RL Basics - MDP, Value functions, Policy

Dinh Dung Tran

Dec 9, 2025

Disclaimer

Please note that this is a work that only reflect my understanding of the materials that I'm learning from. It contains my own understanding and interpretations, which can obviously be wrong since I'm not the expert. However, for those who start learning about RL, I hope this can be a helpful resource. Credit to AI Prism for the amazing recordings of the Deep RL bootcamp. I highly recommend those who want to start learning about RL to watch the videos. Though, it may help to have some understanding of probability and machine learning.

Contents

1 Markov Decision Process

1.1 Definition

In short, a Markov Decision Process (MDP) is a model of an environment in which an agent can take actions to influence the state of the environment, and receive rewards or punishments based on the state of the environment.

There are 4 core components of an MDP:

- State Space $\mathcal{S}$
- Action Space $\mathcal{A}$
- Transition Probability $\Pr(s'|s,a)$
- Reward Function $\mathcal{R}(s,a)$

Note that $\mathcal{S}$ and $\mathcal{A}$ can either be discrete or continuous spaces. For example, the grid world environment has a discrete state space (the coordinates) and a discrete action space (up, down, left, right). On the other hand, the cart pole environment has a continuous state space (position, angle, velocity, angular velocity) and a continuous action space (acceleration).

For the transition probability $\Pr(s'|s,a)$, it is a probability distribution over the next state given the current state and action. It is a Markov property, meaning that the transition probability only depends on the current state and action, and not on the history of the environment.

Lastly, the rewards funciton is a function that maps a state, an action, and the next state the action takes the agent to a value.

Besides the core components, there are 3 more components of an MDP that we need to fully describe our problem:

- Discount Factor $0 \leq \gamma \leq 1$ which weights how important future rewards are.
- Initial State $s_0 \in \mathcal{S}$
- Horizon $H \in \mathbb{N}$ which is the maximum amount of steps the agent can take.

Goal: Given an MDP $(\mathcal{S}, \mathcal{A}, \Pr(s'|s,a), \mathcal{R}, \gamma, s_0, H)$, we wish to perform a sequence of actions (trajectory) such that the expected sum of future rewards is maximized.

With this in mind, let's also look into the definition of a policy. In short, a policy $\pi : \mathcal{S} \to \mathcal{A}$ is a function that maps a state to an action. Note that, we can have stochastic policies, meaning that $\pi(a|s)$ is a probability distribution over actions given a state. For the sake of simplicity, let's consider deterministic policies for now.

2 Value Function

2.1 Definition

The value function is a function that maps a state to a value, representing the expected sum of future rewards starting from that state. That is, its formulation takes the form:

$$V_H(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{H} \gamma^t \mathcal{R}(s_{t+1}, a_t, s_t) \right] \tag{1}$$

Here, π means that we are taking the expected value with a given policy π. Now, note that we can expand the value functions as follows:

$$V_H(s) = \sum_{t=0}^{H} \sum_{s_{t+1}} \Pr(s_{t+1}|s_t, a_t) \gamma^t \mathcal{R}(s_{t+1}, a_t, s_t) \tag{2}$$

where s_{t+1} is a possible next state by performing action a_t in state s_t. Now

One can deduct from the definition of a stochastic policy that $\Pr(s'|s_t, a_t) = \sum_{a'} \pi(a'|s_t) \Pr(s'|s_t, a')$. Substituting this into (2), we get:

$$V_H(s) = \sum_{t=0}^{H} \sum_{s_{t+1}} \sum_{a_t} \pi(a_t|s_t) \Pr(s_{t+1}|s_t, a_t) \gamma^t \mathcal{R}(s_{t+1}, a_t, s_t) \tag{3}$$

However, to make the expression more readable, we will only use deterministic policy for now.

2.2 Bellman Equation

One thing to note is that if we are from state s_t and perform action a that leads us to state s_{t+1}, then

$$V_H(s_t) \text{ can be estimated by } \mathcal{R}(s_{t+1}, a_t, s_t) + \gamma V_{H-1}(s_{t+1})$$

One can understand the Bellman equation as a recursive relationship between the value function at time step t and the value function at time step $t+1$. This is because the value function at time step $t+1$ is the expected sum of future rewards starting from state s_{t+1}, and the value function at time step t is the expected sum of future rewards starting from state s_t. The H turn into $H-1$ because we have performed an action.

Since we have a transition probability and we are maximizing, the correct (tabular) Bellman equation should be:

$$V_H(s_t) = \max_{a_t} \sum_{s_{t+1}} \Pr(s_{t+1}|s_t, a_t) \Big(\mathcal{R}(s_{t+1}, a_t, s_t) + \gamma V_{H-1}(s_{t+1}) \Big) \tag{4}$$

Now, if one have some prior knowledge about Markov processes, most of the time, the model will reach an equilibrium point, meaning that the value function will converge to a fixed point. One can also view this statement as: as the agent has infinite amount of time, it will eventually perform actions that maximize the expected sum of future rewards.

The same happens here. Consider the case when $H \to \infty$, then $V_H(s) \to V^*(s)$, where $V^*(s)$ is the value function of the state s when the agent acts optimally (to maximize the expected sum of future rewards). Then, the Bellman equation states that:

$$\forall s \in \mathcal{S} : V^*(s) = \max_{a} \sum_{s'} \Pr(s'|s, a) \Big(\mathcal{R}(s', a, s) + \gamma V^*(s') \Big) \tag{5}$$

where s' is the next state.

3 Action-Value function

3.1 Definition

The action-value function is a function that maps a state and an action to a value, representing the expected sum of future rewards starting from that state and action. That is, its formulation takes the form:

$$Q_H(s,a) = \mathbb{E}_\pi\left[\sum_{t=0}^{H} \gamma^t \mathcal{R}(s_{t+1}, a_t, s_t)\right] \tag{6}$$

where $a_0 = a$ and $s_0 = s$. Basically, the action-value function is the same as the value function, just that the first action is fixed. A simple relation is

$$V_H(s) = \max_a Q_H(s,a) \tag{7}$$

3.2 Bellman Equation

Similar to the value function, we also have the Bellman equation for the action-value function:

$$Q_H(s,a) = \sum_{s'} \Pr(s'|s,a)\left(\mathcal{R}(s',a,s) + \gamma \max_{a'} Q_{H-1}(s',a')\right) \tag{8}$$

and

$$Q^*(s,a) = \sum_{s'} \Pr(s'|s,a)\left(\mathcal{R}(s',a,s) + \gamma \max_{a'} Q^*(s',a')\right) \tag{9}$$

where equation (9) is (8) when $H \to \infty$.

4 Different approaches to RL

There are two main flavors of RL: **Dynamic Programming** and **Policy Optimization**. The goal of both approaches is to find a policy that maximizes the expected sum of future rewards. However, the ways they obtain the result are different.

4.1 Dynamic Programming

If you have learn Algorithms, you'll probably have some sense that based on the Bellman equation 4 and 8, we can use DP to solve the MDP. There are two main DP methods: **Policy Iteration** and **Value Iteration**. One flavor of Value Iteration is the infamous **Q-learning**. We'll cover Q-learning and other methods in later chapters. Here, we want to focus in the differentiation between Policy Iteration and Value Iteration.

4.1.1 Value Iteration

Based on equation 4, we can initialize a DP table with 2 dimensions: states and horizon. Below is the pseudocode for Value Iteration:

Algorithm 1 Value Iteration

Require: Transition probability $\Pr(s'|s,a)$, Reward function $\mathcal{R}(s',a,s)$, Discount factor γ, the horizon H

Ensure: Value function $V_H(s), \forall s \in \mathcal{S}$

```
Initialize V_0(s) = 0 for all s ∈ S
for h = 0 to H do
    for s ∈ S do
        V_h(s) ← max_a Σ_{s'} Pr(s'|s,a) ( R(s',a,s) + γ V_{h-1}(s') )
        π_h(s) ← arg max_a Σ_{s'} Pr(s'|s,a) ( R(s',a,s) + γ V_{h-1}(s') )
    end for
end for
return V_H(s), π_H(s)
```

As $H \to \infty$, the value function will converges. There is a theorem about this, which I don't go into details here. One can google about it. In short, the theorem states that algorithm 1 will converge to the optimal value function and

$$\forall s \in \mathcal{S} : V^*(s) = \max_a \sum_{s'} \Pr(s'|s,a)\Big(\mathcal{R}(s',a,s) + \gamma V^*(s')\Big)$$

One can check this file for the implementation of Value Iteration. Note that I vectorize the Bellman equation to make it faster.

4.1.2 Policy Iteration

In contrast to Value Iteration, Policy iteration is a two-step process: policy evaluation and policy improvement. Policy evaluation is the same as value iteration, while policy improvement is the same as policy iteration.

Here, given a policy π, we can evaluate a policy based on the resulted value function.

$$V_t^\pi(s) = \sum_{s'} \Pr(s'|s, \pi(s)) \Big(\mathcal{R}(s', \pi(s), s) + \gamma V_{t-1}^\pi(s') \Big) \tag{10}$$

Note that our policy have 'fixed' our action. Again, one can extend this to stochastic policy.

Algorithm 2 Policy Iteration

Require: Transition probability $\Pr(s'|s,a)$, Reward function $\mathcal{R}(s', a, s)$, Discount factor γ, number of iterations T

Initialize π_0 randomly

Compute $V^{\pi_0}(s), \forall s \in \mathcal{S}$

for $t = 1$ to T **do**

Improve: $\forall s \in \mathcal{S} : \pi_t(s) \leftarrow \arg\max_a \sum_{s'} \Pr(s'|s,a) \Big(\mathcal{R}(s', a, s) + \gamma V^{\pi_{t-1}}(s') \Big)$

Evaluation: Compute $V^{\pi_t}(s), \forall s \in \mathcal{S}$

end for

return $V_T(s), \pi_T(s)$

One can check this file for the implementation of Policy Iteration. Note that I vectorize the Bellman equation to make it faster.

Again, there's a theorem that states that Policy Iteration will converge to the optimal policy and value function, but I won't cover it here.

4.2 Policy Optimization

There are two main flavors of Policy Optimization: **DFO/Evolution** and **Policy Gradients**. We won't go into details about DFO/Evolution.

4.2.1 Policy Gradients

I will provide the details when we cover Policy Gradients.

4.2.2 Others

There is a class of methods called Actor-Critic methods. All of value iteration, policy iteration, and policy gradient are special cases of Actor-Critic methods. In simple sense, one given policy is our actor, which interact with the environment and collect data. We will then evaluate (critics) the policy based on the collected data and improve.

5 Q-Learning - Sample-based Approximation

5.1 Motivation and Algorithm

A down-side with Value Iteration and Policy Iteration is that they require a complete knowledge of the environment. That is we need to know the transition probability and reward function, which in real life, we rarely know. Furthermore, due to the DP nature, we can only really solve problems with small state spaces and action spaces.

So, how can we deal with this? It would be nice if one have some knowledge regarding estimates, but in short, if we have some statistics we want to estimate, we can create some 'estimators' which are random variables. An estimator is unbiased if its expected value is equal to the true value. That is if our true statistics is q and our estimator is $\hat{q}$, then $\mathbb{E}[\hat{q}] = q$.

With this in minds, consider the following estimate for the value function:

$$\hat{V}(s) = \mathcal{R}(s', a, s) + \gamma V(s')$$

Note that, s' is a random variable. And thus, if we take the expectation, we will arrive at our good old value function. Hence, this is an unbiased estimator. This inspires the following algorithm

Algorithm 3 Sample-based Approximation

Require: A way to interact with the environment which will get us the reward of an action and the next state.

Sample $s' \sim \Pr(s'|s, a)$

$\hat{V}(s) \leftarrow \mathcal{R}(s', a, s) + \gamma \hat{V}(s')$

Now, it's worth noting that we're define this recursively. However, the key idea is that, if we can take a bunch of transition samples $(s, a, s', r = \mathcal{R}(s', a, s))$, we can estimate the value function by taking the mean of the estimators. Similarly, we can create an estimator for the action-value function $Q(s, a)$. This gives rise to Q-Learning.

Algorithm 4 Q-Learning

Require: A way to interact with the environment which will get us the reward of an action and the next state.

Initialize $Q(s, a) = 0, \forall (s, a) \in \mathcal{S} \times \mathcal{A}$

Initialize an starting state $s \leftarrow s_0$

for $t = 1, 2, 3, \ldots$ until convergence **do**

  Sample an action a, get the next state s', and the reward r

  if s' is terminal **then**

    Initialize a new starting state $s \leftarrow s^*$

    $\texttt{target} \leftarrow r$

  else

    $\texttt{target} \leftarrow r + \gamma \max_{a'} Q(s', a')$

  end if

  $Q(s, a) \leftarrow Q(s, a) + \alpha(t)(\texttt{target} - Q(s, a))$

  $s \leftarrow s'$ if not terminal else $s \leftarrow s^*$

end for

return $Q(s, a)$

There are some uncertainty in the algorithm. Mainly, how do we sample actions? If always choose the action with the highest Q-value, we will get stuck in a local optimum. If we always choose a random action, we will not exploit the best action. To avoid this, we can use an epsilon-greedy policy.

Algorithm 5 ϵ-greedy

Require: $Q(s, a), s, \epsilon(t)$ where $\epsilon(t)$ is a time-dependent value between 0 and 1.

$x \leftarrow \texttt{random}(0, 1)$

if $x < \epsilon(t)$ **then**

  $a \leftarrow \texttt{random}(\mathcal{A})$

else

  $a \leftarrow \arg\max_{a'} Q(s, a')$

end if

return a

Note that we want $\epsilon(t)$ to be close to 1 initially and decreasing over time. This ensures that we explore initially and then exploit once the agent understand the environment. In practice, this is pretty good.

5.2 Properties of Q-Learning

Q-Learning is an off-policy algorithm, that is we learn the Q-value from a non-optimal policy. Furthermore, since we are iterating and improve our estimates, this belongs to the value iteration family.

Now, to ensure convergence, we want our learning rate $\alpha(t)$ to be close to 1 initially and decreasing over time. A general guidelines is to choose $\alpha(t)$ such that

$$\sum_{t=0}^{\infty} \alpha(t) = \infty \quad \text{and} \quad \sum_{y=0}^{\infty} \alpha^2(t) < \infty$$

5.3 Temporal Difference Learning

In Q-Learning, we try to estimate the Q-value. What's about the value function? It turns out, we can use a similar idea to estimate the value function. Of course, this is motivated by the unbiased estimator discussion we have earlier. This is called Temporal Difference Learning.

$$V_t^\pi(s) \leftarrow V_{t-1}^\pi(s) + \alpha(t)(r_t + \gamma V_{t+1}^\pi(s') - V_t^\pi(s)) \tag{11}$$

We'll repeat this process 11 until convergence. Again, we will also update our policy by

$$\pi_t(s) \leftarrow \arg\max_a \sum_{s'} \Pr(s'|s,a)\Big(\mathcal{R}(s',a,s) + \gamma V^{\pi_{t-1}}(s')\Big)$$

Note that the subscript for 11 is for different time step when evaluate the policy, while the superscript for the policy update rule is the 'outer' iteration.

6 Deep Reinforcement Learning

6.1 Motivation

Though Q-Learning is good, there is a major limitation with scalability. Will Q-Learning work for continuous spaces (especially when we don't want dicretize the space)? How about very large state spaces?

Looking over to our friend supervised learning, maybe, instead of trying hard to use a table, maybe we can learn some kind of functions that can take a state and a action as input and output a value. That function may also take learnable parameters θ. This is called Approximate Q-Learning.

We will then use gradient descent to update the parameters θ to minimize the loss function $\mathcal{L}(\theta; s, a) = \frac{1}{2}(r + \gamma \max_{a'} Q(s', a') - Q(s, a))^2$. The update rule is

$$\theta_t \leftarrow \theta_{t-1} - \alpha(t)\nabla_\theta \mathcal{L}(\theta; s, a)$$

We can see that Q-Learning is actually a special case of Approximate Q-Learning where our function is parameterized by $\theta \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$.

But is this enough? Not really.

6.2 Algorithm

There is still a major problem: stability. If one looks at the loss function, we can see that our target is non-stationary (dependent on $Q(\cdot)$). Furthermore, in supervised learning, we often assume that the data points are i.i.d (independently and identically distributed). This is not the case in RL, where picking a starting point may affect the overall trajectory. All of these problems can lead to unstable or even diverging behavior.

To address this, there are 2 basic strategies:

- Experience Replay
- Target Network

6.2.1 Experience Replay

Note that, one way to mitigate the "non-stationary" target and reduce the trajectory-dependence is to use experience replay. The idea is to store a fixed length trajectory, which are a sequence of transitions. Then, transitions are sampled from the replay buffer and used for gradient descent. This methods make the data distribution more stable than the online one.

6.2.2 Target Network

The next solution used is a target network. The idea is to have a target network that is updated less often than the online network. This helps to stabilize the learning process by making the target more 'stationary'.

6.2.3 Native Deep Q-Network (DQN)

With the two solutions in mind, we can create an algorithm that mimics Approximate Q-learning but is much more stable. This is called Deep Q-Network (DQN).

Algorithm 6 DQN

Require: A way to interact with the environment, number of episodes M, number of time steps T

Initialize replay buffer $\mathcal{D}$ of length N

Initialize the action-value function $Q(s, a; \theta)$ for random weights θ

Initialize the target network $\hat{Q}(s, a; \hat{\theta})$ with weights $\hat{\theta} = \theta$ initially.

for episode $0, 1, 2, \ldots, M$ **do**

Initialize the environment and get the initial state s_0

for time step $t = 0, 1, 2, \ldots, T$ **do**

Choose action a_t using ϵ-greedy policy

Take action a_t and observe the next state s_{t+1} and reward r_t

Store the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer. One need to keep track if this is terminal to compute the target.

if the replay buffer is full **then**

Sample a batch of $(s_j, a_j, r_j, s_{j+1}) \in \mathcal{D}$ transitions from the replay buffer.

Compute the target $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \hat{\theta})$ for each sample in the batch.

Update the action-value function $Q(s, a; \theta)$ using the sampled transitions with gradient descent.

For every C steps, update the target network $\hat{Q}(s, a; \hat{\theta})$ with weights $\hat{\theta} = \theta$

end if

end for

end for

Now, there are some niche details about the algorithm itself. First, the loss function used here is the Huber loss, which penalized large errors less severely than MSE. This is to make sure we still let the model make errors to explore the environment. The Huber loss is defined as

$$\mathcal{L}_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{12}$$

Secondly, the optimizer we used for DQN is RMSProp or Adam, which in practice work better than SGD. Lastly, the $\epsilon(t)$ parameter in sampling an action is annealed over time to reduce exploration ($\epsilon(0) = 1 \to \epsilon(10^6) \approx 0.05$ where t is the number of frames or steps).

6.2.4 DQN Variants

6.2.4.1 Double DQN Double DQN is a variant of DQN that addresses the overestimation problem in DQN. Now, note that when we take $\max_a Q(s', a'; \hat{\theta})$, we are using the target network to estimate the value of the next state. This can lead to overestimation (for some actions) if the target network is not updated often enough. To address this, we will choose action based on the online network, but use the target network to evaluate such action. This means that the term $y_j - \hat{y}_j$ becomes

$$r_j + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \hat{\theta}) - Q(s, a; \theta)$$

The arg max is taken with respect to the online network (θ), while the evaluation is done with the target network ($\hat{\theta}$).

6.2.4.2 Prioritized Experience Replay Prioritized Experience Replay is a variant of DQN that boosts convergence rate.

Instead of sampling data from the replay buffer u.a.r., we will sample data based on the Bellman error of the transition. The Bellman error is defined as

$$\delta_j = |y_j - \hat{y_j}| = |r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \hat{\theta}) - Q(s_j, a_j; \theta)|$$

This will lead to faster convergence rate.

6.2.4.3 Dueling DQN Dueling DQN is a variant of DQN that also deals with the overestimation problem.

Now, instead of output the action-value function directly, we will output the value function $V(s)$ and the advantage function $A(s, a)$. The advantage function, as the name suggests, show how advantagous an action is compared to some baseline, in this case, the value function.

Then, the action value function $Q(s, a) = V(s) + A(s, a)$. However, in dueling DQN, we compute the action-value function by

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} A(s, a)$$

6.2.4.4 Noisy Net for Exploration Noisy Net for Exploration is a variant of DQN that addresses the exploration problem in DQN.

Inspired by the idea of adding noises to make the model more robust (data augmentation), we will add noise to the parameters so that the agent will tends to explore more than to exploit initially. Formally, the noisy linear layer is defined as

$$\text{NoisyLinear}(x) = (\mu^w + \sigma^w \epsilon^w)x + (\mu^b + \sigma^b \epsilon^b)$$

where ϵ^w and ϵ^b are independent random variables sampled from a standard normal distribution. The σ^w and σ^b are learnable parameters that are used to control the noise.

7 Policy Gradients