The Complete Reinforcement Learning Dictionary

The Reinforcement Learning Terminology, A to Z

Shaked Zychlinski

Feb 23, 2019 · 14 min read ★

Whenever I begin learning a subject which is new to me, I find the hardest thing to cope with is its new terminology. Every field have many terms and definitions which are completely obscure to an outsider, and can make a newcomer's first step quite difficult.

When I made my first step into the world or Reinforcement Learning, I was quite overwhelmed by the new terms which popped-up every other line, and it always surprised me how behind those complex words stood quite simple and logical ideas. I therefore decided to write them all down in my own words, so I'll always be able to look them up in case I forget. This is how this dictionary came to be.

I will do my best to try and keep on updating this dictionary. Feel free to let me know if I've missed anything important or got something wrong.

. . .

The Dictionary

Action-Value Function: See *Q-Value*.

Actions: Actions are the *Agent's* methods which allow it to interact and change its *environment*, and thus transfer between *states*. Every action performed by the Agent yields a *reward* from the environment. The decision of which action to choose is made by the *policy*.

Actor-Critic: When attempting to solve a *Reinforcement Learning* problem, there are two main methods one can choose from: calculating the *Value Functions* or *Q-Values* of each state and choosing actions according to those, or directly compute a *policy* which defines the probabilities each action should be taken depending on the current state, and act

according to it. Actor-Critic algorithms combine the two methods in order to create a more robust method. A great illustrated-comics explanation can be found here.

Advantage Function: Usually denoted as *A(s,a)*, the Advantage function is a measure of how much is a certain *action* a good or bad decision given a certain *state* — or more simply, what is the advantage of selecting a certain action from a certain state. It is defined mathematically as:

$$A(s,a) = \mathbf{E}\Big[r(s,a) - r(s)\Big]$$

where *r(s,a)* is the expected *reward* of action *a* from state *s*, and *r(s)* is the expected reward of the entire state *s*, before an action was selected. It can also be viewed as:

$$A(s,a) = Q(s,a) - V(s)$$

where *Q(s,a)* is the *Q Value* and *V(s)* is the *Value function*.

Agent: The learning and *acting* part of a *Reinforcement Learning* problem, which tries to maximize the *rewards* it is given by the *Environment*. Putting it simply, the Agent is the model which you try to design.

Bandits: Formally named "k-Armed Bandits" after the nickname "one-armed bandit" given to slot-machines, these are considered to be the simplest type of *Reinforcement Learning* tasks. Bandits have no different *states*, but only one — and the *reward* taken under consideration is only the immediate one. Hence, bandits can be thought of as having single-state *episodes*. Each of the k-arms is considered an *action*, and the objective is to learn the *policy* which will maximize the expected reward after each action (or arm-pulling).
Contextual Bandits are a slightly more complex task, where each state may be different and affect the outcome of the actions — hence every time the *context* is different. Still, the task remains a single-state episodic task, and one context cannot have an influence on others.

Bellman Equation: Formally, Bellman equation defines the relationships between a given *state* (or state-*action* pair) to its successors. While many forms exist, the most common one usually encountered in *Reinforcement Learning* tasks is the Bellman equation for the optimal *Q-Value* , which is given by:

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

or when no uncertainty exists (meaning, probabilities are either 1 or 0):

$$Q^*(s,a) = r(s,a) + \gamma \max_{a'} Q^*(s',a')$$

where the asterisk sign indicates *optimal value*. Some algorithms, such as *Q-Learning*, are basing their learning procedure over it.

Continuous Tasks: *Reinforcement Learning* tasks which are not made of *episodes*, but rather last forever. This tasks have no terminal *states*. For simplicity, they are usually assumed to be made of one never-ending episode.

Deep Q-Networks (DQN): See *Q-Learning*

Deep Reinforcement Learning: The use of a *Reinforcement Learning* algorithm with a deep neural network as an approximator for the learning part. This is usually done in order to cope with problems where the number of possible *states* and *actions* scales fast, and an exact solution in no longer feasible.

Discount Factor (γ): The discount factor, usually denoted as γ, is a factor multiplying the future expected *reward*, and varies on the range of [0,1]. It controls the importance of the future rewards versus the immediate ones. The lower the discount factor is, the less important future rewards are, and the *Agent* will tend to focus on *actions* which will yield immediate rewards only.

Environment: Everything which isn't the *Agent*; everything the Agent can interact with, either directly or indirectly. The environment changes as the Agent performs *actions*; every such change is considered a *state*-transition. Every action the Agent performs yields a *reward* received by the Agent.

Episode: All *states* that come in between an initial-state and a terminal-state; for example: one game of Chess. The *Agent's* goal it to maximize the total *reward* it receives during an episode. In situations where there is no terminal-state, we consider an infinite episode. It is important to remember that different episodes are completely independent of one another.

Episodic Tasks: *Reinforcement Learning* tasks which are made of different *episodes* (meaning, each episode has a terminal *state*).

Expected Return: Sometimes referred to as "overall reward" and occasionally denoted as *G*, is the expected *reward* over an entire *episode*.

Experience Replay: As *Reinforcement Learning* tasks have no pre-generated training sets which they can learn from, the *Agent* must keep records of all the *state*-transitions it encountered so it can learn from them later. The memory-buffer used to store this is often referred to as *Experience Replay*. There are several types and architectures of these memory buffers — but some very common ones are the cyclic memory buffers (which makes sure the Agent keeps training over its new behavior rather than things that might no longer be relevant) and reservoir-sampling-based memory buffers (which guarantees each state-transition recorded has an even probability to be inserted to the buffer).

Exploitation & Exploration: *Reinforcement Learning* tasks have no pre-generated training sets which they can learn from — they create their own *experience* and learn "on the fly". To be able to do so, the *Agent* needs to try many different *actions* in many different *states* in order to try and learn all available possibilities and find the path which will maximize its overall *reward*; this is known as *Exploration*, as the Agent explores the *Environment*. On the other hand, if all the Agent will do is explore, it will never maximize the overall reward — it must also use the information it learned to do so. This is known as *Exploitation*, as the Agent exploits its knowledge to maximize the rewards it receives. The trade-off between the two is one of the greatest challenges of Reinforcement

Learning problems, as the two must be balanced in order to allow the Agent to both explore the environment enough, but also exploit what it learned and repeat the most rewarding path it found.

Greedy Policy, ε-Greedy Policy: A greedy *policy* means the *Agent* constantly performs the *action* that is believed to yield the highest expected *reward*. Obviously, such a policy will not allow the Agent to *explore* at all. In order to still allow some exploration, an ε-greedy policy is often used instead: a number (named ε) in the range of [0,1] is selected, and prior selecting an action, a random number in the range of [0,1] is selected. if that number is larger than ε, the greedy action is selected — but if it's lower, a random action is selected. Note that if $\varepsilon=0$, the policy becomes the greedy policy, and if $\varepsilon=1$, always explore.

k-Armed Bandits: See *Bandits*.

Markov Decision Process (MDP): The Markov Property means that each *state* is dependent solely on its preceding state, the selected *action* taken from that state and the *reward* received immediately after that action was executed. Mathematically, it means: $s' = s'(s,a,r)$, where s' is the future state, s is its preceding state and a and r are the action and reward. No prior knowledge of what happened before s is needed — the Markov Property assumes that s holds all the relevant information within it. A Markov Decision Process is decision process based on these assumptions.

Model-Based & Model-Free: Model-based and model-free are two different approaches an *Agent* can choose when trying to optimize its *policy*. This is best explained using an example: assume you're trying to learn how to play Blackjack. You can do so in two ways: one, you calculate in advance, before the game begins, the winning probabilities of all *states* and all the state-transition probabilities given all the possible *actions*, and then simply act according to you calculations. The second option is to simply play without any prior knowledge, and gain information using "trial-and-error". Note that using the first approach, you're basically *modeling* your *environment*, while the second approach requires no information about the environment. This is exactly the difference between model-based and model-free; the first method is model-based, while the latter is model-free.

Monte Carlo (MC): Monte Carlo methods are algorithms which use repeated random sampling in order to achieve a result. They are used quite often in *Reinforcement Learning* algorithms to obtain expected values; for example — calculating a *state Value function* by returning to the same state over and over again, and averaging over the actual cumulative *reward* received each time.

On-Policy & Off-Policy: Every *Reinforcement Learning* algorithm must follow some *policy* in order to decide which *actions* to perform at each *state*. Still, the learning procedure of the algorithm doesn't have to take into account that policy while learning. Algorithms which concern about the policy which yielded past state-action decisions are referred to as *on-policy* algorithms, while those ignoring it are known as *off-policy*.
A well known off-policy algorithm is *Q-Learning*, as its update rule uses the action which will yield the highest *Q-Value*, while the actual policy used might restrict that action or choose another. The on-policy variation of Q-Learning is known as *Sarsa*, where the update rule uses the action chosen by the followed policy.

One-Armed Bandits: See *Bandits*.

One-Step TD: See *Temporal Difference*.

Policy (π): The policy, denoted as π (or sometimes $\pi(a|s)$), is a mapping from some *state* s to the probabilities of selecting each possible *action* given that state. For example, a *greedy policy* outputs for every state the action with the highest expected *Q-Value*.

Q-Learning: Q-Learning is an *off-policy Reinforcement Learning* algorithm, considered as one of the very basic ones. In its most simplified form, it uses a table to store all *Q-Values* of all possible *state*-*action* pairs possible. It updates this table using the *Bellman equation*, while action selection is usually made with an *ε-greedy policy*.
In its simplest form (no uncertainties in *state*-transitions and expected *rewards*), the update rule of Q-Learning is:

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

A more complex version of it, though far more popular, is the *Deep Q-Network* variant (which is sometimes even referred to simply as *Deep Q-Learning* or just *Q-Learning*). This variant replaces the state-action table with a neural network in order to cope with large-scale tasks, where the number of possible state-action pairs can be enormous. You can find a tutorial for this algorithm in this blogpost.

Q Value (Q Function): Usually denoted as $Q(s,a)$ (sometimes with a π subscript, and sometimes as $Q(s,a;\theta)$ in *Deep RL*), Q Value is a measure of the overall expected *reward* assuming the *Agent* is in *state* s and performs *action* a, and then continues playing until the end of the *episode* following some *policy* π. Its name is an abbreviation of the word "Quality", and it is defined mathematically as:

$$Q(s,a) = \mathbf{E}\left[\sum_{n=0}^{N} \gamma^n r_n\right]$$

where N is the number of states from state s till the terminal state, γ is the *discount factor* and r^0 is the immediate reward received after performing action a in state s.

REINFORCE Algorithms: REINFORCE algorithms are a family of *Reinforcement Learning* algorithms which update their *policy* parameters according to the gradient of the policy with respect to the policy-parameters [paper]. The name is typically written using capital letters only, as it's originally an acronym for the original algorithms group design: "**RE**ward **I**ncrement = **N**onnegative **F**actor x **O**ffset **R**einforcement x **C**haracteristic **E**ligibility" [source]

Reinforcement Learning (RL): Reinforcement Learning is, like Supervised Learning and Unsupervised Learning, one the main areas of Machine Learning and Artificial Intelligence. It is concerned with the learning process of an arbitrary being, formally known as an *Agent*, in the world surrounding it, known as the *Environment*. The Agent seeks to maximize the *rewards* it receives from the Environment, and performs different *actions* in order to learn how the Environment responds and gain more rewards. One of the greatest challenges of RL tasks is to associate actions with postponed rewards — which are rewards received by the Agent long after the reward-generating action was

made. It is therefore heavily used to solve different kind of games, from Tic-Tac-Toe, Chess, Atari 2600 and all the way to Go and StarCraft.

Reward: A numerical value received by the *Agent* from the *Environment* as a direct response to the Agent's *actions*. The Agent's goal is to maximize the overall reward it receives during an *episode*, and so rewards are the motivation the Agent needs in order to act in a desired behavior. All actions yield rewards, which can be roughly divided to three types: *positive rewards* which emphasize a desired action, *negative rewards* which emphasize an action the Agent should stray away from, and *zero*, which means the Agent didn't do anything special or unique.

Sarsa: The Sarsa algorithm is pretty much the *Q-Learning* algorithm with a slight modification in order to make it an *on-policy* algorithm. The Q-Learning update rule is based on the *Bellman equation* for the optimal *Q-Value*, and so in the case on no uncertainties in *state*-transitions and expected *rewards*, the Q-Learning update rule is:

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

In order to transform this into an on-policy algorithm, the last term is modified:

$$Q(s,a) = r(s,a) + \gamma Q(s',a')$$

when here, both *actions* a and a' are chosen by the same *policy*. The name of the algorithm is derived from its update rule, which is based on (s,a,r,s',a'), all coming from the same policy.

State: Every scenario the *Agent* encounters in the *Environment* is formally called a *state*. The Agent transitions between different states by performing *actions*. It is also worth mentioning the *terminal states*, which mark the end of an *episode*. There are no possible states after a terminal state has been reached, and a new episode begins. Quite often, a

terminal state is represented as a special state where all actions transition to the same terminal state with *reward* 0.

State-Value Function: See *Value Function*.

Temporal-Difference (TD): Temporal Difference is a learning method which combines both Dynamic Programming and *Monte Carlo* principles; it learns "on the fly" similarly to Monte Carlo, yet updates its estimates like Dynamic Programming. One of the simplest Temporal Difference algorithms it known as *one-step TD* or *TD(0)*. It updates the *Value Function* according to the following update rule:

$$V(s_t) = V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

where V is the Value Function, s is the *state*, r is the *reward*, γ is the *discount factor*, α is a learning rate, t is the time-step and the '=' sign is used as an update operator and not equality. The term found in the squared brackets is known as the *temporal difference error*.

Terminal State: See *State*.

Upper Confident Bound (UCB): UCB is an *exploration* method which tries to ensure that each *action* is well explored. Consider an exploration *policy* which is completely random — meaning, each possible action has the same chance of being selected. There is a chance that some actions will be explored much more than others. The less an action is selected, the less confident the *Agent* can be about its expected *reward*, and the its *exploitation* phase might be harmed. Exploration by UCB takes into account the number of times each action was selected, and gives extra weight to those less-explored. Formalizing this mathematically, the selected action is picked by:

$$\text{action} = \underset{a}{\text{argmax}}\left[R(a) + c\sqrt{\frac{\ln t}{N(a)}}\right]$$

where *R(a)* is the expected overall reward of action *a*, *t* is the number of steps taken (how many actions were selected overall), *N(a)* is the number of times action *a* was selected and *c* is a configureable hyperparameter. This method is also referred to sometimes as “exploration through optimism”, as it gives less-explored actions a higher value, encouraging the model to select them.

Value Function: Usually denoted as *V(s)* (sometimes with a π subscript), the Value function is a measure of the overall expected *reward* assuming the *Agent* is in *state s* and then continues playing until the end of the *episode* following some *policy* π. It is defined mathematically as:

$$V(s) = \mathbf{E}\left[\sum_{n=0}^{N} \gamma^n r_n\right]$$

While it does seem similar to the definition of *Q Value*, there is an implicit — yet important — difference: for *n=0*, the reward r^0 of *V(s)* is the expected reward from just being in state *s*, *before* any action was played, while in Q Value, r^0 is the expected reward *after* a certain action was played. This difference also yields the *Advantage function*.