

[Open in app](#)[Follow](#)

568K Followers



This is your **last** free member-only story this month. [Upgrade for unlimited access.](#)

A Structural Overview of Reinforcement Learning Algorithms

Actor Critic, Policy Gradient, DQN, VFA, SARSA, Q-learning, Model-based and Model-free Monte Carlo, Dynamic Programming



Siwei Causevic · Jun 18, 2020 · 12 min read ★



Photo by Siwei Xu

Reinforcement learning has gained tremendous popularity in the last decade with a series of successful real-world applications in robotics, games and many other fields.

In this article, I will provide a high-level structural overview of classic reinforcement learning algorithms. The discussion will be based on their similarities and differences in the intricacies of algorithms.

RL Basics

Let's start with a quick refresher on some basic concepts. If you are already familiar with all the terms of RL, feel free to skip this section.

Reinforcement learning models are a type of state-based models that utilize the markov decision process(MDP). The basic elements of RL include:

Episode(rollout): playing out the whole sequence of state and action until reaching the terminate state;

Current state s (or s_t): where the agent is current at;

Next state s' (or s_{t+1}): next state from the current state;

Action a : the action to take at state s ;

Transition probability $P(s'|s, a)$: the probability of reaching s' if taking action a at state s_t ;

Policy $\pi(s, a)$: a mapping from each state to an action that determines how the agent acts at each state. It can be either deterministic or stochastic

Reward r (or $R(s, a)$): a reward function that generates rewards for taking action a at state s ;

Return G_t : total future rewards at state s_t ;

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$$

Value $V(s)$: expected return for starting from state s ;

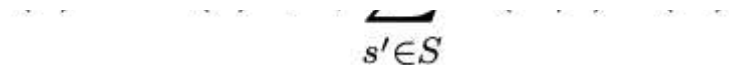
$$V(s) = \mathbb{E}[G_t | s_t = s] = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots | s_t = s]$$

Q value $Q(s, a)$: expected return for starting from state s and taking action a ;

$$Q(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a] = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots | s_t = s, a_t = a]$$

Bellman equation

$$V(s) = R(s) + \gamma \sum_s P(s'|s) V(s')$$


$$s' \in S$$

According to the Bellman equation, the current value is equal to current reward plus the discounted(γ) value at the next step, following the policy π . It can also be expressed using the Q value as:

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s) V(s')$$

This is the theoretical core in most reinforcement learning algorithms.

Prediction vs. Control Tasks

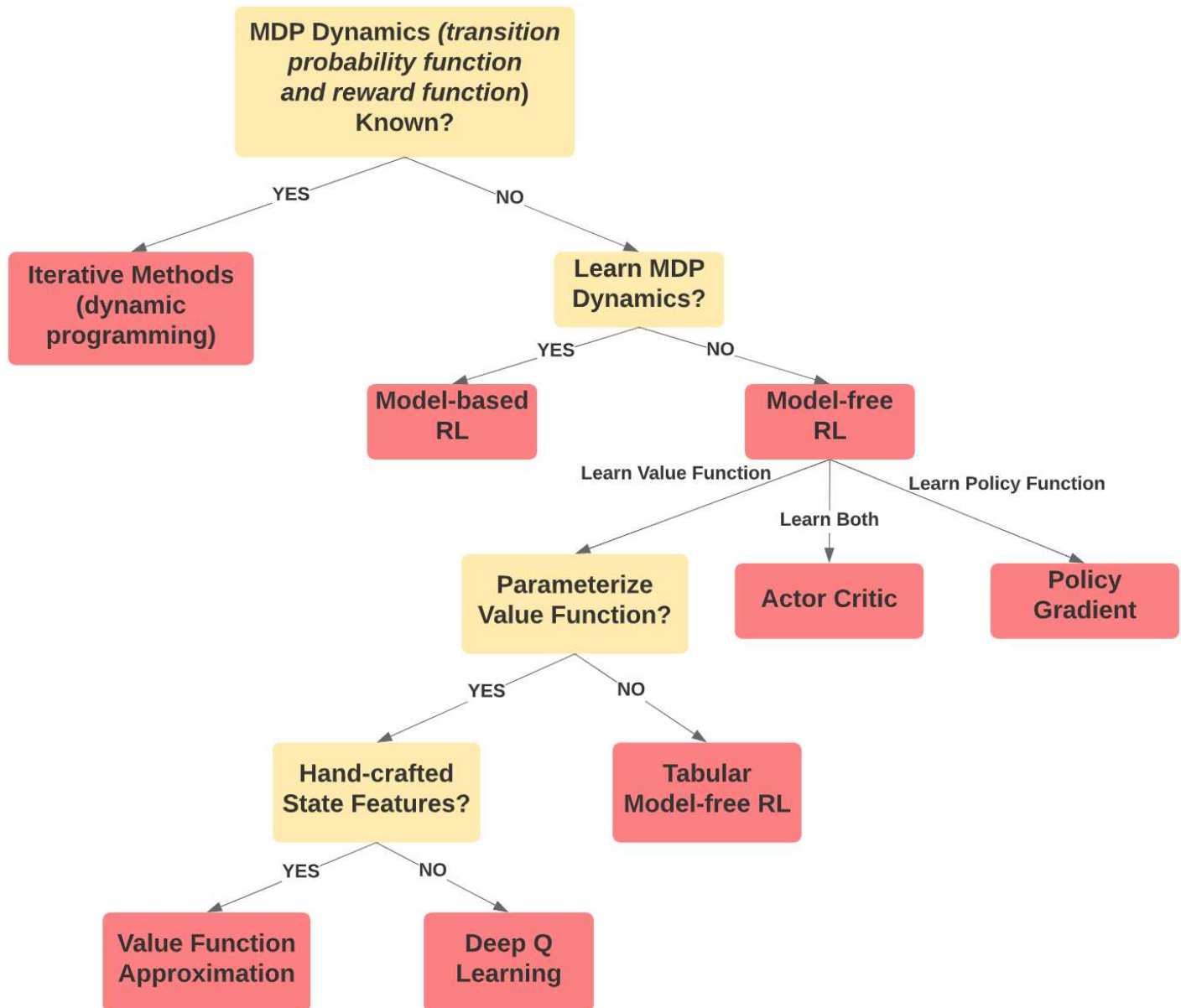
There are two fundamental tasks of reinforcement learning: prediction and control.

In prediction tasks, we are given a policy and our goal is to evaluate it by estimating the value or Q value of taking actions following this policy.

In control tasks, we don't know the policy, and the goal is to find the optimal policy that allows us to collect most rewards. In this article, we will only focus on control problems.

RL Algorithm Structure

Below is a graph I made to visualize the high-level structure of different types of algorithms. In the next few sections, we will delve into the intricacies of each type.



MDP World

In the MDP world, we have a mental model of how the world works, meaning that we know the MDP dynamics (transition $P(s'|s,a)$ and reward function $R(s,a)$), so we can directly build a model using the Bellman equation.

Again, in control tasks our goal is to find a policy that gives us maximum rewards. To achieve it, we use dynamic programming.

Dynamic Programming (Iterative Methods)

1. Policy Iteration

Policy iteration essentially performs two steps repeatedly until convergence: policy evaluation and policy improvement.

In the policy evaluation step, we evaluate the policy π at state s by calculating the Q value using the Bellman equation:

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s')$$

In the policy improvement step, we update the policy by greedily searching for the action that maximizes the Q value at each step.

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a) \quad \forall s \in S$$

Let's see how policy iteration works.

Set $i = 0$

Initialize $\pi_0(s)$ randomly for all states s

While $i == 0$ or $\|\pi_i - \pi_{i-1}\|_1 > 0$ (L1-norm, measures if the policy changed for any state):

- $V^{\pi_i} \leftarrow$ MDP V function policy **evaluation** of π_i
- $\pi_{i+1} \leftarrow$ Policy **improvement**
- $i = i + 1$

2. Value Iteration

Value iteration combines the two steps in policy iteration so we only need to update the Q value. We can interpret value iteration as always following a greedy policy because at each step it always tries to find and take the action that maximizes the value. Once the values converge, the optimal policy can be extracted from the value function.

Set $k = 1$

Initialize $V_0(s) = 0$ for all states s

Loop until [finite horizon, convergence]:

- For each state s

$$V_{k+1}(s) = \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s')$$

- Equivalently, in Bellman backup notation

$$V_{k+1} = BV_k$$

To extract optimal policy if can act for $k + 1$ more steps,

$$\pi(s) = \arg \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_{k+1}(s')$$

In most real-world scenarios, we don't know the MDP dynamics so the applications of iterative methods are limited. In the next section, we will switch gears and discuss reinforcement learning methods that can deal with the unknown world.

Reinforcement Learning World

In most cases, the MDP dynamics are either unknown, or computationally infeasible to use directly, so instead of building a mental model we learn from sampling. In all the following reinforcement learning algorithms, we need to take actions in the environment to collect rewards and estimate our objectives.

Exploration-exploitation Dilemma

In MDP models, we can explore all potential states before we come up with a good solution using the transition probability function. However, in reinforcement learning where the transition is unknown, if we keep greedily searching for the best action, we might end up stuck in a few states without being able to explore the entire environment. This is the exploration-exploitation dilemma.

To get out of the suboptimal states, we usually use a strategy called **epsilon greedy**: when we select the best action, there is a probability of ϵ that we might get a random action.

$$\pi_{\text{act}}(s) = \begin{cases} \arg \max_{a \in \text{Actions}} \hat{Q}^*(s, a) & \text{probability } 1 - \epsilon, \\ \text{random from Actions}(s) & \text{probability } \epsilon. \end{cases}$$

Model-based Reinforcement Learning

One way to estimate the MDP dynamics is sampling. Following a random policy, we sample many (s, a, r, s') pairs and use Monte Carlo (counting the occurrences) to estimate the transition and reward functions explicitly from the data. If the data size is large enough, our estimates should be very close to the true values.

$$\hat{P}(s'|s, a) = \frac{\# \text{ times } (s, a, s') \text{ occurs}}{\# \text{ times } (s, a) \text{ occurs}}$$

$$\hat{r}(s, a) = r \text{ in } (s, a, r, s')$$

Once we have the estimates, we can use iterative methods to search for the optimal policy.

Model-free Reinforcement Learning (Tabular)

Let's take a step back. If our goal is to just find good policies, all we need is to get a good estimate of Q . From that perspective, estimating the model (transitions and rewards) was just a means towards an end. Why not just cut to the chase and estimate Q directly?

This is called model-free learning.

1. Model-free Monte Carlo

Recall that $Q(s, a)$ is the expected utility when the agent takes action a from state s .

$$\hat{Q}(s_t, a_t) = E[G_t] = \frac{1}{K} \sum_{k=1}^K G_t^k$$

The idea of model-free Monte Carlo is to sample many rollouts, and use the data to estimate Q . Let's take a look at the algorithm.


```

Initialize  $Q(s, a) = 0, N(s, a) = 0 \forall (s, a)$ , Set  $\epsilon = 1, k = 1$ 
 $\pi_k = \epsilon\text{-greedy}(Q)$  // Create initial  $\epsilon$ -greedy policy
loop
  Sample  $k$ -th episode  $(s_{k,1}, a_{k,1}, r_{k,1}, s_{k,2}, \dots, s_{k,T})$  given  $\pi_k$ 
   $G_{k,t} = r_{k,t} + \gamma r_{k,t+1} + \gamma^2 r_{k,t+2} + \dots + \gamma^{T-t} r_{k,T}$ 
  for  $t = 1, \dots, T$  do
    if First visit to  $(s, a)$  in episode  $k$  then
       $N(s, a) = N(s, a) + 1$ 
       $Q(s_t, a_t) = Q(s_t, a_t) + \frac{1}{N(s,a)} (G_{k,t} - Q(s_t, a_t))$ 
    end if
  end for
   $k = k + 1, \epsilon = 1/k$ 
   $\pi_k = \epsilon\text{-greedy}(Q)$  // Policy improvement
end loop

```

We first randomly initialize everything and use epsilon greedy to sample an action, and then we start to play rollouts. At the end of each rollout, we calculate the return G_t for each state S_t in the rollouts. To get $Q(s_t, a_t)$, the average of returns G_t , we can store all the returns and update Q when we finish sampling. However, a more efficient way is to update Q incrementally at the end of each rollout using a moving average as is shown below.

$$Q^k(s_t, a_t) \leftarrow Q^{k-1}(s_t, a_t) + \frac{1}{N(s,a)} (G_t^k - Q^{k-1}(s_t, a_t))$$

where $N(s,a) = 1 + \text{the number of episodes played so far}$

2. SARSA

SARSA is a Temporal Difference (TD) method, which combines both Monte Carlo and dynamic programming methods. The update equation has the similar form of Monte Carlo's online update equation, except that SARSA uses $r_t + \gamma Q(s_{t+1}, a_{t+1})$ to replace the actual return G_t from the data. $N(s, a)$ is also replaced by a parameter α .

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Recall that in Monte Carlo, we need to wait for the episode to finish before we can update the Q value. The advantage of TD methods is that they can update the estimate of Q immediately when we move one step and get a state-action pair $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$.

```

Set initial  $\epsilon$ -greedy policy  $\pi$ ,  $t = 0$ , initial state  $s_t = s_0$ 
Take  $a_t \sim \pi(s_t)$  // Sample action from policy
Observe  $(r_t, s_{t+1})$ 
loop
  Take action  $a_{t+1} \sim \pi(s_{t+1})$ 
  Observe  $(r_{t+1}, s_{t+2})$ 
   $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$ 
   $\pi(s_t) = \arg \max_a Q(s_t, a)$  w.prob  $1 - \epsilon$ , else random
   $t = t + 1$ 
end loop

```

3. Q-learning

Q-learning is another type of TD method. The difference between SARSA and Q-learning is that SARSA is an on-policy model while Q-learning is off-policy. In SARSA, our return at state s_t is $r_t + \gamma Q(s_{t+1}, a_{t+1})$, where $Q(s_{t+1}, a_{t+1})$ is calculated from the state-action pair $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ that was obtained by following policy π . However, in Q-learning, $Q(s_{t+1}, a_{t+1})$ is obtained by taking the optimal action, which might not necessarily be the same as our policy.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

In general, on-policy methods are more stable but off-policy methods are more likely to find global optima. We can see from below that except the update equation, the other parts of the algorithm are the same as SARSA.

```

Initialize  $Q(s, a), \forall s \in S, a \in A$   $t = 0$ , initial state  $s_t = s_0$ 
Set  $\pi_b$  to be  $\epsilon$ -greedy w.r.t.  $Q$ 
loop

```

```

Take  $a_t \sim \pi_b(s_t)$  // Sample action from policy
Observe  $(r_t, s_{t+1})$ 
 $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$ 
 $\pi(s_t) = \arg \max_a Q(s_t, a)$  w.prob  $1 - \epsilon$ , else random
 $t = t + 1$ 
end loop

```

Value Function Approximation(VFA)

So far, we have been assuming we can represent the value function V or state-action value function Q as a tabular representation (vector or matrix). However, many real world problems have enormous state and/or action spaces for which tabular representation is insufficient. Naturally, we might wonder if we can parameterize the value function so we don't have to store a table.

VFA is the very method that represents the Q value function with a parameterized function \hat{Q} .

$$\hat{Q}(s, a; \mathbf{w}) \approx Q(s, a)$$

The state and action are represented as a feature vector $\mathbf{x}(s, a)$, and the estimated \hat{Q} is the score of the linear predictor.

$$\hat{Q}(s, a; \mathbf{w}) = \mathbf{x}(s, a)^T \mathbf{w} = \sum_{j=1}^n x_j(s, a) w_j$$

The objective is to minimize the loss between the estimated Q (prediction) and real Q (target), and we can use stochastic gradient descent to solve this optimization problem.

$$J(\mathbf{w}) = E[(Q(s, a) - \hat{Q}(s, a; \mathbf{w}))^2]$$

How do we get our target — the real Q value in the objective function?

Recall the Q value is the expected returns (G_t), so one way to get Q value is to use Monte Carlo: playing many episodes and counting the occurrences.

We have the following objective functions and gradients for parameterized Monte Carlo:

$$J(\mathbf{w}) = E[(G_t - \hat{Q}(s_t, a_t; \mathbf{w}))^2]$$

$$\Delta \mathbf{w} = \alpha (G_t - \hat{Q}(s_t, a_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

Another way is to utilize the recursive expression of Q value: $Q(s_t, a_t) = r_t + \gamma Q(s_{t+1}, a_{t+1})$. As we discussed earlier, Temporal Difference (TD) methods combine both Monte Carlo and dynamic programming and allow real-time update. Hence we can also obtain the target Q value using TD methods: SARSA and Q-learning.

SARSA:

$$J(\mathbf{w}) = E[(r_t + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w}))^2]$$

$$\Delta \mathbf{w} = \alpha (r_t + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

Q-learning

$$J(\mathbf{w}) = E[(r_t + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w}))^2]$$

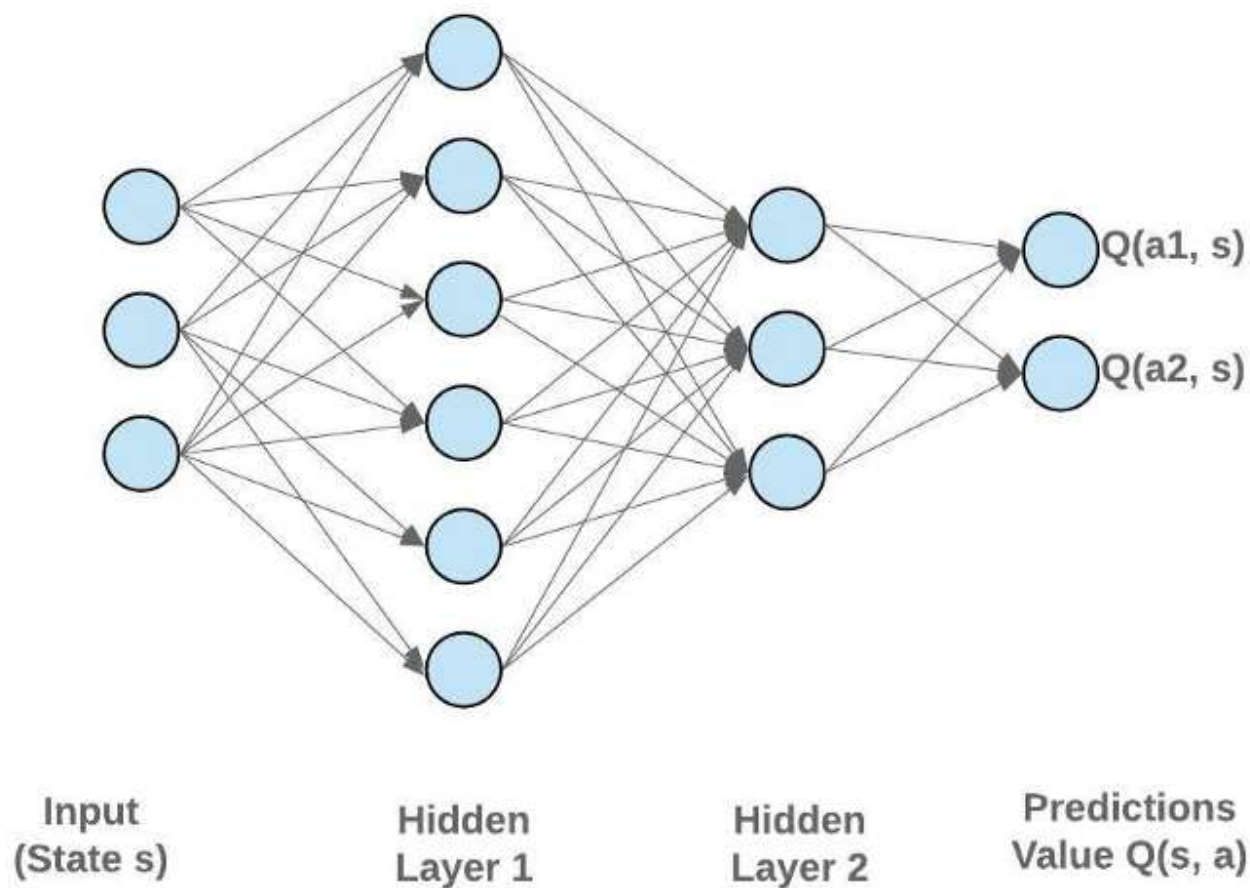
$$\Delta \mathbf{w} = \alpha (r_t + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

Notice that in the above TD methods, we are actually using the model prediction to approximate the real target value $Q(s_{t+1}, a_{t+1})$, and this type of optimization is called semi-gradient.

Deep Q Networks (DQN)

Linear VFA often works well if we have the right set of features, which usually require careful hand designing. An alternative is to use deep neural networks that directly use the states as input without requiring an explicit specification of features.

For example, in the graph below we have a neural network with the state s as the input layer, 2 hidden layers, and the predicted Q values as the output. The parameters can be learned through backpropagation.



There are two important concepts in DQN: target net and experience replay.

As you may have realized, a problem of using semi-gradient is that the model updates could be very unstable since the real target will change each time the model updates itself. The solution is to create a **target network** that copies the training model at a certain frequency so the target model updates less frequently. In equation below, w - are the weights of the target network.

$$\Delta \mathbf{w} = \alpha (r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^-) - \hat{Q}(s, a; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a; \mathbf{w})$$

We also create an experience replay buffer that stores the (s, a, r, s', a') pairs from prior episodes. When we update the weights, instead of using the most recent pair generated from the episode, we randomly select an experience from the experience replay buffer to run stochastic gradient descent. This will help avoid overfitting.

I have previously implemented DQN with Tensorflow to play the CartPole game. If you are interested in learning more about the implementation, check out my article [here](#).

Policy Gradient

Different from the previous algorithms that model the value function, policy gradient methods directly learn the policy by parameterizing it as:

$$\pi_{\theta}(s, a) = \mathbb{P}[a|s; \theta]$$

However, when it comes to optimization we still have to go back to the value function as the policy function can't be used as an objective function by itself. Our objective can be expressed as the value function $V(\theta)$, which is the expected total rewards we get from trajectories τ following stochastic policy π . Here θ are parameters for the policy. Be careful not to misinterpret $V(\theta)$ as the parameterized value function.

$$V(\theta) = \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=0}^T R(s_t, a_t); \pi_{\theta} \right] = \sum_{\tau} P(\tau; \theta) R(\tau)$$

Where τ is a state-action trajectory:

$$\tau = (s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$$

$R(\tau)$ is sum of rewards for a trajectory τ :

$$R(\tau) = \sum_{t=0}^T R(s_t, a_t)$$

$$\pi(a_t | s_t) = \arg \max_{a_t} \pi(a_t | s_t)$$

Now, the goal is to find the policy parameters (θ) that maximize the value $V(\theta)$. To do so, we search for the maxima in $V(\theta)$ by ascending the gradient of the policy, w.r.t parameters θ .

$$\arg \max_{\theta} V(\theta) = \arg \max_{\theta} \sum_{\tau} P(\tau; \theta) R(\tau)$$

In the gradients shown as follows, the policy π_{θ} is usually modeled using softmax, Gaussian or neural networks to ensure it's differentiable.

$$\begin{aligned} \nabla_{\theta} V(\theta) &= \nabla_{\theta} E[R] \\ &= E \left[\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t, s_t) \sum_{t'=t}^{T-1} r_{t'} \right] \\ &= E \left[\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t, s_t) G_t \right] \end{aligned}$$

Let's see what policy gradient is like.

```

Initialize policy parameters  $\theta$  arbitrarily
for each episode  $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_{\theta}$  do
  for  $t = 1$  to  $T - 1$  do
     $\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(s_t, a_t) G_t$ 
  endfor
endfor
return  $\theta$ 

```

Actor-critic

Actor-critic methods differ from the policy gradient methods in that actor-critic methods estimate both policy and the value function, and update both. In policy gradient

methods, we update θ using G_t , an estimation of the value function at s_t from a single rollout. Although this estimation is unbiased, it has high variance.

To address this issue, Actor-critic methods introduce bias using bootstrapping and function approximation. Instead of using G_t from the roll out, we estimate the value with a parameterized function, and this is where the critic comes in.

Here is “vanilla” actor-critic policy gradient:

```

Initialize policy parameter  $\theta$ , baseline  $b$ 
for iteration=1, 2, ... do
  Collect a set of trajectories by executing the current policy
  At each timestep  $t$  in each trajectory  $\tau^i$ , compute
    Target  $\hat{R}_t^i$ 
    Advantage estimate  $\hat{A}_t^i = G_t^i - b(s_t)$ .
  Re-fit the baseline, by minimizing  $\sum_i \sum_t ||b(s_t) - \hat{R}_t^i||^2$ ,
  Update the policy, using a policy gradient estimate  $\hat{g}$ ,
    Which is a sum of terms  $\nabla_{\theta} \log \pi(a_t | s_t, \theta) \hat{A}_t^i$ .
  (Plug  $\hat{g}$  into SGD or ADAM)
endfor

```

In the above procedure, two new terms are introduced: advantage(A_t) and baseline($b(s)$).

$b(t)$ is the expected total future rewards at state S_t , equivalent to $V(S_t)$, and this is the value function the critic estimates.

$$b(s_t) \approx \mathbb{E}[r_t + r_{t+1} + \dots + r_{T-1}]$$

After every episode, we update both the value function $b(s)$ and the policy function. Different from that in policy gradient, the policy function here is updated with A_t instead of G_t , which helps reduce variance of the gradient

On top of the vanilla actor-critic, there are two popular actor-critic methods A3C and A2C that update the policy and value functions with multiple workers. Their main

difference is that A3C performs asynchronous updates while A2C is synchronous.

Conclusions & Thoughts

In this article, we had an overview of many classic and popular reinforcement learning algorithms, and discussed their similarities and differences in the intricacies.

It's worthwhile to mention that there are a lot of variants in each model family that we've not covered. For example, in the DQN family, there are dueling DQN and double DQN. In the policy gradient the actor-critic families, there are DDPG, ACER, SAC etc.

Additionally, there is another type of RL methods: evolution strategies(ES). Inspired by the theory of natural selection, ES solves problems when there isn't a precise analytic form of an objective function. As they are beyond the scope of MDP, I didn't include them in this article but I might have a discussion in my future articles. Stay tuned! :)

References

Stanford CS234 course notes: <https://web.stanford.edu/class/cs234/slides/>

Lilian's blog: <https://lilianweng.github.io/lil-log/>

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

Get this newsletter

Emails will be sent to pchakraborty@hyderabad.bits-pilani.ac.in.
[Not you?](#)

Reinforcement Learning

Markov Decision Process

Towards Data Science

Artificial Intelligence

Deep Learning

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

