Member-only story

THOUGHTS AND THEORY

The Four Policy Classes of Reinforcement Learning

A comprehensive classification of solution strategies for reinforcement learning

Wouter van Heeswijk, PhD · Following
Published in Towards Data Science · 7 min read · May 11, 2021

Image by Hans Braxmeier via Pixabay

Policies in Reinforcement Learning (RL) are shrouded in a certain mystique. Simply stated, a policy $\pi: s \rightarrow a$ is any function that returns a feasible action for a problem. No less, no more. For instance, you could simply take the first action that comes to mind, select an action at random, or run a heuristic. However, what makes RL special is that we actively anticipate the downstream impact of decisions and learn from our observations; we therefore expect some intelligence in our policies. In his framework on sequential decision-making[1], Warren Powell argues there are four policy classes for RL. Techniques from all four classes are widely used across domains, but not yet universally recognized. This article will provide a brief — and no doubt incomplete — introduction to this classification of solution strategies.

Solving MDP models

Before moving on to reinforcement learning, let's first refresh our memory on analytical solutions a bit. Typically, we aim to formulate an RL problem as a Markov Decision Process (MDP) model. If we stay close to the MDP, the goal of reinforcement learning would be solving the corresponding system of Bellman equations and thereby find the optimal policy π^*:

$$V^{\pi^*}(s) = \max_{a \in \mathcal{A}} \left\{ R(s,a) + \sum_{\forall s' \in \mathcal{S}'} \mathbb{P}(s' \mid s,a) V^{\pi^*}(s') \right\} \forall s \in \mathcal{S}$$

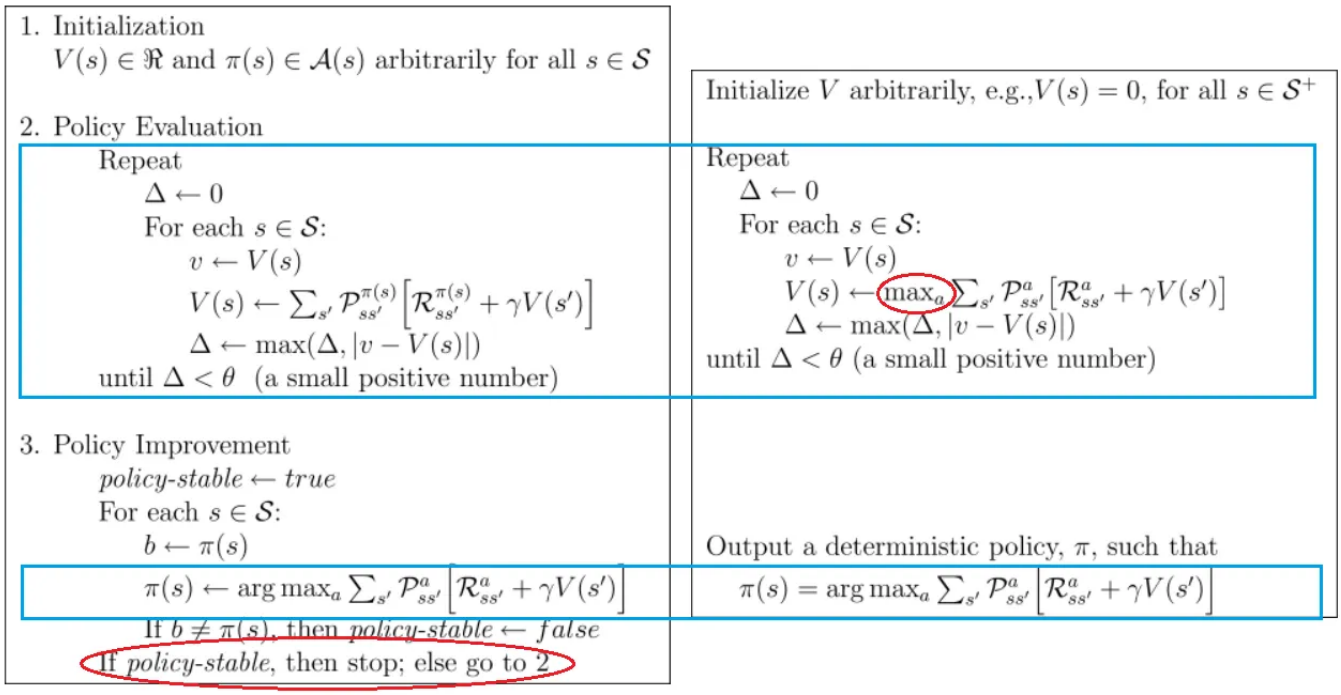
Bellman equation for MDP. Finding the optimal policy π* yields the optimal value functions V(s) and vice versa.

We actually don't actually need the Bellman equation though. Ultimately, we simply try to maximize a cumulative reward; the optimal policy does exactly that. It also eliminates the need to consider value functions *V^π*(s')*, which is something we only do in one of the four policy classes. Thus, we can state our objective as follows:

$$\max_{\pi \in \Pi} \mathbb{E} \left\{ \sum_{t=0}^{T} R(s_t, x^{\pi}(s_t)) \mid s_0 \right\}$$

Reward function over a time horizon. The optimal policy π* maximizes the cumulative reward.

To solve an MDP model to optimality, there are basically two approaches: (i) policy iteration and (ii) value iteration. *Policy iteration* fixes a policy, computes the corresponding policy value, and subsequently updates the policy using the new value. The algorithm iterates between these steps until the policy remains stable. *Value iteration* in fact relies on quite similar steps (see figure below), but aims to directly maximize the value functions and only updates the policy afterwards. Note that finding the optimal value functions equates finding the optimal policy; either suffices to solve the system of Bellman equations.



Comparison between policy iteration (left) and value iteration (right). Note the iterative character of policy iteration and the maximum operator in value iteration. Adapted from Sutton & Barto[2]

Most — if not all — RL algorithms are rooted either in policy iteration or value iteration (or a combination of both). As the deployed simulation methods typically do not guarantee finding optimal policies, in RL we speak of policy *approximation* and value *approximation*, respectively. Powell states that both strategies may be subdivided into two classes, yielding a total of

four classes to be discussed soon. Just some basic notation, and we are good to go:

s: state (information needed to make decision)

a: action (feasible operation on state)

π: policy (maps state to action)

ϕ: basis function (derives features from state)

θ: feature weights (parametrization of policy)

t: time epoch (discrete point in time)

R: reward function (direct reward for taking action in state)

V: value function (downstream reward for certain state)

Policy approximation

In policy approximation solutions, we directly modify the policy itself. Such solution strategies tend to work best when the policy has a clear structure. We may distinguish two classes: PFA and CFA.

Policy function approximation (PFA)

A policy function approximation (PFA) is essentially a parameterized function of the policy. Plugging in the state directly returns a feasible action.

A linear PFA might look like:

$$\pi^{PFA}(s_t \mid \theta) = \theta_0 + \theta_1\phi_1(s_t) + \theta_2\phi_2(s_t) + \dots$$

Example of policy function approximation (PFA)

The key challenge here is to find appropriate basis functions $\phi(s)$ that capture the essence of the decision-making process. To achieve this, a good insight into the structure of the problem is required. The design effort might be alleviated by selecting more generic function representations such as neural network (actor networks), using the state as input and outputting an action. The drawback of such PFA representations is that parameter tuning becomes harder and interpretability suffers. Regardless, a solid understanding of the problem structure remains necessary.

Cost function approximation (CFA)

Like the PFA, a Cost Function Approximation (CFA) also directly searches the policy space. However, the CFA does not directly return an action, but rather solves an optimization problem in a constrained action space. A sample formulation would be:

$$\pi^{CFA}(s_t \mid \theta) = \underset{a \in \mathcal{A}^\pi(s_t,\theta)}{\arg\max}\ \tilde{R}_t^\pi(s_t, a \mid \theta)$$

Example of cost function approximation (CFA)

Whereas a PFA directly returns an action, the CFA requires us to first solve a parameterized maximization function. Note the true reward- and value functions have been replaced by an approximate reward function.

Furthermore, the action space A^π is constrained by the policy π and its parameterization θ, typically yielding a smaller action space than the original one. Note that the simplest form of the CFA is simply a greedy heuristic, yet the modified reward function may embed exploration elements. Computational effort per iteration is likely higher than for the PFA (due to the maximization step), yet less design effort is required.

Value approximation

Value approximations explicitly consider the downstream impact of current decisions, taking into account the full decision horizon. Remind that optimal value functions equate the optimal policy; they both yield identical solutions to the Bellman equations. Value approximation may be suitable when the policy structure is not eminent or we cannot properly oversee the downstream effects of current decisions.

Value function approximation (VFA)

A value function approximation (VFA) represents downstream values as a function. One of the problems with the Bellman equation is that after taking an action, random events may lead us to many new states $s' \in S'$. Thus, for each action we should consider the value of all reachable states s' and the probability of ending up there. VFAs circumvent this problem by replacing the stochastic expectation term with a deterministic approximation function $V_t(s_t,a_t)$. In canonical form the VFA looks like:

Example of value function approximation (VFA)

The simplest VFA is a lookup table, in which we store the average observed values for each state-action pair. Sufficient iterations allow us to learn accurate values for each pair, yet this is rarely computationally tractable. Therefore we typically we resort to features that capture the essence of the state, which we could manually design or extract by, e.g., using auto-encoders. Thus, we capture the state-action values in compact functions (e.g., a critic network) and tune the feature weights by observation.

Direct lookahead approximation (DLA)

Designing a VFA typically requires a good understanding of the problem structure, although neural networks alleviate this problem to some extent (at the cost of additional tuning). Instead of deriving an explicit function, a direct lookahead approximation (DLA) simply samples downstream values. The DLA may be represented by:

Example of direct lookahead approximation (DLA)

Admittedly, this equation looks rather cumbersome, but in fact may be the simplest of the lot. The expectation terms imply we sample the future and apply some (suboptimal) policy to estimate the downstream values. Whereas we maximize over all feasible actions at the current time *t*, for the future time epochs *t'* we typically use a computationally lighter policy (e.g., a heuristic) and/or a simplified problem representation (e.g., assuming perfect foresight). The DLA strategy brings its own challenges (scenario sampling, aggregation techniques, etc.), but unlike the other three policies it does not require estimating a function (so no, replacing 'Function' by 'Lookahead' is

not just semantics). As such, it often serves as a last resort for complex problems at which the other three strategies fail.

Hybrid classes

We would be remiss not to mention the options to combine strategies from multiple classes. For instance, the classical actor-critic framework contains elements of both PFA (the actor) and VFA (the critic). However, there are many more combinations, such as embedding a VFA as a downstream policy into a direct lookahead algorithm. Combined strategies may negate each other's weaknesses, often producing superior results when compared to single-class solutions.

Takeaways

According to Powell, virtually any solution strategy may be categorized in one (or multiple) of the four policy classes. Aside from that, there are some interesting takeaways that may be derived from the classification:

- **No one-size-fits-all.** Although there are some rules-of-thumb, multiple strategies may yield good solutions. A tangible example of this statement can be found in Powell & Meisel[3], demonstrating implementations of all four solution strategies on the same problem.
- **Academics likes elegance.** PFA and VFA appear to be most popular in academia. After all, there is a certain mathematical beauty to capturing a complicated decision-making policy in a single function.
- **Industry likes results.** When problems get too large or complex, CFA and DLA may yield surprisingly good results. Although relying somewhat

more on brute force and enumeration, the design effort is substantially less.

- **Everything has its price.** There are always tradeoffs between convenience, design effort, computational complexity, interpretability, etc. The nature of the problem dictates how heavily these tradeoffs weigh.
- **Classification is key.** There are many RL communities, many techniques, many notational styles, many algorithms. To streamline the domain and foster advancement, a clear overarching framework is needed.

References

[1]Powell, Warren B. "A unified framework for stochastic optimization." *European Journal of Operational Research* 275.3 (2019): 795–821.

[2]Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[3]Powell, Warren B., and Stephan Meisel. "Tutorial on stochastic optimization in energy — Part II: An energy storage illustration." *IEEE Transactions on Power Systems* 31.2 (2015): 1468–1475.

Reinforcement Learning | Machine Learning | Data Science | Editors Pick | Thoughts And Theory

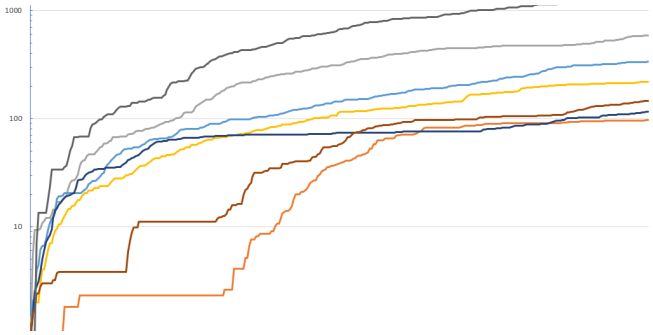
Written by Wouter van Heeswijk, PhD

Following

1.5K Followers · Writer for Towards Data Science

Assistant professor in Financial Engineering and Operations Research. Writing about reinforcement learning, optimization problems, and data science.

More from Wouter van Heeswijk, PhD and Towards Data Science



Wouter van Heeswijk, PhD in Towards Data Science

Five Ways To Handle Large Action Spaces in Reinforcement Learning

Action spaces, particularly in combinatorial optimization problems, may grow unwieldy i...

✦ · 14 min read · Aug 18

36

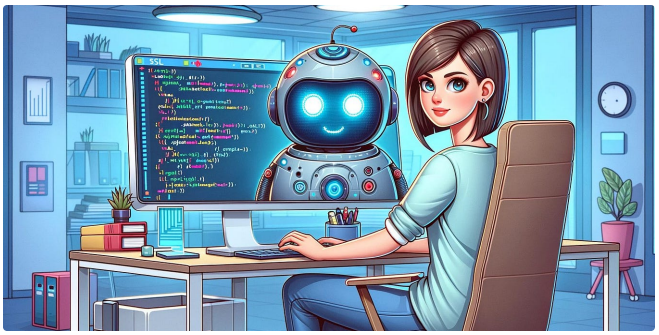
Pau Blasco i Roca in Towards Data Science

My Life Stats: I Tracked My Habits for a Year, and This Is What I...

I measured the time I spent on my daily activities (studying, doing sports, socializing...

12 min read · Nov 20

4.5K 80



Mariya Mansurova in Towards Data Science

LMQL—SQL for Language Models

Yet another tool that could help you with LLM applications

17 min read · Nov 27

676 10

Wouter van Heeswijk, PhD in Towards Data Science

Python's Map(), Filter(), and Reduce() Functions Explained

Demonstrating the building blocks of functional programming in Python with code...

✦ · 8 min read · Aug 10, 2022

44

See all from Wouter van Heeswijk, PhD

See all from Towards Data Science

Recommended from Medium

Navneet Singh

Understanding Q-Learning: A Powerful Reinforcement Learning...

In the field of machine learning, specifically in the realm of reinforcement learning, Q-...

4 min read · Jul 10

8

Ayeshanasim

Portfolio Management Using Multi-Agent Reinforcement Learning

9 min read · Oct 14

58

Lists

Predictive Modeling w/ Python

20 stories · 687 saves

Practical Guides to Machine Learning

10 stories · 785 saves

Natural Language Processing

980 stories · 475 saves

data science and AI

38 stories · 4 saves

Madhur Prashant

RLHF + Reward Model + PPO on LLMs

Purpose

12 min read · Sep 10

81

Kim Rodgers

Temporal-difference RL: Sarsa vs Q-learning

Introduction

5 min read · Aug 7

13

Faruk Hussain Abdullahi

REINFORCEMENT LEARNING

What is reinforcement learning?

5 min read · Jul 16

William Seymour

Training an AI to play a game using Deep Reinforcement Learning

This article builds on tutorials on Reinforcement Learning (DQN, or Deep Q...

12 min read · Sep 27



See more recommendations