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

We will now move on to the next class of algorithms: policy gradients. To set up the framework, we will now use a more 'flexible' policy - stochastic policy $\pi_\theta(u|s)$. To avoid confusion, we will now denote action by u.

Compared to the old 'deterministic' policy, the stochastic policy allows for a much smoother optimization problem. That is instead of a shift from one 'spiky' distribution to another, we can change the probability mass or density of the action, which allows for much smoother optimization landscape.

Nonetheless, the ultimate goal is still to find the optimal policy that maximizes the expected return $\mathbb{E}_{u\sim\pi_\theta(u|s)}[\mathcal{R}(\tau)]$ for some trajectory τ which starts at some state s_0.

7.1 Why Policy Gradients?

Before going into what policy gradients are, let's first think about why we use policy gradients. Compared to using the value function $V(s)$ and/or action-value function $Q(s,a)$, policy gradients allows us to directly optimize the policy. In value-based methods, we either need a dynamic models (for $V(s)$) or an efficient way to compute $\arg\max_a Q(s,a;\theta)$ for some state s. However, as the problem scales larger, these approaches aren't that good. Table 1 show the comparison between value-based methods and policy gradients.

	Policy Gradients	**Dynamic Programming**
Concepts	Optimize directly what we care about	Indirect, exploit problem structure (Bellman Eq.), self-consistent
Empirical	Compatible with rich architectures (parameterized policy); versatile; auxiliry objectives	Exploration and off-policy learning; sample-efficient (if works)

Table 1: Comparison of Value-based (DP) Methods and Policy Gradients

7.2 Likelihood Ratio Policy Gradients - REINFORCE

Let τ be a trajectory $(s_0, u_0, s_1, u_1, \ldots, s_H, u_H)$. Then, define

$$\mathcal{R}(\tau) = \sum_{t=0}^{H-1} \mathcal{R}(s_t, u_t) \tag{13}$$

to be the reward of the trajectory τ. Then, the utility function is defined as

$$U(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\mathcal{R}(\tau)] = \sum_\tau \Pr(\tau;\theta)\mathcal{R}(\tau) \tag{14}$$

to be the expected reward of the policy π_θ over the distribution of the trajectory τ. The goal is to find θ such that $U(\theta)$ is maximized.

7.3 Policy Gradient Theorem

Since we're maximize $U(\theta)$ w.r.t. θ, we can try taking the gradient of $U(\theta)$ w.r.t. θ.

$$
\begin{aligned}
\nabla_\theta U(\theta) &= \nabla_\theta \sum_\tau \Pr(\tau;\theta)\mathcal{R}(\tau) \\
&= \sum_\tau \nabla_\theta \Pr(\tau;\theta)\mathcal{R}(\tau) \\
&= \sum_\tau \frac{\Pr(\tau;\theta)}{\Pr(\tau;\theta)} \nabla_\theta \Pr(\tau;\theta)\mathcal{R}(\tau) \\
&= \sum_\tau \Pr(\tau;\theta) \frac{\nabla_\theta \Pr(\tau;\theta)}{\Pr(\tau;\theta)} \mathcal{R}(\tau) \\
&= \sum_\tau \Pr(\tau;\theta) \nabla_\theta \log \Pr(\tau;\theta)\mathcal{R}(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \Pr(\tau;\theta)\mathcal{R}(\tau)]
\end{aligned}
$$

This is the policy gradient theorem. It states that the gradient of the expected reward w.r.t. the policy parameters is equal to the expected gradient of the log probability of the trajectory w.r.t. the policy parameters.

$$
\nabla_\theta U(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \Pr(\tau;\theta)\mathcal{R}(\tau)] \tag{15}
$$

One thing worth mentioning is the interpretation of this gradient. Now, say we have two trajectories τ_1 and τ_2 and $\mathcal{R}(\tau_1) > 0$ and $\mathcal{R}(\tau_2) < 0$. Then, the gradient $\nabla_\theta \log \Pr(\tau_1;\theta)\mathcal{R}(\tau_1)$ is positive while $\nabla_\theta \log \Pr(\tau_2;\theta)\mathcal{R}(\tau_2)$ is negative. One implication is that the gradient increase the likelihood of the trajectory with positive reward and decrease the likelihood of the trajectory with negative reward. However, it's worth noting that the gradient only changes the probability, not the trajectory itself.

7.4 REINFORCE - 'Vanilla' Policy Gradients

7.4.1 Gradient Estimation

With this in mind, we can use gradient descent with an approximation of the gradient $\hat{g} \approx \nabla_\theta U(\theta)$. The key problem is now how to estimate $\nabla_\theta U(\theta)$. In fact, the core issue is to estimate $\nabla_\theta \log \Pr(\tau;\theta)$. Let's open the gradient and see what we get

$$
\begin{aligned}
\nabla_\theta \log \Pr(\tau;\theta) &= \nabla_\theta \log \left[\prod_{t=0}^{H-1} \Pr(s_{t+1}|s_t,u_t)\pi_\theta(u_t|s_t) \right] \\
&= \nabla_\theta \sum_{t=0}^{H-1} \log \Pr(s_{t+1}|s_t,u_t) + \log \pi_\theta(u_t|s_t) \\
&= \sum_{t=0}^{H-1} \nabla_\theta \log \Pr(s_{t+1}|s_t,u_t) + \nabla_\theta \log \pi_\theta(u_t|s_t)
\end{aligned}
$$

Now, note that the transition probability $\Pr(s_{t+1}|s_t,u_t)$ is fixed and doesn't depend on the policy parameters θ. Thus, it's gradient w.r.t. θ is zero. Therefore,

$$
\nabla_\theta \log \Pr(\tau;\theta) = \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(u_t|s_t)
$$

Now, there's also a different way to arrive at this. Consider

$$
U(\theta) = \mathbb{E}_{\tau\sim\theta_{\text{old}}}\left[\frac{\Pr(\tau|\theta)}{\Pr(\tau|\theta_{\text{old}})} \mathcal{R}(\tau) \right] \tag{16}
$$

Here, the trajectories τ are sampled from the old policy θ_{old}; thus, we add a correction term $\frac{1}{\Pr(\tau|\theta_{\text{old}})}$.

We can now take the gradient of $U(\theta)$ w.r.t. θ at the point $\theta = \theta_{\text{old}}$ to get

$$\nabla_\theta U(\theta)|_{\theta=\theta_{\text{old}}} = \mathbb{E}_{\tau\sim\theta_{\text{old}}}\left[\nabla_\theta \frac{\Pr(\tau|\theta)|_{\theta=\theta_{\text{old}}}}{\Pr(\tau|\theta_{\text{old}})}\mathcal{R}(\tau)\right] \tag{17}$$

$$= \mathbb{E}_{\tau\sim\theta_{\text{old}}}\left[\nabla_\theta \log \Pr(\tau|\theta)|_{\theta=\theta_{\text{old}}}\mathcal{R}(\tau)\right] \tag{18}$$

This is precisely the policy gradient theorem. We can see that both formulas give us the same gradient value at $\theta = \theta_{\text{old}}$. We can again break down the probability and arrive at the same result above.

7.4.2 Stablizing Techniques

Now, with that in minds, there is still a problem that we need to address. Though the estimation is unbiased, it is also very noisy. To explain this, consider the gradient estimator

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log \Pr(\tau_i|\theta)\mathcal{R}(\tau_i)$$

Note that this estimation is based on the set of trajectories τ_i. What happens if we collect some samples, and all of them have positive rewards? Similarly, what if we collect some samples, and all of them have negative rewards? This means that the estimation is highly sensitive to noise, and one way to address this is to collect a huge amount of data. However, computation resources aren't infinite, and we need to find a balance between the number of samples and the variance of the gradient estimator.

To reduce the variance, we can reduce the scale of the values that the gradient can take ($\mathbb{V}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$). One way to do this is to use a baseline. To be more specific, we want to 'center' the rewards and ask the question: how good is this reward compared to the average reward?

Baseline Formally, the estimation becomes

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log \Pr(\tau_i|\theta)(\mathcal{R}(\tau_i) - b)$$

where b is the baseline. One key observation is that this estimator is still unbiased! Below is a proof:

$$\begin{aligned}
\mathbb{E}_\tau[\nabla_\theta \log \Pr(\tau|\theta)b] &= \sum_\tau \Pr(\tau|\theta)\nabla_\theta \log \Pr(\tau|\theta)b \\
&= \sum_\tau \Pr(\tau|\theta)\frac{\nabla_\theta \Pr(\tau|\theta)}{\Pr(\tau|\theta)}\mathbb{E}_\tau[b] \\
&= \sum_\tau \nabla_\theta \Pr(\tau|\theta)\mathbb{E}_\tau[b] \\
&= \nabla_\theta \Pr(\tau|\theta)\mathbb{E}_\tau[b] \\
&= \nabla_\theta \mathbb{E}_\tau[b]
\end{aligned}$$

Now, if the baseline is independent of the policy (or the actions), then $\nabla_\theta \mathbb{E}_\tau[b] = 0$, and the gradient estimator is unbiased. Note that b can depend on the states s. With some hindsight, we will see that the value function $V^\pi(s)$ is a good baseline.

Temporal Structure To further reduce the variance (reduce the scale), we can make use of the temporal structure of the trajectory. Consider the current estimate

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})\Big(\sum_{k=0}^{t-1}\mathcal{R}(s_k^{(i)},u_k^{(i)}) + \sum_{k=t}^{H-1}\mathcal{R}(s_k^{(i)},u_k^{(i)}) - b\Big)$$

Note that we can remove the term $\sum_{k=0}^{t-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})$ since the reward obtained before time step t is independent of the action at time step t. More formally, consider

$$\begin{aligned}
\mathbb{E}_\tau\Big[\nabla_\theta \log \pi_\theta(u_t|s_t)\sum_{k=0}^{t-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})\Big] &= \sum_{k=0}^{t-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})\mathbb{E}_\tau\Big[\nabla_\theta \log \pi_\theta(u_t|s_t)\Big] \\
&= \sum_{k=0}^{t-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})\mathbb{E}_\tau\Big[\nabla_\theta \log \pi_\theta(u_t|s_t)\Big] \\
&= \sum_{k=0}^{t-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})\mathbb{E}_\tau\Big[\frac{\nabla_\theta \pi_\theta(u_t|s_t)}{\pi_\theta(u_t|s_t)}\Big] \\
&= \sum_{k=0}^{t-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})\nabla_\theta\mathbb{E}[1] = 0
\end{aligned}$$

Thus, we can simpify our gradient estimator to

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})\Big(\sum_{k=t}^{H-1}\mathcal{R}(s_k^{(i)},u_k^{(i)}) - b\Big)$$

But, what is this term $\sum_{i=t}^{H-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})$? This is precisely the action-value function for state s_t under policy π_θ, denoted as $Q_\theta(s_t,u_t)$. Note that, for stability, a discount factor is often added, i.e.

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})\Big(\sum_{k=t}^{H-1}\gamma^{k-t}\mathcal{R}(s_k^{(i)},u_k^{(i)}) - b\Big) \tag{19}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})\Big(Q^{\pi,\gamma}(s_t^{(i)},u_t^{(i)}) - b\Big) \tag{20}$$

Now, the last puzzle is the value of b. There are some common choices. They are

- Constant baseline: $b = \mathbb{E}[\mathcal{R}(\tau)] = \frac{1}{m}\sum_{i=1}^{m}\mathcal{R}(\tau_i)$
- Optimal (minimum variance) function: $b = \frac{\sum_i(\nabla_\theta \log \Pr(\tau_i|\theta))^2\mathcal{R}(\tau_i)}{\sum_i(\nabla_\theta \log \Pr(\tau_i|\theta))^2}$
- Time-dependent function: $b(t) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=t}^{H-1}\mathcal{R}(s_k^{(i)},u_k^{(i)})$
- Value (State-dependent) function: $b = V^\pi(s_t)$

Among these, the value function is the one that makes sense. Why? Because $A(s_t,u_t) = Q^\pi(s_t,u_t) - V^\pi(s_t)$, which is the advantage function, denotes how advantagous we are by choosing action u_t in state s_t compared to the 'current' best return we know we can get from state s_t.

So, the problem now boils down to estimating $V_\phi^\pi(s)$ (ϕ is the parameters for estimating the value function). We have seen the temporal difference learning before (see equation 11). We still start by collecting sample transitions (s, u, s', r) and initialize $V_{\phi_0}^\pi(s)$. However, there is a small twist to make the training stable:

$$\phi_\pi \leftarrow \arg\min_\phi \frac{1}{m}\sum_{i=1}^{m} ||r + \gamma V_\phi^\pi(s') - V_\phi^\pi(s)||_2^2 + \kappa||\phi - \phi_i||_2^2 \tag{21}$$

The second regularization term is used to prevent overfitting and thus unstability.

Another method is to regress against empirical data. We first collect a set of trajectories τ_i, and then we try to fit a function $V_\phi^\pi(s)$ to the data.

$$\phi_\pi \leftarrow \arg\min_\phi \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\left(V_\phi^\pi(s_t^{(i)}) - \sum_{k=t}^{H-1}\gamma^{k-t}\mathcal{R}(s_k^{(i)}, u_k^{(i)})\right)^2 \tag{22}$$

7.4.3 REINFORCE

With everything in place, we are ready to see the algorithm.

Algorithm 7 REINFORCE - Vanilla Policy Gradient

Initialize policy parameters θ, baseline $b(s_t)$
for each episode **do**
  Generate a set of trajectories $\tau_1, \tau_2, \ldots$
  At each time step of each trajectory, compute $\mathcal{R}_t^{(i)} = \sum_{k=t}^{H-1}\gamma^{k-t}\mathcal{R}(s_k^{(i)}, u_k^{(i)})$ and the advantage estimate $\hat{A}_t^{(i)} = \mathcal{R}_t^{(i)} - b(s_t^{(i)})$
  Refit the baseline $b(s_t)$ by minimizing $||b(s_t) - R_t||^2$ summed over all time steps for all trajectories
  Update policy parameters θ by using gradient estimate $\hat{g}$
end for

Note that we use discount to increase the stability of the algorithm.

Depending on how one refit $b(s_t)$ and compute $\hat{A}_t^{(i)}$, the algorithm can be different.

7.4.4 Actor-Critic Methods

We now extends to a family of RL algorithms called Actor-Critic methods. The REINFORCE algorihtm uses Monte-Carlo methods. More specifically, it samples trajectories and estimate the gradient of the policy using the Monte-Carlo estimate of the return. This approach may still have high variance due to long-horizon task. To address this, we use the idea that we have seen in DQN - an evaluating function or critic.

In short, Actor-Critic methods are RL algorithms that has two components: the actor (the policy that we used to interact with the environment) and the critic (the evaluating function that we used to evaluate the policy).

General Actor-Critic algorithm Some examples have been shown in the analysis in section 7.4.2. For example, Instead of using $\mathcal{R}_t$, we can use $Q^\pi(s_t, u_t)$ (see equation 20) to examine how well the policy performs. There are some estimators for $Q^\pi(s_t, u_t)$ that we can use, we will see some of the ways below as we inspect some variants of the REINFORCE algorithm that implements Actor-Critic methods.

Algorithm 8 General Actor-Critic algorithm

Initialize π_{θ_0} and $V^{\pi}_{\phi_0}$

Collect roll-outs $\{(s_t, u_t, s_{t+1}, r_t)\}$ and compute $\hat{Q}_i(s_t, u_t)$ for all time steps of all roll-outs

Update:

- $\theta_{T+1} \leftarrow \theta_T \alpha \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H-1} \nabla \log \pi_{\theta_T}(u_t|s_t)(\hat{Q}_i(s_t, u_t) - V^{\pi}_{\phi_T}(s_t))$
- $\phi_{T+1} \leftarrow \arg\min_{\phi} \sum_{(s_t,u_t,s_{t+1},r_t)} ||\hat{Q}_i(s_t, u_t) - V^{\pi}_{\phi}(s_t)||_2^2 + \kappa||\phi - \phi_i||_2^2$

Repeat until convergence

Asynchronous Advantage Actor-Critic: A2C and A3C The core idea here is that we can express $Q^{\pi}(s_t, u_t)$ as $Q^{\pi}(s_t, u_t) = \mathbb{E}_{\tau}[r_1 + \gamma r_2 + \cdots + \gamma^{H-1} r_H]$. But we have seen before that this can be simplified, with a parameter k, as $Q^{\pi}(s_t, u_t) = \mathbb{E}_{\tau}[r_1 + \gamma r_2 + \ldots \gamma r_{k-1} + \gamma^k V^{\pi}(s_{k+1})]$. That is, we'll look a head k steps to estimate the value of $Q^{\pi}(s_t, u_t)$. To make it clearer,

$$\begin{aligned} \hat{Q}^{\pi,k}(s_t, u_t) &= r_1 + \gamma r_2 + \ldots \gamma r_{k-1} + \gamma^k V^{\pi}(s_{k+1}) \\ \mathcal{R}_t^{(i)} &= \hat{Q}^{\pi,k}(s_t^{(i)}, u_t^{(i)}) - V^{\pi}(s_t^{(i)}) \end{aligned} \tag{23}$$

Generalized Advantage Estimation: GAE Using a somewhat similar idea to A3C, GAE extends the idea of A3C to have an exponentially decayed average to estimate the advantage function. To be more specific, given a parameter $0 \leq \lambda < 1$, we have

$$\hat{Q}^{\pi,\lambda}(s_t, u_t) = \sum_{k=t}^{H-1} (1-\lambda)\lambda^{k-t} \hat{Q}^{\pi,k}(s_t, u_t) \tag{24}$$

where $\hat{Q}^{\pi,k}(s_t, u_t)$ is the same as in equation 23.

7.5 Trust Region Policy Optimization (TRPO)

7.5.1 REINFORCE drawbacks

REINFORCE has a few drawbacks. First, it is not stable. To be more specific, one can imagine this instability by looking over the over-stepping in supervised learning. In SL, if our learning rate is too high, we may over-step and our model performs even worse after the update. Similarly, if our update makes the policy worse, we will sample data from an even worse policy, which leads to even worse update. This will make the training unstable.

Second, it is not sample-efficient. Note that we collect a bunch of trajectories, but then only perform ONE updates. This means that we need to perform a lot of episodes in order to improve the policy.

So, how can we address this?

7.5.2 Trust Region Policy Optimization

The core idea is to only perform a 'small' step. To be more specific, we don't want outr policy to change radically. Consider the loss function in REINFORCE:

$$\mathcal{L}^{PG}(\theta) = \hat{\mathbb{E}}_t[\log \pi_{\theta}(u_t|s_t)\hat{A}_t]$$

If an action has a large empirical advantage, the update may 'overpush' the probability to 1. Similarly, if the empirical advantage is negative, the update may 'overdecrease' the probability to 0. This leads to instability with the logarithm. Now, recall equation 16. This yields the same gradient that we use in REINFORCE. Of course, this doesn't change anything yet. However, notice this means we can use importance sampling.

Now, to the main idea: how do we incorporate the idea that the policy shouldn't change much? If one recalls, in statistics, we have a powerful tool called KL-divergence. Basically, it quantifies how different two probability distributions are from each other. From this, the problem is now an optimization problem:

$$\begin{aligned} &\text{maximize}_\theta && \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(u_t|s_t)}{\pi_{\theta_{\text{old}}}(u_t|s_t)}\hat{A}_t\right] \\ &\text{subject to} && D_{KL}(\pi_\theta(\cdot|s_t)\|\pi_{\theta_{\text{old}}}(\cdot|s_t)) \leq \delta \end{aligned}$$

That is, we now add the constraint that the KL-divergence between the old policy and the new policy should be small. Then, we have our algorithm:

Algorithm 9 Trust Region Policy Optimization

Initialize π_{θ_0}

Collect roll-outs $\{(s_t, u_t, s_{t+1}, r_t)\}$ and compute $\hat{A}_t$ for all time steps of all roll-outs

Find θ which maximizes $\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(u_t|s_t)}{\pi_{\theta_{\text{old}}}(u_t|s_t)}\hat{A}_t\right]$ subjected to $D_{KL}(\pi_\theta(\cdot|s_t)\|\pi_{\theta_{\text{old}}}(\cdot|s_t)) \leq \delta$

Repeat until convergence

There are a number of ways one can solve the optimization problem and one can search about this online. I won't go into details here (Check 'Natural Policy Gradient' or 'Conjugate Gradient' for more details).

There are two prominent variants of TRPO: KFAC, which leads to ACKTR, and the penalized KL. For the first one, I recommend reading the paper. For the second one, the idea is quite simple - adding the constraint to the objective:

$$\text{maximize}_\theta \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(u_t|s_t)}{\pi_{\theta_{\text{old}}}(u_t|s_t)}\hat{A}_t - \beta D_{KL}(\pi_\theta(\cdot|s_t)\|\pi_{\theta_{\text{old}}}(\cdot|s_t))\right]$$

where β is a hyperparameter that controls how much we care about the KL-divergence. There are also methods to solve this, one of the is the linear-quadratic approximation. This leads to a different TRPO algorithm

Algorithm 10 KL-penalized TRPO

Initialize π_{θ_0}

Collect roll-outs $\{(s_t, u_t, s_{t+1}, r_t)\}$ and compute $\hat{A}_t$ for all time steps of all roll-outs

Find θ which maximizes $\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(u_t|s_t)}{\pi_{\theta_{\text{old}}}(u_t|s_t)}\hat{A}_t - \beta D_{KL}(\pi_\theta(\cdot|s_t)\|\pi_{\theta_{\text{old}}}(\cdot|s_t))\right]$

If the KL-divergence is too large, increase $\beta \leftarrow 1.5\beta$. If it's too low, decrease $\beta \leftarrow \beta/1.5$

Repeat until convergence

The coefficients 1.5 and $\frac{1}{1.5}$ are well-tuned hyperparameters found by the original authors.

Now, as a reality check, note that:

- If we remove the constraint (or the penalty), we have our policy iteration (start with some policy, evaluate, improve, repeat)

- If we replace the KL-divergence penalty with L2-regularization, we arrive at our REINFORCE algorithm.

7.5.3 Proximal Policy Optimization

Note that the idea in TRPO is great, however, solving the optimization problem is not easy and fast. To make the implementation easier (and faster), we have Proximal Policy Optimization (PPO). The idea is very staightforward. Instead of have a KL-penalty or constraint to keep the policies close to each other, we instead clip the value of the ratio $r_t(\theta) = \dfrac{\pi_\theta(u_t|s_t)}{\pi_{\theta_{\text{old}}}(u_t|s_t)}$. More specifically, the loss is

$$\mathcal{L}^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{25}$$

where ϵ is a hyperparameter. Now, the full loss function does include an entropy term for exploration and a value function term for stability. The loss in that case is

$$\mathcal{L}^{PPO}(\theta) = \hat{\mathbb{E}}_t\Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big] + c_1\hat{\mathbb{E}}_t[(V_\theta(s_t) - R_t)^2] + c_2 S_{\pi_\theta}(s_t) \tag{26}$$

where R_t is the return at time t and c_1, c_2 are hyperparameters.

Algorithm 11 PPO

Initialize π_{θ_0}
Collect roll-outs $\{(s_t, u_t, s_{t+1}, r_t)\}$ and compute $\hat{A}_t$ for all time steps of all roll-outs
Sample some batches of data from the roll-outs.
Find θ which maximizes $\hat{\mathbb{E}}_t\Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big] + c_1\hat{\mathbb{E}}_t[(V_\theta(s_t) - R_t)^2] + c_2 S_{\pi_\theta}(s_t)$.
This can be done with SGD.
Repeat until convergence

Note that we now use the data much more efficient and the implementation is rather straight forward.