



Reinforcement Learning



Anant Nawalgaria
Machine Learning Specialist,
Google Cloud



Alex Erfurt
Machine Learning Specialist,
Google Cloud



Welcome to reinforcement learning, one of the modules that is part of the Recommendation Systems course.

Agenda

Introduction to reinforcement learning (RL)

RL framework and workflow

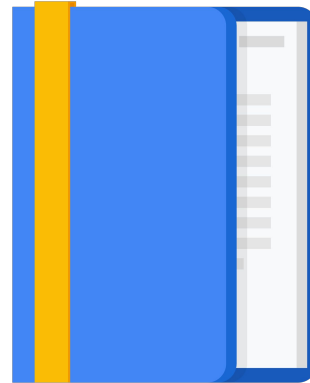
Model-based and model-free RL

Value-based RL

Policy-based RL

Contextual bandits

Applications of RL



In this module, we will discuss few reinforcement learning topics including:

- An introduction to reinforcement learning (RL)
- The RL learning framework and workflow
- Model-based and model-free RL
- Value-based RL
- Policy-based RL
- Contextual bandits
- Applications of RL

Objectives

- 1 Explain how reinforcement learning fits within the different machine learning types.



By the end of this module, you will be able to:

- Explain how reinforcement learning fits within the different machine learning types. We'll be comparing three different types including:
 - Supervised
 - Unsupervised
 - Reinforcement learning

Objectives

- 1 Explain how reinforcement learning fits within the different machine learning types.
- 2 Define the primary branches and useful types of reinforcement learning.



- Define the primary branches and types of RL.
 - Model-based methods
 - Model-free methods (including value-based, policy-based, and contextual bandits)

Objectives

- 1 Explain how reinforcement learning fits within the different machine learning types.
- 2 Define the primary branches and useful types of reinforcement learning.
- 3 Describe the framework and workflow to solve a common reinforcement learning problem.



- Describe the framework and workflow to solve a common reinforcement learning problem.

Objectives

- 1 Explain how reinforcement learning fits within the different machine learning types.
- 2 Define the primary branches and useful types of reinforcement learning.
- 3 Describe the framework and workflow to solve a common reinforcement learning problem.
- 4 Identify use cases where RL is the ideal approach to solving an ML problem and where other methods are preferable to RL.



And finally,

- Identify use cases where RL is the ideal approach to solving an ML problem and where other methods are preferable to RL.

Agenda

Introduction to reinforcement learning (RL)

RL framework and workflow

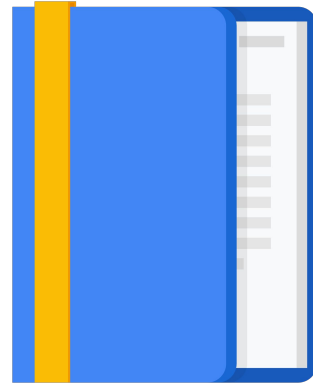
Model-based and model-free RL

Value-based RL

Policy-based RL

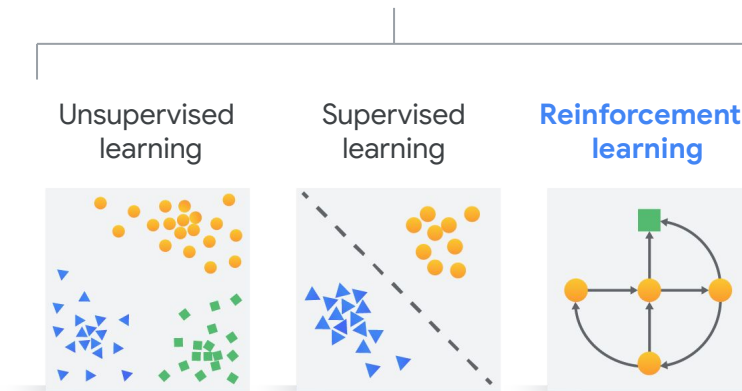
Contextual bandits

Applications of RL



Before we can explore the details, we'll talk about reinforcement learning in general.

Machine learning types



Recall from an earlier module that there are three types of common machine learning approaches:

- 1) Unsupervised learning, where a learning system establishes a model for data distribution based on unlabeled examples.
- 2) Supervised learning, where a learning system learns a latent map based on labeled examples.
- 3) reinforcement learning, where a decision-making system is trained to make optimal decisions.

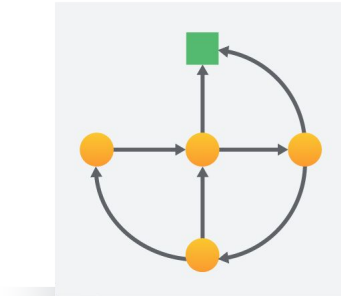
These three types are the focus of this module.

What is reinforcement learning?

Reinforcement learning is an area of machine learning where an agent learns by interacting with its environment.

Agents learn to:

- Achieve a goal.
- Achieve the *optimal* behavior.
- Obtain the maximum reward.



The term “reinforcement learning” is borrowed from an area of behavior psychology known as “operant conditioning.” The area of operant conditioning deals with learning the relationship between stimuli, actions, and consequences (the occurrence of rewards or punishments). Rewards and punishments guide the learner on the desired behavior or policy.

Reinforcement learning in software is an area of machine learning where an agent (or system of agents) learns to achieve a goal by interacting with its environment.

By “goal” we mean that we want the agent to learn the optimal path or behavior that collects the maximum reward. The agent might start with trial and error and, after many interactions with the environment, eventually have enough information to make smarter decisions. For simplicity, we often refer to either positive rewards (pleasant events) or negative rewards (unpleasant events) given to the agent.

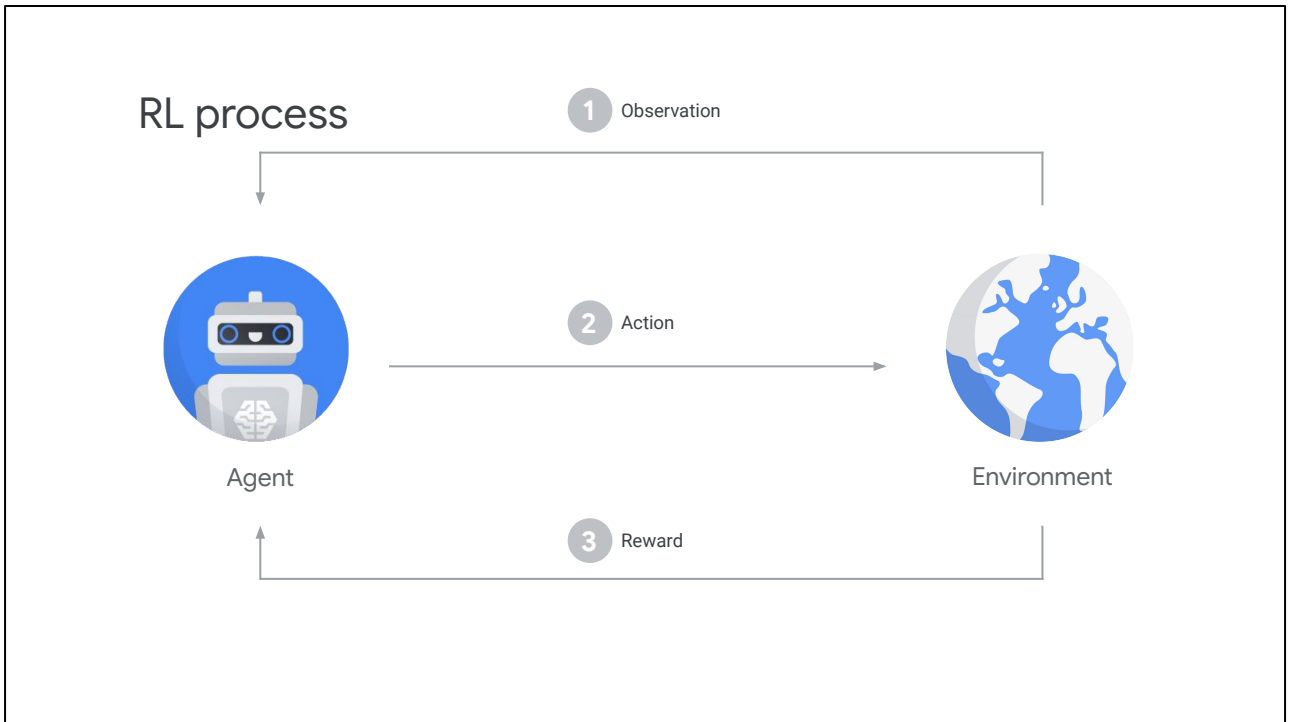
The optimal behavior is learned through interactions with the environment and observations of how it responds. The behavior is learned by:

- **Exploring** new states and situations that haven't been explored before.
- **Exploiting** knowledge of what has worked well by repeated trial and error.

With the information the agent obtains after each action, it adjusts future actions.

Furthermore, where supervised and unsupervised learning work with a static dataset, reinforcement learning works with a dynamic environment. The goal is not to cluster

or label data, but to find the best sequence of actions that will generate the optimal outcome. Unlike more traditional supervised learning techniques, not every data point is labeled. The agent might also only have access to “sparse” rewards. That is, the agent may only get feedback when it’s close to or at the target.



Let's analyze the general process agents use. Agents collect knowledge through exploration and exploitation during repeated trials to learn which actions they should perform to obtain maximum reward. The most basic steps in the process include the following:

1. Observation
 - The agent identifies something in the environment that it should act on.
2. Action
 - The agent does something in reaction to the observation.
3. Reward
 - The environment issues feedback for the behavior. Though we refer to it as a reward, think of it as feedback because the reward can be neutral (0), positive (+), or negative (-) values after each action.

Now, what do you suppose happens in scenarios where the agent takes actions that result in zero or negative rewards? You're right. The agent learns from these situations too.

Decision trees are useful representations of the possible outcomes, and yet they become unwieldy when there are many possible observations and actions. Reinforcement learning is ideally suited for solving problems with many possible outcomes.

Reinforcement learning in dog training



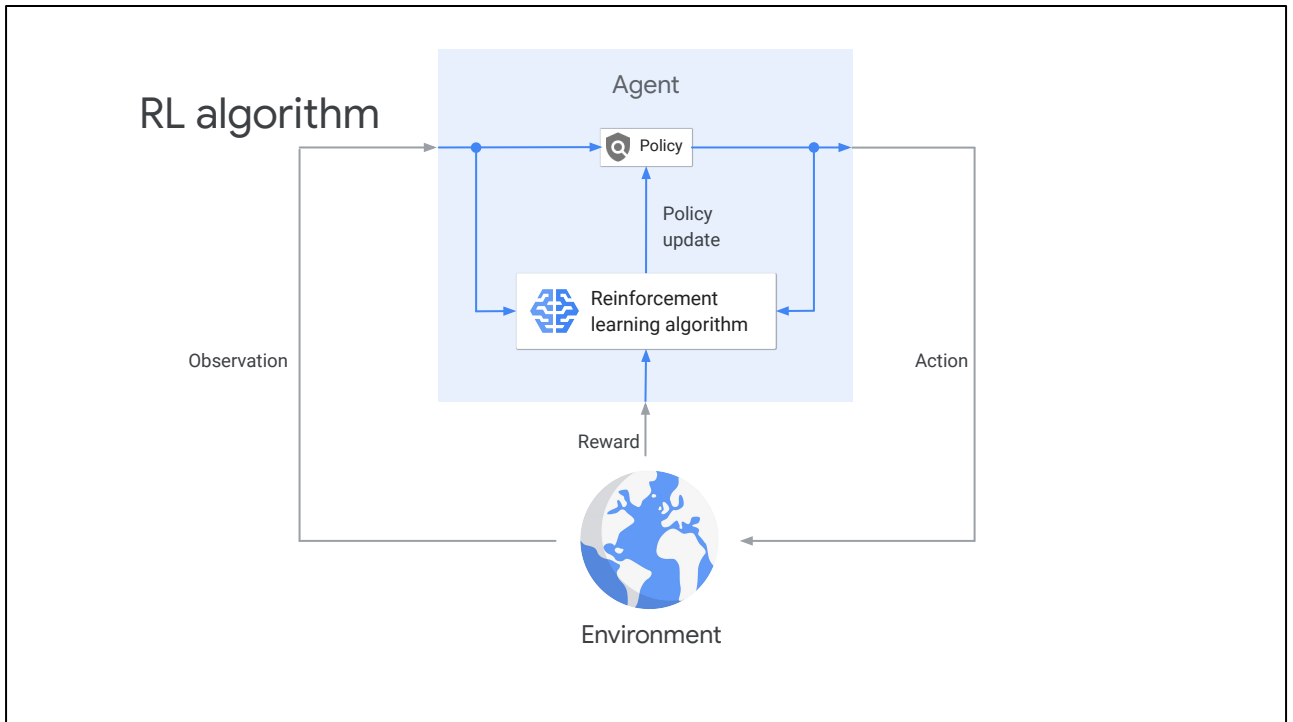
A common example from operant conditioning highlights reinforcement learning.

Let's say we have an owner who wants to train a dog. The goal of reinforcement learning in this case is to train the dog (agent) to complete a task within an environment (trainer). The environment could include all surroundings of the dog, but for simplicity let's focus only on the trainer.

First, the trainer issues a command which the dog *observes*. The dog then responds with an *action*. If the action is close to the desired behavior, the trainer will likely provide a *reward*, such as a toy; otherwise, no reward or a negative reward will be provided. At the beginning of training, the dog will likely take random actions, such as rolling over, when a "sit" command is given. The dog displays this behavior because it tries to associate specific observations with actions and rewards. The association, or mapping, between observations and actions is called a *policy*. We'll talk about that more shortly.

To summarize, the basic steps of RL in this example include:

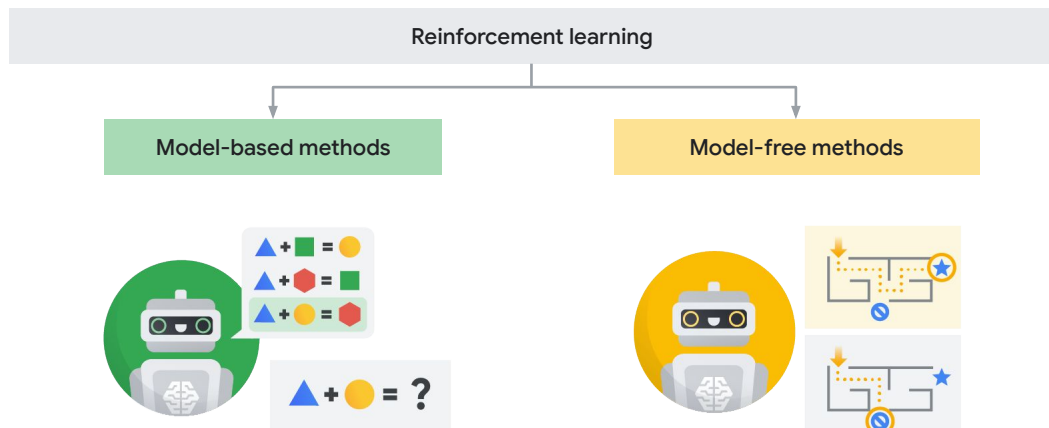
1. Observation: the trainer says "sit."
2. Action: the dog sits down.
3. Reward: the trainer gives the dog a toy.



Now that you can see how it works with the dog and trainer example, let's generalize the algorithm a bit more. In the absence of a supervisor, the agent must independently discover the sequence of actions that maximize the reward. The discovery process is initially a trial and error search. We measure the quality of actions by both the immediate reward they return and the delayed reward they might fetch.

Because the agent can learn the actions that result in eventual success in an unseen environment without the help of a supervisor, reinforcement learning is a powerful algorithm. The agent refines the policy over time based on its experience of which actions resulted in the optimal rewards.

Reinforcement learning methods

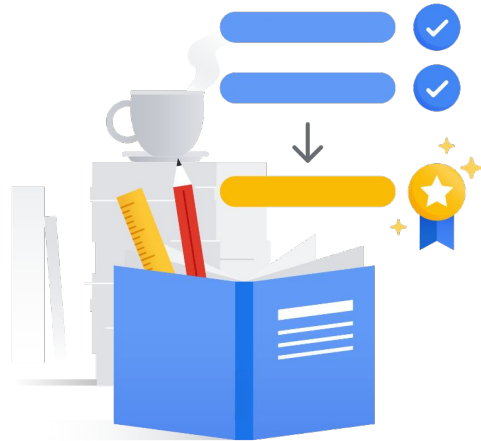


RL can be divided in two branches: model-based and model-free. Each method takes slightly different approaches to developing algorithmic possibilities:

- **Model-based**
 - A system uses a predictive model of the environment to determine what happens when certain actions are taken.
- **Model-free**
 - A system that doesn't need a modeling step because the control policy can be learned directly.

Characteristics of reinforcement learning

- There is no supervisor; there is only a real number or reward signal.

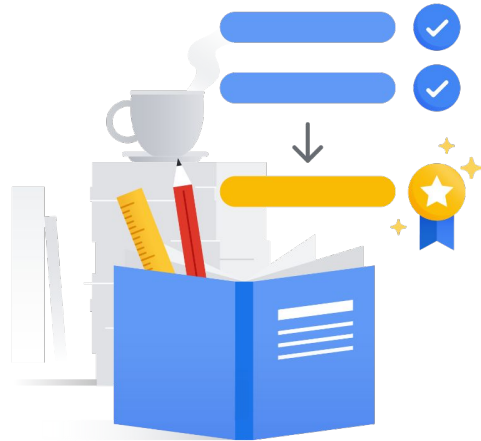


Some important characteristics of reinforcement learning include the following:

- There is no supervisor, only a real number or reward signal.
 - For example, the reward could be defined as the values -1.0, 0.0, and +1.0, negative, neutral, and positive respectively.

Characteristics of reinforcement learning

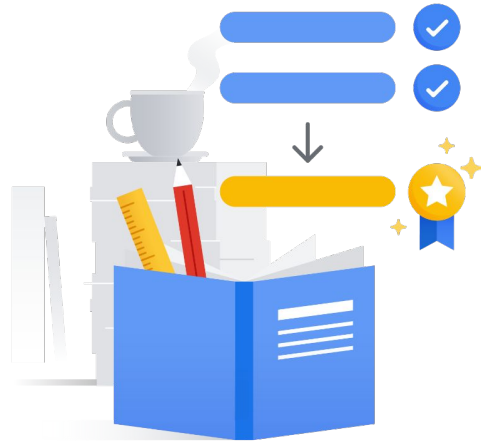
- There is no supervisor; there is only a real number or reward signal.
- Decision making is sequential.



- Decision making is sequential.
 - That is, the decision on what action to take in state S_2 comes after the decision has been made on state S_1 , and the environment changes to state S_2 .

Characteristics of reinforcement learning

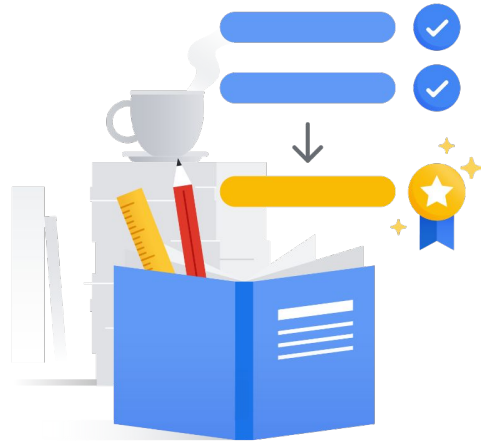
- There is no supervisor; there is only a real number or reward signal.
- Decision making is sequential.
- Time plays a crucial role in RL problems.



- Time plays a crucial role in RL problems.
 - Depending on when an action occurs (in time), there are different possibilities of what reward can be given and what state the environment can transition to.

Characteristics of reinforcement learning

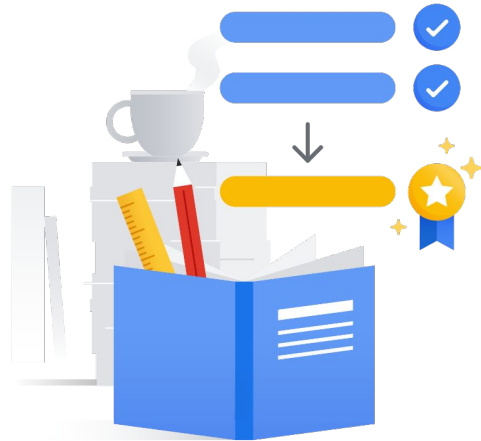
- There is no supervisor; there is only a real number or reward signal.
- Decision making is sequential.
- Time plays a crucial role in RL problems.
- Feedback is always delayed, not instantaneous.



- Feedback is always delayed, not instantaneous.
 - The agent can't know at the beginning whether it will receive a positive reward.

Characteristics of reinforcement learning

- There is no supervisor; there is only a real number or reward signal.
- Decision making is sequential.
- Time plays a crucial role in RL problems.
- Feedback is always delayed, not instantaneous.
- The agent's actions determine the subsequent data it receives.

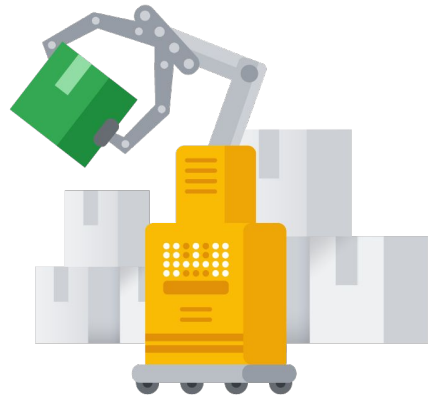


- The agent's actions determine the subsequent data it receives.
 - Depending on what action an agent takes, there are different possibilities of how the policy will be shaped.

Reference use case

Warehouse fulfillment center:

- It spans 28 football fields.
- Online merchants outsource to have their products stored, managed, and shipped from the fulfillment center warehouse.
- Robots need to choose optimal paths.



Let's look at a reference use case where reinforcement learning is a great choice. We'll refer to this use case later in this module.

Fulfillment centers are used by ecommerce merchants to store and manage their inventory as their clients make purchases and their packages must be shipped. By using fulfillment centers, online businesses are relieved of the necessary physical space to store all of their products and the need to directly manage inventory.

Why did we choose a fulfillment center as a reference use case? Because it's a good representation of how reinforcement learning can optimize the work that a robot agent needs to do when it moves products around a warehouse.

Imagine a robot moving around a warehouse through aisles of racks, which store products over an average of 28 football fields of space. The robot might go the long or short way around to find something. How does it know which is optimal? Certainly, if you have a small warehouse with only one robot, you could give it heuristics to follow. However, this method doesn't work when you have an enormous warehouse with many robots.

With RL, a robot can learn for itself to choose optimal paths and update the policy whenever something changes.

Agenda

Introduction to reinforcement learning (RL)

RL framework and workflow

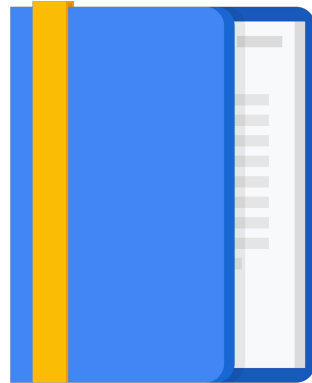
Model-based and model-free RL

Value-based RL

Policy-based RL

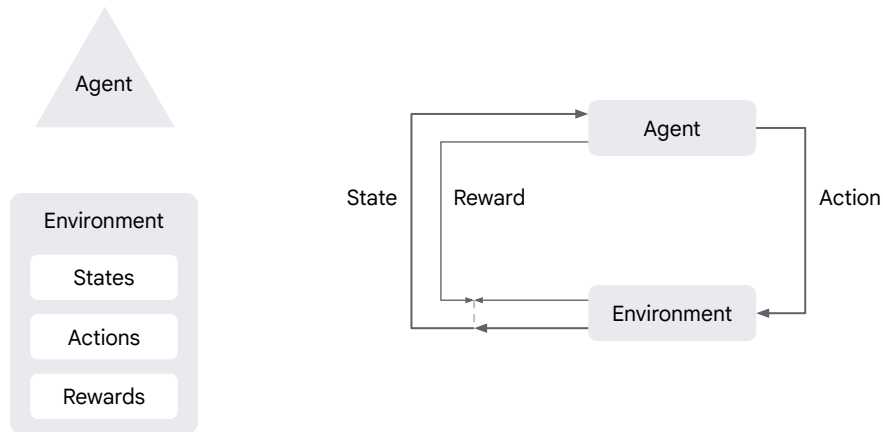
Contextual bandits

Applications of RL



Now that you've heard about reinforcement learning in general, we'll discuss the RL framework and the workflow you'll use to apply it to your own use cases.

The reinforcement learning framework



The reinforcement learning framework encompasses all the tools, notations, and algorithms you'll use to design and implement your RL solution. Its purpose is to help you create higher-level abstractions of the core components of an RL algorithm. Using an RL framework makes code easier to develop and read, and it can also help improve your overall efficiency.

Before we discuss the framework in more detail, let's take a moment to analyze the terminology used to describe the RL framework.

Terminology in reinforcement learning

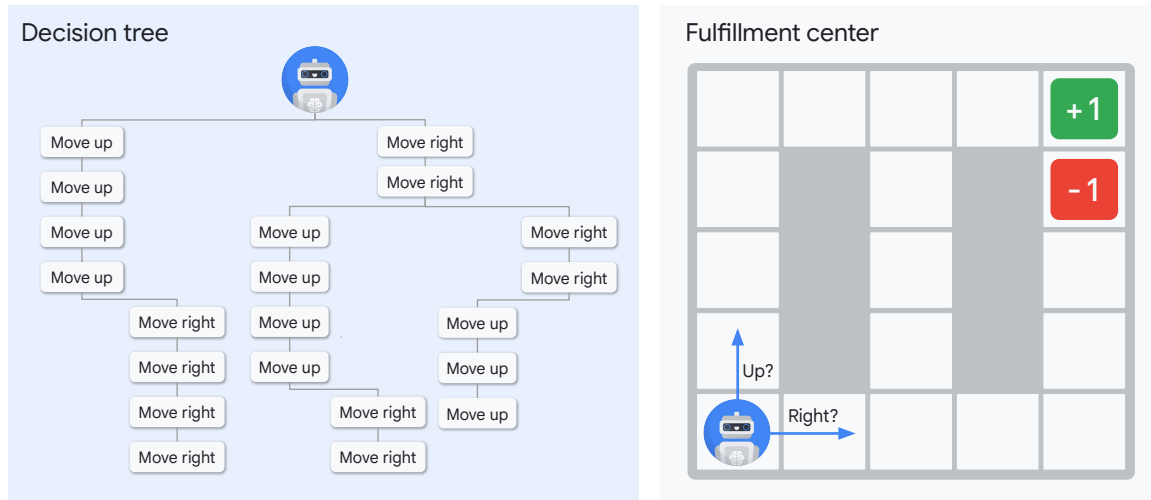
Term	Definition
State	Summary of events so far; the current situation
Action	One or more events that alter the state
Environment	The scenario the agent has to respond to
Agent	The learner entity that performs actions in an environment
Reward	Feedback on agent actions, also known as <i>reward signal</i>
Policy	Method to map the agent's state to actions
Episode	A termination point
Value	Long-term reward gained by the end of an episode
Value function	Measure of potential future rewards from being in a particular state, or $V(S)$
$Q(S,A)$	"Q-value" of an action in various state/action pairs
SARSA	State Action Reward State Action

The reinforcement learning framework consists of the following terms and notations, which we'll use at different times during this module:

- **State:** the current situation returned by the environment. In other words, the state suggests awareness of the situation that summarizes the history. The state is sufficient for the agent to determine the next step it should take. The state might not include all the information, and the agent might not see the whole grid board here. That is, it might be only partially observable. When your state is better, it contains more information and its performance improves.
- **Action:** one or more events that alter the state. The action space is essentially a set of actions which the agent can take to interact with the environment. Then the agent will move to a new state, unless the state is a terminal one.
- **Environment:** the world in which an agent interacts. A scenario that an agent has to face.
- **Agent:** the learner entity, or the brain, which takes action for each time step toward a goal. It is an assumed entity which performs actions in an environment to gain some reward. The agent generates the actions and updates a policy through learning. In fact, the agent doesn't need to know anything about the environment.
- **Reward (or reward signal):** the immediate feedback from the environment. It's what the agent observes after taking actions or tasks. Each time the agent acts, the environment gives it an instantaneous reward (which may be positive, negative, or neutral). Each step is independent from the previous one and requires the agent to decide on an action. Rewards can be given in multiple ways:

- At every time step.
- Sparse; this means it is given after long sequences of actions.
- At the end of an episode.
- **Policy:** a method to map the agent's state to actions. The policy dictates what the agent will do when it takes action. It is a strategy applied by the agent to decide the next action based on the current state.
- **Episode:** a termination point.
- **Value:** the total expected reward that an agent is expected to receive in the future by taking an action in a particular state. It is an expected long-term return, as compared to the short-term reward. The difference between reward and value is as follows: the reward is instantly given to the agent after it takes a certain action in a state. In contrast, the value is the cumulative sum of all rewards the agent obtains through its actions at all steps from the beginning to the end of an episode.
- **Value function:** it specifies the long-term desirability, or goodness, of any particular state. Where the reward signal captures the immediate reward, the value function gives a measure of potential future rewards from being in a state. **V(S)** is shorthand for the value function. That is, the value, V, of the state, S.
- **Q (S,A):** the value, Q, of taking an action, A, in a particular situation or state, S. By value we mean the long-term, accumulated sum of rewards in the future time steps taken until the end of an episode. In other words, how good is this one action compared to all the possible actions.
- **SARSA:** State Action Reward State Action; it is a shorthand way to reference the interaction between agent and environment.

Using the RL framework: Example



Let's look at an example that uses an RL framework. Think of this diagram as a representation of our fulfillment center (on a small scale). In a fulfillment center, you want the agent to find the optimal path to the warehouse item you need to ship.

A (partial) decision tree in our diagram represents the choices the agent has at each cell as it moves through the warehouse.

The agent starts at the bottom left of the grid, knowing nothing about the environment, so its first action could be chosen randomly. After each action, the agent receives a reward and moves to a new situation. That is, if it takes a step up, it's no longer in the same cell or state.

After the agent takes the next action, it's again in a new state, a cyclic situation. The agent is repeatedly faced with a series of actions it can take in each new state and cell. Each time it has to decide which direction to go, and sometimes not all directions are possible because an adjacent cell is blocked. The agent repeats the cyclical behavior until it reaches a termination point where the whole simulation resets and it ends up at the beginning again.

In the example shown, the reward is given at the end of the episode. Your goal is that the agent reaches the green box while it avoids the red box. If the agent gets to the

green box, it gets a reward of (+1). If it gets to the red box, it gets (-1). Both are terminal states which lead to a restart of the episode. Another way to look at it is the agent receives a neutral reward in the intermediate steps before getting to the terminal state.

Think of a scenario where you want the agent to learn the *optimal* path to the goal, not just any path. If the agent received negative rewards, (-1), for each step it took, then it would avoid too many penalties by taking a path with the least number of steps. Two of the shortest paths it can take are the L-shaped ones, which require a minimum of 8 steps. It needs to learn and derive a policy that will optimize positive rewards and avoid negative ones.

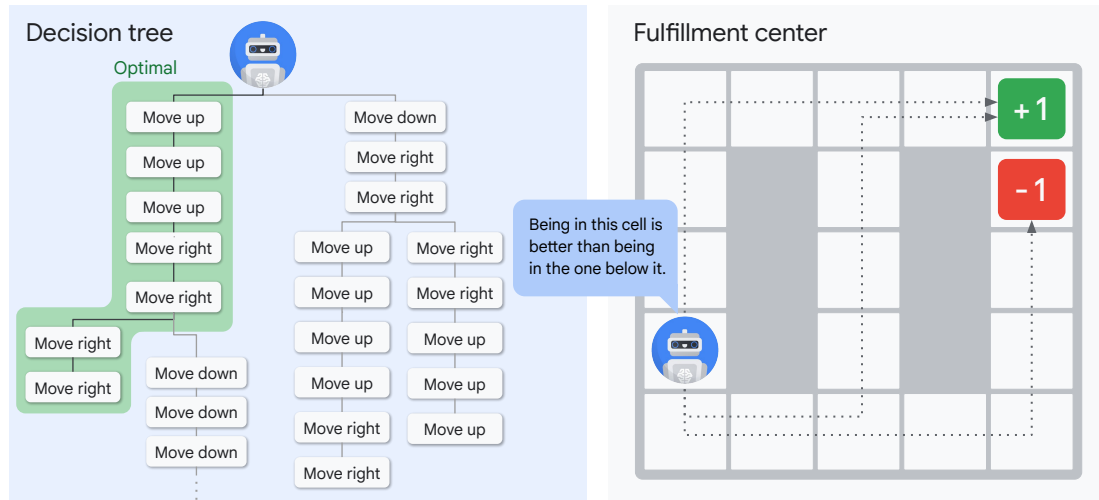
The agent assesses each reward (immediate) and not the value (long-term). If the agent reaches the goal, it's a termination point and it's done. Think of it like a game scenario. The first time around, the agent might not take the optimal path. It might just move randomly and accidentally reach the goal. After some episodes, the fact that it reached the goal will reinforce the behavior that led it to the goal.

<text below should be on the last breakdown slide as a summary of what we just walked through>

In our example, the episode terminates when it reaches one of the two endpoints (highlighted in red and green). The episode will reset and the value is all the cumulative sum of rewards the agent collected after moving between all the cells.

The agent aims to reach the green block. However, in each one of these cells it can finish in a separate state representation because of the different situation it is in. You could think, what is the better state to be in? Since the red cell is a pit (the agent will lose), the agent is better off in any other cell besides that one. No matter what action it takes, at least it will not lose. Let's look at the value function to see how it can help determine how good a state is.

The value function as the RL algorithm



The objective of an RL algorithm is to discover the action policy that maximizes the average value that it can extract from every state of the system. How can an agent determine which box it should move to next in our fulfillment center example?

The value function gives us a way to capture the measure of *potential future* rewards from being in a particular state. In other words, how good is this situation to be in? While the reward signal represents the immediate benefit of being in a certain state, the value function captures the cumulative reward to be collected from that state onward.

In shorthand, we refer to the value function as the $V(S)$, or the value of being in a state.

In the fulfillment center example, what is the value of moving to the 'right' from the agent's starting point? You might say it's lower than the value of going up. If the agent moves 'up' from its current position, it might end the trajectory in the green cell, which is the goal. If the agent instead moves 'right', the value of this action is most likely lower than the value of the action 'up'. That is, the chance is higher for the agent to land in the red cell, which should be avoided.

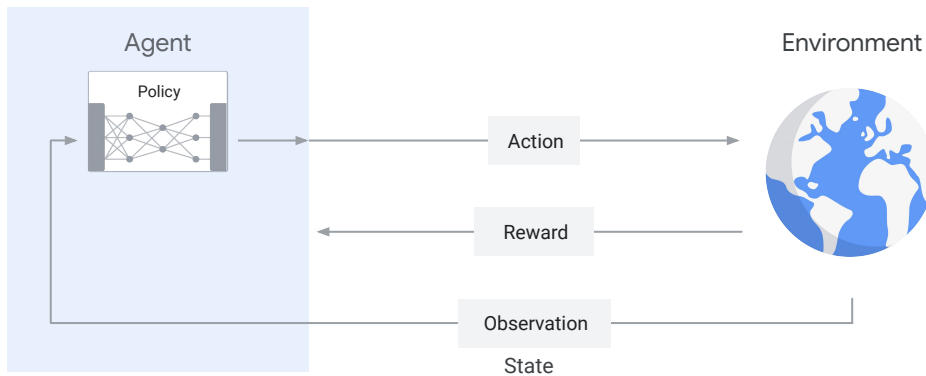
In the diagram, the box directly below the agent's current position is where it was at the start of the episode. We could say, the agent's current position is better than the one it started in because it's now closer to the green box.

Now, how would you generalize the value function for your RL problem?

<https://towardsdatascience.com/reinforcement-learning-value-function-57b04e911152>

Find the best sequence of actions that will generate the optimal outcome

Collect the most reward!



You use a value function to determine the best sequence of actions that will generate the optimal outcome.

Within the agent, a brain maps state observations (the inputs) to actions (the outputs). In RL nomenclature, this mapping is called the policy. Given a set of observations, the policy decides which actions to take. Just like with supervised learning, we can represent the policy as a deep neural network. Representing the policy as a deep neural network allows our agents to input thousands of states at the same time and still be able to develop a meaningful action.

In the fulfillment center example, your agent had a first iteration until the end of the episode. The agent's behavior was reinforced and it learned that in the first state, moving 'up' gives it an award of X . We can write X as $Q(S,A)$. Because the sample warehouse is small, we can easily see the endpoint. However, let's say it's a much bigger warehouse and moving to the right from its starting point could lead to $5X$ or $10X$ the reward over moving up. The agent has never tried it, so you still don't know. This is where a technique called exploration is useful to ensure you generate trajectories of experiences which might be more optimal than the knowledge you already have.

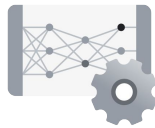
Setting up the problem for the RL workflow



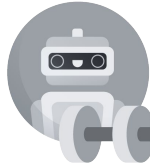
Environment



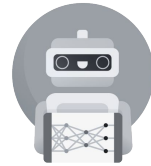
Reward



Policy



Training



Deployment

Now, we can't unleash RL on a problem if we don't know what setting up the problem correctly means. So the next thing we'll look at includes some of the basic questions we need to answer for ourselves.

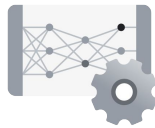
Setting up the problem for the RL workflow



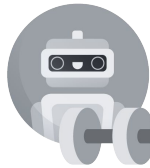
Environment



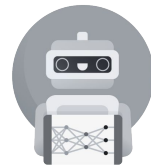
Reward



Policy



Training



Deployment

↑
How is it defined?
Real or simulated.

What exactly does the environment consist of? How will you define its properties? Is it a real or simulated environment?

Setting up the problem for the RL workflow



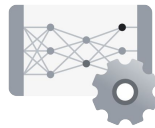
Environment

How is it defined?
Real or simulated.

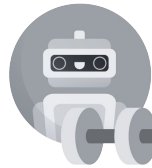


Reward

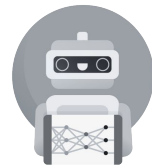
Incentivize
the agent to do
what we want.



Policy



Training



Deployment

How do we give the agent incentive, that is, reward the agent, to do what we want?

Setting up the problem for the RL workflow



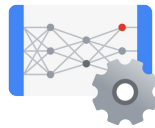
Environment

How is it defined?
Real or simulated.



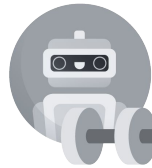
Reward

Incentivize
the agent to do
what we want.

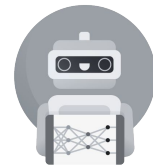


Policy

Structure the
logic and
parameters.



Training



Deployment

How should we structure the logic and parameters in order to derive a policy?

Setting up the problem for the RL workflow



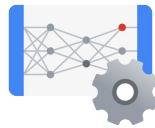
Environment

How is it defined?
Real or simulated.



Reward

Incentivize
the agent to do
what we want.



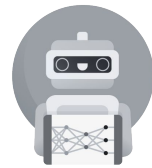
Policy

Structure the
logic and
parameters.



Training

Choose a training
algorithm.



Deployment

Which training algorithm should we choose to train the agent?

Setting up the problem for the RL workflow



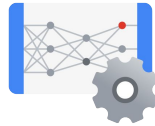
Environment

How is it defined?
Real or simulated.



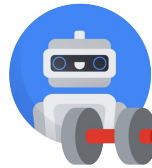
Reward

Incentivize
the agent to do
what we want.



Policy

Structure the
logic and
parameters.



Training

Choose a training
algorithm.



Deployment

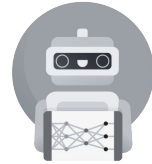
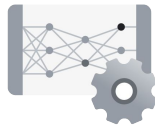
Put the agent
to work.

Finally, we put the agent to work, so it can determine the optimal solution.

Reinforcement learning workflow



Environment



1 Create the Environment

- Define the environment, including the interface with the agent.
- It can be either a simulation model or a real physical system.
- Simulated environments are safer and allow experimentation.

The general workflow for training an agent by using reinforcement learning includes the following steps:

1. Create the environment

First you need to define the environment within which the agent operates, including the interface between agent and environment. The environment can be either a simulation model or a real physical system. Simulated environments are usually a good first step because they are safer (real hardware can be expensive!) and allow experimentation.

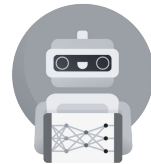
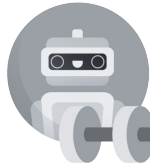
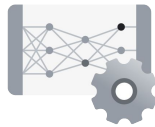
Reinforcement learning workflow



Environment



Reward



2

Define the reward

- Specify the reward signal.
- Iterate several times to shape the reward.

2. Define the reward

Next, specify the reward (also known as reward signal) that the agent uses to measure its performance against the task goals and how this signal is calculated from the environment. Reward shaping can be tricky and might require a few iterations to get right.

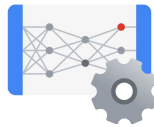
Reinforcement learning workflow



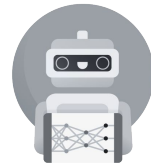
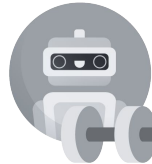
Environment



Reward



Policy



3 Create the agent

- Choose how to represent the policy.
- Select a training algorithm.

3. Create the agent

In this step, you create the agent, which consists of the policy and training algorithm.

- Choose a way to represent the policy. For instance, use neural networks or lookup tables.
- Select the appropriate training algorithm. Different representations are often tied to specific categories of training algorithms. In general, most modern algorithms rely on neural networks because they are good candidates for large state/action spaces and complex problems.

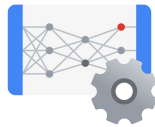
Reinforcement learning workflow



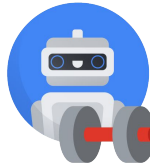
Environment



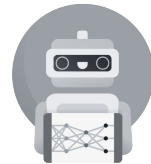
Reward



Reward



Training



4 Train and validate the agent

- Setup stopping criteria.
- Train the agent to tune the policy.
- Utilize multiple CPUs, GPUs, and clusters for complex applications.

4. Train and validate the agent

Set up training options. For instance, set up stopping criteria, and train the agent to tune the policy. Remember to validate the trained policy after training ends. Training can take minutes or days, depending on the application. For complex applications, parallelizing training on multiple CPUs, GPUs, and computer clusters will accelerate the process.

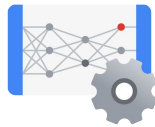
Reinforcement learning workflow



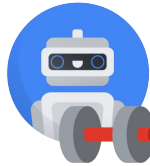
Environment



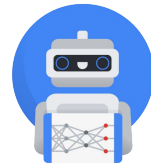
Reward



Reward



Training



Deployment

5 Deploying the policy

- Use generated code to deploy the policy.
- No agents and training algorithms needed yet.
- Revisit earlier stages again as needed.

5. Deploying the policy

Deploy the trained policy representation by using, for example, generated C/C++ or CUDA code. No need to worry about agents and training algorithms at this point; the policy is a standalone decision-making system!

Training an agent with reinforcement learning is an iterative process. Decisions and results in later stages can require you to return to an earlier stage in the learning workflow. For example, if the training process does not converge on an optimal policy within a reasonable amount of time, you might have to update any of the following items before retraining the agent:

- Training settings
- Learning algorithm configuration
- Policy representation
- Reward signal definition
- Action and observation signals
- Environment dynamics

Workflow



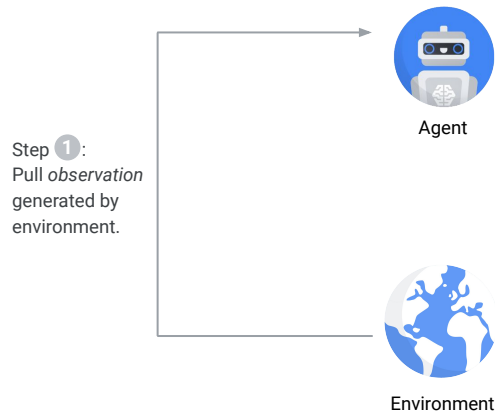
Agent



Environment

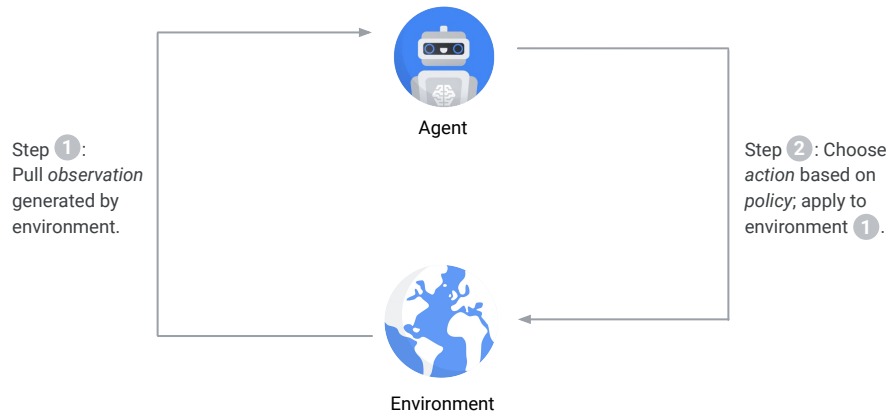
Now how is the workflow applied? After we have the basics of the model defined, we can execute the steps of the workflow.

Workflow



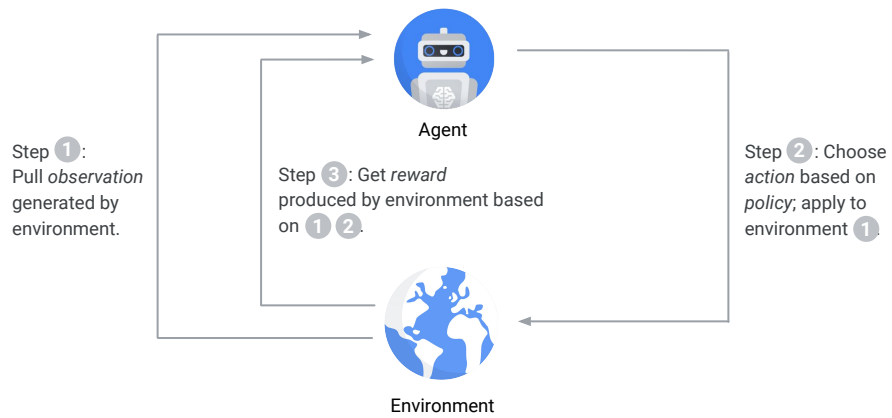
1. Pull the *observation* generated by the environment.

Workflow



2. Choose an action based on a defined policy and apply it to the environment.

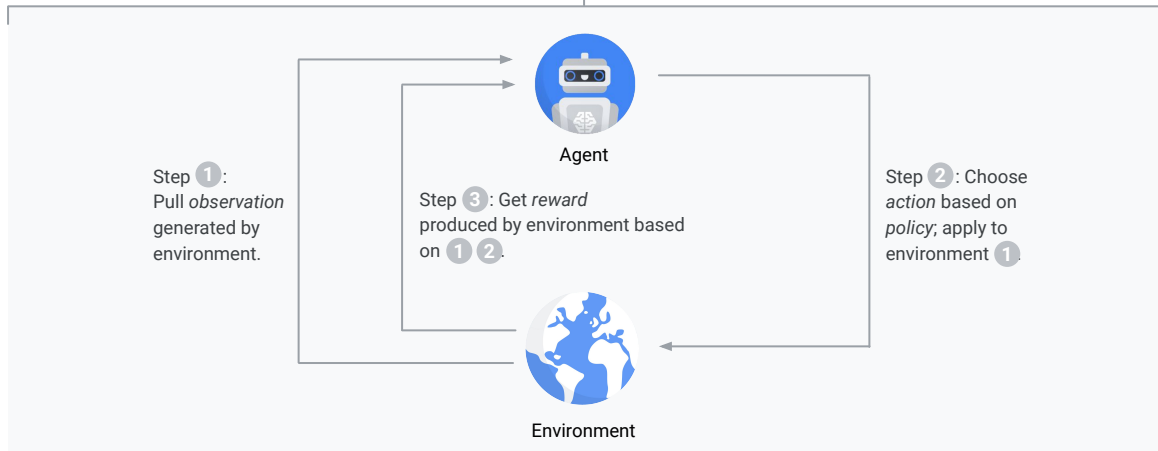
Workflow



3. Get a *reward* produced by the environment.

Workflow

Step ④: Use agent to *train the target policy* on trajectory data including observation, action, and reward from ① ② ③.



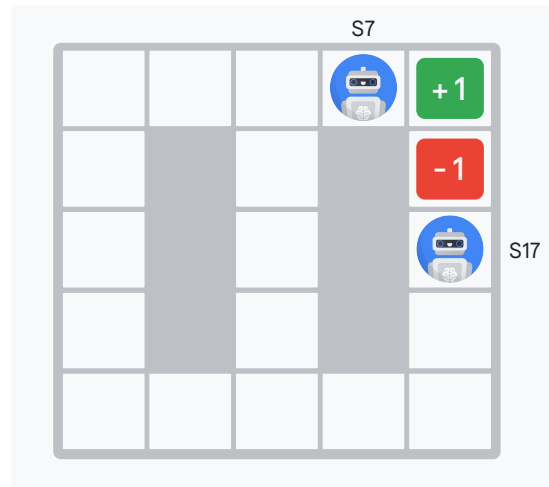
4. Use the agent to *train the target policy* on trajectory data including observation, action, and rewards from the first three parts of the workflow.

SARSA

State Action Reward State Action represents the quintuple — $Q(S_t, A)$ — where an agent interacts with its environment and updates its policy based on the feedback it received.

For instance:

- $Q(S7, \text{right})$ would have possible reward of $r = +1$
- $Q(S17, \text{up})$ would have possible reward of $r = -1$



As we begin to explore the functions used by an agent to determine actions, looking at the SARSA (State Action Reward State Action) algorithm can be useful. In simple terms, a SARSA agent interacts with its environment and then updates its policy based on the feedback it gets from those actions.

SARSA can be represented in a quintuple: $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$.

- S_t : the state at the beginning of the episode which is at time t .
- A_t : the action taken based on S_t .
- R_t : the reward given based on A_t .

And when the environment is in a new state, S_t is incremented.

- S_{t+1} : the state at time $t+1$.
- A_{t+1} : the action taken based on S_{t+1} .

The cycle repeats until the end of the episode. We can use a Q-value—or $Q(S,A)$ —to represent the value, Q , of taking an action, A , in a particular situation or state, S .

So, $Q(S_t, A)$ would have a possible reward r .

If $S7$ and $S17$ represent two possible states for the fulfillment example in the diagram, we could say the following:

- $Q(S7, \text{right})$ would have a possible reward of $r = +1$.
- $Q(S17, \text{up})$ would have a possible reward of $r = -1$.

Agenda

Introduction to reinforcement learning

RL framework and workflow

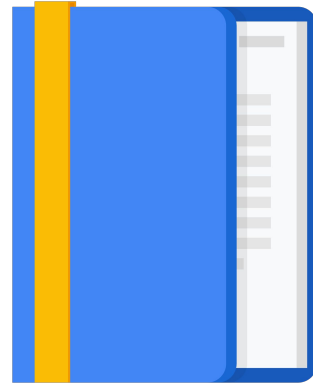
Model-based and model-free RL

Value-based RL

Policy-based RL

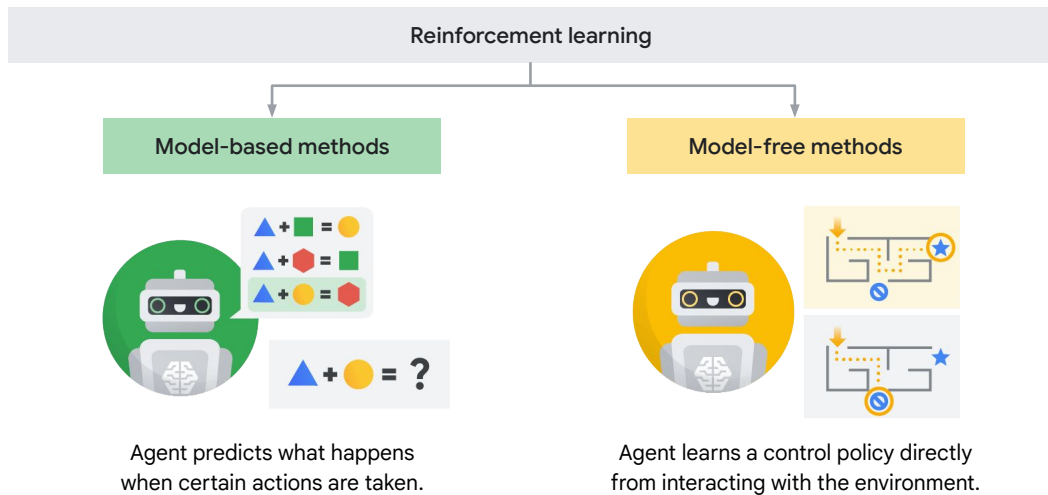
Contextual bandits

Applications of RL



Earlier in this module, we introduced the concept of various RL types. Next, we'll explore model-based and model-free types in more detail.

Model-based and model-free RL



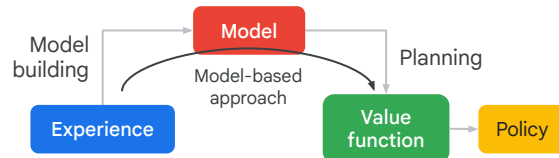
Earlier in this module, we introduced the two branches of reinforcement learning, model-based and model-free methods. Let's revisit them at a high level before we begin discussing them in more detail.

- **Model-based** is a system that uses a predictive model of the environment to determine what happens when certain actions are taken. In other words, the policy is set in advance and the agent predicts whether or not each potential step is the best one or not.
- **Model-free** is a system where the modeling step is skipped altogether in favor of learning a control policy directly. In other words, the policy is undefined at first and the agent explores its environment; updating the policy as it learns where it gets rewards.

Categorizing the approaches into two branches is useful, but keep in mind they can sometimes overlap.

Model-based method

The agent learns what is considered optimal behavior through actions and observing the resulting state and reward outcomes.



In the model-based method, the agent learns what is considered optimal behavior by learning a model of the environment. It does so through actions and by observing the resulting state and reward outcomes. You provide the agent with heuristics, a model, or part of the model of the environment. Through experiences built on the foundation of a model, a value function emerges which in turn results in a policy the agent can use in its interaction with the environment.

Model-based method

The agent learns what is considered optimal behavior through actions and observing the resulting state and reward outcomes.



With existing knowledge of the environment, such as areas not worth exploring, you can supply the information that the agent needs in advance. Think of the model-based method as giving a basis of knowledge about the environment to the agent.

You can read more about model-based algorithms in the following article on [Model-Based Reinforcement Learning: Theory and Practice](#)

Model-free method

The agent learns by exploring all areas of the state space to fill out its value function as it searches for the best reward.



The model-free RL method is called this because the agent does not need to know anything about the environment. It is powerful because you can drop an RL equipped agent into any system. So long as you've given the policy access to observations, actions, and enough internal states, the agent will learn how to collect the most rewards on its own.

Without any understanding of the environment, an agent needs to explore all areas of the state space to complete its value function. That means it will spend some time exploring low reward areas during the learning process.

You can think of a model-free method as the agent is free to roam around as it searches for the best reward.

Model-based vs model-free RL

	Model-based	Model-free
You have access to or knowledge about the environment.	Yes	No
You can avoid needless exploration by focusing on areas you already know are worthwhile.	Yes	No
Need to make more assumptions and approximations.	Yes	No
Need lots of samples.	No	Yes
Over many episodes, results become <i>less</i> optimal.	Yes	No
Over many episodes, results become <i>more</i> optimal.	No	Yes
Applicable across a wide variety of applications.	No	Yes

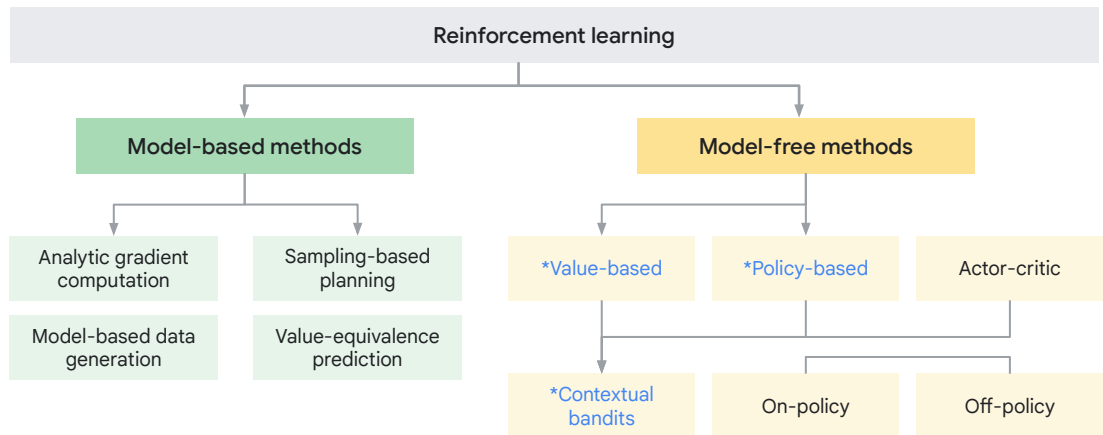
Now that we've looked at model-based and model-free reinforcement learning types, you might be wondering, which should I use? What are the tradeoffs?

In the model-based approach, you might have access to the model (environment). If you have access to the environment, you know the probability distribution over states that you move to. If you don't have access to the environment, you first try to build a model yourself (which is an approximation). Building a model yourself can be useful because it lets you plan. You can think about potential moves without actually performing any actions. The agent learns to perform in that specific environment.

Building a model is helpful, but it has a disadvantage. By nature, the model-based method uses a model to learn the environment first, and then an agent learns from the model in a feedback loop. Both operations are subject to errors, and the agent relies on the model that learned the environment to be accurate. Unfortunately, the model is not accurate, especially in the beginning, and worse, it can lead to double accumulation of errors. The growing deviation over many episodes begins to show results that are less and less optimal.

The model-free method is more popular because it is more suitable across applications. In the model-free approach, you're not given a model and you're not trying to explicitly determine how it works. You just collect some experience and then (hopefully) derive the optimal policy.

Examples of model-based and model-free RL methods



* Discussed in this module.

Various algorithmic approaches are listed under each model-based and model-free branch. Examples of the methods categorized under these branches include:

Model-based

- Analytic gradient computation
- Sampling-based planning
- Model-based data generation
- Value-equivalence prediction

Model-free

- Value-based
 - Contextual bandits
- Policy-based
 - On-policy
 - Off-policy
- Actor-critic

For the remainder of this module, we will be focusing on the most commonly used model-free methods.

Agenda

Introduction to reinforcement learning

RL framework and workflow

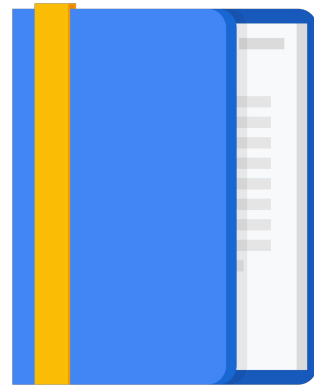
Model-based and model-free RL

Value-based RL

Policy-based RL

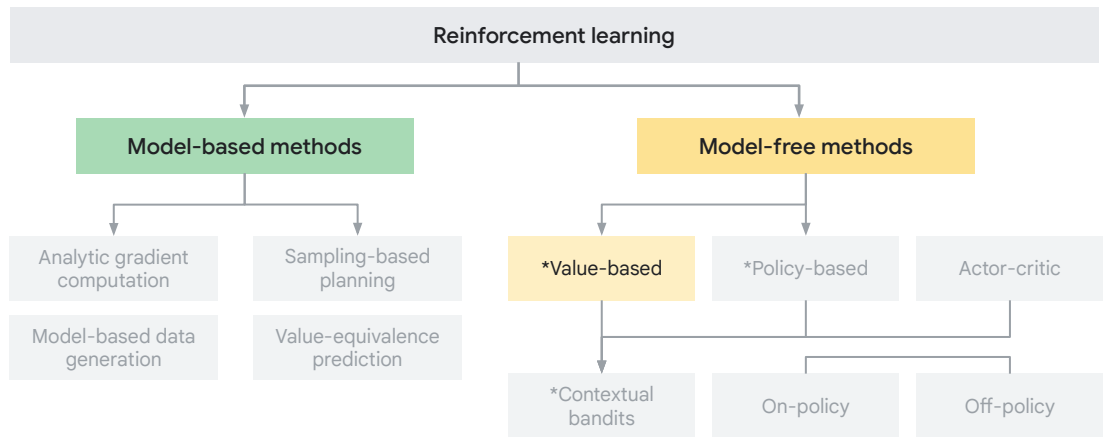
Contextual bandits

Applications of RL



Now that we've introduced the model-based and model-free RL branches, let's turn our attention to some specific model-free methods.

RL methods

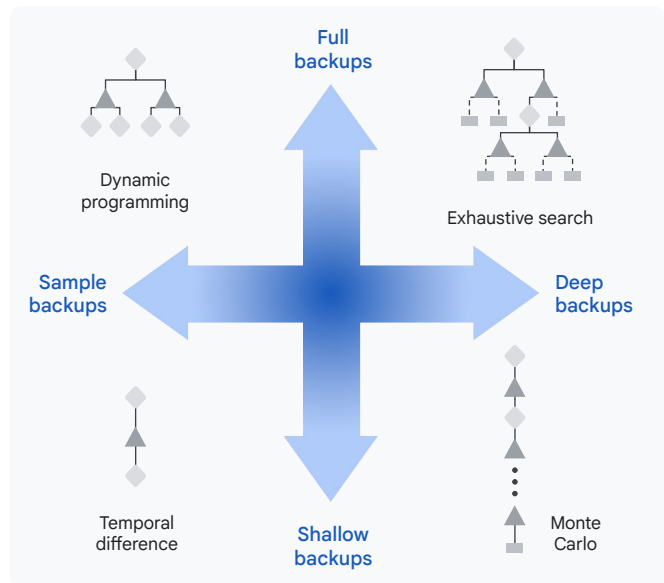


Earlier in this module, we introduced the concept of various reinforcement learning (RL) types. We'll first explore more detail on the value-based type of RL.

Value-based approach

You explore in order to learn state-action values and maximize a value function, $V(S)$.

The agent can sample exhaustively or sample and generalize to derive a policy, π , that maximizes the value of action for each state.



The value-based reinforcement learning approach is often used in real life applications.

In a value-based method, your goal is to maximize a value function, $V(S)$. To derive the value function, the agent learns the states through exploration and saving the information as backups. The agent expects a long-term return of the current states under a policy (where policy is denoted " π "). To get a good approximation of the value function, the agent exhaustively samples the whole state space (complete backup) or samples from a shallow backup and generalizes to unseen states. We'll discuss more on value functions later.

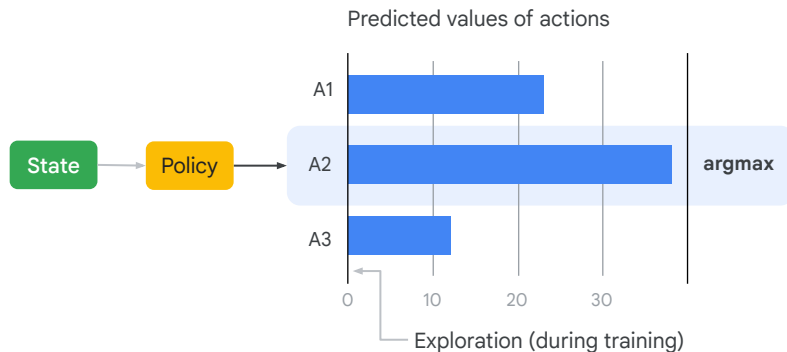
There are different ways of deriving a value function. An exhaustive search requires full and deep backups to capture every value for each state. Dynamic programming, Temporal difference learning, and Monte Carlo methods require less sampling but to varying degrees.

In comparing the different value-based approaches, the spectrum goes from sample to deep backups and from shallow to full backups which can be described as follows.

- Sample backup: the agent learns from environment sampling which may provide an incomplete picture of the environment dynamics. With enough samples, the sample backup approaches come closer to the full backup approaches.
- Deep backup: the agent learns the whole trajectory of the chosen action up to

- the termination point. This can be the whole trajectory of the sample and not necessarily the full environment.
- Shallow backup: the agent learns one time step at a time, in a breadth first search manner, of the chosen action trajectory.
- Full backup: the agent learns from the ability to access the complete environment, not just samples.

Enforcing control in value methods



We talked a lot about estimating the long-term value of a state, which is predictive in nature. Now we'll explore how we enforce control (decide) in value methods by using these predicted value estimations.

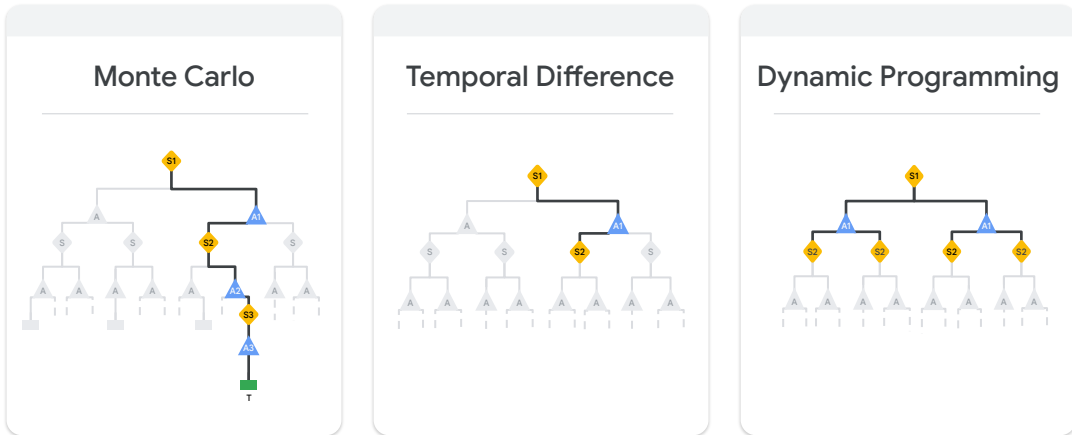
In the diagram, you can see the following:

- **State:** the state which is fed into the policy.
- **Policy:** the policy which the agent uses to map states to actions.
- **Predicted values of actions:** the bar graph represents the value function approximation. There is a function approximator embedded inside the policy, which is used to predict the value of a given action for a given state. For instance, it could be a multi-layer perceptron.
- **A1, A2, and A3:** these represent actions the agent could take.
- **Blue bars:** each blue bar represents the value function or sum of rewards possible, if that particular action were taken.

The goal is to maximize $V(S)$, which is denoted in the diagram as **argmax**. Notice that the predicted value of taking action A2 appears to be the highest.

Consider a [simple epsilon-greedy](#) type exploration strategy. If the agent chooses to exploit during the training, it will choose the action with the maximum value (argmax), which is A2. During the exploration cycle, the agent would randomly pick among the three actions with uniform probability. During training, both exploitation and exploration take place. During inference and serving, typically only exploitation occurs because exploring new options is risky.

Three approaches of value-based RL algorithms






There are three basic approaches to value-based RL algorithms:

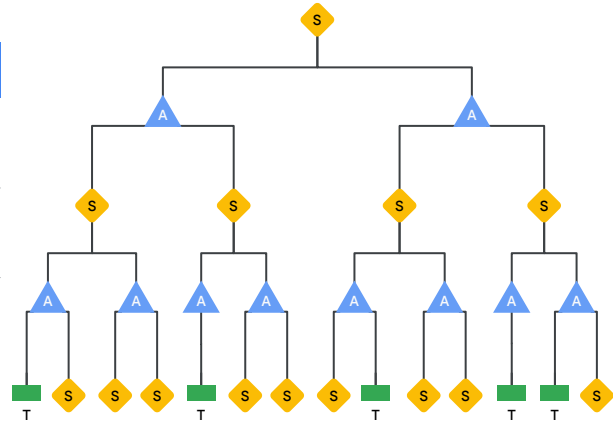
- Monte Carlo
- Temporal Difference
- Dynamic Programming

For additional information, see "[A \(Long\) Peek into Reinforcement Learning.](#)"

Dynamic programming will not be covered in this module due to time constraints, but you can find lots of information on the internet.

Terms of the decision tree

Symbol	What it represents	Examples
	State (S) at each time (t) step, or S_t , until termination (T).	$S_1, S_2, S_3, \dots S_T$.
	Action (A) at each time (t) step, or A_t , until termination (T).	$A_1, A_2, A_3, \dots A_T$.
	Termination point, or end of the episode.	



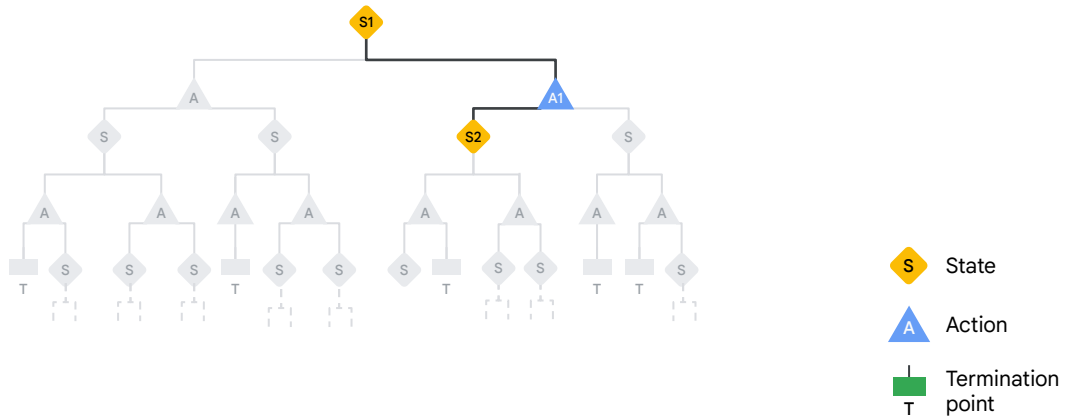
A decision tree is an easy way to visualize what happens in the different approaches. Let's talk about what's in the diagram, in general, before we discuss the specifics:

- Yellow squares: represent states.
- Blue triangles: represent possible actions. Each action can lead to one or more states based on some probability.
- T boxes: are terminal states where the episode ends and the agent starts over at the very beginning.
- T: represents each time step, a point in time.
- S_t : represents the state at each time step. That is, S_1 is the root node and it's the state at the beginning of the episode. The agent determines which action to take based on the state.
- A_t : represents the action at each time step. Each action can lead to one or more states based on some probability. That is, A_1 is the action to the left of S_1 , and A_2 is the action to the right of S_1 . Based on which action the agent chose, the journey will progress down one side of the tree.

The interaction between the agent and the environment involves a sequence of actions and observed rewards in time, where $t = 1, 2, 3, \dots T$. During the process, the agent accumulates the knowledge about the environment, learns the optimal policy, and decides which action to take next so as to efficiently learn the best policy. Let's label the state, action, and reward at time step t as S_t , A_t , and R_t respectively. Thus the interaction sequence represents one episode (also known as "trial" or "trajectory") and the sequence ends at the terminal state: $S_1, A_1, R_1, S_2, A_2, R_2, \dots S_T$.

learns that these actions led to a certain cumulative reward. For instance, say 10 was the reward for the trajectory to the right. Now, if the agent takes a different trajectory, say to the left, that led to a reward of 20. Then the agent learns taking a left action here is better. The agent tunes to improve the state value, knowledge of the action, which action to take, and the value of the state. The tuning done by the agent improves rewards and the value function. When you get $V(S \text{ of } T)$, you learn how good the situation is. Because how good or bad the situation is depends on the value, you get all the actions you can pick from here.

Temporal Difference backup



Another option is Temporal Difference (TD) backup. Like Monte Carlo, the TD method can learn directly from raw experience without a model of the environment's dynamics. Unlike the Monte Carlo method, TD estimates based in part on other learned estimates, without waiting until the end of the episodes ([bootstrapping](#)). The agent learns from one or more intermediate time steps in a recursive fashion. The recursive learning helps in accelerating overall learning even in cases where there might not be any well-defined terminal states.

In TD backup, you take an action, you get some reward, you get a new state, then you back up, and so forth, in a recursive fashion. From each state, you will make another choice of actions, end up in a new state and so on. When you get the reward at the end, you learn a better value function for the policy given the state, which action, and so forth. The agent can recursively learn which action would have been the best here without having to finish the complete episode. It's useful because you might not be able to explore all the states' spaces.

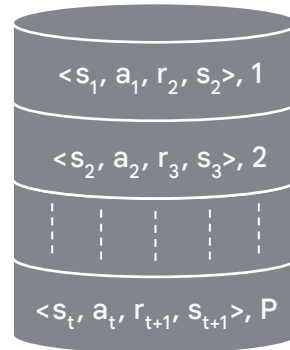
Something to consider is this method comes with a caveat: because TD backups haven't seen the whole set of trajectories, they have a narrow perspective and tend to underfit, especially in the beginning. Even so, TD backups are used more often than Monte Carlo backups, despite higher complexity and high bias.

Achieving data efficiency

The problem

- Data inefficiency
- Rare event loss
- Policy drift
- Correlated experience
- Large state space

The solution



Experience replay buffer

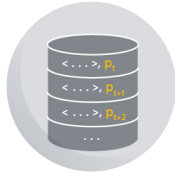
Now, there are some issues that arise when you use value-based approaches including:

- **Data inefficiency:**
 - These approaches require much trial and error, meaning they're not good at data efficiency. The trajectory of experience is lost because data points might be used only once.
- **Rare event loss:**
 - Events which only happen occasionally are lost. Ideally you want to prevent the neural network from forgetting about them.
- **Policy drift:**
 - The complete environment might change. That is, there could be a non-stationary policy concept drift and the distribution might change. If you wanted to revert to the previous environment, the network would have already forgotten about it, unless you stored the information somewhere.
- **Correlated experience:**
 - The action you take in a particular state is often correlated with the next action you end up in. It's often dangerous to learn from correlated experience. To break the correlation between these two elements, you might want to store it somewhere and sample it in a non-sequential format.
- **Large state space:**
 - The large state space is where you might want to remember and learn from the previous states too.

What is the solution?

An experience replay buffer could help. Let's explore this in detail.

Collecting and learning from experience



Experience replay buffer



Batch of experiences



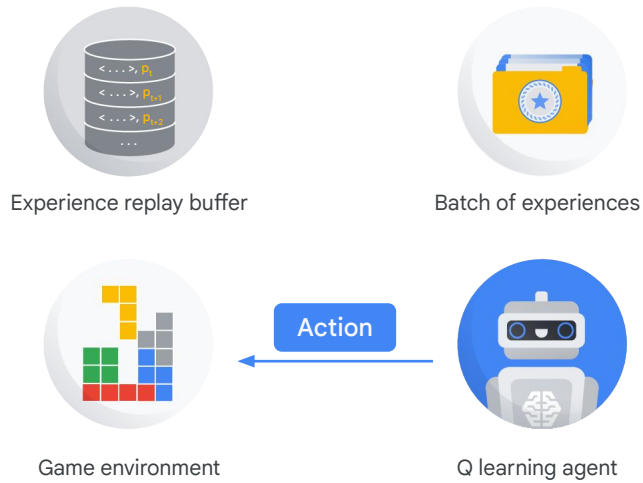
Game environment



Q learning agent

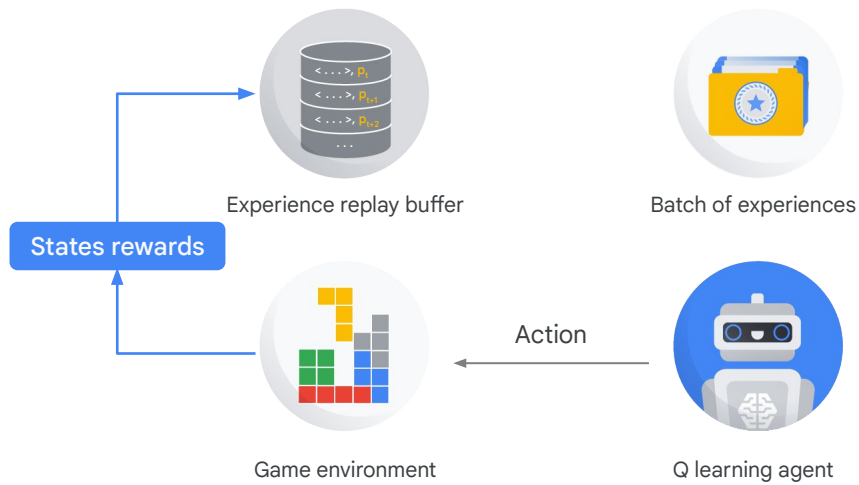
An agent can use an experience replay buffer to collect and learn from experience. It's like a last recently used (LRU) cache where the element unused for the longest period is evicted. You have a buffer where you store experience until it is full. Then you remove the oldest experience to make space for a new one. It stores past experiences including rare events. Sampling from the replay buffer can be done in a more random fashion to break the correlation. The replay buffer can also keep previous environment information in case you go back or the environment shifts between two different distributions.

Collecting and learning from experience



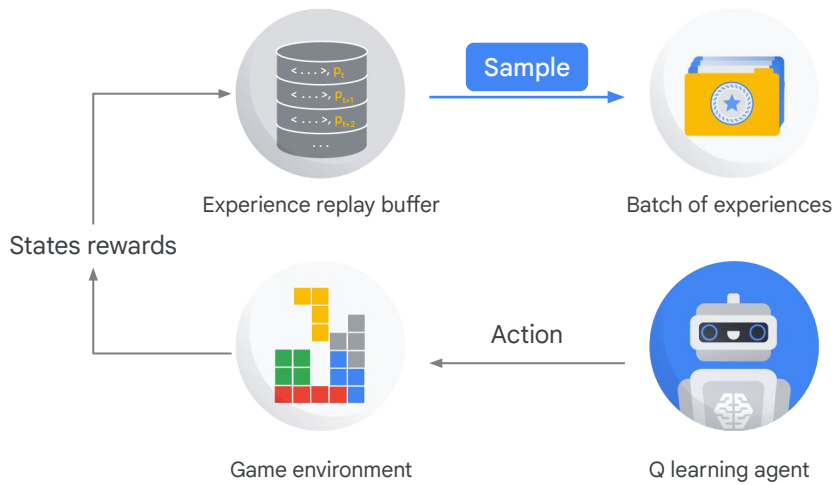
With an initial policy, the agent takes a particular action in the environment.

Collecting and learning from experience



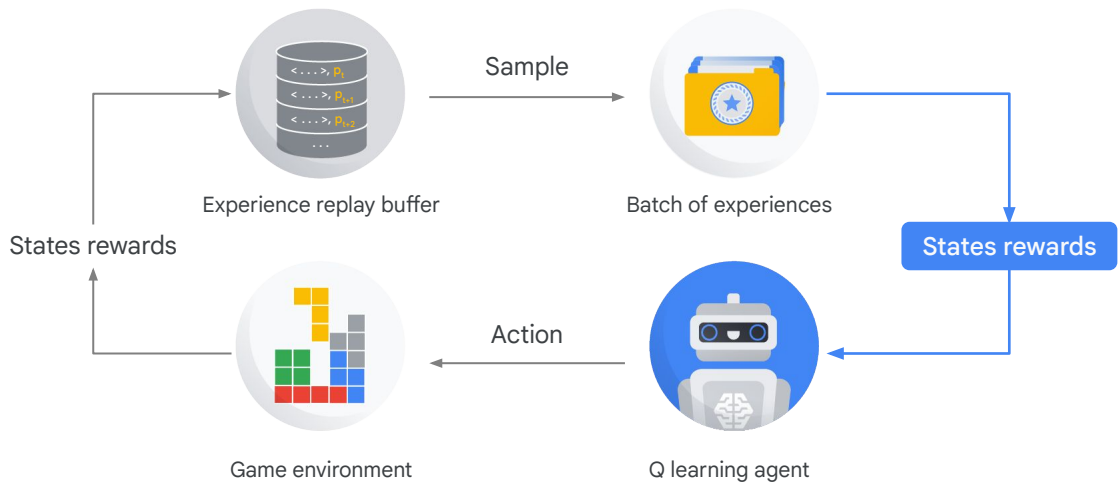
The environment returns a reward and the next state and these outcomes are stored in the experience replay buffer. The reward should be correlated with the desirability of the action toward the goal.

Collecting and learning from experience



From the experience replay buffer, the agent samples the experience trajectories in a non-sequential format, perhaps the shuffle format. It shuffles random samples to counter correlation.

Collecting and learning from experience



From samples of trajectories, or a batch of experiences, the agent learns to use a backup method such as TD backup or Monte Carlo. Then the agent updates the policy. Based on the new policy, it acts again and the cycle repeats until the episode ends.

Deep Learning called DQN (or deep Q learning), is an algorithm that is built keeping experience replay buffers in mind.

Agenda

Introduction to reinforcement learning

RL framework and workflow

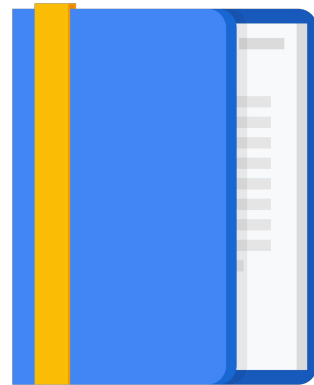
Model-based and model-free RL

Value-based RL

Policy-based RL

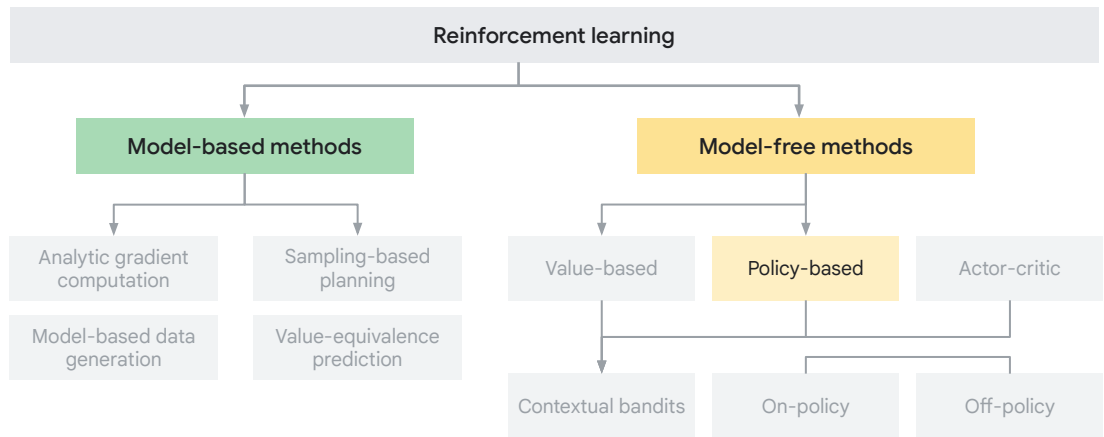
Contextual bandits

Applications of RL



Earlier in this module, we introduced the concept of various RL types. Next, we'll explore in more detail the policy-based type of RL.

RL methods



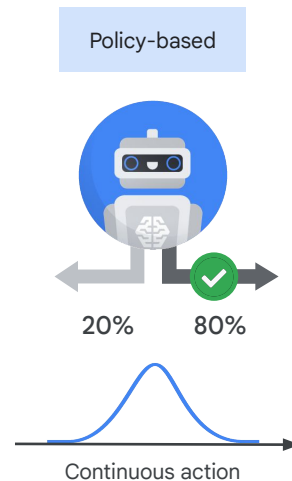
We looked at value-based approaches. Now we'll look at policy-based approaches.

Policy-based RL

You want the action performed in every state to help you to gain maximum reward in the future.

The agent will:

- Learn the stochastic policy function that maps state to action.
- Act by sampling policy.
- Utilize exploration techniques.



In a policy-based RL method, you try to develop a policy, π , such that the action performed in every state helps you to gain maximum reward in the future. The goal is a mapping of optimal states to actions ($s \rightarrow a$).

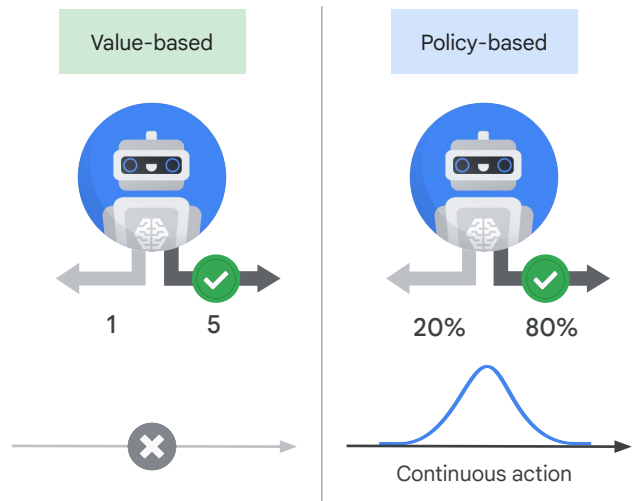
In general, the agent will:

- learn the stochastic policy function that maps state to action.
- act by sampling policy.
- utilize exploration techniques.

Policy-based vs value-based

A policy-based approach is preferable over value-based when:

- There are large action spaces.
- Stochasticity is needed.
- An agent will learn the policy directly.
- Lower bias in the policy is needed.



In policy-based methods, we explicitly build a representation of a policy (mapping $\pi: s \rightarrow a$) and keep it in memory during learning. In value-based, we don't store any explicit policy, only a value function. The policy is implicit and can be derived directly from the value function. That is, the policy picks the action with the best value.

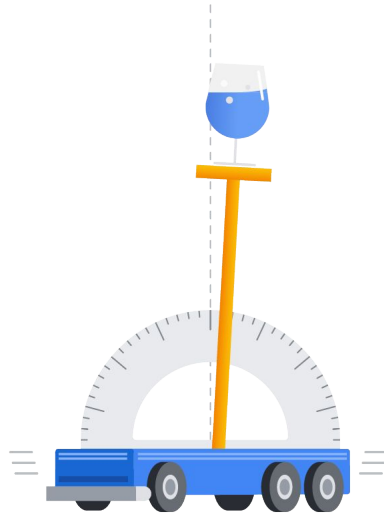
A policy-based approach is preferable over a value-based one when any of the following are true:

- You have a large continuous action space.
- You would like stochasticity (lacking any predictability or order) in your policy.
- An agent will learn the policy directly.
- You want a lower bias in your policy.

Example: CartPole problem

A policy-based approach outputs an optimal action for a selected state

	Min	Max
Cart position	-2.4	2.4
Cart velocity	-Inf	Inf
Pole angle	-41.8 deg	41.8 deg
Pole velocity at tip	-Inf	Inf



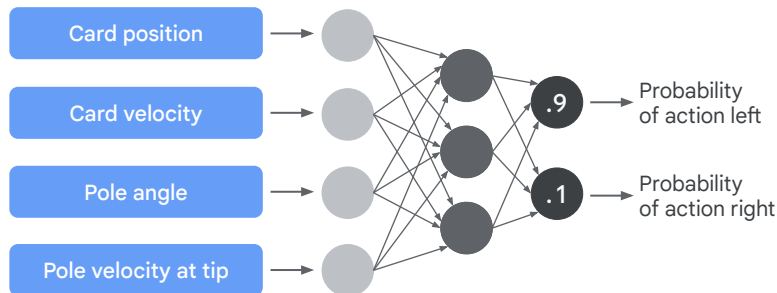
Let's consider the CartPole problem as an example. As a reminder, the CartPole problem is also known as the inverted pendulum problem. In this problem, a pole or handle is attached to a cart. Because the pole's center of mass is above its pivot point, it's an unstable system. Without a control system to monitor the angle, it will fall over.

Say the action space consists of continuous number variables as follows:

- Velocity between (-) infinity and (+) infinity
- Cart position between (-) 2.4 and (+) 2.4

An infinite number of discrete values lies within these intervals. Therefore, we would need an infinite number of actions to model this with a value-based approach. However, a policy-based approach outputs an optimal action for a state: in this situation, the optimal action is a floating point vector.

Policy-based algorithms



Model free

- State → Optimal policy
- REINFORCE
- Proximal policy optimization (PPO)

Some of the policy-based algorithms include REINFORCE, proximal policy optimization (PPO), and actor-critic.

When stochasticity is required, policy-based approaches can be useful in other areas. Policy-based algorithms can be used to output parameters of a distribution. They can also output probabilities of each action in the instance of a discrete action space (for instance, with a softmax activation), which can then be sampled from.

Function approximation with Deep Learning

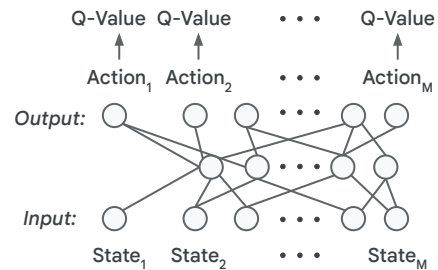
Used in CartPole example

Q-Learning

	Action ₁	Action ₂	...	Action _M
State ₁	Q ₁₁	Q ₁₂	...	Q _{1M}
State ₂	Q ₂₁	Q ₂₂	...	Q _{2M}
...
State _N	Q _{N1}	Q _{N2}	...	Q _{NM}

A better way

DQN



In the CartPole example, we used a tabular approach typical of Q-Learning. There's a better way, but before we look at the Deep Q-network (DQN) approach, let's review Q-Learning.

In Q-Learning, we represent the states and actions in matrix form. For instance, if you have five actions, then they are denoted as action one to five in the columns. With the states as rows, you might wonder the value of a particular action in a particular state. You can find the answer in the particular cell.

The primary disadvantage is Q-Learning in large matrices is computationally intractable. A second disadvantage is that it requires you to have already explored. You've taken every action in every particular state to fill this matrix completely. What if you have never explored Q22, which is action two in state two? Because you have never explored it, the agent would not know anything about it. You could certainly improve the situation in some ways, but let's just say that it's not ideal.

In contrast, Deep Q-networks are neural networks that utilize Deep Q-Learning to provide models. They train on inputs that represent agents in areas or other experienced samples and learn to match those data with the desired outputs. They involve a compact function approximation, which can then generalize well across state and action per combination that the agent hasn't seen before. Deep Q-network implementation often also samples from a replay buffer.

Agenda

Introduction to reinforcement learning

RL framework and workflow

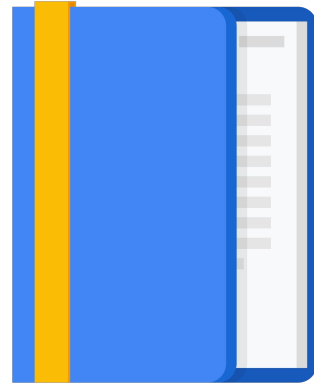
Model-based and model-free RL

Value-based RL

Policy-based RL

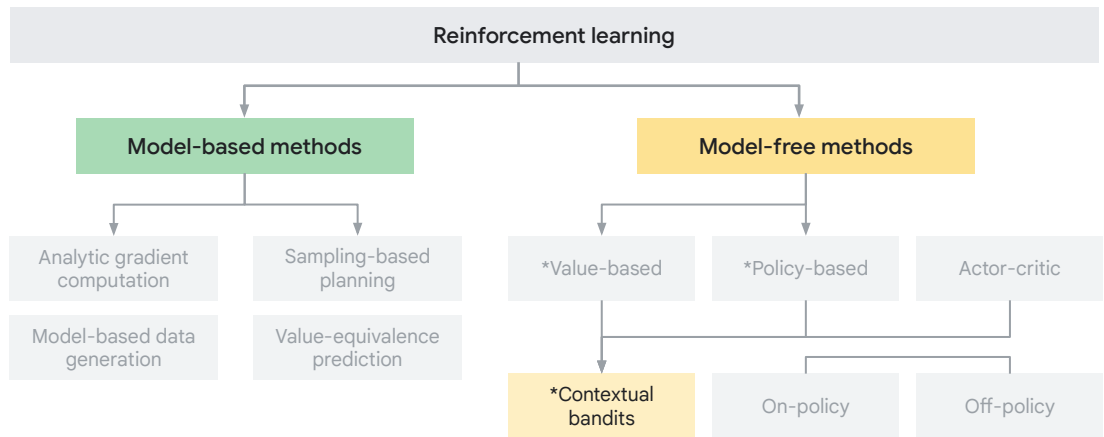
Contextual bandits

Applications of RL



Earlier in this module, we introduced the concept of various RL types. Next, we'll explore in more detail the contextual bandits type of RL.

RL methods

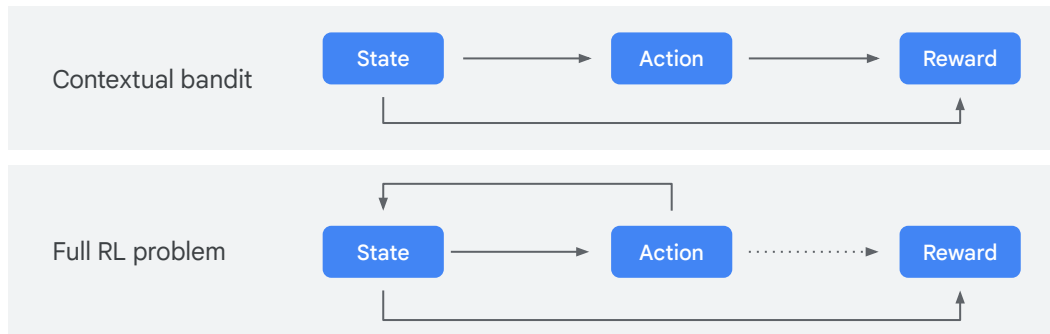


We've looked at value-based and policy-based approaches. Now we'll look at contextual bandits.

Contextual bandits

An extension of multi-armed bandits or simplified RL.

- In a sequence of trials, the agent acts based on a given context.

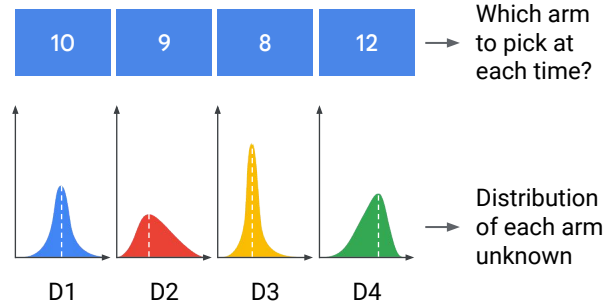


The contextual bandits approach is an extension of multi-armed bandits. A contextual multi-armed bandit problem is a simplified reinforcement learning algorithm where the agent takes an action from a set of possible actions. The agent tries to maximize the total reward of the chosen actions. The problem involves a sequence of independent trials. The agent bases its decision for next actions on a given context (side information).

What are multi-armed bandits?

An agent simultaneously attempts to:

- Explore (acquire new knowledge).
- Exploit (optimize its decisions based on existing knowledge).



First, we'll look into multi-armed bandits as a foundation and then we'll move to contextual bandits.

The bandits problem is a simplified reinforcement learning problem, which has only one time step and no state transition dynamics. In the diagram, we have four casino machines. The reward distribution is shown for pulling each arm. However, the reward distribution itself is unknown to the user and they can only infer it from repeated pulls of the arms. So, the user tries to estimate, based on past values they got, which machine arm they should pull to get the highest points. Over time they will learn the distribution and know what arm they have to pull to maximize the rewards.

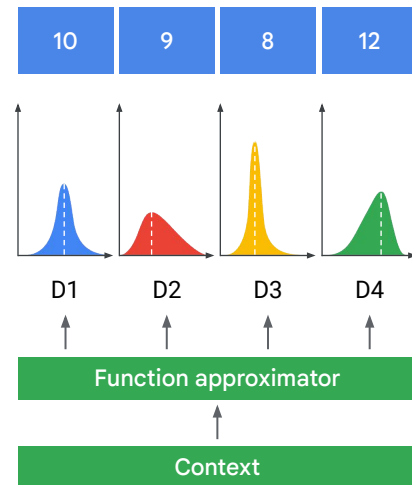
For example, a couple of times the user will pull the arm of the different machines. At some point, they realize what they learned from past iterations: that D4 has a higher probability to get higher points (12), a better distribution in terms of maximizing the long-term value.

In the multi-armed bandit approach, an agent simultaneously attempts to acquire new knowledge (exploration) and optimize its decisions based on existing knowledge (exploitation).

A metric often used to measure the progress in contextual bandits is called "regret": simply put, it is the reward it got versus the maximum reward it could have gotten. Every iteration is independent of other iterations.

What are contextual bandits?

- Each data point is a new episode.
- Value of exploration strategies is much easier to quantify/tune.
- Context can be the input feature space (recommender/personalization systems).
- A policy as the function approximator:
 - Estimating value gain from an action.



Now that you've got an idea of multi-armed bandits in general, we'll look at an extension of that called contextual bandits.

In this approach, the agent still has to choose between different arms that have an unknown distribution. However, before making the choice this time, the agent will also see a context, which is an N dimensional feature vector that the agent needs to consider.

For example, in an ad placement or recommender system, the context could be information about the user and their relevant past preferences.

The agent tries to estimate the reward of each action in interaction with the context by understanding the relation between the context and the reward distribution for each action.

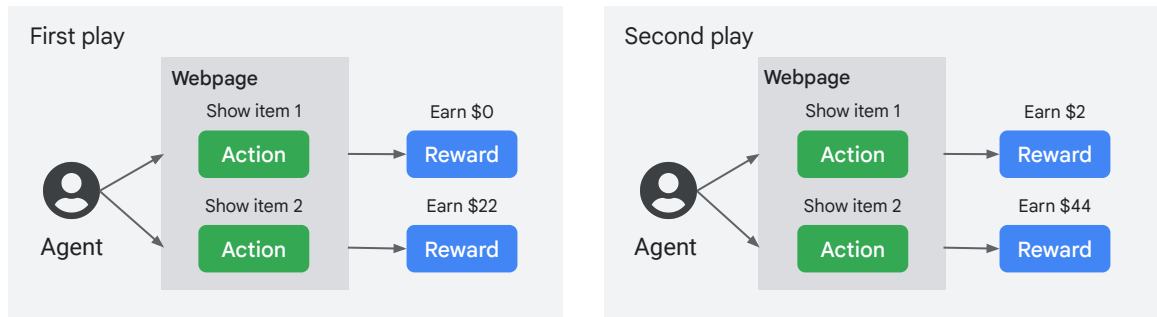
The objective remains the same as before: to minimize regret by exploring and exploiting as necessary. Each agent can then also be initialized with the policy that is the function approximator. It can be a linear or nonlinear approximator or something that estimates the Q values. The agent then takes the context in using this policy (the function approximator), then it adds what we learned before the exploration and exploitation component on top of that.

You notice one difference from other RL problems: the user doesn't have a state transition. The episode ends after the user pulls an arm, and they start from the very

beginning. There is no complex state space to traverse, making it much easier to explore, see the value of exploration strategies, and quantify and tune them.

Another analogy for how contextual bandits work is the following: imagine you wake up in the morning, and you have to choose a way to travel to work. You have four options: a bicycle, car, train, or bus. Your reward, the least amount of time in transit, depends on which travel option you choose. In multi-armed bandits, you don't have any contexts, so you choose one based on history. One time you chose to cycle, next time you chose a car, and perhaps you took the bus one time. Based on the time it's taken from the actions, you learn this distribution and try to choose the optimal arm next time. However, you don't know anything about the context: the weather. Now with the contextual bandit, you have weather information, so you wake up in the morning and you figured "hey! it's sunny, so cycling might be better." Or let's say it's rainy, so taking a bus might be better. With the extra information, you form a basis of which action you'll take.

A multi-armed bandits agent



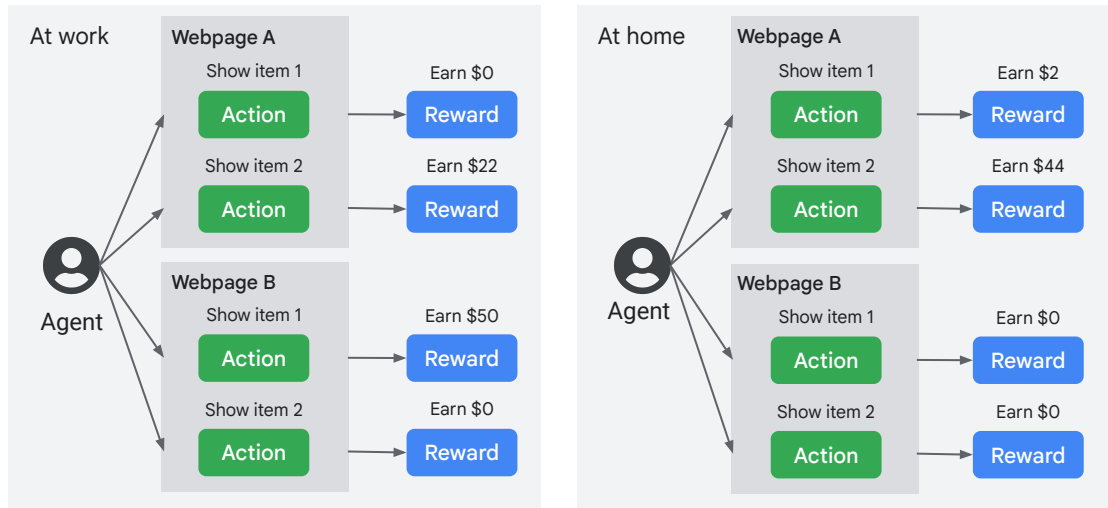
There's no state.
Every episode is independent.

Let's discuss another example of multi-armed bandits.

Here you have a multi-armed bandit without state. That is, there is no state because every play is a full episode, and it's independent of any others. Rewards received are only related to the action executed. So, the agent learns which action yields the best reward most often.

This example focuses on an online retailer and their website. They want to offer a product recommendation. In this case, you – the agent – try showing one item, then another item, and learn that showing the second item is somehow more rewarding. The user actually clicks and buys it, which earns you 22 dollars. Sometimes they will even buy it twice, in which case you earned 44 dollars. However, it's completely independent. The agent doesn't really understand why it works, especially for item two.

Contextual bandits agent



There's context. Other episodes may influence the agent.

Alternatively, we look at the problem with some context. The reward is conditional to the state of the customer environment. Rewards vary according to the state or context that the agent is operating. The agent has more data points to analyze to decide which action to take. Now you consider whether the user was on your website, whether the user got a recommendation, what time of day it is. Is the user at work or at home? What is their environment, the context of the user? Also, on what web page of my retail shop am I showing my products? For example, if it's a website about photography, showing some cameras or related products is desirable. There's also another context: where should my recommendation be placed. In that case, it can look different. Given another website and an understanding of what the user looked into in past sessions, what their interests might be, showing item one can now lead to rewards. How? Because then other users are also interested in and can buy the product. Now the agent has a better way to understand the time they should place the order and the right context.

Agenda

Introduction to reinforcement learning

RL framework and workflow

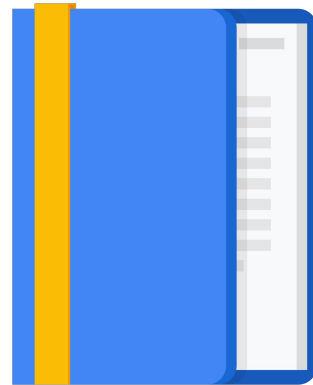
Model-based and model-free RL

Value-based RL

Policy-based RL

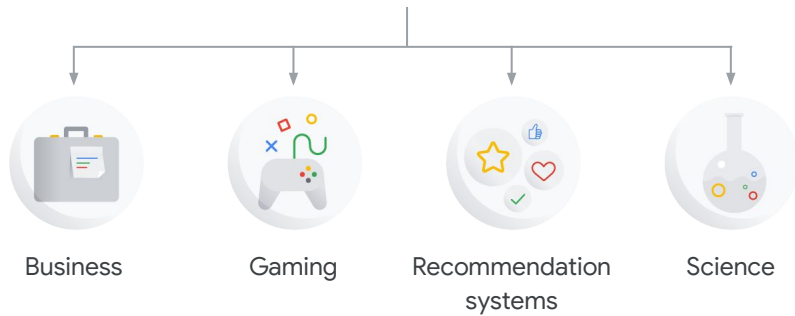
Contextual bandits

Applications of RL



Next, we'll look in more detail at different applications of reinforcement learning and where RL could be better suited than other ML types for your use cases.

Industries that use reinforcement learning

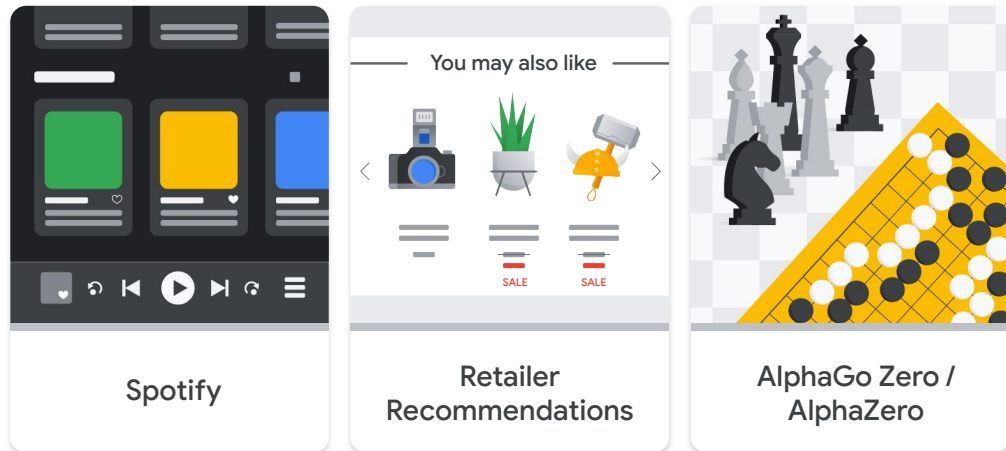


The intended application of reinforcement learning is to evolve and improve systems without human or programmatic intervention. It is used successfully in different industries for various use cases.

Some industries that use reinforcement learning include:

- Business
- Gaming
- Recommendation systems
- Science

Real-life applications



Let's look at some real-life applications that optimize for multi-objective (multiple rewards versus just one), where they optimize for themselves and all other parties involved.

- **Spotify**

Spotify and other similar cloud audio platforms give music recommendations. But there's more. Their goal is to ensure both that you get the best possible selection of songs in your lists and that artists get visibility and traction from their side. So these platforms have to balance two objectives.

- **Retailer recommendations**

RL is well suited for online ecommerce retailer shops. Normally retailers collect customer behavior data on their server, including which/when pages are opened, what was searched for, and whether product recommendations were selected. When customers spend more time browsing the retailer's site, the behavior data the retailer has to work with increases. The goal is that our agent can influence the recommendations.

So why don't preset rules work well? Unfortunately, not all customers behave the same way. Some know what they want and search for something in particular. They might spend longer on average on each page they visit but skip category overview pages. Other customers browse to find inspiration. The variations can get too complex for simple rules.

We want an agent that adapts to various customer behaviors. The agent learns to differentiate between groups and determines the most appropriate action for each. So RL helps a retailer increase sales through customer-specific recommendations rather than perform the same action for every customer based on a set of strict rules. The retailer has better sales and the customers have more relevant recommendations.

- **AlphaGo Zero/DeepMind**

Google DeepMind's AlphaGo Zero is an example of an incredible achievement. This software shows how an agent can be trained in the highly complex domain of Go from a blank slate, with no human expert play used as training data, to a superhuman level.

AlphaGo Zero is known for not only beating the best player on earth, but when the algorithm was recreated without human input, it was even better in a shorter time frame.

There are so many more examples, but here are a few more to think about:

- Play Store
- Strategy-planning
- Robotics for industrial automation
- Aircraft control
- Machine learning and data processing
- Training systems that provide custom instruction and materials according to the requirements of learners

In summary, the problems mentioned here can be thought of as ones where the solution can both optimize for individuals and all the parties.

When to use RL rather than SL or USL

Supervised learning:

Prediction problems
Short term
Offline training on cold historic data
IID data
No trial and error required
Suitable for lower variance / dynamism
Transfer learning
Differentiable non-noisy loss

Reinforcement learning:

Control/optimization/decision-making problems
Optimized for long term/delayed value oriented
Real-time training or offline simulation
Non-IID data possible
Trial and error necessary
Lower variance/dynamism
Transfer learning not yet possible
Differentiability not critical, noisy reward OK

Use supervised learning and not reinforcement learning when:

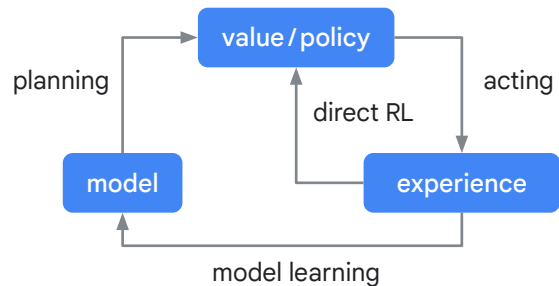
- You have a *prediction problem* such as regression or classification problems with labels. You want to ensure that the distance between a predicted distribution and the actual distribution is as low as possible. You want to ensure that the prediction matches reality. It is more for an instantaneous prediction at the time step.
- The outcome is *short term*.
- You can do *offline training on cold historic data*. In these situations, you can train a model-based agent on the static data.
- *All data is IID* (independent and identically distributed, which means that there are no overall trends). In supervised learning, the data is often IID.
- *No trial and error* is required, because you have the target, the input features, and the environment is static.
- The problem is suitable for *lower variance and dynamism*.
- You want to apply *transfer learning*, which is very mature in supervised learning, especially for image and text-based tasks. You can pre-train a model and then transfer all the knowledge to another task, which may not be identical to the previous one, and require much less training.
- You want *differentiable non-noisy loss*. In supervised learning, you want the loss function to be differentiable and preferably not noisy. Loss might not be the exact metric you're trying to optimize for, but it is the closest proxy, which is differentiable to the actual business metric.

Use reinforcement learning and not supervised/unsupervised learning when:

- You have *control/optimization/decision-making* problems where you are both predicting something and responsible for taking an action among the set of actions.
- You want to *optimize for the long term* where the outcome is delayed and value-oriented. You get the reward after a longer time for optimizing a strategic long-term goal.
- You want *real-time training* or *offline simulation*. You can do real-time training or offline simulation in RL, but you normally don't learn from static data unless it's contextual bandit. You cannot simply dump data into a data warehouse like BigQuery and learn from it.
- It's possible to have *non-IID data*. In RL, the current state, the action you take in the current state, the next state and the next situation you move to might be correlated. That is, if you move to the right, then the final state you finish in is probably in the right side of the room. Due to this correlation, data is not necessarily IID.
- *Trial and error are necessary* because the state space is large, and the agent must try lots of scenarios before it learns optimal decisions. There is no static definition. This trial and error is necessary, either by real time or by an offline simulation.
- You want *lower variance/dynamism*.
- *Transfer learning is not yet possible*. In RL, it is not mainstream yet.
- Outcomes *do not need to be differentiable* (a *noisy reward is OK*). Unlike supervised learning, where you try to minimize the loss, minimize the distance between the predicted and the target distribution instead of trying to maximize the value. In RL, the reward and value functions don't need to be differentiable, and it's OK if the reward is noisy too. It's preferable if the reward is not noisy, but the restriction to differentiability is not critical. You can directly optimize for the metric that you care for and not necessarily a proxy.

Why not combine model-based and model-free approaches?

- Combining allows the agent to learn from both real experience online and the trajectories generated by the model.
- Agent uses both sources to take an action on the environment.



We have looked at model-free approaches where the agent learns directly from its interaction with the environment. We have also looked at model-based approaches where the model learns from the environment and then the agent learns from the model.

You might be wondering, why not combine both approaches? Combining lets the agent learn from both real experience online and the trajectories generated by the model (the offline planning phase). The agent uses both sources to act on the environment.

Just like actor critic approaches combine both value and policy approaches, the agent can benefit from a combination of model-free and model-based methods.

Lab intro

Applying reinforcement learning



In this hands-on lab, you will build a contextual bandits agent in order to recommend to a user another movie to watch (based on the Movielens dataset).

You will first instantiate a Vertex AI Workbench notebook instance, eventually load data to Tensorflow (TF), and build an agent using the TF Agents library.

Most of the lab, in particular the creation of the agent, will be inside a Jupyter notebook that you will clone into your Workbench instance.

Although you can copy and paste commands from the lab to the appropriate place, students should type the commands themselves to reinforce their understanding of the core concepts.

What you'll do:

- Perform the initial setup of a Vertex AI Workbench Notebook.
- Clone a Jupyter notebook into your Vertex AI Workbench instance.
- Initialize the environment and the agent.
- Create and train the model.
- Apply inference on the trained model.
- Evaluate the model with Vertex AI TensorBoard (TB).

Lab review

Applying reinforcement learning



Now that you've completed the lab you know how to:

- Perform the initial setup of a Vertex AI Workbench Notebook.
- Clone a Jupyter notebook into your Vertex AI Workbench instance.
- Initialize the environment and the agent.
- Create and train the model.
- Apply inference on the trained model.
- Evaluate the model with Vertex AI TensorBoard (TB).

