Notes on Deep RL

Fengyu Li

Summer 2022

These notes are based on the materials for these courses, all of which are publicly available:

1. CS 188 Introduction to Artificial Intelligence, UC Berkeley
2. CS 4789 Introduction to Reinforcement Learning, Cornell University
3. CS 285 Deep Reinforcement Learning, UC Berkeley

Contents

1 Markov Decision Process

1.1 Definitions

A **Markov decision process** (MDP) is a non-deterministic discrete search. An MDP is specified by:

1. A set of states S.
2. A set of actions A.
3. A transition function $T(s, a, s') = \Pr(s' \mid s, a)$.
4. A reward function $R(s, a, s')$ represents the reward the agent gains by taking action a in state s and transitioning to state s'.
5. A start state.
6. (Optional) One or more terminal states.
7. (Optional) A discount factor γ.

In a partially observed MDP, which is the usual case in reinforcement learning, we will also have a set of possible observation O and an emission function $\varepsilon(o, s) = \Pr(o \mid s)$.

Similar to the Markov chain, given the current state, the future states and the states in the past are independent. (Hence "Markov" decision process.)

For a Markov decision process, we want to find an optimal **policy** $\pi^* : S \to A$ that gives an action given each possible state. An optimal policy maximizes the expected utility.

Discounting is a technique used to encourage the agent to prefer rewards sooner than later and to limit the total number of steps the agent takes. Each time the agent takes an action, the reward is discounted by the **discount factor** γ. Now, the objective becomes to maximize the discounted utility:

$$U([s_0, a_0, s_1, a_1, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \cdots$$

Another technique to prevent infinite steps and utilities is **finite horizon**. This simply terminates actions after a finite number of steps. A disadvantage of finite horizon is that it causes π^* to be affected by the amount of time left.

For an MDP, a few other optimal quantities can be defined besides the optimal policy. We define the **value function** of a state, $V^*(s)$, as the expected utility of the agent acting optimally given the current state. Notice that when the agent takes an action, we can define an intermediary state before it transitions from s to s'. This non-deterministic state that leads to an uncertain next state is called a **Q-state**. We define the **Q-function** (or Q-value) of a Q-state, $Q^*(s, a)$, as the expected utility of the agent acting optimally given the current state and an action already taken. Equivalently,

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^*(s'))$$

which are known as the **Bellman equations**.

1.2 MDP Algorithms

From the Bellman equations, we can generalize the **value iteration**, a dynamic-programming method for computing the value function under the optimal policy:

$$V_0^*(s) = 0$$

$$V_{k+1}^*(s) \leftarrow \max_a \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V_k^*(s'))$$

where the subscript k refers to the remaining time steps. The value iteration converges to the unique optimal value.

Theorem 1. *The value iteration always converges to the optimal policy. (assuming $\gamma \leq 1$ and transition probabilities are correct.)*

Informal Proof. Suppose the search tree of an MDP has maximum depth m, then V_m^* obviously holds the optimal value. For any $k \neq m$, the difference between V_k^* and V_{k+1}^* is at most $\gamma^k \max |R|$. Therefore, the value iteration converges to the optimal value as $k \to m$. □

We can compute the optimal policy if we know the value function. The procedure is called **policy extraction** and is similar to the Bellman's equations:

$$\pi^*(s) = \arg\max_a Q^*(s, a) = \arg\max_a \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^*(s'))$$

Although the value iteration always converges, it has some problems:

1. It is slow. Each iteration takes $O(S^2 A)$ time.
2. Iterations are wasted because the argmax rarely changes across states.
3. Iterations are wasted because policy converges much sooner than the value function.

Alternatively, we can approach the optimal policy using the **policy iteration**. The process works analoguesly to the gradient descent method. It contains two steps that are repeated until convergence:

1. (Policy evaluation) Calculate the utility for some fixed policy $V^\pi(s)$:

   $$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')(R(s, \pi(s), s') + \gamma V^\pi(s'))$$

   which can be solved as a system of linear equations, as the above equation maps each $V^\pi(s)$ to a linear combination of all other $V^\pi(s')$. We can also find the values using an iterative approach:

   $$V^\pi_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')(R(s, \pi(s), s') + \gamma V^\pi_k(s'))$$

2. Update the policy using policy extraction. If the policy yields a better expected utility, the model has converged. Otherwise, feed the updated policy back into the previous step.

Lemma 2. *The updates in policy iteration are monotonic improvements. Equivalently,*

$$\forall s, V^\pi_{k+1}(s) \geq V^\pi_k(s)$$

Theorem 3. *The policy iteration always converges to the optimal policy.*

Explicitly computing value iteration and policy iterations are examples of **offline** algorithms because they don't involve actually playing with the machine and all relevant figures are computed based on the details of the MDP. If that is not the case and the MDP is itself not clear, different approaches are needed. The study of these different, learning-based techniques is called **reinforcement learning**.

2 Introduction to the Problems

2.1 Imitation Learning

Before we start with reinforcement learning, it is helpful to first examine a way to learn the policy under a more traditional machine learning setting. In **imitation learning**, the agent can observe the environment and an expert taking actions. The agent can imitate the expert's behavior and learn under a **supervised learning** framework.

A problem of imitation learning is that the learned policy $\pi_\theta(a_t \mid o_t)$, where o_t is the observation at time t and π_θ is the policy learned under parameters θ. (We shall continue a parameterized notation of policy for the rest of this note.) It is not guaranteed to work because of **distributional shift**, i.e., different distribution of observations during training and testing (because the agent makes suboptimal actions and thus enters new, unseen states / observations, which makes the errors prone to propagate.) This problem is especially prevalent in **behavioral cloning** algorithms where the agent learns a mapping from actions to observations given labelled observations. To solve this problem, we need to make sure the distribution of the observation in the original datasets is the same as under the learned policy:

$$p_{\text{data}}(o_t) = p_{\pi_\theta}(o_t)$$

Since it is hard to directly affect $p_{\pi_\theta}(o_t)$, we make efforts to make $p_{\text{data}}(o_t)$ constantly similar to the observations actually seen by the agent. A technique that handles this is **dataset aggregation** (DAgger), where we iteratively generate a new, aggregated dataset of observations under the recently learned policy, label it, and feed it back to the agent for further training. Imitation learning has several problems:

1. It could be hard to label the observations acquired by the agent. It could be hard to make near-optimal actions for humans at the first place.

2. States could be non-Markovian, e.g., human might have different actions given the same observation because we also make decisions based on previous input.

3. The agent could have causal confusion because it incorrectly associates a common feature of the observation with an action.

4. Multimodal behaviors.

It is sometimes believed that this supervised learning framework is insufficient for certain tasks. In this case, we need a more self-enhancing framework: reinforcement learning.

2.2 Reinforcement Learning

Reinforcement learning involves an agent that actively interacts with the environment and gains rewards. Neither the transition function nor the reward function is known. The agent learns to optimize its utility by observing the environment and learning to make decisions based on the rewards received. Therefore, reinforcement learning is a type of machine learning.

There several general paths to carry out RL (under traditional frameworks):

1. **Model-based RL**: learn an approximation of the true MDP by approximating the transition function T and finding R by experiencing the transitions in T.

2. **Value-based RL**: given a fixed policy, learn the state values policy evaluation.

$$V_{k+1}^*(s) \leftarrow \underbrace{\max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_k^*(s'))}_{\text{approximate through sampling}}$$

An important work on passive RL is **temporal-difference learning**, the latter term in the above equation is approximated using exponential moving average, i.e.,

$$V^{\pi_\theta}(s) \leftarrow V^{\pi_\theta}(s) + \alpha(\text{sample} - V^{\pi_\theta}(s))$$

However, although we can approach the value functions, it is still unrealistic to propose a better policy based on those. All we can do from them is policy evaluation.

Alternatively, we can also learn the Q-values. This technique is also referred to as **Q-learning**.

$$Q_{k+1}(s,a) \leftarrow \underbrace{\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q_k(s',a'))}_{\text{approximate through sampling}}$$

Q-values are updated using exponential moving average. To obtain a policy, we simply take the argmax of all the actions on their Q-values.

3. Policy Search: Different from previous methods that learn the value function or the Q-function, policy search starts with an "okay" policy, which could be the result of Q-learning, and then improves it by nudging feature weights (if using feature representation.)

Theorem 4. *Q-learning converges to the optimal policy (under the correct transition probabilities.)*

In reinforcement learning, we face the problem of **exploration-exploitation trade-off**. The agent has to balance between the two, while exploration teaches it about the environment and exploitation (following current optimal policy) could yield better utility.

ϵ-greedy introduces a hyperparameter, ϵ, which denotes the probability that the agent acts randomly and not following the current optimal policy. We typically lower ϵ as the agent learns.

Exploration function is a clever way to decide whether or what to explore instead of letting the agent explore randomly. It is a function $f(u, n)$ where u is the utility estimate and n is the number of times the agent has visited the state. It gives bonus to fresh visits and penalizes otherwise. A simple example would be $f(u, n) = u + k/n$. When updating the Q-value, we took the modified maximum that incorporates the exploration function:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} f(Q_k(s',a')), N(s',a'))$$

The performance difference between ϵ-greedy and exploration function can be measured by **regret**, which measures the total mistake cost. Minimizing the regret means to optimally learning the optimal policy. The exploration function has a lower regret than ϵ-greedy.

In reality, an MDP could have so many states and actions that it is impossible for Q-learning to experience them all multiple times. We can use feature-based representations to significantly reduce the number of states by mapping each state to a feature vector. In this way, we can express the value and the Q-value for any state as a linear combination of the feature vectors. Feature vectors should be carefully designed so that similar feature represents similar states in the MDP. We can reexpress the Q-learning process as a linear machine learning problem:

$$Q(s, a) = \boldsymbol{w} f(s, a)$$

where $f \colon S \times A \to \mathbb{R}$ is the feature extraction function. Using this new expression for Q-values, we can carry out Q-learning under the traditional ERM framework and applying optimization algorithms such as gradient descent. However, the optimum attained under this linear assumption is not guaranteed to be the true optimum.

2.3 Deep Reinforcement Learning

In practice, it is hard to design reinforcement learning algorithms that are powerful enough to handle complex systems. It has been constantly a question how features can be extracted from the state space so that they perform well under complex, end-to-end scenarios. **Deep reinforcement learning** tackles this problem by introducing deep learning methods such as deep neural networks. Deep RL algorithms are typically much better **end-to-end** learning algorithms that penetrates the entire problem without manually designing intermediary features and states.

We can formally define the objective of a (deep) RL problem as bettering the policy $\pi_\theta \colon (a \mid s)$. A primary difference of this new definition of the policy is that it outputs a probability distribution of actions given a state, i.e., it is non-deterministic. This is made feasible by the characteristics of deep neural networks. The policy function is now a linear operator on the state space. Now we can reexpress the value function and the Q-function under this non-deterministic policy as expectations. (discounting is neglected.)

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi(a_t \mid s_t)}[Q^\pi(s_t, a_t)]$$

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_\pi[R(s'_t, a'_t) \mid s_t, a_t]$$

We define a **trajectory** τ as a consecutive sequence of states and actions. A probability distribution of the trajectories can thus be expressed as a function of the model parameters:

$$p_\theta(\tau) = p_\theta(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) T(s_t, a_t, s_{t+1})$$

The objective is to find the trajectory that maximizes the expected reward:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t R(s_t, a_t) \right]$$

A nice property of adopting a non-deterministic policy is that it makes the following into a Markov chain:

$$\prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) T(s_t, a_t, s_{t+1})$$

Thus, by linearity, the objective becomes

$$\begin{aligned} \theta^* &= \arg\max_\theta \sum_t \mathbb{E}_{(s_t, a_t) \sim p_\theta(s_t, a_t)} \left[R(s_t, a_t) \right] \\ &= \arg\max_\theta \mathbb{E}_{s_1} V^\pi(s_1) \end{aligned}$$

Notice that both $p_\theta(s_t, a_t)$ and $p_\theta(\tau)$ are now (likely) smooth over θ, which thus allows using optimization methods that are typical in deep learning (since it is now differentiable.) To attain this objective, a deep RL algorithm typically iterates on the following track:

1. Generate samples by running the policy exploratively.
2. Fit a model / estimate the return.
3. Improve the policy.

In deep RL, unlike in Q-learning, we typically don't fix an action that maximizes the Q-value, i.e., we don't set $\pi(a' \mid s) = 1$ if $a' = \arg\max_a Q(s, a')$. Instead, we leverage the property of the policy being a probability distribution and just increase the probability of selecting a'. That is, we increase $\pi(a' \mid s)$ if $Q^\pi(s \mid a') > V(s)$. Methods for achieving this will be covered in later sections.

The algorithms that will be introduced can be classified into two categories: **off-policy** algorithms and **on-policy** algorithms. The former is able to improve the policy without generating new samples while the latter can't. Deep, gradient-based methods, e.g., policy gradient, actor critic, are typically on-policy and thus, similar to most other DL tasks, require much data to train.

2.4 Related Topics

In the real world, the agent does not always learn to maximize the reward, and even the question of what the reward is has not been always clear. Several other fields in RL research focus on these problems:

1. Learning from demonstrations: let the agent watch a series of expert demonstrations and learn. Instead of just copying the expert's behavior, an exciting topic, **inverse reinforcement learning**, lets the agent infer the rewards based on these demonstrations.

2. Learning from observing the real world: in addition to limited demonstrations, the agent can also observe the real world and learn from it. This involves the use of **unsupervised learning**.

3. Learning from other tasks: a weakness of RL algorithms is that they could be over-specialized to a certain task. **Transfer learning** studies how a model that is already adept on a task can quickly become proficient on other related tasks. **Meta learning** studies how the agent can learn the ability to learn, i.e., to acquire an efficient learning style to tackle a group of problems.

These topics will also be covered in the following sections.

3 Model-free RL

In **model-free RL**, we typically don't assume that the transition function is known. We don't intend to model the underlying MDP and instead focus on improving the policy.

3.1 Policy Gradient

Policy gradient is a method that optimizes the policy using purely gradient-based methods.

3.1.1 Direct Differentiation

Recall that the RL objective is a function of θ which is now defined as:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t R(s_t, a_t) \right] = \mathbb{E}_{\tau \sim p_\theta(\tau)} R(\tau)$$

Which can be optimized by direct differentiation. First, express $J(\theta)$ using the law of expectations:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p_\theta(\tau)} R(\tau) \\ &= \int p_\theta(\tau) R(\tau) d\tau \end{aligned}$$

Next, compute the gradient,

$$\begin{aligned} \nabla_\theta J(\theta) &= \int \nabla_\theta p_\theta(\tau) R(\tau) d\tau \\ &= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) R(\tau) d\tau \\ &= \mathbb{E}_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p_\theta(\tau) R(\tau)] \end{aligned}$$

Note that in

$$\log p_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} [\log \pi_\theta(a_t \mid s_t) + \log p(s_{t+1} \mid s_t, a_t)]$$

only the middle term affects the gradient. Therefore,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right) R(\tau) \right]$$

To approximate this, we can simply run the policy:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \right) \left(\sum_{t=1}^{T} R(s_{i,t}, a_{i,t}) \right) \right]$$

Given the gradient, we perform a gradient descent step to improve the expected utility:

Algorithm 1 Policy Gradient

```
1: Estimate $\nabla_\theta J(\theta) \approx \nabla_\theta \log(\pi_\theta)(a \mid s) R(\tau)$
2: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
```

The above direct differentiation approach is the idea of the REINFORCE algorithm. The intuition behind the method is to make the probability of obtaining high-reward trajectories higher by improving θ and to make it less likely to obtain a low-reward trajectory. Notice that in the above gradient expression, if without the reward term, the equation would resemble that in a typical MLE case. This similarity suggests that policy gradient can be viewed as a weighted version of MLE, where observed samples have their probabilities increase or decrease based on their rewards. Thus, the policy gradient can be very efficiently computed using automatic differentiation frameworks. Since its derivation does not involve using the Markov property, policy gradient would work almost the same in a partially observed MDP.

A main problem with policy gradient is that it has high variance, i.e., its result is highly dependent on the samples. Oftentimes the amount of data is insufficient for preventing overfitting. To reduce variance, we can utilize the causality assumption, which states that the policy at a later time cannot affect itself at an earlier time. Formally,

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \right) \left(\sum_{t'=t}^{T} R(s_{i,t'}, a_{i,t'}) \right) \right]$$

Recall that the policy gradient would increase the probability of a trajectory if the reward is positive. In some cases when the sample rewards are far from zero-centered, this leads to bad results. To correct this, we can subtract from all sample rewards the sample mean to force it to be zero-centered.

This method is called **baselines**. When using baselines, we need not always subtract the mean, we could also select an optimal value by zeroing the derivative of the variance of the objective regarding that value.

3.1.2 Off-Policy Policy Gradient

The REINFORCE algorithm is an on-policy algorithm because we need to generate new samples to approximate the expectations by approaching $p_\theta(\tau)$. Imagine now we cannot simulate $p_\theta(\tau)$ but instead have trajectories sampled from another policy $\bar{p}(\tau)$. We can use **importance sampling** to compute the original objective:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} R(\tau) = \int \bar{p}(\tau) \frac{p_\theta(\tau)}{\bar{p}(\tau)} R(\tau) d\tau = \mathbb{E}_{\tau \sim \bar{p}(\tau)} \left[\frac{p_\theta(\tau)}{\bar{p}(\tau)} R(\tau) \right]$$

where

$$\frac{p_\theta(\tau)}{\bar{p}(\tau)} = \frac{\prod_{t=1}^{T} \pi_\theta(s_t, a_t)}{\prod_{t=1}^{T} \bar{\pi}(s_t, a_t)}$$

Thus, the gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\prod_{t=1}^{T} \frac{\pi_\theta(s_t, a_t)}{\bar{\pi}(s_t, a_t)} \right) \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right) R(\tau) \right]$$

which can be approximated similarly as in the on-policy case.

3.1.3 Natural Gradient

In policy gradient, choosing the step size α could be delicate because different parameters affect the gradient by different degrees. Instead of solving this problem using some involved optimization algorithms, we can just modify the original gradient descent step in policy gradient. Notice that the traditional gradient descent step can be formulated as an optimization step:

$$\theta \leftarrow \arg\max_{\theta'} (\theta' - \theta)^T \nabla_\theta J(\theta) \text{ s.t. } \|\theta' - \theta\|^2 \leq \epsilon$$

which iteratively optimizes in an "ϵ" ball in the parameter space. Instead, we want it to optimize in the **trust region** of the policy space, which is more natural since we are improving the policy at the end. Thus, we can modify the constraint:

$$\theta \leftarrow \arg\max_{\theta'} (\theta' - \theta)^T \nabla_\theta J(\theta) \text{ s.t. } D(\pi_{\theta'}, \pi_\theta) \leq \epsilon$$

where D is a divergence measure that is parameter-independent. A good choice would be the KL-divergence:

$$D_{\mathrm{KL}}(\pi_{\theta'} \| \pi_\theta) = \mathbb{E}_{\pi_{\theta'}}[\log \pi_\theta - \log \pi_{\theta'}] \approx (\pi_{\theta'} - \pi_\theta)^T \mathbf{F} (\pi_{\theta'} - \pi_\theta)$$

where $\mathbf{F}$ is the Fisher information matrix:

$$\mathbf{F} = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s)^T]$$

The gradient descent update step thus becomes

$$\theta \leftarrow \theta + \mathbf{F}^{-1} \nabla_\theta J(\theta)$$

3.2 Actor-Critic Algorithms

3.2.1 Improving Policy Gradient

One of the reasons the policy gradient has high variance is that the sum of the rewards term is not fixed in reality; instead, it is probabilistic on the trajectory sampled. That is,

$$\sum_{t=1}^{T} R(s_{i,t}, a_{i,t}) = \hat{Q}(i, t) \neq \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[R(s_{t'}, a_{t'} \mid s_t, a_t)]$$

In order to approximate the expectation instead of a single reward that might be due to random chance, we can apply the baselines. In 3.1, the baseline is a scalar value. Here, we can make the baseline dependent on the state, i.e., it reflects the average reward given a state under the policy. The baseline thus becomes the value function. Formally,

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \right) \left(\sum_{t=1}^{T} A^{\pi_\theta}(s_{i,t}, a_{i,t}) \right) \right]$$

where $A^{\pi_\theta}(s_{i,t}, a_{i,t})$ is called the **advantage function** and is the result after the baseline is applied:

$$A^{\pi_\theta}(s_{i,t}, a_{i,t}) = Q^{\pi_\theta}(s_{i,t}, a_{i,t}) - V^{\pi_\theta}(s_{i,t})$$

If we can approximate the advantage function, then the variance of the policy gradient could be significantly reduced. Recall that the Q-function (without discounting) is defined as

$$Q^{\pi}(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + V^{*}(s'))$$

where $\sum_{s'} T(s,a,s') = 1$. This can be approximated by

$$Q^{\pi}(s,a) \approx R(s,a,s') + V^{*}(s')$$

where s' is a single sample. This would, however, generate better result than policy gradient because although s' is a single sample, there is no further uncertainty involved with $V(s')$ (unlike in policy gradient where every future step in the trajectory space is completely probabilistic.) Using this approximation, we have

$$A^{\pi_\theta}(s_t,a_t) \approx R(s_t,a_t) + V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)$$

Thus we would only need to fit the value function, which can be achieved by supervised learning (regression) where we fit a model, usually a neural network with parameters ϕ, that turns input states into predictions of the value function. The "true" label y in this case is the single sample Monte Carlo estimate. We can even do better by setting the true labels using the approximation of the value function that is similar to the approximation of the Q-function introduced above.

$$y_{i,t} = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[R(s_{t'},a_{t'})] \approx R(s_{i,t},a_{i,t}) + V_{\phi}^{\pi}(s_{i,t+1})$$

where $a_{i,t}$ are single samples and $V^{\pi}(s_{i,t+1})$ is computed in previous iterations. This technique is referred to as the "bootstrapped" estimate.

Now that we can compute the advantage function, we can finally improve the policy gradient using this new baselined gradient. This algorithm is called the batch **Actor-Critic** algorithm. Its main difference from REINFORCE is that it requires computing the value functions during training instead of letting unscaled sample rewards dominate everything. It thus has lower variance, but is no longer unbiased.

A problem of using the bootstrapped estimate is that in the infinite-horizon case, the value function could grow infinitely large. The discount factor solves this problem:

$$y_{i,t} \approx R(s_{i,t}, a_{i,t}) + \gamma V_\phi^\pi(s_{i,t+1})$$

with the new estimated policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(\left(\sum_{t'=t}^{T} \gamma^{t'-t} R(s_{i,t'}, a_{i,t'}) \right) - V_\phi^\pi(s_{i,t}) \right)$$

with the latter baseline term estimated with bias.

3.2.2 Online Actor-Critic Algorithm

The online Actor-Critic algorithm is similar to above. It considers an infinite-horizon MDP with discounting and learns from each new action sampled. This algorithm is called Actor-Critic because it involves an "actor", $\pi_\theta(a \mid s)$, that samples actions. It also involves a "critic", $V^\pi(s)$, that evaluates actions and promote better ones. Note that all the V^π terms in the algorithm are not accurate; they are estimates from another model.

Algorithm 2 Online Actor-Critic Algorithm

```
1: Sample an action $a \sim \pi_\theta(a \mid s)$, get a tuple $(s, a, s', r)$
2: Update estimated $V_\phi^\pi$ using target $r + \gamma V_\phi^\pi(s')$
3: Evaluate $A^\pi(s, a) \approx R(s, a) + \gamma V^\pi(s') - V_\phi^\pi(s)$
4: $\nabla_\theta J(\theta) \approx \nabla_\theta \log(\pi_\theta)(a \mid s) A^\pi(s, a)$
5: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
```

The algorithm by itself only uses batch size of 1. If we use parallel workers, i.e., if we have multiple simulators of the agent that all updates the model parameters. This would resemble the SGD algorithm with mini-batches.

This algorithm, in mini-batch case, can be improved by introducing a **replay buffer** $\mathcal{R}$ that stores the sampled actions. If so, we would then extract a batch of actions from $\mathcal{R}$ in every iteration. In step 2, we would not fit the value function, but fit the Q-function:

$$\text{Estimate } Q_\phi^\pi \text{ with target } y_i = r_i + \gamma Q^\pi(s_i, a_i) \text{ for each } \underbrace{s_i, a_i}_{\text{not from } \mathcal{R}}$$

Then, we use Q values to estimate the policy gradient. Notice the equivalence between the Q value and the sampled reward.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \right) Q^\pi_\phi(s_{i,t}, a_{i,t}) \right]$$

3.2.3 Critic as Baselines

Recall that the Actor-Critic algorithm is not unbiased because the advantage function cannot be perfectly fit. Nevertheless, a variant of Actor-Critic could be worked out that retains the unbiased policy gradient. Again, we would only need to make sure the critics, which act as state-dependent baselines, have unbiased estimates. That is, again, we would need to have unbiased estimate of this expression:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(\left(\sum_{t'=t}^{T} \gamma^{t'-t} R(s_{i,t'}, a_{i,t'}) \right) - V^\pi_\phi(s_{i,t}) \right)$$

which is equivalent to having the advantage function

$$A^\pi(s_t, a_t) = \left(\sum_{t'=t}^{T} \gamma^{t'-t} R(s_{t'}, a_{t'}) \right) - V^\pi_\phi(s_t)$$

which is hard to estimate in Actor-Critic and is only feasible if we use single point Monte Carlo estimation. Instead, we may go even further from having state-dependent baselines: we can make the baselines state-action-dependent. That is:

$$A^\pi(s_t, a_t) = \left(\sum_{t'=t}^{T} \gamma^{t'-t} R(s_{t'}, a_{t'}) \right) - Q^\pi_\phi(s_t, a_t)$$

However, this advantage function results in an incorrect policy gradient! This is because the value function, as a baseline, would let the advantage function integrate to 0 in expectation (given the definition of value function as expectation over different actions.) The Q-function would not. Therefore, we need to re-derive the policy gradient to account to the error induced by the Q-function term:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(\left(\sum_{t'=t}^{T}\gamma^{t'-t}R(s_{i,t'}, a_{i,t'})the\right) - Q_\phi^\pi(s_{i,t}, a_{i,t})\right)$$
$$+\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \mathbb{E}_{a\sim\pi_\theta(a_t|s_{i,t})}[Q_\phi^\pi(s_{i,t,}, a_t)]$$

where the second term can be easily approximated. This produces the unbiased Actor-Critic variant called **control variates**.

Alternatively, in **n-step return**, we could adopt both the unbiased, single point Monte Carlo estimate and the Actor-Critic version of the advantage function and produce a new, combined one:

$$A_n^\pi(s_t, a_t) \approx \sum_{t'=t}^{t+n}\gamma^{t'-t}R(s_{t'}, a_{t'}) + \gamma^n V_\phi^\pi(s_{t+n}) - V_\phi^\pi(s_t)$$

where we can adjust the value of n. The larger it is, the more the critic looks into future (thus the higher the variance is.)

3.3 Value-based RL

In value-based RL, we omit the policy gradient completely. If we know $A^\pi(s_t, a_t)$ why not directly select $\arg\max_a A^\pi(s_t, a_t)$? Given the policy π, we can implement a better, deterministic policy π':

$$\pi'(s,a) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} A^\pi(s,a') \equiv \arg\max_{a'} Q^\pi(s,a') \\ 0 & \text{otherwise} \end{cases}$$

3.3.1 Fitted Q-Iteration and Q-Learning

We now revisit the policy iteration algorithm. A difference from the version in 1.2 is that we can fit not only the value function, but also the Q-function or the advantage function of a fixed policy. We then make the update shown above.

Algorithm 3 Policy Iteration

```
1: $\forall s, V^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')(R(s,\pi(s),s') + \gamma V^\pi(s'))$
2: Extract $\pi'$
3: $\pi \leftarrow \pi'$, go back to step 1
```

We can eliminate the policy improvement step and get the value iteration:

Algorithm 4 Value Iteration

```
1: Repeat $\forall s, V^{\pi}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')(R(s,\pi(s),s') + \gamma V^{\pi}(s'))$
2: Extract optimal $\pi$
```

In model-free RL, we typically don't know the transition probabilities; thus we cannot exactly fit the value function. Instead, we train a neural network to do this job. Note that $\mathcal{L}()$ denotes a regression loss function.

Algorithm 5 Fitted Value Iteration

```
1: Repeat the following; collect new data as needed
2: $\forall s_i, y_i \leftarrow \max_a \sum_{s'} T(s,\pi(s),s')(R(s,\pi(s),s') + \gamma V_{\phi}(s'))$
3: Set model parameter $\phi \leftarrow \arg\min_{\phi} \sum_i \mathcal{L}(V_{\phi}(s_i), y_i)$
```

Apparently, we need to know the outcomes of carrying out different actions for the first step of fitted value iteration. Recall that something we can estimate by sampling is the Q-values. Our samples are policy-independent, meaning that however the policy improves, our sampled distribution is always valid. (Thus the algorithm becomes off-policy.) Recall the Q-learning equation (which is just a definition of Q-values that we can fit through sampling:)

$$Q_{k+1}(s,a) \leftarrow \underbrace{\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q_k(s',a'))}_{\text{approximate through sampling}}$$

From this, we can fit Q-values without knowing the transition probabilities:

Algorithm 6 Fitted Q-Iteration

```
1: Repeat the following; collect new data as needed
2: $\forall s_i, y_i \leftarrow \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q_k(s',a'))$ (approximate)
3: Set model parameter $\phi \leftarrow \arg\min_{\phi} \sum_i \mathcal{L}(Q_{\phi}(s_i), y_i)$
```

where ϕ here is the parameter of a model that takes in a state-action pair and outputs the corresponding Q-value. If, ideally, $\mathcal{L}(Q_{\phi}(s_i), y_i)$ decreases to 0, we call the resulting set of Q-values an optimal Q-function. However, unlike in the tabular case, such optimality is not guaranteed when we use models.

The fitted Q-iteration algorithm can be very easily modified to become an online algorithm.

Algorithm 7 Online Q-Learning

```
Repeat the following; take action $a$ and observe the tuple $(s_i, a_i, s'_i, r_i)$
$\forall s_i, y_i \leftarrow \sum_{s'} T(s_i, a_i, s')(R(s_i, a_i, s') + \gamma \max_{a'} Q_k(s', a'))$ (approximate)
$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y)$
```

A problem about the naive Q-iteration and Q-learning introduced above is that in the first step, when we collect data from a fixed deterministic policy, we will get stuck at certain state-action pairs and will never sample certain state-action pairs, which might be better. This problem is solved by exploration as introduced in 2.2. For example, the ϵ-greedy strategy:

$$\pi(s, a) = \begin{cases} 1 - \epsilon & \text{if } a = \arg\max_{a'} Q^\pi_\phi(s, a') \\ \epsilon/(|A| - 1) & \text{otherwise} \end{cases}$$

Another set of commonly-used exploration rules is to weigh the actions by their Q-values. For example:

$$\pi(s, a) \propto \exp(Q_\phi(s, a))$$

which is called the **Boltzmann exploration** rule.

3.3.2 Value Function Learning Theory

This section slightly deviates from deep RL. First, we revisit theorem 1 and give it a somewhat more formal proof. Recall that in theorem 1, we state that the value iteration converges to the optimal policy.

Proof. Define an operator $\mathcal{B}$ (the Bellman operator) such that

$$\mathcal{B}V = \max_a R_a + \gamma T_a V$$

where R_a is the stacked vector of rewards at all states for action a, T_a is the transition probability matrix for action a, and V is the value function.

Notice that the optimal value function $V^*(s) = \max_a R(s,a) + \gamma \mathbb{E}[V^*(s')]$ is a fixed point of $\mathcal{B}$, that is, $V^* = \mathcal{B}V^*$.

We can prove that the value iteration converges to V^* because $\mathcal{B}$ is a **contraction** under max-norm. That is,

$$\forall V, \bar{V}, \|\mathcal{B}V - \mathcal{B}\bar{V}\|_\infty \leq \gamma \|V - \bar{V}\|_\infty$$

which intuitively means V and $\bar{V}$ becomes closer to each other when they are operated on $\mathcal{B}$. Here, if we set $\mathcal{B}\bar{V} = \mathcal{B}V^* = V^*$, the value iteration step just becomes $V \leftarrow \mathcal{B}V$. As explained, V would converge to V^*. □

Similar to the procedure in the proof above, the fitted value iteration (under square loss) can be expressed as

$$V' \leftarrow \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(s) - (\mathcal{B}V)(s)\|^2$$

where V is the old value function and Ω is the hypothesis class of the supervised learning model with parameter ϕ. Intuitively, we are trying to find a new value function $V' \in \Omega$ that is as close as possible to $\mathcal{B}V$. For convenience, we define another operator Π such that $\Pi V = \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(s) - (\mathcal{B}V)(s)\|^2$. Observe that Π is a projection onto Ω in l_2 norm. Π is also a contraction.

However, notice that Π and $\mathcal{B}$ are contractions defined under different norms, and that $\Pi\mathcal{B}$ would not be a contraction. This directly results in the conclusion that fitted value iteration is not guaranteed to converge. Similarly:

Theorem 5. *Fitted Q-iteration might not converge.*

Proof. We re-define the two operators for the Q-values:

$$\mathcal{B}Q = r + \gamma T \max_a Q$$

$$\Pi Q = \arg\min_{Q' \in \Omega} \frac{1}{2} \sum \|Q'(s,a) - Q(s,a)\|^2$$

Then the Q-iteration becomes $Q \leftarrow \Pi\mathcal{B}Q$. Since $\Pi\mathcal{B}$ is not a contraction, fitted Q-iteration might not converge. □

Similarly, online Q-learning is not guaranteed to converge. Although its update step looks like gradient descent and SGD always converges, Q-learning is actually not using SGD on a well-defined objective. This is because the target values y_i themselves depend on the fitted Q-values and thus are not guaranteed to be accurate.

Corollary 5.1. *The batch Actor-Critic algorithm might not converge.*

3.3.3 Deep RL with Q-Functions

Despite lacking theoretical guarantees on their convergence, Q-learning algorithms could still perform well in practical settings.

As mentioned above, the main reason that Q-learning would not converge is that the target values keep changing and are not accurate. This problem is more severe than it seems because in online Q-learning, samples would represent adjacent states in the MDP, thus are highly correlated with each other. This would lead to dangerous overfitting to the wrong target values. In practice, this problem is solved by introducing a replay buffer that is similar to the one in the online Actor-Critic algorithm. We periodically feed the reply buffer and in each iteration, we sample a batch of transitions from the buffer and update the Q-function.

4 Model-based RL

5 Exploration

6 Offline Learning

7 Related Topics

7.1 RL Theory

7.2 Tabular Methods

7.3 Linear Quadratic Regulator

7.4 Optimal Control and Planning

7.5 Variational Inference and Generative Methods

7.6 Inverse RL

7.7 Transfer and Multi-Task Learning

7.8 Meta Learning