Chapter 2 - Mathematical foundations of reinforcement learning

In this chapter:

-- You learn about the core components of reinforcement learning.

-- You learn to represent sequential decision-making problems as reinforcement learning environments using a mathematical framework known as Markov Decision Processes.

-- You build from scratch environments that reinforcement learning agents learn to solve in later chapters.

Mankind's history has been a struggle against a hostile environment. We finally have reached a point where we can begin to dominate our environment [...]. As soon as we understand this fact, our mathematical interests necessarily shift in many areas from descriptive analysis to control theory.— Richard Bellman American applied mathematician an IEEE medal of honor recipient

You pick up this book and decide to read one more chapter despite having limited free time, a coach benches their best player for tonight's match ignoring the press criticism, a parent invests long hours of hard work and unlimited patience in teaching their child good manners. These are all examples of complex sequential decision-making under uncertainty.

The second word I used is "*sequential*," and this one refers to the fact that in many problems, there are *delayed consequences*. In the coaching example, again, let's say the coach benched their best player for a seemingly unimportant match midway through the season. But, what if the action of resting players lowers their morale and performance that only manifest in finals? In other words, what if the actual consequences are delayed? The fact is that assigning credit to your past decisions is challenging because we learn from sequential feedback.

Finally, the word "*uncertainty*" refers to the fact that we don't know the actual inner workings of the world to understand how our actions affect it; everything is left to our interpretation. Let's say the coach did bench their best player, but they got injured in the next match. Was the benching decision the reason the player got injured because the player got out of shape? What if the injury becomes a team motivation throughout the season, and the team ends up winning the final? So, again, was benching the right decision?
This *uncertainty* gives rise to the need for *exploration*. Finding the appropriate balance between *exploration* and *exploitation* is challenging because we learn from *evaluative* feedback.

In this chapter, you'll learn to represent these kinds of problems using a mathematical framework known as Markov Decision Processes (MDPs). The general framework of MDPs allows us to model virtually any complex sequential decision-making problem under uncertainty in a way that RL agents can interact with and learn to solve solely through experience

We'll dive deep into the challenges of learning from sequential feedback in chapter 3, then into the challenges of learning from evaluative feedback in chapter 4, then into the challenges of learning from feedback that is simultaneously sequential and evaluative in chapters 5 through 7, and then chapters 8 through 12 will add "complex" into the mix.

2.1 Components of reinforcement learning

Figure 2.1 The reinforcement learning interaction cycle

Miguel'sAnalogy

The parable of a Chinese farmer

There is an excellent parable that shows how difficult it is to interpret feedback that is simultaneously sequential, evaluative, and sampled. The parable goes like this:

A Chinese farmer gets a horse, which soon runs away. A neighbor says, “So, sad. That’s bad news.” The farmer replies, “Good news, bad news, who can say?”

The horse comes back and brings another horse with him. The neighbor says. “How lucky. That’s good news.” The farmer replies, “Good news, bad news, who can say?”

The farmer gives the second horse to his son, who rides it, then is thrown and badly breaks his leg. The neighbor says, “So sorry for your son. This is definitely bad news.” The farmer replies, “Good news, bad news, who can say?”

In a week or so, the emperor’s men come and take every healthy young man to fight in a war. The farmer’s son is spared.

So, good news or bad news? Who can say?

Interesting story, right? In life, it is challenging to know with certainty what are the long- term consequences of events and our actions. Often, we find misfortune responsible for our later good fortune, or our good fortune responsible for our later misfortune.

Even though this story could be interpreted as a lesson that “beauty is in the eye of the beholder,” in reinforcement learning, we assume there is a correlation between actions we take and what happens in the world. It’s just that it is so complicated to

understand these relationships, that it is difficult for humans to connect the dots with certainty. But, perhaps this is something that computers can help us figure out. Exciting, right?

Have in mind that when feedback is simultaneously evaluative, sequential, and sampled, learning is a hard problem. And, deep reinforcement learning is a computational approach to learning in these kinds of problems.

Welcome to the world of deep reinforcement learning!

2.1.1 Examples of problems, agents, and environments

Problem: you are training your dog to sit. **Agent**: the part of your brain that makes decisions. **Environment**: your dog, the treats, your dog's paws, the loud neighbor, etc. **Actions:** Talk to your dog. Wait for dog's reaction. Move your hand. Show treat. Give treat. Pet. **Observations:** Your dog is paying attention to you. Your dog is getting tired. Your dog is going away. Your dog sat on command.
Problem: your dog wants the treats you have. **Agent**: the part of your dog's brain that makes decisions. **Environment**: you, the treats, your dog's paws, the loud neighbor, etc. **Actions:** Stare at owner. Bark. Jump at owner. Try to steal the treat. Run. Sit. **Observa- tions:** Owner keeps talking loud at me. Owner is showing the treat. Owner is hiding the treat. Owner gave me the treat.
Problem: a trading agent investing in the stock market. **Agent**: the executing DRL code in memory and in the CPU. **Environment**: your Internet connection, the ma- chine the code is running on, the stock prices, the geopolitical uncertainty, other inves- tors, day-traders, etc. **Actions:** Sell n stocks of y company. Buy n stocks of y company.
Hold. **Observations:** Market is going up. Market is going down. There are economic tensions between two powerful nations. There is danger of war in the continent. A global pandemic is wreaking havoc in the entire world.
Problem: you are driving your car. **Agent**: the part of your brain that makes decisions. **Environment**: the make and model of your car, other cars, other drivers, the weather, the roads, the tires, etc. **Actions:** Steer by x, Accelerate by y. Break by z. Turn the head- lights on. Defog windows. Play music. **Observations:** You are approaching your desti- nation. There is a traffic jam on Main Street. The car next to you is driving recklessly. It's starting to rain. There is a police officer driving in front of you.

As you can see, problems can take many forms: from high-level decision-making problems that require long-term thinking and broad general knowledge, such as investing in the

stock market, to low-level control problems, in which geopolitical tensions don't seem to play a direct role, such as driving a car.

Also, you can represent a problem from multiple agents' perspective. In the dog training example, in reality, there are two agents each interested in a different goal and trying to solve a different problem.

Let's dig in some more. Let's zoom into each of these components independently.

2.1.2 The agent: The decision-maker

For now, the only important thing for you to know about agents is that *there are agents* and that *they are the decision-makers* in the RL big picture. They have internal components and processes of their own, and that is what makes each of them unique and good at solving specific problems.

If we were to zoom in, we would see that most agents have a three-step process: all agents have an *interaction* component, a way to gather data for learning, all agents *evaluate* their current behavior, and all agents *improve* something in their inner components that allows them to improve their overall performance (or at least attempt to improve).

Figure 2.2 The three internal steps that every reinforcement learning agent goes through

We'll continue discussing the inner-workings of agents starting with the next chapter. So, for now, let's spend some time discussing a way for *representing* environments, how they look, how we should model them, which is the goal of this chapter.

2.1.3 The environment: Everything else

The environment is represented by a set of *variables* related to the problem. The combination of all the possible values this set of variables can take is referred to as the state space. A state is a specific set of *values* the variables take at any given time

Agents may or may not have access to the *actual* environment's *state*; however, one way or another, agents can *observe* something from the environment. The set of variables the agent *perceives* at any given time is called an observation.

The combination of all possible values these variables can take is the observation space. Know that "*state*" and "*observation*" are terms used interchangeably in the RL community. This is because, very often, agents are allowed to see the internal state of the environment, but this is not always the case. In this book, I use state and observation interchangeably as well. But you need to know that there might be a difference between states and observations, even though the RL community often uses them interchangeably.

At every state, the environment makes available a set of actions the agent can choose from. Often the set of actions is the same for *all states*, but this is *not* required. The set of all actions in all states is referred to as the action space.

The agent attempts to *influence* the environment through these actions. The environment *may change states* as a response to the agent's action. The function that is responsible for this transition is called the transition function.

After a transition, the environment *emits* a new observation. The environment may also provide a reward *signal* as a response. The function responsible for this mapping is called the reward function. The set of *transition* and *reward function* is referred to as the model of the environment.

A Concrete
Example

The Bandit
Walk
environment

Let's make these concepts concrete with our first RL environment: I created this very simple environment for this book; I call it the Bandit Walk (BW).

BW is a very simple grid-world (GW) environment. GWs are a common type of environments for studying RL algorithms that are grids of any size. GWs can have any model (transition and reward functions) you can think of, and can make any kind of actions available.

But, they all commonly make move actions available to the agent: LEFT, DOWN, RIGHT, UP (or WEST, SOUTH, EAST, NORTH, which is more precise because the agent has no heading and usually has no visibility of the full grid, but cardinal directions can also be more confusing). And, of course, each action with the logical transitions; E.g. a left moves the agent left most of the time, etc. Also, they all tend to have a fully-observable discrete state and observation spaces (that is state == observation) with integers representing the cell id location of the agent. A "Walk" is a special case of grid-world environments with a single row. In reality, what I call a "Walk," is more commonly referred to as a "Corridor." But, in this book, I use the term "Walk" for all the grid-world environments with a single row.

The Bandit Walk (BW) is a walk with 3 states, but only 1 non-terminal state. Environments that have a single non-terminal state are called "bandit" environments. "Bandit" here is an analogy to slot machines, which are also known as "one-armed bandits"; they have one arm and, if you like gambling, can empty your pockets, just like a bandit would.

The BW environment has just 2 actions available: a left (action 0) and an right (action 1) action. BW has a deterministic transition function:

a left actions always moves the agent to the left, and

a right action always moves the agent to the right. The reward signal is a +1 when landing on the rightmost cell, 0 otherwise. The agent starts in the middle cell.

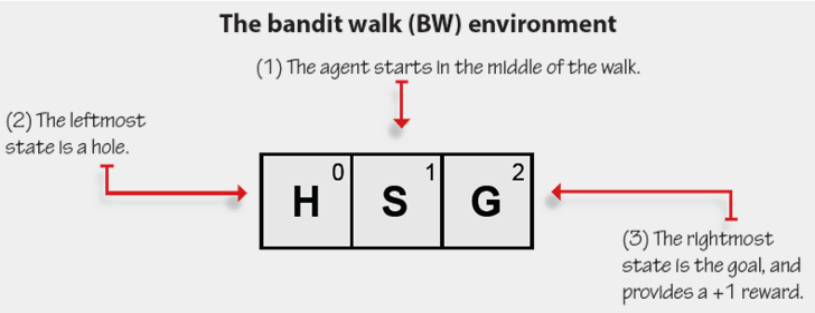


Figure 2.3 The bandit walk (BW) environment

We can also represent this environment in a table form:

Table 2.1

State	Action	Next state	Transition probability	Reward signal
0 (Hole)	0 (Left)	0 (Hole)	1.0	0
0 (Hole)	1 (Right)	0 (Hole)	1.0	0
1 (Start)	0 (Left)	0 (Hole)	1.0	0
1 (Start)	1 (Right)	2 (Goal)	1.0	+1
2 (Goal)	0 (Left)	2 (Goal)	1.0	0
2 (Goal)	1 (Right)	2 (Goal)	1.0	0

Interesting, right? Let's look at another simple example.

A Concrete Example

The Bandit Slippery Walk environment

OK, so how about we make this environment stochastic?

Let's say the surface of the walk is slippery and each action has 20% chance of sending the agent backwards. I call this environment the Bandit Slippery Walk (BSW).

BSW is still a one-row grid world, a walk, a corridor, with only left and right actions available.

So, again 3 states and 2 actions. The reward is the same as before, +1 when landing at the rightmost state (except when coming from the rightmost state - from itself), and 0 otherwise.

However, the transition function is different: 80% of the time the agent moves to the intended cell, 20% of time in the opposite direction.

A depiction of this environment would look as follows:

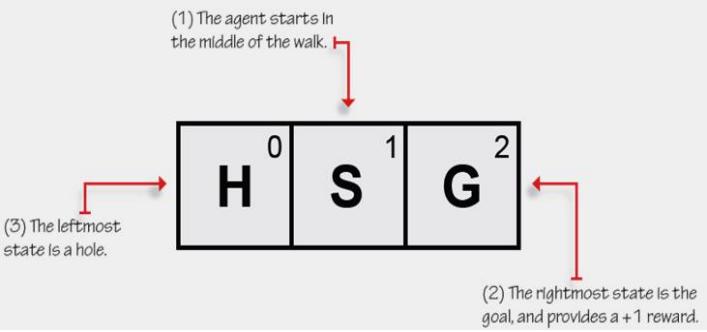


Figure 2.5 The Bandit Slippery Walk (BSW) environment

Identical to the BW environment! Interesting...

So, how do we know that the action effects are stochastic? How do we represent the "slippery" part of this problem?

The graphical and table representations can help us with that.

Figure 2. 6 Bandit Slippery Walk graph

Table 2.2

State	Action	Next state	Transition probability	Reward signal
0 (Hole)	0 (Left)	0 (Hole)	1.0	0
0 (Hole)	1 (Right)	0 (Hole)	1.0	0
1 (Start)	0 (Left)	0 (Hole)	0.8	0
1 (Start)	0 (Left)	2 (Goal)	0.2	+1
1 (Start)	1 (Right)	2 (Goal)	0.8	+1
1 (Start)	1 (Right)	0 (Hole)	0.2	0
2 (Goal)	0 (Left)	2 (Goal)	1.0	0
2 (Goal)	1 (Right)	2 (Goal)	1.0	0

Notice I didn't add the transition function to this table. That is because, while you can look at the code implementing the dynamics for some environments, other implementations are not easily accessible. For instance, the transition function of the Cart Pole environment is a small Python file defining the mass of the cart and the pole and implementing basic physics equations, while the dynamics of ATARI games, such as Pong, are hidden inside an ATARI emulator and the corresponding game-specific ROM file.

Notice that what we are trying to represent here is the fact that the environment "reacts" to the agent's actions in some way, perhaps even by ignoring the agent's actions. But at the end of the day, there is an internal process that is *uncertain* (except in this and next chapter). To represent the ability to interact with an environment in an MDP we need states, observations, actions, a transition and a reward function.

Figure 2.7 Process the environment goes through as a consequence of agent's actions

Table 2.3

Environment Description	Observation space	Observation sample	Action space	Action sample	Reward function
Hotter Colder: Guess a randomly selected number using hints.	Int range 0-3. 0 means no guess yet submitted, 1 means guess is lower than the target, 2 means guess is equal to the target and 3 means guess is higher than the target.	2	Float from -2000.0-2000.0. The float number the agent is guessing.	-909.37	The reward is the squared percentage of the way the agent has guessed toward the target.
Cart Pole: Balance a pole in a cart.	A 4-element vector with ranges: from [-4.8, -Inf, -4.2, -Inf] to [4.8, Inf, 4.2, Inf]. First element is the cart position, second is the cart velocity, third is pole angle in	[-0.16, -1.61, 0.17, 2.44]	Int range 0-1. 0 means push cart left, 1 means push cart right.	0	The reward is 1 for every step taken, including the terminatio n step.

	radians, fourth is the pole velocity at tip.				
Lunar Lander: Navigate a lander to its landing pad.	An 8-element vector with ranges: from [-Inf, -Inf, -Inf, -Inf, -Inf, -Inf, 0, 0] to [Inf, Inf, Inf, Inf, Inf, Inf, 1, 1]. First element is the x position, the second the y position, the third is the x velocity, the fourth is the y velocity, fifth is the vehicle's angle, sixth is the angular velocity, last two values are booleans indicating legs contact with the ground.	[0.36 , 0.23, -0.63, -0.10, -0.97, -1.73 , 1.0, 0.0]	Int range 0-3. No-op (do nothing), fire left engine, fire main engine, fire right engine.	2	Reward for landing is 200. There is reward for moving from the top to the landing pad, for crashing or coming to rest, for each leg touching the ground, and for firing the engines.

Pong: Bounce the ball past the opponent, and avoid letting the ball pass you.	A tensor of shape 210, 160, 3. Values ranging 0-255. Represents a game screen image.	[[[246, 217, 64], [55, 184, 230], [46, 231, 179], ..., [28, 104, 249], [25, 5, 22], [173, 186, 1]],...]]	Int range 0-5. Action 0 is No-op, 1 is Fire, 2 is up, 3 is right, 4 is left, 5 is down. Notice how some actions don't affect the game in any way. In reality the paddle can	3	The reward is a 1 when the ball goes beyond the opponent, and a -1 when your agent's paddle misses the ball.
Humanoid: Mak e robot run as fast as possible and not fall.	A 44-element (or more, depending on the implementation) vector. Values ranging from	[0.6, 0.08, 0.9, 0. , 0., 0., 0., 0., 0.045, 0., 0.47, ..., 0.32, 0., -0.22,..., 0.]	A 17-element vector. Values ranging from -Inf to Inf.	[-0.9, -0.06, 0.6, 0.6, 0.6, -0.06, -0.4, -0.9, 0.5, -0.2,	The reward is calculated based on forward motion with a small

	-Inf to Inf. Represents the positions and velocities of the robot's joints.		Represents the forces to apply to the robot's joints.	0.7, -0.9, 0.4, -0.8, -0.1, 0.8,-0.03]	

The interactions between the agent and the environment go on for several cycles. Each cycle is called a time step. A time step is a unit of time, which can be a millisecond, a second, 1.2563 seconds, a minute, a day, or any other period of time.

At each time step, the agent *observes* the environment, takes *action*, and receives a *new* observation and reward. Notice that, even though rewards can be negative values, they are still called *rewards* in the RL world. The set of the observation (or state), the action, the reward, and the new observation (or new state) is called an experience tuple.

The task the agent is trying to solve may or may not have a natural ending. Tasks that have a natural ending, such as a game, are called episodic tasks. Tasks that do not, such as learning forward motion, are called continuing tasks. The sequence of time steps from the *beginning* to the *end* of an *episodic task* is called an episode. Agents may take several time steps and

episodes to learn to solve a task. The *sum of rewards* collected in a single episode is called a return. Agents are often designed to *maximize the return*. A time step limit is often added to continuing tasks, so they become episodic tasks, and agents can maximize the return.

2.1.4 Agent-environment interaction cycle

For the rest of this chapter, we'll put aside the agent and the interactions, and we'll examine the environment and inner MDP in depth. In chapter 3, we'll pick back up the agent, but there will be *no interactions* because the agent won't need them as it'll have access to the MDPs. In chapter 4, we'll remove the agent's access to MDPs and *add interactions* back into the equation, but it'll be in *single-state environments (bandits)*. Chapter 5 is about *learning to estimate returns* in multi-state environments when agents have no access to MDPs. Chapter 6 and 7 are about *optimizing behavior*, which is the full reinforcement learning problem.

Chapters 5, 6 and 7 are about agents learning in environments where there is *no need for function approximation*. After that, the rest of the book is all about agents that use *neural networks for learning*.

2.2 MDPs: The engine of the environment

Let's build MDPs for a few environments as we learn about the components that make them up. We'll create Python dictionaries representing MDPs from descriptions of the problems. In the next chapter, we'll study algorithms for planning on MDPs. These methods can devise solutions to MDP and will allow us to find optimal solutions to all problems in this chapter.

The ability to build environments yourself is an important skill to have. However, often you find environments for which somebody else has already created the MDP. Also, often, the dynamics of the environments are hidden behind a simulation engine and are too complex to examine in detail, some dynamics are even inaccessible and hidden behind the real world. In reality, RL agents do not need to know the precise MDP of a problem to learn robust behaviors, but knowing *about* MDPs, in general, is crucial for *you* as agents are commonly designed with the assumption that an MDP, even if inaccessible, is running under the hood.

For the rest of this chapter, we'll put aside the agent and the interactions, and we'll examine the environment and inner MDP in depth. In chapter 3, we'll pick back up the agent, but there will be no interactions because the agent won't need them as it'll have access to the MDPs. In chapter 4, we'll remove the agent's access to MDPs and add interactions back into the equation, but it'll be in single-state environments (bandits). Chapter 5 is about learning to estimate returns in multi-state

environments when agents have no access to MDPs. Chapter 6 and 7 are about optimizing behavior, which is the full reinforcement learning problem.

2.2.1 States: Specific configurations of the environment

2.2 MDPs: The engine of the environment

Figure 2.9 State space: A set of sets

For the BW, BSW and FL environments, the state is composed of a *single variable* containing the *id* of the cell the agent is at any given time. The agent's location cell id is a **discrete** variable. But state variables can be of any kind, and the set of variables can be larger than one. We could have the euclidean distance, that would be a **continuous** variable and an infinite state space. E.g.: 2.124, 2.12456, 5.1, 5.1239458, and so on. We could also have multiple variables defining the state. For instance, the number of cells away from the goal in the x- and y-axis. That would be two variables representing a single state. Both variables would be discrete, therefore the state space finite. However, we could also have variables of mixed types, for instance, one could be discrete, another continuous, another boolean.

A Concrete Example

The Frozen Lake environment

This is another, more-challenging problem for which we will build an MDP in this chapter. This environment is called the Frozen Lake (FL).

FL is a simple grid-world (GW) environment. It also has discrete state and action spaces. However, this time, four actions are available, move LEFT, DOWN, RIGHT, or UP.

The task in the FL environment is very similar to the task in the BW and BSW environments: to go from a start location to a goal location, while avoiding falling into holes. The challenge is similar to the BSW, in that the surface of the FL environment is slippery, it is a frozen lake after all. But the environment itself is larger. Let's take a look at a depiction of the FL.

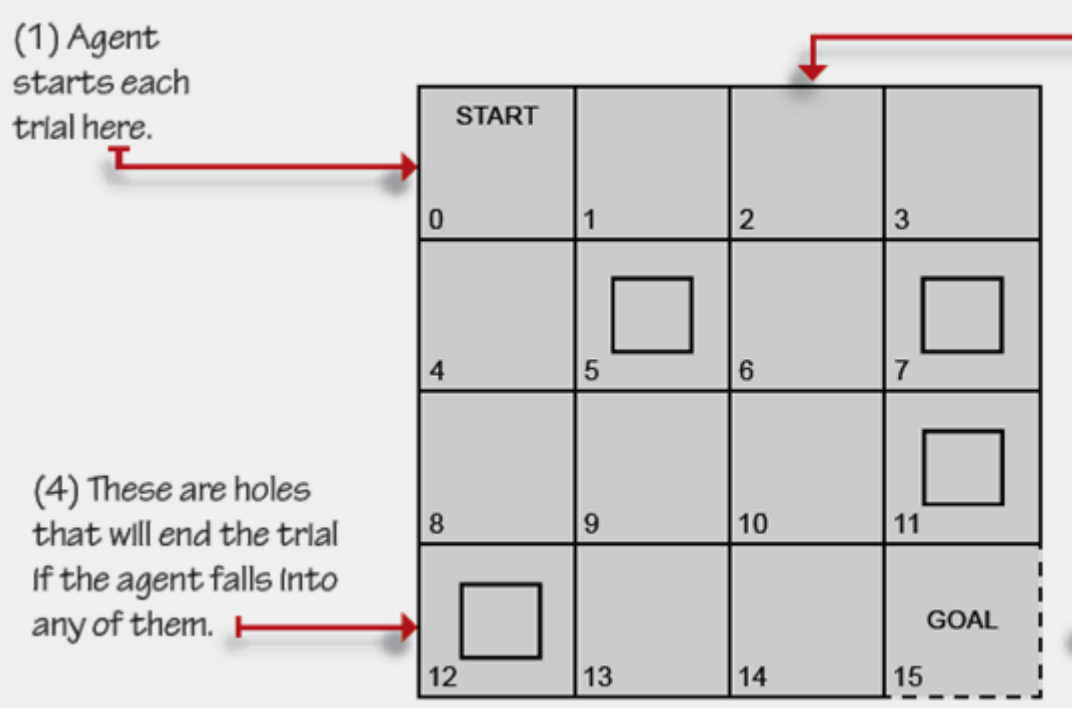


Figure 2.8 The Frozen Lake (FL) environment

The FL is a 4x4 grid (it has 16 cells, ids 0-15). The agent shows up in the START cell every new episode. Reaching the GOAL cell, which gives a +1 reward, anything else is a 0. Because the surface is slippery, the agent moves only a third of the time as intended. The other two- thirds is split evenly in orthogonal directions. For example, if the agent chooses to move DOWN,

there is a 33.3% chance it moves down, 33.3% moves LEFT and 33.3% RIGHT. There is a fence around the lake, so if the agent tries to move out of the grid world, it will bounce back to the cell from which it tried to move. There are four holes in the lake. If the agent falls into one of these holes, it's game over.

So, are you ready to start building a representation of these dynamics? We need a Python dictionary representing the MDP as described here. Let's start building the MDP.

2.2.1 States: Specific configurations of the environment

Figure 2.10 States in the FL contain a single variable indicating the id of the cell in which the agent is at any given time step

Figure 2.9 State space: A set of sets

Partially-Observable Markov Decision Processes (POMDPs), is a more general framework for modeling environments in which observations, which still depend on the internal state of the environment, are the only thing the agent can see instead of the state. Notice that for the BW, BSW and FL environments, we are creating an *MDP*, so the agent will be able to observe the internal state of the environment.

States must contain all the necessary variables needed to make them **independent** of all other states. In the FL environment, you only need to know the current state of the agent to tell its next possible states. That is, you don't need the history of states visited by the agent for anything. You know that from state 2 the agent can only transition to state 1, 3, 6, or 2 and this is true regardless of whether the agent's previous state was 1, 3, 6, or 2.

The probability of the next state, given the current state and action, is independent of the history of interactions. This memoryless property of MDPs is known as the Markov

property: the probability of moving from one state *s* to another state *s'* on two separate occasions, given the same action *a*, is the same regardless of all previous states or actions encountered before that point.

Figure 2.10 States in the FL contain a single variable indicating the id of the cell in which the agent is at any given time step

But why do you care about this? Well, in the environments we've explored so far it's not that obvious and it's not that important. But because most RL (and DRL) agents are designed to take advantage of the Markov assumption, you must make sure you feed your agent the necessary variables to make it hold as tightly as possible (completely keeping the Markov assumption is impractical, perhaps impossible).

For example, if you are designing an agent to learn to land a spacecraft, the agent must receive all variables that indicate *velocities* along with its *locations*. Locations alone are not sufficient to land a spacecraft safely, and because you must assume the agent is memoryless, you need to feed the agent more information than just its x, y, z coordinates away from the landing pad.

But, you probably know that acceleration is to velocity what velocity is to position: the derivative. You probably also know that you can keep taking derivatives beyond acceleration. So, to make the MDP completely Markovian, how deep do you have to go? This is more of an art than a science, the more variables you add, the longer it takes to train an agent, but the fewer variables, the higher the chance the information fed to the agent is not sufficient and the harder it is to learn anything useful. For the spacecraft example, often locations and velocities are adequate, and for grid-world environments, only the state id location of the agent is sufficient.

The set of all states in the MDP is denoted $S+$. There is a subset of S^+ called the set of starting or initial states, denoted *Si*. To begin interacting with an MDP, we draw a state from *Si* from a probability distribution. This distribution can be anything, but it must be fixed

throughout training, that is the probabilities must be the same from the first to the last episode of training and for agent evaluation

he Markov Property

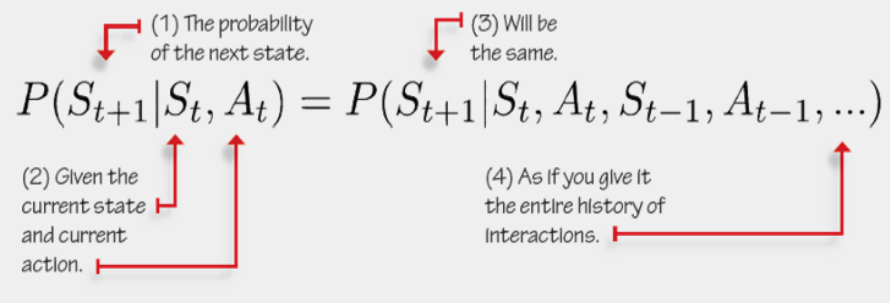


Figure 2.11

As expected? Yes. A terminal state is a special state: it must have all available actions transitioning, with probability 1, to itself, and these transitions must provide no reward. Note that I'm referring to the transitions *from* the terminal state, not *to* the terminal state.

It is very common the case that the end of an episode provides a non-zero reward. For instance, in a chess game you win, you lose or you draw, a logical reward signal would be +1, -1, and 0 respectively. But it is a compatibility convention which allows for all algorithms to converge to the same solution to make all actions available in a terminal state transition *from that terminal state to itself* with probability 1 and reward 0. Otherwise, you run the risk of infinite sums and algorithms that may not work altogether. Remember how the BW and BSW environments had these terminal states?

In the FL environment, for instance, there is only one starting state (which is state 0) and five terminal states (or five states that transition to a single terminal state, whichever you prefer). For clarity, I use the convention of multiple terminal states (5, 7, 11, 12 and 15) for the illustrations and code;

again, each terminal state is a separate terminal state.

Figure 2.12 States in the frozen lake environment

2.2.2 Actions: A mechanism to influence the environment

MDPs make available a set of actions *A* that depends on the state. That is, there might be some actions that are simply not allowed in a state—in fact, *A* is a function that takes a state as an argument, that is *A(s)*. This function returns the set of available actions for state *s*. If needed, you can define this set to be constant across the state space, that is all actions are available at every state. You can also set all transitions from a state-action pair to zero if you want to **deny** an action in a given state. You could also set all transitions from state *s* and action *a* to the same state *s* to denote action *a* as a **no-intervene** or **no-op** action.

Just as with the state, the action space may be finite or infinite, and the set of *variables* of a single action may contain more than one element and must be finite. However, unlike the number of state variables, the number of variables that compose an action may not be constant. The actions available in a state may change depending on that state. For simplicity, most environments are designed with the same number of actions in all states.

The environment makes the set of all available actions known in advance. Agents can select actions either **deterministically** or **stochastically**. And, this is different than saying the environment reacts deterministically or stochastically to agent's actions. Both are true statements, but I'm referring here to the fact that agents can either *select actions* from a look- up table or from a per-state probability distributions.

Figure 2.12 States in the frozen lake environment

2.2.2 Actions: A mechanism to influence the environment

2.2.3 Transition function: Consequences of agent actions

The way the environment changes as a response to actions is referred to as the state-transition probabilities or more simply the transition function and is denoted by *T(s, a, s')*. The transition function *T* maps a transition tuple *s*, *a*, *s'* to a probability; that is you pass in a state *s* an action *a* and a next state *s'*, and it'll return the corresponding probability of transition from state *s* to state *s'* when taking action *a*. You could also represent it as *T(s, a)* and return a *dictionary* with the next states for its keys and probabilities for its values.

Notice that *T* also describes a probability distribution *p(.|s,a)* determining how the system will evolve in an interaction cycle from selecting action *a* in state *s*. So, when integrating over the next states *s'*, as any probability distribution, the sum of these probabilities must equal one.

In the BW, BSW and FL environments, actions are singletons representing the direction the agent will attempt to move. In FL, there are four available actions in all states: UP, DOWN, RIGHT, or LEFT. There is one variable per action and the size of the action space is four.

The BW environment was deterministic, that is, the probability of the next state *s'* given the current state *s* and action *a* was always 1. There was always a single possible next state *s'*. The BSW and FL environments are stochastic, that is, the probability of the next state *s'* given the current state *s* and action *a* is less than 1. There are more than one possible next state *s'*.

2.2.3 Transition function: Consequences of agent actions

In the FL environment, we know that there is a 33.3% chance we will transition to the intended cell (state) and a 66.6% chance we will transition to orthogonal directions. There is also a chance we will bounce back to the state we are coming from if it is next to the wall.

For simplicity and clarity, I have added to the image below only the transition function for all actions of states 0, 2, 5, 7, 11, 12, 13, and 15 of the FL environment. This subset of states allows for the illustration of all possible transition without too much clutter.

Transition Function

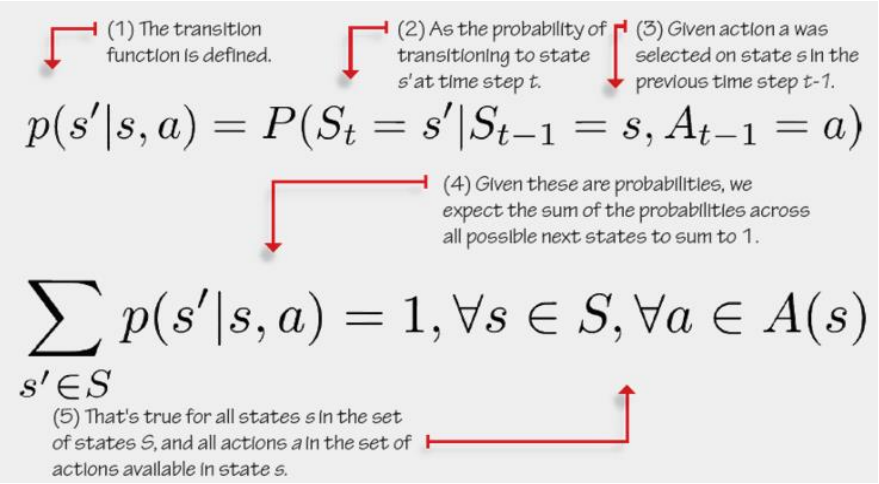


Figure 2.14

It might still be a bit confusing, but look at it this way: for consistency each action in non-terminal states has three separate transitions (some actions in corner states could be represented with only two, but again, let me be consistent): one to the intended cell and two to the cells in orthogonal directions.

2.2.4 Reward signal: Carrots and sticks

The reward function *R* maps a transition tuple *s, a, s'* to a **scalar**. The reward function gives a numeric signal of goodness to transitions. When the signal is positive, we can think of the reward as an **income** or a **reward**. Most problems have at least one positive signal—winning a chess match or reaching the desired destination, for example. But, rewards can also be negative, and we can see these as **cost, punishment**, or **penalty**. In robotics, adding a time step cost is a common practice because we usually want to reach a goal, but within a number of time steps. One thing to clarify is that whether positive or negative, the scalar coming out of the reward function is always referred to as *the reward*. RL folks are happy folks.

It is also important to highlight that while the reward function can be represented as *R(s,a,s')*, which is very explicit, we could also use *R(s,a)*, or even *R(s)*, depending on our needs. Sometimes rewarding the agent based on state is what we need, sometimes it makes more sense to use the action and the state. However, the most explicit way to represent the reward function is to use a state, action and next state triplet. With that, we can simply compute the marginalization over next states in *R(s,a,s')* to obtain *R(s,a)*, and the and the

Figure 2.15 The transition function of the Frozen Lake environment

R(s,a) or *R(s,a,s'),* and once we are on *R(s,a)* we can't recover *R(s,a,s').*

2.2.4 Reward signal: Carrots and sticks

In the FL environment, the reward function is simply +1 for *landing* in state 15, 0 otherwise. Again, for clarity to the following image, I've only added the reward signal to transitions that give a non-zero reward, landing on the final state (state 15.)

There are only three ways to land on 15. (1) Selecting the *RIGHT* action in state 14 will transition the agent with 33.3% chance there (33.3% to state 10 and 33.3% back to 14).

But, (2) selecting the *UP* and (3) the *DOWN* action from state 14 will unintentionally also transition the agent there with 33.3% probability for each action. See the difference between actions and transitions? It's interesting to see how stochasticity complicates things, right?

B(c,c) te Y(c,s,z'), pnz nzkk wo tvs nv C(a,c) wo znc'r errvoce C(a,s,a').

Show Me The Math

The Reward Function

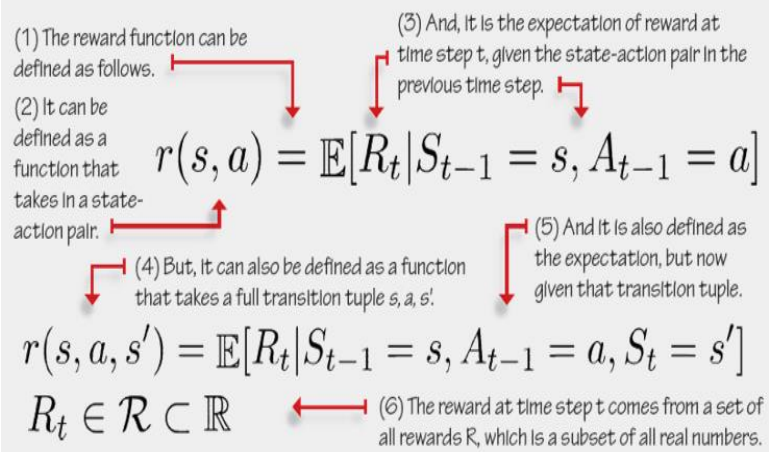


Figure 2.16

In the FL environment, the reward function is simply +1 for landing in state 15, 0 otherwise. Again, for clarity to the following image, I've only added the reward signal to transitions that give a non-zero reward, landing on the final state (state 15.)

2.2.5 Horizon: Time changes what's optimal

We can represent time in MDPs as well. A time step, also referred to as **epoch**, **cycle**, **iteration**, or even **interaction**, is a global clock syncing all parties and *discretizing* time. Having a clock gives rise to a couple of possible types of tasks.
An episodic task is a task in which there is a finite number of time steps, either because the clock stops or because the agent reaches a terminal state. There are also continuing tasks, which are tasks that go on forever; there are no terminal states, so there is an infinite number of time steps. In this type of task, the agent must be stopped manually.

Figure 2.17 Reward signal for states with non-zero reward transitions

On the other hand, an infinite horizon is when the agent doesn't have a predetermined time step limit, so the agent plan for an infinite number of time steps. Such task may still be episodic and therefore terminate, but from the perspective of the agent, its planning horizon is infinite. We

refer to this type of infinite planning horizon tasks as an indefinite horizon task. The agent plans for infinite, but interactions may stop at any time by the environment.

Table 2.4

State	Action	Next state	Transition probability	Reward signal
0	LEFT	0	0.33	0
0	LEFT	0	0.33	0
0	LEFT	4	0.33	0
0	DOWN	0	0.33	0
0	DOWN	4	0.33	0
0	DOWN	1	0.33	0
0	RIGHT	4	0.33	0
0	RIGHT	1	0.33	0
0	RIGHT	0	0.33	0
0	UP	1	0.33	0
0	UP	0	0.33	0
0	UP	0	0.33	0
1	LEFT	1	0.33	0
1	LEFT	0	0.33	0
1	LEFT	5	0.33	0
1	DOWN	0	0.33	0
1	DOWN	5	0.33	0
1	DOWN	2	0.33	0
1	RIGHT	5	0.33	0
1	RIGHT	2	0.33	0
1	RIGHT	1	0.33	0

2	LEFT	1	0.33	0
2	LEFT	2	0.33	0
2	LEFT	6	0.33	0
2	DOWN	1	0.33	0
...	...	...	...	...
14	DOWN	14	0.33	0
14	DOWN	15	0.33	1
14	RIGHT	14	0.33	0
14	RIGHT	15	0.33	1
14	RIGHT	10	0.33	0
14	UP	15	0.33	1
14	UP	10	0.33	0
...	...	...	...	...
15	LEFT	15	1.0	0
15	DOWN	15	1.0	0
15	RIGHT	15	1.0	0
15	UP	15	1.0	0

2.2.5 Horizon: Time changes what's optimal

2.2.6 Discount: The future is uncertain, value it less

Because of the possibility of infinite sequences of time steps in infinite horizon tasks, we need a way to discount the value of rewards over time; that is, we need a way to tell the agent that getting +1's is better sooner than later. So, we commonly use a positive real value less than one to exponentially discount the value of future rewards. The further into the future we receive the reward, the less valuable it is in the present.

This number is called the discount factor, or gamma. The discount factor adjusts the importance of rewards over time. The later we receive rewards, the less attractive they are to present calculations. Another important reason why the discount factor is commonly used is to reduce the variance of return estimates. Given that the future is uncertain, and that the later we look into the future, the more stochasticity we accumulate and the more variance our value estimates will have, the discount factor helps reducing the degree to which future rewards affect our value function estimates which stabilizes learning for most agents.

Figure 2.18 Effect of discount factor and time on the value of rewards

Interestingly, gamma is actually part of the MDP definition, the problem,

and not the agent. However, very often you'll find no guidance for the proper value of gamma to use for a given environment. Again, this is because gamma is also used as a hyperparameter for reducing variance, and therefore left for the agent to tune.

You can also use gamma as a way to give a sense of "urgency" to the agent. To wrap your head around that, imagine that I tell you I'll give you $1,000 once you finish reading this book, but I'll discount (gamma) that reward by 0.5 daily. Meaning, every day, I cut the value that I pay in half. You'll probably finish reading this book today. If I say gamma is 1, then it doesn't matter when you finish it, you still get the full amount.

2.2.6 Discount: The future is uncertain, value it less

Because of the possibility of infinite sequences of time steps in infinite horizon tasks, we need a way to discount the value of rewards over time; that is, we need a way to tell the agent that getting +1's is better sooner than later. So, we commonly use a positive real value less than one to exponentially discount the value of future rewards. The further into the future we receive the reward, the less valuable it is in the present.

2.2.7 Extensions to MDPs

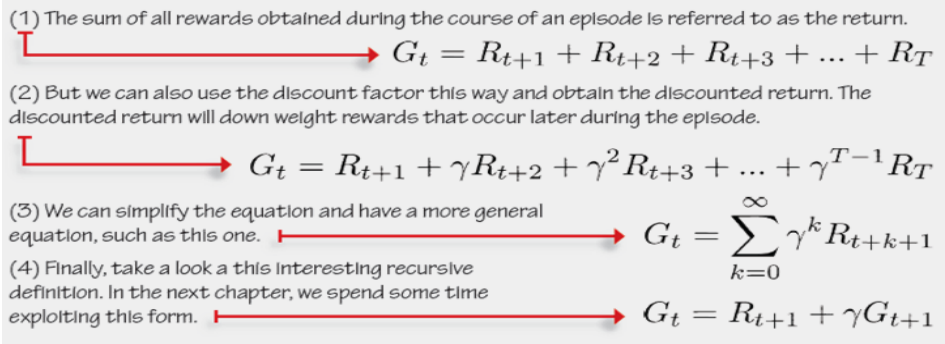
There are many extensions to the MDP framework as we just discussed. They allow us to target slightly different types of RL problems. The following list is not comprehensive, but it should give you an idea of how large the field is. Know that the acronym “MDPs” is often used to refer to all types of MDPs. We are currently looking only at the tip of the iceberg.

Partially-Observable Markov Decision Process (POMDP): When the agent cannot fully observe the environment state.
Factored Markov Decision Process (FMDP): Allows the representation of the transi- tion and reward function more compactly so that we can represent very large MDPs.
Continuous [Time|Action|State] Markov Decision Process: When either time, action, state or any combination of them are continuous.

Discount Factor

(Gamma)

Figure 2.19



2.2.7 Extensions to MDPs

Multi-Agent Markov Decision Process (MMDP): Allows the inclusion of multiple agents in the same environment.
Decentralized Markov Decision Process (Dec-MDP): Allows for multiple agents to collaborate and maximize a common reward.

· Factored Markov Decision Process (FMDP): Allows the representation of the transi- tion and reward function more compactly so that we can represent very large MDPs.

· Continuous [Time|Action|State] Markov Decision Process: When either time, action, state or any combination of them are continuous.

· Relational Markov Decision Process (RMDP): Allows the combination of probabilistic and relational knowledge.

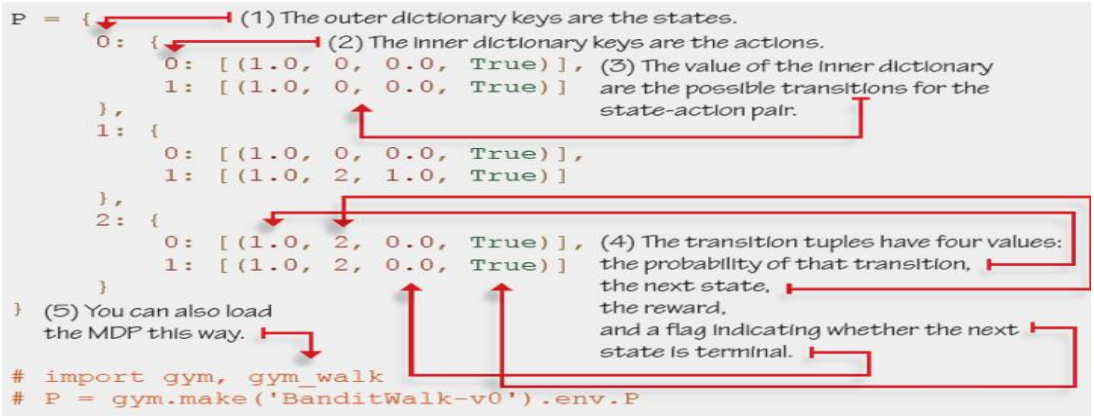
2.2.8 Putting it all together

Unfortunately, when you go out to the real world, you'll find many different ways that MDPs are defined. Moreover, some sources describe POMDPs and refer to them as MDPs without the full disclosure. All of this creates confusion to the newcomers, so I have a few points to clarify for you going forward. First, what you see above as Python code is not a complete MDP, but instead only the transition functions and reward signals. From these, we can easily infer the state and action spaces. These code snippets come from a few packages containing several environments I developed for the OpenAI Gym framework, and the FL environment is part of the OpenAI Gym core. Some of the additional components of an MDP that are missing from the dictionaries above, such as the initial state distribution $S\theta$ that comes from the set of initial state Si, are handled

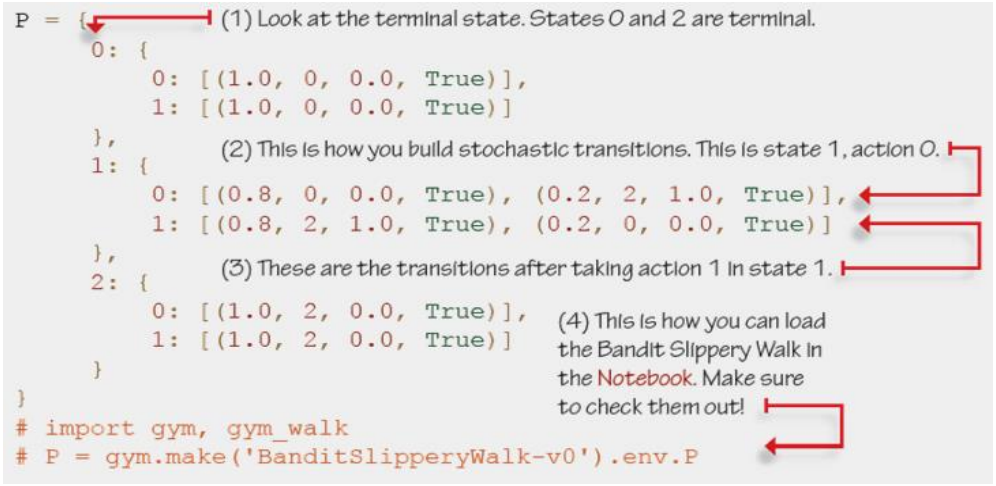
internally by the Gym framework and not shown here. Further, other components, such as the discount factor γ and the horizon H, are not shown in the dictionary above, and the OpenAI Gym framework doesn't provide them to you.

Like I said before, discount factors are commonly considered hyperparameters, for better or worse. And the horizon is very often assumed to be infinity.

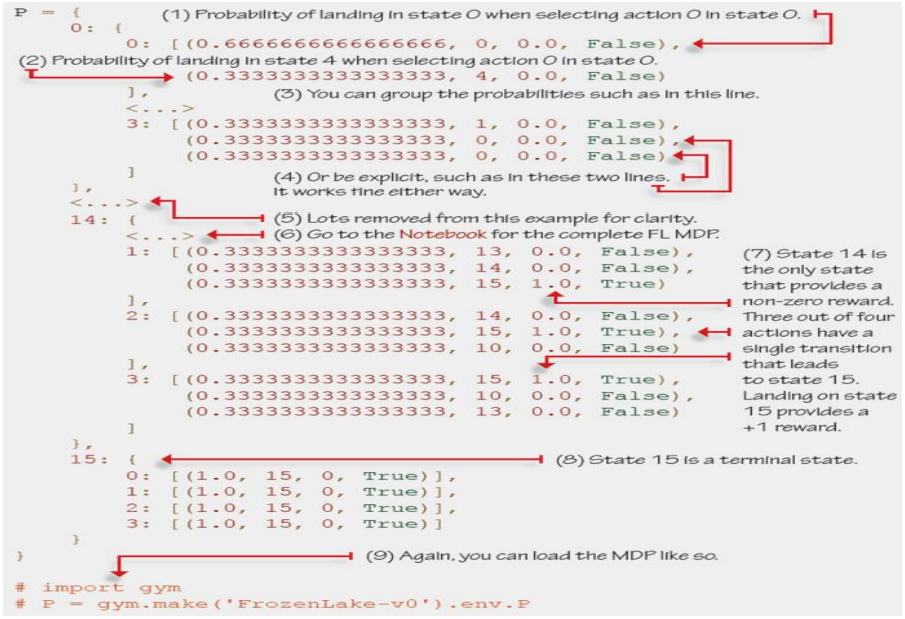
The
Bandit
Walk
(BW)
MDP



The Bandit Slippery Walk (BSW) MDP



he Frozen Lake (FL) MDP



2.2.8 Putting it all together

At the highest level, a reinforcement learning problem is about the interactions between an agent and the environment in which the agent exists. A large variety of issues can be modeled under this setting. Markov decision process is a mathematical framework for representing complex decision-making problems under uncertainty.

Markov decision processes (MDPs) are composed of a set of systems *states*, a set of per-state *actions*, a *transition function*, a *reward signal*, a *horizon*, a *discount factor*, and an *initial state distribution*. States describe the configuration of the environment. Actions allow agents to interact with the environment. The transition function tells how the environment evolves and reacts to the agents' actions. The reward signal encodes the goal to be achieved by the agent. The horizon and discount factor add a notion to time to the interactions.

The state space, the set of all possible states, can be infinite or finite. The number of variables that make up a single state, however, must be finite. States can be fully observable, but in a more general case of MDPs, a POMDP, the states are partially observable. This means the agent is not able to observe the full state of the system, but a noisy state instead, called an observation.

MDPs and POMDPs

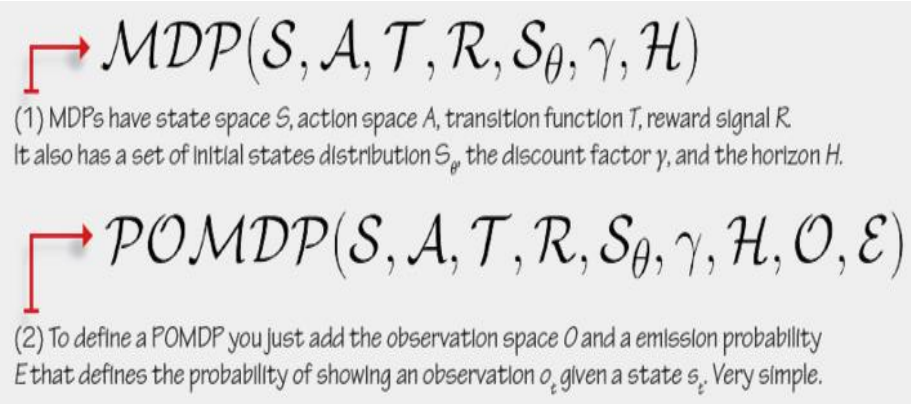


Figure 2.23

2.3 Summary

OK. I know this chapter is heavy on new terms, but that's its intent. The best summary for this chapter is on the previous page, more specifically, the definition of an MDP. Take another look at the last two equations and try to remember what each letter means. Once you do so, you can be assured that you got the necessary out of this chapter to proceed.

At the highest level, a reinforcement learning problem is about the interactions between an agent and the environment in which the agent exists. A large variety of issues can be modeled under this setting. Markov decision process is a mathematical framework for representing complex decision-making problems under uncertainty.

Markov decision processes (MDPs) are composed of a set of systems states, a set of per-state actions, a transition function, a reward signal, a horizon, a discount factor, and an initial state distribution. States describe the configuration of the environment. Actions allow agents to interact with the environment. The transition function tells how the environment evolves and reacts to the agents' actions. The reward signal encodes the goal to be achieved by the agent. The horizon and discount factor add a notion to time to the interactions.

The action space is a set of actions which can vary from state to state. However, the convention is to use the same set for all states. Actions can be composed with more than one variable, just like the states. Action variables may be discrete or continuous.

The transition function links a state (a next state) to a state-action pair, and it defines the probability of reaching that future state given the state-action pair. The reward signal, in its more general form, maps a transition tuple s, a, s' to scalar and it indicates the goodness of the transition. Both, the transition function and reward signal, define the model of the environment and assume to be stationary, meaning probabilities stay the same throughout

By now you:

· Understand the components of a reinforcement learning problem and how they inter- act with each other.

· Recognize Markov Decision Processes and know what they are composed from and how they work.

· Can represent sequential decision-making problems as MDPs.

The state space, the set of all possible states, can be infinite or finite. The number of variables that make up a single state, however, must be finite. States can be fully observable, but in a more general case of MDPs, a POMDP, the states are partially observable. This means the agent is not able to observe the full state of the system, but a noisy state instead, called an observation.