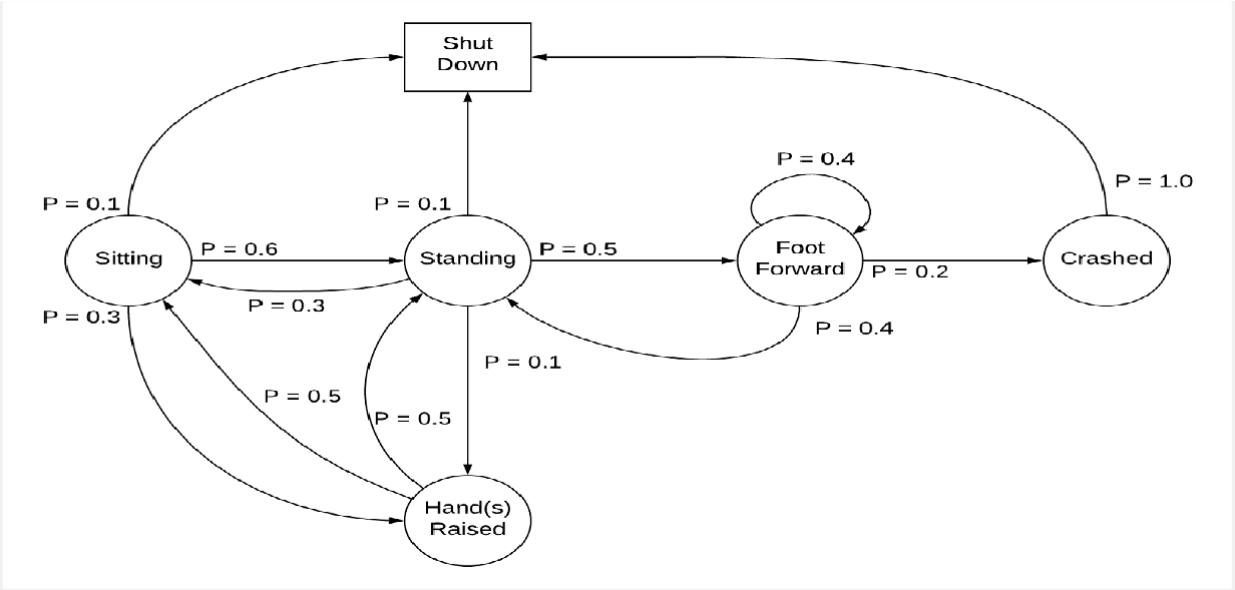
Markov Process or Markov Chain

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

State Transition Probability

A Markov Process is defined by ***(S, P)*** where ***S*** are the states, and ***P*** is the state-transition probability. It consists of a sequence of **random** states S_1 **,** S_2 **, …** where all the **states obey the Markov Property**.

The state transition probability or ***P_ss'*** is the probability of jumping to a state ***s'*** from the current state ***s***.



A Sample Markov Chain for the Robot Example

To get an intuition of the concept, consider the figure above. Sitting, Standing, Crashed, etc. are the states, and their respective state transition probabilities are given.

Markov Reward Process (MRP)

$$\mathcal{P}_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s\right]$$

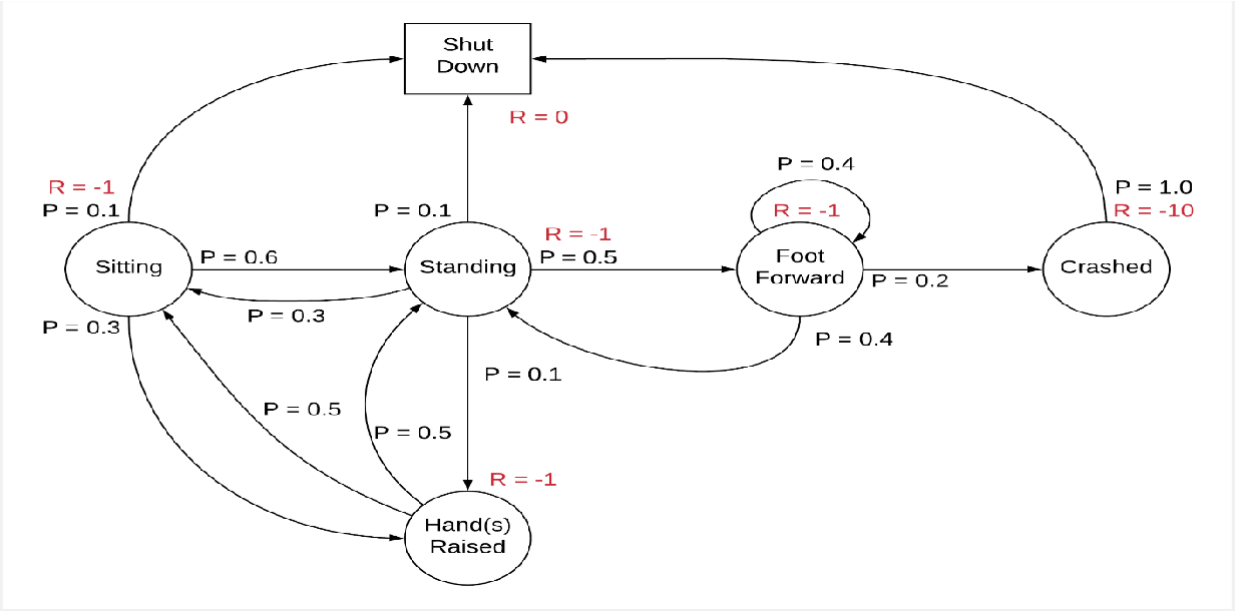
$$\mathcal{R}_s = \mathbb{E}\left[R_{t+1} \mid S_t = s\right]$$

State Transition Probability and Reward in an MRP

An MRP is defined by ***(S, P, R, γ)***, where ***S*** are the states, ***P*** is the state-transition probability, ***R_s*** is the reward, and ***γ*** is the discount factor (will be covered in the coming sections).

The state reward ***R_s*** is the **expected reward** over all the possible states that one can transition to from state ***s***. This reward is received for being at the state ***S_t***. **By convention**, it is said to be received after the agent leaves the state and hence, regarded as ***R_(t+1)***.

For example:



A Simple MRP Example

Markov Decision Process (MDP)

$$\mathcal{P}^{a}_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$$

$$\mathcal{R}^{a}_{s} = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$$

State Transition Probability and Reward in an MDP

An MDP is defined by ***(S, A, P, R, γ)***, where ***A*** is the set of actions. It is essentially **MRP with actions.** Introduction to actions elicits a notion of control over the Markov Process, i.e., previously, the state transition probability and the state rewards were more or less stochastic (random). However, now the rewards and the next state also depend on what action the agent picks. Basically, the agent can now control its own fate (to some extent).

Now we will discuss how to use MDPs to address RL problems.

Return (*G_t*)

The *return* G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Return

Rewards are temporary. Even after picking an action that gives a decent reward, we might be missing on a greater total reward in the long-run. This long-term total reward is the **Return**. However, in practice, we consider discounted Returns.

Discount (γ)

The variable $\gamma \in [0, 1]$ in the figure is the discount factor. The intuition behind using a discount is that there is no certainty about the future rewards; i.e., as important it is to consider the future rewards to increase the Return, it is also equally important to limit the contribution of the future rewards to the Return (Since you can't be 100% sure of the future).

And also because using a discount is mathematically convenient.

Policy (π)

A *policy* π is a distribution over actions given states,

$$\pi(a|s) = \mathbb{P}[A_t = a \mid S_t = s]$$

Policy

As mentioned earlier, a policy defines the thought behind making a decision (picking an action). It defines the behavior of an RL agent.

Formally, a policy is a **probability distribution** over the set of actions ***a***, given the current state ***s*** i.e., it gives the probability of picking an action ***a*** at state ***s***.

Value Functions

A value function is the long-term value of a state or an action i.e., the expected Return over a state or an action. **This is something that we are actually interested in optimizing.**

State Value Function (for MRP)

$$v(s) = \mathbb{E}[G_t \mid S_t = s]$$

State Value Function for an MRP

The state value function ***v(s)*** is the expected Return starting from state ***s.***

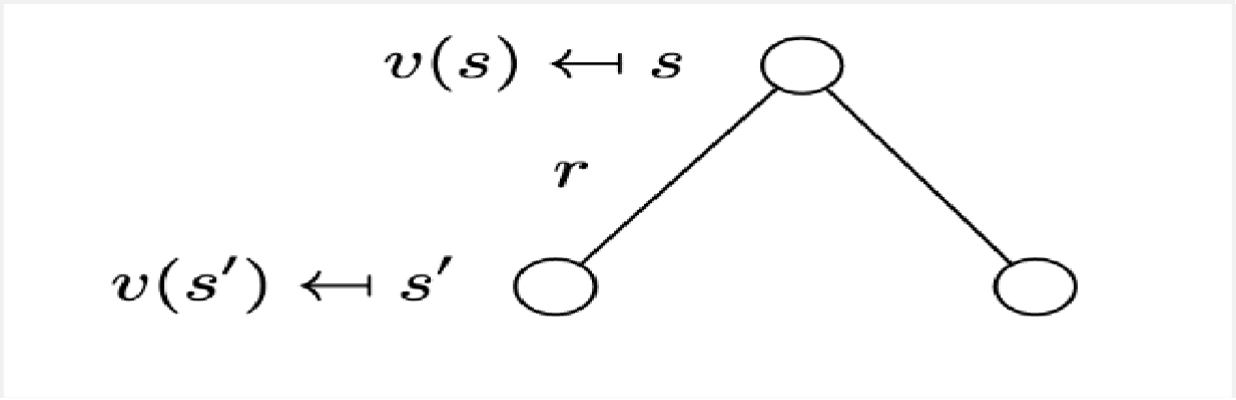
Bellman Expectation Equation (for MRP)

The Bellman Equation gives a standard representation for value functions. It essentially decomposes the value function into two components:

1. The immediate reward ***R_(t+1)***
2. Discounted value of the future state ***γ.v(S_(t+1))***

$$
\begin{aligned}
v(s) &= \mathbb{E}\left[G_t \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \ldots\right) \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]
\end{aligned}
$$

Solution for the Bellman Equation of the State Value Function



Intuition on Bellman Equation

The agent can transition from the current state ***s*** to some state ***s'***. Now, the state value function is basically the **expected value of Returns over all *s'***. Now, using the same definition, we can recursively substitute the Return of the next state ***s'*** with the value function of ***s'***. This is exactly what the Bellman Equation does:

$$
v(s) = \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]
$$

The Bellman Equation

Now let's solve this equation:

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

Solution for the Value Function

So, since expectation is distributive, we can solve for both ***R_(t+1)*** and ***v(s')*** separately. We have already seen that the expected value of ***R_(t+1)*** over ***S_t=s*** is the state reward ***R_s.*** And the expectation of ***v(s')*** over all ***s'*** is taken by the definition of Expected Value.

Another way of saying this would be— the **state reward is the constant value that we are anyway going to receive for being at the state *s***. And the **other term is the average state value over all *s'***.

State Value Function (for MDP)

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

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

State Value Function for an MDP

This is similar to the value function for MRP, but there is a small difference that we'll see shortly.

Action Value Function (for MDP)

The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right]$$

Action Value Function for an MDP

MDPs introduce control in MRPs by considering actions as the parameter for state transition. So, it is **necessary to evaluate actions along with states.** For this, we define action value functions that essentially give us the expected Return over actions.

State value functions and action value functions are closely related. We'll see how in the next section.

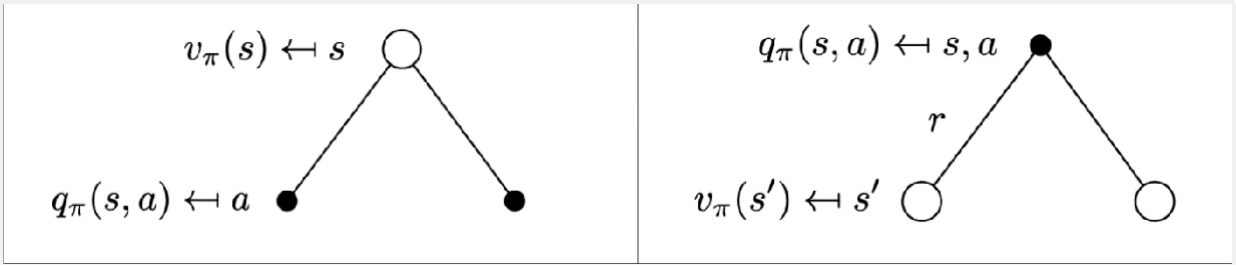
Bellman Expectation Equation (for MDP)

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

$$q_\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right]$$

Bellman Expectation Equations for an MDP

Since we know the basics of the Bellman Equation now, we can directly jump to the solution of this equation and see how this differs from the Bellman Equation for MRPs:



Intuition on Bellman Equation

Note that we represent the **states using circles** and **actions using dots**; both the diagrams above are a different level view of the same MDP, left being the 'state-centric' view and right being the 'action-centric' view.

Let's first understand the figure:

- **Circle to dot:** The agent is in a state ***s***; it picks an action ***a*** according to the policy. For eg: say we're training the agent to play chess. One time step is equivalent to one whole move (one white and one black move resp.). So in this part of the transition, the agent picks an action (makes a move). This part is completely controllable by the agent as it gets to pick the action.
- **Dot to Circle:** The environment acts on the agent and sends it to a state based on the transition probability. For eg: continuing the chess-playing agent example, this is the part of the transition where the opponent makes a move. After both moves, we call it a complete state transition. This part is not controllable by the agent as it cannot control how the environment acts, just its own behavior.

Now we treat these as two individual mini transitions:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a)$$

Solution for the State Value Function

Since we have a state to action transition, we take the expected action value over all the actions.

And this completely satisfies the Bellman Equation as the same is done for the action value function:

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

Solution for the Action Value Function

i.e., we can substitute this equation in the state value function to obtain the value in terms of recursive state value functions (and vice versa) similar to MRPs:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')$$

Substituting the Action Value Function in the State Value Function and vice versa

Optimal Value Functions

$$v_*(s) = \max_\pi v_\pi(s)$$

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

Optimal State and Action Value Functions

Say, we obtain the value for all the states/actions of an MDP for all possible patterns of actions that can be picked, i.e., all the policies. Then, we can simply pick the policy with the highest value for the states and actions. The equations above represent this exact thing.

If we obtain q∗(s, a), the problem is solved.

We can simply assign probability 1 for the action that has the max value for q∗ and 0 for the rest of the actions for all given states, i.e.

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_*(s, a) \\ 0 & \textit{otherwise} \end{cases}$$

Optimal Policy

Bellman Optimality Equation

$$v_*(s) = \max_a q_*(s, a)$$

Bellman Optimality Equation for Optimal State Value Function

Since we are anyway going to pick the action that yields the max q∗, we can simply assign this value as the optimal value function.

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Bellman Optimality Equation for Optimal Action Value Function

Nothing much can change for this equation as this is the part of the transition where the environment acts; hence, uncontrollable by the agent. However, since we are following the optimal policy, the state value function will be the optimal one.

Utility Functions in Artificial Intelligence

Utility functions are one of the elements of artificial intelligence (AI) solutions that are frequently mentioned but seldom discussed in details in AI articles. That basic AI theory has become an essential element of modern AI solutions. In some context, we could generalize the complete spectrum of AI applications as scenarios that involve a utility function that needs to be maximized by a rational agent. Before venturing that far, we should answer a more basic question: What is a utility function?

Utility functions are a product of Utility Theory which is one of the disciplines that helps to address the challenges of building knowledge under uncertainty. Utility theory is often combined with probabilistic theory to create what we know as decision-theoretic agents. Conceptually, a decision-theoretic agent is an AI program that can make rational decisions based on what it believes and what it wants. Sounds rational, right? :)

Ok, let's get a bit more practical. In many AI scenarios, agents don't have the luxury of operating in an environment in which they know the final outcome of every possible state. Those agents operate under certain degree of uncertainly and need to rely on probabilities to quantify the outcome of possible states. That probabilistic function is what we call Utility Functions.

Diving Into Utility Theory and MEU

Utility Theory is the discipline that lays out the foundation to create and evaluate Utility Functions. Typically, Utility Theory uses the notion of Expected Utility (EU) as a value that represents the average utility of all possible outcomes of a state, weighted by the probability that the outcome occurs. The other key concept of Utility Theory is known as the Principle of Maximum Utility(MEU) which states that any rational agent should choose to maximize the agent's EU.

The principle of MEU seems like an obvious way to make decisions until you start digging into it and run into all sorts of interesting questions. Why using the average utility after all? Why not to try to minimize the loss instead of maximizing utility? There are dozens of similar questions that challenge the principle of MEU. However, in order to validate the principle of MEU, we should go back to the laws of Utility Theory.

In AI, a utility function assigns values to certain actions that the AI system can take. An AI agent's preferences over possible outcomes can be captured by a function that maps these outcomes to a utility value; the higher the number the more that agent likes that outcome.

In economics, utility function is an important concept that measures preferences over a set of goods and services. Utility represents the satisfaction that consumers receive for choosing and consuming a product or service.

So, to create a "rational AI agent" that understands how to take the appropriate actions, the AI programmer must define the utility function across a variety of often-conflicting value dimensions. For example, increase financial value, while reducing operational costs and risks, while improving customer satisfaction and likelihood to recommend, while improving societal value and quality of life, while reducing environmental impact and carbon footprint

he agents use the utility theory for making decisions. It is the mapping from lotteries to the real numbers. An agent is supposed to have various preferences and can choose the one which best fits his necessity.

Utility scales and Utility assessments

To help an agent in making decisions and behave accordingly, we need to build a decision-theoretic system. For this, we need to understand the utility function. This process is known as **preference elicitation** In this, the agents are provided with some choices and using the observed preferences, the respected utility function is chosen. Generally, there is no scale for the utility function. But, a scale can be established by fixing the boiling and freezing point of water. Thus, the utility is fixed as:

U(S)=uT for best possible cases

U(S)= u? for worst possible cases.

A normalized utility function uses a utility scale with value uT=1, and u? =0. For example, a utility scale between uT and u? is given. Thereby an agent can choose a utility value between any prize Z and the standard lottery [p, u_; (1?p), u?]. Here, p denotes the probability which is adjusted until the agent is adequate between Z and the standard lottery.

Like in medical, transportation, and environmental decision problems, we use two measurement units: **micromort** or **QUALY**(quality-adjusted life year) to measure the chances of death of a person.

Money Utility

Economics is the root of utility theory. It is the most demanding thing in human life. Therefore, an agent prefers more money to less, where all other things remain equal. The agent exhibits a monotonic preference(more is preferred over less) for getting more money. In order to evaluate the more utility value, the agent calculates the **Expected Monetary Value**(EMV) of that particular thing. But this does not mean that choosing a monotonic value is the right decision always.

Multi-attribute utility functions

Multi-attribute utility functions include those problems whose outcomes are categorized by two or more attributes. Such problems are handled by multi-attribute utility theory.

Terminology used

- **Dominance:** If there are two choices say A and B, where A is more effective than B. It means that A will be chosen. Thus, A will dominate B. Therefore, multi-attribute utility function offers two types of dominance:
- **Strict Dominance:** If there are two websites T and D, where the cost of T is less and provides better service than D. Obviously, the customer will prefer T rather than D. Therefore, T strictly dominates D. Here, the attribute values are known.

- **Stochastic Dominance:** It is a generalized approach where the attribute value is unknown. It frequently occurs in real problems. Here, a uniform distribution is given, where that choice is picked, which stochastically dominates the other choices. The exact relationship can be viewed by examing the cumulative distribution of the attributes.
- **Preference Structure:** Representation theorems are used to show that an agent with a preference structure has a utility function as:

U(x1, . . . , xn) = F[f1(x1), . . . , fn(xn)],

where F indicates any arithmetic function such as an addition function.

Therefore, preference can be done in two ways :

- **Preference without uncertainty:** The preference where two attributes are preferentially independent of the third attribute. It is because the preference between the outcomes of the first two attributes does not depend on the third one.
    - **Preference with uncertainty:** This refers to the concept of preference structure with uncertainty. Here, the utility independence extends the preference independence where a set of attributes X is utility independent of another Y set of attributes, only if the value of attribute in X set is independent of Y set attribute value. A set is said to be mutually utility independent (MUI) if each subset is utility-independent of the remaining attribute.

Value Iteration

Value iteration is a method of computing an optimal MDP policy and its value.
Value iteration starts at the "end" and then works backward, refining an estimate of either Q^* or V^*. There is really no end, so it uses an arbitrary end point. Let V_k be the value function assuming there are k stages to go, and let Q_k be the Q-function assuming there are k stages to go. These can be defined recursively. Value iteration starts with an arbitrary function V_0 and uses the following equations to get the functions for $k+1$ stages to go from the functions for k stages to go:

$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a) (R(s,a,s')+ \gamma V_k(s'))$ for $k \geq 0$

$V_k(s) = \max_a Q_k(s,a)$ for $k>0$.

It can either save the *V[S]* array or the *Q[S,A]* array. Saving the *V* array results in less storage, but it is more difficult to determine an optimal action, and one more iteration is needed to determine which action results in the greatest value.

1: Procedure Value_Iteration(S,A,P,R,θ)

2: Inputs

3: S is the set of all states

```
4:              A is the set of all actions
5:              P is state transition function specifying P(s'|s,a)
6:              R is a reward function R(s,a,s')
7:              θ a threshold, θ>0
8:      Output
9:              π[S] approximately optimal policy
10:             V[S] value function
11:     Local
12:             real array V_k[S] is a sequence of value functions
13:             action array π[S]
14:     assign V_0[S] arbitrarily
15:     k ←0
16:     repeat
17:             k ←k+1
18:             for each state s do
19:                     V_k[s] = max_a ∑_s' P(s'|s,a) (R(s,a,s')+ γV_k-1[s'])
20:     until As |V_k[s]-V_k-1[s]| < θ
21:     for each state s do
22:             π[s] = argmax_a ∑_s' P(s'|s,a) (R(s,a,s')+ γV_k[s'])
23:     return π,V_k
```

Figure 9.14: Value iteration for MDPs, storing V

Figure 9.14 shows the value iteration algorithm when the V array is stored. This procedure converges no matter what is the initial value function V_0. An initial value function that approximates V^* converges quicker than one that does not. The basis for many abstraction techniques for MDPs is to use some heuristic method to approximate V^* and to use this as an initial seed for value iteration.

Example 9.26: Consider the 9 squares around the *+10* reward of Example 9.25. The discount is $\gamma=0.9$. Suppose the algorithm starts with $V_0[s]=0$ for all states s.
The values of V_1, V_2, and V_3 (to one decimal point) for these nine cells is

0	0	*-0.1*
0	10	*-0.1*
0	0	*-0.1*

0	*6.3*	*-0.1*
6.3	*9.8*	*6.2*
0	*6.3*	*-0.1*

4.5	*6.2*	*4.4*
6.2	*9.7*	*6.6*
4.5	*6.1*	*4.4*

After the first step of value iteration, the nodes get their immediate expected reward. The center node in this figure is the *+10* reward state. The right nodes have a value of *-0.1*, with the optimal actions being up, left, and down; each of these has a *0.1* chance of crashing into the wall for a reward of *-1*.
The middle grid shows V_2, the values after the second step of value iteration. Consider the node that is immediately to the left of the *+10* rewarding state. Its optimal value is to go to the right; it has a 0.7 chance of getting a reward of 10 in the following state, so that is worth 9 (10 times the discount of *0.9*) to it now. The expected reward for the other possible resulting states is *0*. Thus, the value of this state is *0.7×9=6.3*.
Consider the node immediately to the right of the *+10* rewarding state after the second step of value iteration. The agent's optimal action in this state is to go left. The value of this state is

	Prob	Reward		Future Value	
	0.7×(	0	+	0.9×10)	*Agent goes left*
+	0.1×(	0	+	0.9×-0.1)	*Agent goes up*
+	0.1×(	-1	+	0.9×-0.1)	*Agent goes right*
+	0.1×(	0	+	0.9×-0.1)	*Agent goes down*

which evaluates to 6.173.
Notice also how the *+10* reward state now has a value less than 10. This is because the agent gets flung to one of the corners and these corners look bad at this stage.
After the next step of value iteration, shown on the right-hand side of the figure, the effect of the +10 reward has progressed one more step. In particular, the corners shown get values that indicate a reward in 3 steps.
An applet is available on the book web site showing the details of value iteration for this example.

The value iteration algorithm of Figure 9.14 has an array for each stage, but it really only must store the current and the previous arrays. It can update one array based on values from the other.
A common refinement of this algorithm is **asynchronous value iteration**. Rather than sweeping through the states to create a new value function, asynchronous value iteration updates the states one at a time, in any order, and store the values in a single array. Asynchronous value iteration can store either the *Q[s,a]* array or the *V[s]* array. Figure 9.15 shows asynchronous value iteration when the *Q* array is stored. It

converges faster and uses less space than value iteration and is the basis of some of the algorithms for reinforcement learning. Termination can be difficult to determine if the agent must guarantee a particular error, unless it is careful about how the actions and states are selected. Often, this procedure is run indefinitely and is always prepared to give its best estimate of the optimal action in a state when asked.

1: **Procedure** Asynchronous_Value_Iteration(*S,A,P,R*)
2: **Inputs**
3: *S* is the set of all states
4: *A* is the set of all actions
5: *P* is state transition function specifying *P(s'|s,a)*
6: *R* is a reward function *R(s,a,s')*
7: **Output**
8: $\pi[s]$ approximately optimal policy
9: *Q[S,A]* value function
10: **Local**
11: real array *Q[S,A]*
12: action array $\pi[S]$
13: assign *Q[S,A]* arbitrarily
14: **repeat**
15: select a state *s*
16: select an action *a*
17: $Q[s,a] = \sum_{s'} P(s'|s,a) (R(s,a,s') + \gamma max_{a'} Q[s',a'])$
18: **until** termination
19: **for each** state *s* **do**
20: $\pi[s] = argmax_a Q[s,a]$
21: **return** π, Q

Figure 9.15: Asynchronous value iteration for MDPs

Asynchronous value iteration could also be implemented by storing just the *V[s]* array. In that case, the algorithm selects a state *s* and carries out the update:

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

Although this variant stores less information, it is more difficult to extract the policy. It requires one extra backup to determine which action *a* results in the maximum value. This can be done using

$\pi[s] = argmax_a \sum_{s'} P(s'|s,a) (R(s,a,s') + \gamma V[s'])$.

Policy Iteration

Once a policy, π, has been improved using V^{π} to yield a better policy, π', we can then compute $V^{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} V^{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} V^{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} V^*,$$

where $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in Figure 4.3. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

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

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s') \right]$
   If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop; else go to 2

Figure 4.3: Policy iteration (using iterative policy evaluation) for V^*. In the "$\arg\max$" step in 3, it is assumed that ties are broken in a consistent order.

Policy iteration often converges in surprisingly few iterations. This is illustrated by the example in Figure 4.2. The bottom-left diagram shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration.

Partially observable Markov decision process

A **partially observable Markov decision process** (**POMDP**) is a generalization of a Markov decision process (MDP). A POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state. Instead, it must maintain a sensor model (the probability distribution of different observations given the underlying state) and the underlying MDP. Unlike the policy function in MDP which maps the underlying states to the actions, POMDP's policy is a mapping from the observations (or belief states) to the actions.
The POMDP framework is general enough to model a variety of real-world sequential decision processes. Applications include robot navigation problems, machine maintenance, and planning under uncertainty in general. The general framework of Markov decision processes with imperfect information was described by Karl Johan Åström in 1965 [1] in the case of a discrete state space, and it was further studied in the operations research community where the acronym POMDP was coined. It was later adapted for problems in artificial intelligence and automated planning by Leslie P. Kaelbling and Michael L. Littman.
An exact solution to a POMDP yields the optimal action for each possible belief over the world states. The optimal action maximizes (or minimizes) the expected reward (or cost) of the agent over a possibly infinite horizon. The sequence of optimal actions is known as the optimal policy of the agent for interacting with its environment.

Formal definition [edit]

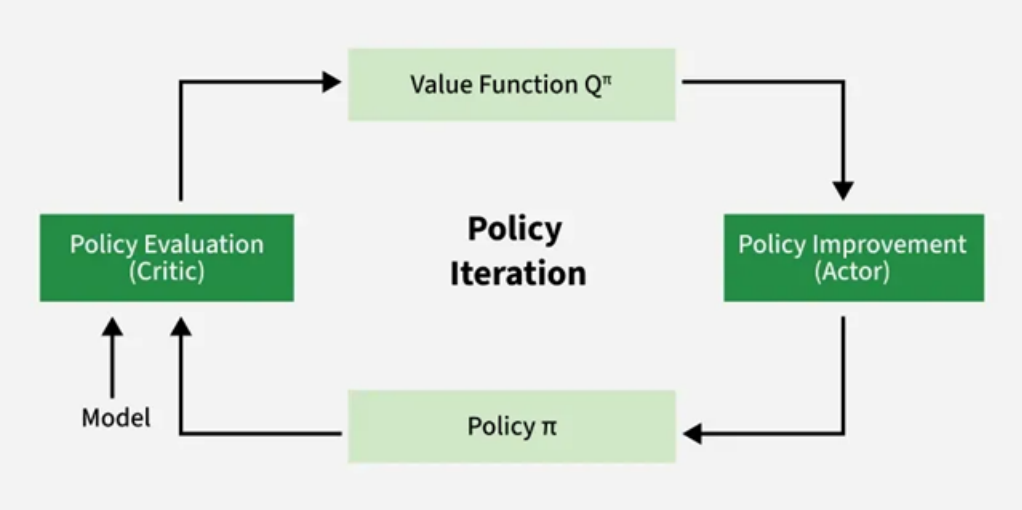
A discrete-time POMDP models the relationship between an agent and its environment. Formally, a POMDP is a 7-tuple $(S, A, T, R, \Omega, O, \gamma)$, where

- S is a set of states,
- A is a set of actions,
- T is a set of conditional transition probabilities between states,
- $R : S \times A \to \mathbb{R}$ is the reward function.
- Ω is a set of observations,
- O is a set of conditional observation probabilities, and
- $\gamma \in [0, 1]$ is the discount factor.

At each time period, the environment is in some state $s \in S$. The agent takes an action $a \in A$, which causes the environment to transition to state s' with probability $T(s' \mid s, a)$. At the same time, the agent receives an observation $o \in \Omega$ which depends on the new state of the environment, s', and on the just taken action, a, with probability $O(o \mid s', a)$ (or sometimes $O(o \mid s')$ depending on the sensor model). Finally, the agent receives a reward r equal to $R(s, a)$. Then the process repeats. The goal is for the agent to choose actions at each time step that maximize its expected future discounted reward: $E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where r_t is the reward earned at time t. The discount factor γ determines how much immediate rewards are favored over more distant rewards. When $\gamma = 0$ the agent only cares about which action will yield the largest expected immediate reward; when $\gamma = 1$ the agent cares about maximizing the expected sum of future rewards.

What is Policy Iteration?

Policy Iteration is another dynamic programming algorithm used to compute the optimal policy. It alternates between two steps:



Policy Iteration

- **Policy Evaluation**: For a given policy $\pi\pi$, the value function $V\pi(s)V_\pi(s)$ is computed using the Bellman Expectation Equation:

$$V\pi(s)=R(s,\pi(s))+\gamma\sum s'P(s'|s,\pi(s))V\pi(s')V_\pi(s)=R(s,\pi(s))+\gamma\sum_{s'}P(s'|s,\pi(s))V_\pi(s')$$

- **Policy Improvement**: Once the value function for the current policy is calculated the policy is updated to improve it by selecting the action that maximizes the expected return from each state:

$$\pi'(s)=arg\,max\,a[R(s,a)+\gamma\sum s'P(s'|s,a)V\pi(s')]\pi'(s)=\text{argmax}_a[R(s,a)+\gamma\sum_{s'}P(s'|s,a)V_\pi(s')]$$

This process repeats until the policy converges meaning it no longer changes between iterations.

.

Comparison Between Value Iteration and Policy Iteration

Feature	Value Iteration	Policy Iteration
Approach	Updates the value function iteratively until convergence.	Alternates between policy evaluation and policy improvement.
Convergence	Converges when the value function converges.	Converges when the policy stops changing.
Computational Cost	More computationally expensive per iteration due to full evaluation of all states.	Requires more iterations but may converge faster in terms of fewer iterations.
Policy Output	The policy is derived after the	The policy is updated during each

Feature	Value Iteration	Policy Iteration
	value function has converged.	iteration.
Speed of Convergence	May require many iterations for convergence, especially in large state spaces.	Tends to converge faster in practice, especially when the policy improves significantly at each iteration.
State Space	Typically suited for smaller state spaces due to computational complexity.	Can handle larger state spaces more efficiently.

When to Use Value Iteration and Policy Iteration

Use Value Iteration:

- When you have a small state space and can afford the computational cost of updating the value function for each state.
- When you want to compute the value function first and derive the policy later.

Use Policy Iteration:

- When you have a larger state space and want to reduce the number of iterations for convergence.
- When you can afford the computational cost of policy evaluation but want faster policy improvement.

Value Iteration is simpler and more direct in its approach and Policy Iteration often converges faster in practice by improving the policy iteratively. The choice between the two methods depends largely on the problem's scale and the computational resources available. In many real-world applications Policy Iteration may be preferred for its faster convergence especially in problems with large state spaces.