Part 1: Key Concepts in RL

Table of Contents

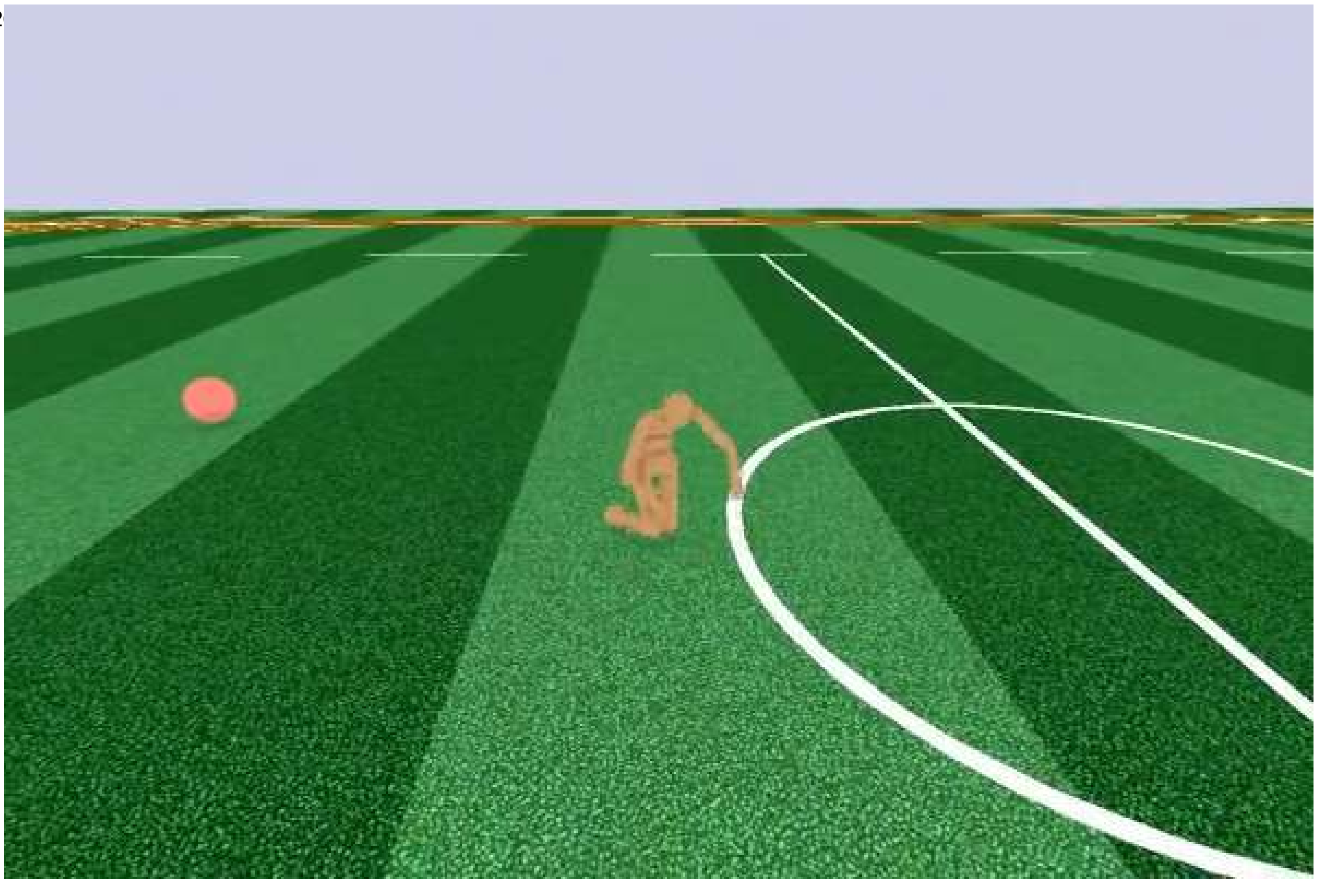
Welcome to our introduction to reinforcement learning! Here, we aim to acquaint you with

- the language and notation used to discuss the subject,
- a high-level explanation of what RL algorithms do (although we mostly avoid the question of *how* they do it),
- and a little bit of the core math that underlies the algorithms.

In a nutshell, RL is the study of agents and how they learn by trial and error. It formalizes the idea that rewarding or punishing an agent for its behavior makes it more likely to repeat or forego that behavior in the future.

What Can RL Do?

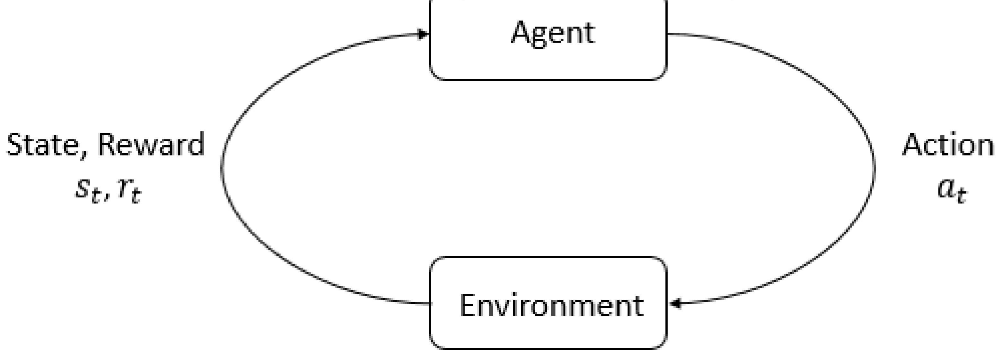
RL methods have recently enjoyed a wide variety of successes. For example, it's been used to teach computers to control robots in simulation...



...and in the real world...

It’s also famously been used to create breakthrough AIs for sophisticated strategy games, most notably Go and Dota, taught computers to play Atari games from raw pixels, and trained simulated robots to follow human instructions.

Key Concepts and Terminology



Agent-environment interaction loop.

The main characters of RL are the **agent** and the **environment**. The environment is the world that the agent lives in and interacts with. At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it, but may also change on its own.

The agent also perceives a **reward** signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward, called **return**. Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.

To talk more specifically what RL does, we need to introduce additional terminology. We need to talk about

- states and observations,
- action spaces,
- policies,
- trajectories,
- different formulations of return,
- the RL optimization problem,
- and value functions.

States and Observations

A **state** s is a complete description of the state of the world. There is no information about the world which is hidden from the state. An **observation** o is a partial description of a state, which may omit information.

In deep RL, we almost always represent states and observations by a real-valued vector, matrix, or higher-order tensor. For instance, a visual observation could be represented by the RGB matrix of its pixel values; the state of a robot might be represented by its joint angles and velocities.

When the agent is able to observe the complete state of the environment, we say that the environment is **fully observed**. When the agent can only see a partial observation, we say that the environment is **partially observed**.

You Should Know

Reinforcement learning notation sometimes puts the symbol for state, s, in places where it would be technically more appropriate to write the symbol for observation, o. Specifically, this happens when talking about how the agent decides an action: we often signal in notation that the action is conditioned on the state, when in practice, the action is conditioned on the observation because the agent does not have access to the state.

In our guide, we'll follow standard conventions for notation, but it should be clear from context which is meant. If something is unclear, though, please raise an issue! Our goal is to teach, not to confuse.

Action Spaces

Different environments allow different kinds of actions. The set of all valid actions in a given environment is often called the **action space**. Some environments, like Atari and Go, have **discrete action spaces**, where only a finite number of moves are available to the agent. Other environments, like where the agent controls a robot in a physical world, have **continuous action spaces**. In continuous spaces, actions are real-valued vectors.

This distinction has some quite-profound consequences for methods in deep RL. Some families of algorithms can only be directly applied in one case, and would have to be substantially reworked for the other.

Policies

A **policy** is a rule used by an agent to decide what actions to take. It can be deterministic, in which case it is usually denoted by μ:

$$a_t = \mu(s_t),$$

or it may be stochastic, in which case it is usually denoted by π:

$$a_t \sim \pi(\cdot|s_t).$$

Because the policy is essentially the agent's brain, it's not uncommon to substitute the word "policy" for "agent", eg saying "The policy is trying to maximize reward."

In deep RL, we deal with **parameterized policies**: policies whose outputs are computable functions that depend on a set of parameters (eg the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm.

We often denote the parameters of such a policy by θ or ϕ, and then write this as a subscript on the policy symbol to highlight the connection:

$$a_t = \mu_\theta(s_t)$$
$$a_t \sim \pi_\theta(\cdot|s_t).$$

Deterministic Policies

Example: Deterministic Policies. Here is a code snippet for building a simple deterministic policy for a continuous action space in Tensorflow:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
actions = tf.layers.dense(net, units=act_dim, activation=None)
```

where `mlp` is a function that stacks multiple `dense` layers on top of each other with the given sizes and activation.

Stochastic Policies

The two most common kinds of stochastic policies in deep RL are **categorical policies** and **diagonal Gaussian policies**.

Categorical policies can be used in discrete action spaces, while diagonal Gaussian policies are used in continuous action spaces.

Two key computations are centrally important for using and training stochastic policies:

- sampling actions from the policy,
- and computing log likelihoods of particular actions, $\log \pi_\theta(a|s)$.

In what follows, we'll describe how to do these for both categorical and diagonal Gaussian policies.

Categorical Policies

A categorical policy is like a classifier over discrete actions. You build the neural network for a categorical policy the same way you would for a classifier: the input is the observation, followed by some number of layers (possibly convolutional or densely-connected, depending on the kind of input), and then you have one final linear layer that gives you logits for each action, followed by a softmax to convert the logits into probabilities.

Sampling. Given the probabilities for each action, frameworks like Tensorflow have built-in tools for sampling. For example, see the tf.distributions.Categorical documentation, or tf.multinomial.

Log-Likelihood. Denote the last layer of probabilities as $P_\theta(s)$. It is a vector with however many entries as there are actions, so we can treat the actions as indices for the vector. The log likelihood for an action a can then be obtained by indexing into the vector:

$$\log \pi_\theta(a|s) = \log \left[P_\theta(s)\right]_a .$$

❗ Diagonal Gaussian Policies

A multivariate Gaussian distribution (or multivariate normal distribution, if you prefer) is described by a mean vector, μ, and a covariance matrix, Σ. A diagonal Gaussian distribution is a special case where the covariance matrix only has entries on the diagonal. As a result, we can represent it by a vector.

A diagonal Gaussian policy always has a neural network that maps from observations to mean actions, $\mu_\theta(s)$. There are two different ways that the covariance matrix is typically represented.

The first way: There is a single vector of log standard deviations, $\log \sigma$, which is **not** a function of state: the $\log \sigma$ are standalone parameters. (You Should Know: our implementations of VPG, TRPO, and PPO do it this way.)

The second way: There is a neural network that maps from states to log standard deviations, $\log \sigma_\theta(s)$. It may optionally share some layers with the mean network.

Note that in both cases we output log standard deviations instead of standard deviations directly. This is because log stds are free to take on any values in $(-\infty, \infty)$, while stds must be nonnegative. It's easier to train parameters if you don't have to enforce those kinds of constraints. The standard deviations can be obtained immediately from the log standard deviations by exponentiating them, so we do not lose anything by representing them this way.

Sampling. Given the mean action $\mu_\theta(s)$ and standard deviation $\sigma_\theta(s)$, and a vector z of noise from a spherical Gaussian ($z \sim \mathcal{N}(0, I)$), an action sample can be computed with

$$a = \mu_\theta(s) + \sigma_\theta(s) \odot z,$$

where $\odot$ denotes the elementwise product of two vectors. Standard frameworks have built-in ways to compute the noise vectors, such as tf.random_normal. Alternatively, you can just provide the mean and standard deviation directly to a tf.distributions.Normal object and use that to sample.

Log-Likelihood. The log-likelihood of a k -dimensional action a, for a diagonal Gaussian with mean $\mu = \mu_\theta(s)$ and standard deviation $\sigma = \sigma_\theta(s)$, is given by

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i-\mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right).$$

Trajectories

A trajectory τ is a sequence of states and actions in the world,

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

The very first state of the world, s_0, is randomly sampled from the **start-state distribution**, sometimes denoted by ρ_0:

$$s_0 \sim \rho_0(\cdot).$$

State transitions (what happens to the world between the state at time t, s_t, and the state at $t+1$, s_{t+1}), are governed by the natural laws of the environment, and depend on only the most recent action, a_t. They can be either deterministic,

$$s_{t+1} = f(s_t, a_t)$$

or stochastic,

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

Actions come from an agent according to its policy.

You Should Know

Trajectories are also frequently called **episodes** or **rollouts**.

Reward and Return

The reward function R is critically important in reinforcement learning. It depends on the current state of the world, the action just taken, and the next state of the world:

$$r_t = R(s_t, a_t, s_{t+1})$$

although frequently this is simplified to just a dependence on the current state, $r_t = R(s_t)$, or state-action pair $r_t = R(s_t, a_t)$.

The goal of the agent is to maximize some notion of cumulative reward over a trajectory, but this actually can mean a few things. We'll notate all of these cases with $R(\tau)$, and it will either be clear from context which case we mean, or it won't matter (because the same equations will apply to all cases).

One kind of return is the **finite-horizon undiscounted return**, which is just the sum of rewards obtained in a fixed window of steps:

$$R(\tau) = \sum_{t=0}^{T} r_t.$$

Another kind of return is the **infinite-horizon discounted return**, which is the sum of all rewards *ever* obtained by the agent, but discounted by how far off in the future they're obtained. This formulation of reward includes a discount factor $\gamma \in (0,1)$:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

Why would we ever want a discount factor, though? Don't we just want to get *all* rewards? We do, but the discount factor is both intuitively appealing and mathematically convenient. On an intuitive level: cash now is better than cash later. Mathematically: an infinite-horizon sum of rewards may not converge to a finite value, and is hard to deal with in equations. But with a discount factor and under reasonable conditions, the infinite sum converges.

You Should Know

While the line between these two formulations of return are quite stark in RL formalism, deep RL practice tends to blur the line a fair bit—for instance, we frequently set up algorithms to optimize the undiscounted return, but use discount factors in estimating **value functions**.

The RL Problem

Whatever the choice of return measure (whether infinite-horizon discounted, or finite-horizon undiscounted), and whatever the choice of policy, the goal in RL is to select a policy which maximizes **expected return** when the agent acts according to it.

To talk about expected return, we first have to talk about probability distributions over trajectories.

Let's suppose that both the environment transitions and the policy are stochastic. In this case, the probability of a T -step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

The expected return (for whichever measure), denoted by $J(\pi)$, is then:

$$J(\pi) = \int_{\tau} P(\tau|\pi) R(\tau) = \underset{\tau\sim\pi}{\mathrm{E}}\left[R(\tau)\right].$$

The central optimization problem in RL can then be expressed by

$$\pi^* = \arg \max_{\pi} J(\pi),$$

with π^* being the **optimal policy**.

Value Functions

It's often useful to know the **value** of a state, or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. **Value functions** are used, one way or another, in almost every RL algorithm.

There are four main functions of note here.

1. The **On-Policy Value Function**, $V^{\pi}(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\left| s_0 = s\right.\right]$$

2. The **On-Policy Action-Value Function**, $Q^{\pi}(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s,a) = \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\left| s_0 = s, a_0 = a\right.\right]$$

3. The **Optimal Value Function**, $V^*(s)$, which gives the expected return if you start in state s and always act according to the *optimal* policy in the environment:

$$V^*(s) = \max_{\pi} \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\left| s_0 = s\right.\right]$$

4. The **Optimal Action-Value Function**, $Q^*(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the *optimal* policy in the environment:

$$Q^*(s,a) = \max_{\pi} \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\left| s_0 = s, a_0 = a\right.\right]$$

You Should Know

When we talk about value functions, if we do not make reference to time-dependence, we only mean expected **infinite-horizon discounted return**. Value functions for finite-horizon undiscounted return would need to accept time as an argument. Can you think about why? Hint: what happens when time's up?

❗ You Should Know

There are two key connections between the value function and the action-value function that come up pretty often:

$$V^{\pi}(s) = \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a)\right],$$

and

$$V^*(s) = \max_a Q^*(s,a).$$

These relations follow pretty directly from the definitions just given: can you prove them?

The Optimal Q-Function and the Optimal Action

There is an important connection between the optimal action-value function $Q^*(s,a)$ and the action selected by the optimal policy. By definition, $Q^*(s,a)$ gives the expected return for starting in state s, taking (arbitrary) action a, and then acting according to the optimal policy forever after.

The optimal policy in s will select whichever action maximizes the expected return from starting in s. As a result, if we have Q^*, we can directly obtain the optimal action, $a^*(s)$, via

$$a^*(s) = \arg\max_a Q^*(s,a).$$

Note: there may be multiple actions which maximize $Q^*(s,a)$, in which case, all of them are optimal, and the optimal policy may randomly select any of them. But there is always an optimal policy which deterministically selects an action.

Bellman Equations

All four of the value functions obey special self-consistency equations called **Bellman equations**. The basic idea behind the Bellman equations is this:

> The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.

The Bellman equations for the on-policy value functions are

$$\begin{aligned} V^{\pi}(s) &= \underset{\substack{a\sim\pi \\ s'\sim P}}{\mathrm{E}}\left[r(s,a) + \gamma V^{\pi}(s')\right], \\ Q^{\pi}(s,a) &= \underset{s'\sim P}{\mathrm{E}}\left[r(s,a) + \gamma \underset{a'\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s',a')\right]\right], \end{aligned}$$

where $s' \sim P$ is shorthand for $s' \sim P(\cdot|s,a)$, indicating that the next state s' is sampled from the environment's transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

The Bellman equations for the optimal value functions are

$$V^*(s) = \max_a \underset{s'\sim P}{\mathrm{E}} \left[r(s,a) + \gamma V^*(s')\right],$$
$$Q^*(s,a) = \underset{s'\sim P}{\mathrm{E}} \left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right].$$

The crucial difference between the Bellman equations for the on-policy value functions and the optimal value functions, is the absence or presence of the $\max$ over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

You Should Know

The term "Bellman backup" comes up quite frequently in the RL literature. The Bellman backup for a state, or state-action pair, is the right-hand side of the Bellman equation: the reward-plus-next-value.

Advantage Functions

Sometimes in RL, we don't need to describe how good an action is in an absolute sense, but only how much better it is than others on average. That is to say, we want to know the relative **advantage** of that action. We make this concept precise with the **advantage function.**

The advantage function $A^\pi(s,a)$ corresponding to a policy π describes how much better it is to take a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming you act according to π forever after. Mathematically, the advantage function is defined by

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

You Should Know

We'll discuss this more later, but the advantage function is crucially important to policy gradient methods.

(Optional) Formalism

So far, we've discussed the agent's environment in an informal way, but if you try to go digging through the literature, you're likely to run into the standard mathematical formalism for this setting: **Markov Decision Processes** (MDPs). An MDP is a 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$, where

- S is the set of all valid states,
- A is the set of all valid actions,
- $R : S \times A \times S \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,

- $P : S \times A \to \mathcal{P}(S)$ is the transition probability function, with $P(s'|s,a)$ being the probability of transitioning into state s' if you start in state s and take action a,
- and ρ_0 is the starting state distribution.

The name Markov Decision Process refers to the fact that the system obeys the Markov property: transitions only depend on the most recent state and action, and no prior history.