Foundations of Deep Reinforcement Learning

Max De Jong

Abstract

These notes come from the “Foundations of Deep Reinforcement Learning” series by Pieter Abbeel.

Chapter 1

Lecture 1: MDPs Foundations and Exact Solution Methods

We will start with small problems with small, discrete state-action spaces to get the main ideas across. In future lectures, we will move beyond these exact solutions to consider larger state spaces.

1.1 Motivation

Deep reinforcement learning really gained momentum as a field in 2013, with the Deepmind result applying DQN to Atari games. Prior to this result, RL had mostly been restricted to small, toy problems. Following this groundbreaking work, the field exploded with many new results impacting fields ranging from locotion problems, robotics control, and mastering complex games. In all of these results, the agents performed their task not through hand-crafted routines but through learning from their own trial and error.

1.2 Markov Decision Processes (MDPs)

Reinforcement learning aims to solve problems within the Markov decision process framework. Within this framework, we have an agent and an environment. Our agents gets to choose an action a_t, and after choosing that action the environment will change to some state s_{t+1}. The agents observes this changed environment, gets to select another action, and so on. And with the current state of the environment, there is an associated reward r_t. The goal is for the agent to repeatedly interaction with its environment and figure out the optimal action for the environment to maximize its total reward. Note that in the MDP setting, we assume that the agent observes the entire state. There is an alternative, partially-observed MPDs, that relax this assumption, but this variant will not be our focus.

Formally, to define an MDP we require

- A set of states $\mathcal{S}$.
- A set of actions $\mathcal{A}$.
- A transition function $P(s'|s, a)$ between states of the system.
- A reward function $R(s, a, s')$.
- An initial state s_0.

- A discount factor γ to discount future rewards.
- A horizon H dictating how long the agent will be acting.

For a given MDP defined by $(\mathcal{S}, \mathcal{A}, P, R, s_0, \gamma, H)$, our goal is to find a policy π such that

$$\max_{\pi} E\left[\sum_{t=0}^{H} \gamma^t R(S_t, A_t, S_{t+1})|\pi\right]. \tag{1.1}$$

That is, we wish to maximize our expected rewards discounted over time. If we can map our problem of interest onto an MDP, we can hope to solve our problem within this framework.

To get a feel for problems within this framework, we will rely on a canonical example in this lecture: grid world. This simple environment will allow us to gain intuition about MPDs and reinforcement learning. Within grid world, we imagine

- An agent located at a particular grid cell.
- The actions of the agent include moving to any neighboring cell.
- There is one cell with a reward of +1 and one cell with a reward of -1 if reached.

Clearly, the goal is for our agent to navigate to the cell containing the positive reward. Mathematically, this is equivalent to finding a policy π satisfying

$$\max_{\pi} E\left[\sum_{t=0}^{H} \gamma^t R(S_t, A_t, S_{t+1})|\pi\right], \tag{1.2}$$

which is just the objective that we have already seen. Note that the dynamics within this world can be adjusted. Perhaps we consider a case with deterministic actions, in which the agent can move to the desired square deterministically. We could also consider the case of noisy actions, in which the agent moves to the desired cell with some fixed probability. In this way, we can probe both deterministic and stochastic systems with this simple toy example. In any case, because our world is so simple, we can very intuitively understand the behavior of the agent. And for this example, we can think of a policy of a map of actions for each grid cell. It is worth reflecting on the utility of the discount factor $\gamma \in [0, 1]$. For $\gamma \neq 1$, future rewards are discounted. This leads to favoring the shortest path to the grid cell with the positive reward, an important attribute of our desired policy solution.

1.3 Exact Solution Methods

1.3.1 Value Iteration

An optimal value function $V^*(s)$ is the maximum we can achieve in expected discounted sum of rewards if we are acting optimally. Mathematically, this is given by

$$V^*(s) = \max_{\pi} E\left[\sum_{t=0}^{H} \gamma^t R(s_t, a_t, s_{t+1})|\pi, s_0 = s\right] \tag{1.3}$$

for an initial state s_0. In our grid world example, this value function can be thought of as the best case scenario for our agent. If the actions are performed deterministically and without a discount factor, our agent can reach the desired cell, receiving a reward of +1, from all initial cells (assuming our H is large enough). However, if it starts in the cell with the reward of -1, the highest score achievable is -1. So we can write

$$V^*(s_x, s_y) = 1 - 2 \cdot \mathbb{1}_{(s_x,s_y),(x',y')} \tag{1.4}$$

for cell at coordinates (s_x, s_y) if the cell containing the negative reward is located (x', y').

The situation becomes more complicated if we add a discount factor of $\gamma = 0.9$. If we start at the cell with the reward, we receive the +1 reward without any discount, so we would have $V^* = 1$ for this state. What happens if we start one grid cell away from the reward? We can still reach the reward, but we must first step towards it. Since this requires an additional action, this cell would have $V^* = \gamma \cdot 1$. And if we start two cells away from the reward, we require two steps and thus have $V^* = \gamma^2$.

We can make the situation again more complicated if we add in a non-unity action success probability. Let's say that the agent moves in the desired direction with probability $p = 0.8$. Now we have more complex dynamics. If we start at the square with the reward, we immediately receive our reward and again have $V^* = 1$. If we start in the adjacent cell, however, we obviously want to move towards the reward. But this only happens if our action was successful, which occurs with probability p. So this leads to a contribution $V^* = p \cdot \gamma \cdot 1$. But there is also a 10% chance that the agent moves down or stays in place (due to the boundary conditions of the environment). To make this concrete, let's assume that the reward is located at square $(3, 4)$. The adjacent square then has

$$V^*(3, 3) = 0.8\gamma \cdot 1 + 0.1\gamma V^*(3, 3) + 0.1\gamma V^*(3, 2). \tag{1.5}$$

This is an interesting expression, because we have a recursive expression for the values of the neighboring cells. This is the key idea behind value iteration, which we will now explore more properly. For this more careful treatment, we will also index our value function by how many time steps still remain in the future. With $H = 0$, we will consider $V_0^*(s)$ as the optimal value for state s with $H = 0$. This is our initialization, and we trivially have $V_0^*(s) = 0 \; \forall \; s$. With one remaining time step, we can look at all actions available in our given state and then sum over all future states given our initial state, tabulating the discounted rewards. Mathematically, this is

$$V_1^*(s) = \max_a \sum_{s'} P(s'|s, a)\big[R(s, a, s') + \gamma V_0^*(s')\big], \tag{1.6}$$

knowing that we have $V_0^*(s') = 0$. This is the key idea of value iteration: we can decompose a problem with an arbitrary number of timesteps remaining into the immediate reward plus the value function with one fewer timestep remaining. In general, we can write

$$V_k^*(s) = \max_a \sum_{s'} P(s'|s, a)\big[R(s, a, s') + \gamma V_{k-1}^*(s')\big], \tag{1.7}$$

for a general value iteration.

We now have enough to write down the algorithm to perform value iteration (Algorithm 1). This update scheme is termed a "value update" or a "Bellan update." After enough steps, our estimates converge to the optimal value for every state in our system. This is guaranteed to converge according to a formal theorem, and at convergence we have an optimal value function V^* for the discounted infinite horizon problem satisfying the Bellman equation. Once we have V^*, we can easily form the optimal policy π^*. Also, note that the infinite horizon optimal policy is stationary, so that the optimal action depends only on s and not in time. This makes the policy efficient to store.

Algorithm 1: Value iteration algorithm

```
Initialize $V_0^*(s) = 0$ for all $s$;
for $k \leftarrow 1$ to $H$ do
    for $s \in \mathcal{S}$ do
        $V_k^*(s) \leftarrow \max_a \sum_{s'} P(s',|s,a)\big[R(s,a,s') + \gamma V_{k-1}^*(s')\big]$;
        $\pi_k^*(s) \leftarrow \text{argmax}_a \sum_{s'} P(s',|s,a)\big[R(s,a,s') + \gamma V_{k-1}^*(s')\big]$;
    end
end
```

We can form some intuition behind this convergence. We know that $V^*(s)$ is the expected sum of rewards accumulated starting with state$s$ acting optimally for ∞ timesteps. And $V_H^*(s)$ is the expected sum of rewards acting optimally for H steps. The additional reward collected during timesteps $H+1$, $H+2$, ... are given by

$$\gamma^{H+1}R(s_{H+1}) + \gamma^{H+2}R(s_{H+2}) + \dots \tag{1.8}$$

which we bound using $R(s) \leq R_{\max}$ to write

$$\begin{aligned}\gamma^{H+1}R(s_{H+1}) + \gamma^{H+2}R(s_{H+2}) + \dots &\leq (\gamma^{H+1} + \gamma^{H+2} + \dots)R_{\max} && (1.9)\\ &\leq \frac{\gamma^{H+1}}{1-\gamma}R_{\max} && (1.10)\end{aligned}$$

using the result for a geometric series. We see that this quantity decays as we increase our horizon, so that the difference between $V^*(s)$ and $V_H^*(s)$ for finite horizon H is bounded by a number the decreases with H. Therefore, we see that $V_H^* \to V^*$ as $H \to \infty$.

In addition to the value function, there is another important abstraction for thinking about these problems. We define $Q^*(s,a)$ as the expected utility of taking action a from state s and thereafter acting optimally. It is like a value function, but for a state-action pair rather than just a state. We can also form a Bellman equation for these Q-values:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\Big[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\Big] \tag{1.11}$$

Here we measure the expected reward for the first action in the first term and then the expected discounted reward for future state s' in the second term. Note that this second term is really equivalent to $V^*(s')$, just written in terms of Q-values. We similarly can perform Q-value iteration using this equation according to

$$Q_{k+1}^*(s,a) \leftarrow \sum_{s'} P(s'|s,a)\Big[R(s,a,s') + \gamma \max_{a'} Q_k^*(s',a')\Big] \tag{1.12}$$

where k indexes the Q-values. This too will converge to the optimal set of Q-values that we can use to determine a policy for our agent.

1.3.2 Policy Iteration

We just saw two flavors of value iteration (using V and Q) that allow us to converge to the optimal policies for small MDPs. So why do we need to consider another method? As we will see as we cover more advanced methods, some of these will build upon the value iteration approach and others will build upon a policy iteration approach. So we need to understand both of these methods for a complete understanding of the foundation of modern reinforcement learning.

Recall that in value iteration, we had an update rule according to

$$V^*_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V^*_k(s')\big]. \tag{1.13}$$

In policy evaluation, we fix the policy using a given $\pi(s)$. Practically, this means that we can no longer rely on taking a maximum over all of the available actions but instead take actions according to our policy $\pi(s)$. This means that our policy evaluation now takes the form

$$V^\pi_{k+1}(s) \leftarrow \sum_{s'} P(s'|s,\pi(s))\big[R(s,\pi(s),s') + \gamma V^*_k(s')\big]. \tag{1.14}$$

So if we are given some policy π, we can use it to run this policy evaluation to determine the updates on our value functions. And upon convergence, we will find the value for that specific policy, given by

$$V^\pi(s) \leftarrow \sum_{s'} P(s'|s,\pi(s))\big[R(s,\pi(s),s') + \gamma V^*(s')\big]. \tag{1.15}$$

Since this is just a special case of value iteration, we know it will converge. Note that our policy can also be stochastic, so that we have $\pi(a|s)$ rather than a deterministic policy $\pi(s)$. In this case, the update to perform policy evaluation is given by

$$V^\pi_{k+1}(s) \leftarrow \sum_{s'} \sum_a \pi(a|s) P(s'|s,a)\big[R(s,a,s') + \gamma V^\pi_k(s')\big] \tag{1.16}$$

Now that we can perform policy evaluation for both deterministic and stochastic policies, we can form an algorithm to perform policy iteration. The idea is that given a policy, we can evaluate it and then use that evaluation to improve our policy. This improvement in our policy is performed iteratively, leading to the name "policy iteration." For one of these iterations, the algorithm has two steps:

1. Policy evaluation for current policy π_k: Iteration until convergence

$$V^{\pi_k}_{i+1}(s) \leftarrow \sum_{s'} P(s'|s,\pi_k(s))\big[R(s,\pi(s),s') + \gamma V^{\pi_k}_i(s')\big] \tag{1.17}$$

2. Policy improvement: find the best action according to the one-step look-ahead

$$\pi_{k+1}(s) \leftarrow \text{argmax}_a \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V^{\pi_k}(s')\big] \tag{1.18}$$

   Here we are finding the action in state s that maximizes our immediate reward plus discounted future rewards using policy π_k.

We can repeat these iterations until convergence, and at convergence we have an optimal policy. This approach often converges in fewer iterations than value iteration. But this comes with a trade-off, since we have many small optimizations happening that are akin to value iteration within each loop of our policy iteration. So while the overall number of "outer" iterations might be fewer, there is more work on the inner iterations.

We know that policy iteration is guaranteed to converge and that it will converge so that both its policy and value function are the optimal policy and optimal value function. Let's try to form some intuition behind these guarantees. To sketch the proof, we look at both of these guarantees in turn.

1. Policy iteration is guaranteed to converge: the policy improves at every step. When we perform a one step look-ahead, we are choosing the best action now and then using the current policy. This is necessarily better than applying the current policy now and then the current policy from the next step. Thus, the policy does indeed improve at every step, and therefore a given policy can be encountered at most once. So after we have iterated many times, we have covered all possibile policies and hence have converged.

2. Optimality is guaranteed at convergence: by definition of convergence, we have $\pi_{k+1}(s) = \pi_k(s)$ for all states s. So in our policy improvement step, taking the argmax is equivalent to taking what is already prescribed by the policy. But this is just the value iteration equation. This means that V^{π_k} satisfies the Bellman equation, so that we are at the optimal V^*.

1.4 Maximum Entropy Formulation

To motivate this approach, let's think about finding a distribution over near-optimal solutions instead of a single optimal policy. In this case, we would have a more robust policy. If the environment changes, the distribution over near-optimal solutions may still have some good policies for the new situation. We would also have more robust learning. If we can retain a distribution over near-optimal solutions, our agent will collect more interesting exploratory data during learning. Data collection becomes more important as we move beyond small MDPs, so collecting more "interesting" data will help introduce more variation in the data collected, leading to better exploration and thus identification of optima.

With this motivation in place, let's define entropy. Physically, it is a measure of the uncertainty of a random variable. Specifically, it is the number of bits required to encode some random variable on average. For a random variable X, it is given mathematically by

$$\begin{aligned}\mathcal{H}(X) &= \sum_i p(x_i) \log_2 \frac{1}{p(x_i)} && (1.19)\\ &= -\sum_i p(x_i) \log_2 p(x_i) && (1.20)\end{aligned}$$

We can think of entropy as a measure of the variance associated with a sample from a distribution. With this in mind, let's bring entropy into our MDP framework. With the standard formulation, an MDP aims to find

$$\max_\pi E\left[\sum_{t=0}^{H} r_t\right] \tag{1.21}$$

With a maximum entropy approach, we instead maximize the sum of the rewards plus an extra term encoding the entropy of the policy. If we make the policy deterministic, the entropy will be zero. But if we instead maximize

$$\max_\pi E\left[\sum_{t=0}^{H} r_t + \beta\mathcal{H}(\pi(\cdot|s_t))\right], \tag{1.22}$$

we account for not just the reward but also the entropy of the policy in each state. This introduces a trade-off: if we can better control our actions, we can more precisely control the rewards achieved by the agent. The larger we make β, then, the more entropy we will have likely at the expense of the total accumulated rewards. For the reasons we mentioned with more advanced training schema, this trade-off can be worthwhile even if it comes with a decrease to our accumulated rewards.

To solve the maximum entropy MDP, we need to use constrained optimization. As a quick recap, a general constrained optimization problem aims to perform $\max_x f(x)$ such that $g(x) = 0$. To solve this problem, we typically form a Lagrangian

$$\mathcal{L}(s, \lambda) = f(x) + \lambda g(x) \tag{1.23}$$

Our constrained optimization problem can then be expressed as $\max_x \min_\lambda \mathcal{L}(x, \lambda)$. This Lagrangian is useful to us because at the optimum of the original problem, we must satisfy

$$\frac{\partial \mathcal{L}(x, \lambda)}{\partial x} = 0, \tag{1.24}$$
$$\frac{\partial \mathcal{L}(x, \lambda)}{\partial \lambda} = 0. \tag{1.25}$$

This provides us with a coupled set of equations that can be solved to find both x and λ. For a one-step maximum entropy problem, we want to calculate

$$\max_{\pi(a)} \{E[r(a)] + \beta \mathcal{H}(\pi(a))\} = \max_{\pi(a)} \left\{ \sum_a \pi(a) r(a) - \beta \sum_a \pi(a) \log \pi(a) \right\} \tag{1.26}$$

This becomes a constrained optimization problem since we require $\sum_a \pi(a) = 1$ since it represents a probability distribution. So the constrained optimization can be written as

$$\max_{\pi(a)} \min_{\lambda} \mathcal{L}(\pi(a), \lambda) = \sum_a \pi(a) r(a) - \beta \sum_a \pi(a) \log \pi(a) + \lambda \left(\sum_a \pi(a) - 1 \right) \tag{1.27}$$

Taking partial derivatives, we find

$$\pi(a) = \frac{1}{Z} e^{\beta^{-1} r(a)} \tag{1.28}$$

with normalization constant

$$Z = \sum_a e^{\beta^{-1} r(a)}. \tag{1.29}$$

This tells us exactly how actions with high reward receive high probability and actions with low reward are exponentially suppressed. And as $\beta \to 0$, our exponential approaches a delta function and our policy becomes deterministic. And as β increases, the rewards become more uniformly exponentiated and all of the actions receive a similar amount of probability mass. If we now plug in this expression for $\pi(a)$, we find

$$V = \beta \log \sum_a e^{\beta^{-1} r(a)}, \tag{1.30}$$

after some algebra, which just is a softmax function. So adding entropy changes the maximum that we previously saw into a softmax, with the sharpness determined by the entropy scaling β.

Finally, we can perform maximum entropy value iteration for more than one step. We now wish to find

$$\max_\pi E\left[\sum_{t=0}^{H} r_t + \beta\mathcal{H}(\pi(\cdot|s_t))\right] \tag{1.31}$$

with a value function given by

$$V_k(s) = \max_\pi E\left[\sum_{t=H-k}^{H} r(s_t, a_t) + \beta\mathcal{H}(\pi(a_t|s_t))\right] \tag{1.32}$$

For a one-step look ahead, we write

$$V_k(s) = \max_\pi E\big[r(s,a) + \beta\mathcal{H}(\pi(a|sJ) + V_{k-1}(s'))\big] \tag{1.33}$$

Now using

$$Q_k(s,a) = E\big[r(s,a) + V_{k-1}(s')\big] \tag{1.34}$$

we can write

$$V_k(s) = \max_\pi E[Q_k(s,a) + \beta\mathcal{H}(\pi(a|s))] \tag{1.35}$$

This now resembles the same problem that we saw before in the single step problem, just with r replaced by Q. So we know that the solution is given by

$$V_k(s) = \beta \log \sum_a e^{\beta^{-1}Q_k(s,a)} \tag{1.36}$$

and the policy is given by

$$\pi_k(a|s) = \frac{1}{Z}e^{\beta^{-1}Q_k(s,a)} \tag{1.37}$$

Chapter 2

Lecture 2: Deep Q-Learning

Previously, we looked at exact methods for solving MPDs. There are two main limitations with these exact methods:

1. We require access to a dynamics model
2. We require iterations over all possible states and actions (impractical for large MDPs)

To move beyond these limitations, we have solutions for each:

1. Using sampling-based approximations for dynamics
2. Fitting either Q or V function (and later policy fitting)

The remainder of these lectures will look at various strategies to accomplish these two solutions.

2.1 Q-Learning

Recall that $Q^*(s,a)$ is the expected utility of state s, taking action a, and thereafter acting optimally. These satisfy a Bellman equation:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{2.1}$$

and then Q-value iteration took the form

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]. \tag{2.2}$$

Using this equation assumes access to the transition model as well as the ability to iterate in our system. First, let's work on removing the necessity of access to a transition model. We can rewrite our expression as an expectation value:

$$Q_{k+1} \leftarrow E_{s'\sim P(s'|s,a)}\left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right] \tag{2.3}$$

Once we have an expectation, we can approximate using sampling. By replacing expectation by samples taken from the agent, we no longer require intimate knowledge of the dynamics model. This leads to tabular Q-learning:

1. For a state-action pair (s, a), we receive $s' \sim P(s'|s, a)$
2. We consider our old estimate $Q_k(s, a)$
3. And we consider our new sample estimate, given by

$$\text{target}(s') = R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \tag{2.4}$$

   If this single sample were fully representative, this target value would become our new value. But in general, this one-sample estimate will not be very precise.

4. So we will instead use a running average to incorporate our new estimate:

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha \cdot \text{target}(s') \tag{2.5}$$

   for some learning rate α.

This leads to the algorithm showin in Algorithm 2. This allows the agent to explore new state-action pairs and allows the Bellman updates using the sampled values.

Algorithm 2: Tabular Q-learning algorithm

```
Initialize $Q_0(s, a)$ for all $s$, $a$;
Get initial state $s$;
for $k \leftarrow 1$ to $N$ do
    Sample action $a$, get next state $s'$;
    if $s'$ is terminal then
        target $\leftarrow R(s, a, s')$;
        Reset agent;
        Sample new initial state $s'$;
    else
        target $\leftarrow R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$;
    end
    $Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha \cdot$ target;
    $s \leftarrow s'$;
end
```

While this algorithm is simple enough, we have not decided how new actions are chosen. We could choose random actions (zero greed). Alternatively, we could choose the action that maximizes $Q_k(s, a)$ (full greed). One popular approach is to be ϵ-greedy: a random action is chosen with probability ϵ, and a greedy action is chosen with probability $1 - \epsilon$. This allows exploration (the random action) while also performing exploitation (the greedy action). This permits learning of new state-action pairs and aids in converging. One amazing result of Q-learning is that it converges to an optimal policy, even if we are acting suboptimally. This is called "off-policy" learning, and the result that the optimal policy can be found while acting suboptimally is not trivial. However, this comes with a few caveates:

- We require enough exploration
- We need to anneal the learning rate α
- ... but not decrease it too quickly

Technically, these can be expressed precisely as

- All states and actions are visited infinitely often
- The learning rate schedule must obey

$$\sum_{t=0}^{\infty} \alpha_t(s,a) = \infty \tag{2.6}$$

so that there is enough "power" left to correct for poor samples that may mislead us but also that

$$\sum_{t=0}^{\infty} \alpha_t^2(s,a) < \infty \tag{2.7}$$

to bound the variance.

2.1.1 Deep Learning

So far, we have been dealing with small discrete environments. For grid world, for example, we have $\mathcal{O}(10^1)$ states, so storing $Q(s,a)$ is not problematic. But even in a simple game like Tetris, we have $\mathcal{O}(10^{60})$ unique states, and with an Atari game, we have $\mathcal{O}(10^{308})$ states. Clearly, storing a table for these games would not be practical. We run into similar issues with continuous environments crudely discretized. So we need an approach beyond relying on table storing all of the game information. One alternative to storing a tabular entry for each state is to instead work with a parameterized Q-function $Q_\theta(s,a)$. This approach is termed "approximate Q-learning," since we are now approximating the Q-values for a given state-action pair by some function parameterized by θ. By changing θ, we can represent different Q-functions. Traditionally, we could imagine that this takes the explicit form of a linear function in features $f_i(s,a)$:

$$Q_\theta(s,a) = \theta_0 f_0(s,a) + \theta_1 f_1(s,a) + \dots \tag{2.8}$$

Today, neural networks are a more popular representation for $Q_\theta(s,a)$. Now θ represents the weights of our neural network. Recall that in our Q-learning algorithm, we need to compute

$$\text{target}(s') = R(s,a,s') + \gamma \max_{a'} Q_{\theta_k}(s',a') \tag{2.9}$$

Here we can rely on a neural network to decide the action a' leading to the highest Q-value. We also need a way of updating our weights so that our neural network learns to better approximate the true Q-values. We apply a simple update rule such as

$$\theta_{k+1} \leftarrow \theta_k - \alpha \boldsymbol{\nabla}_\theta \left[\frac{1}{2}\big(Q_\theta(s,a) - \text{target}(s')\big)^2\right]\Bigg|_{\theta=\theta_k} \tag{2.10}$$

This is a loss function that penalizes our network from deviations between target(s') and $Q_\theta(s,a)$.

For a basic single layer perceptron, we have a linear function $f(x) = Wx$. For a multi-layer perceptron, we will stack linear functions followed by some non-linear activation function. For example, a two-layer network will look have an output

$$f(x) = W_1 \max(0, W_0 x) \tag{2.11}$$

using a basic ReLU activation. With enough hidden layers, we can use a neural network to approximate any function of interest. Of course, this requires some optimization, which is a non-convex problem. However, gradient-based methods have proven to be surprisingly effective. Minibatch stochastic gradients are often used rather than full gradients, and gradient calculation is handled by automatic differentiation and then used in backpropogation to update network weights. The most common optimiziers today include SGD with mommentum and preconditioning, which can take the form of RMSProp, Adam, Adamax, and others. All of these details are hidden by a modern package like `TensorFlow` or `PyTorch`.

2.2 Deep Q-Networks

With this brief recap of neural networks, let's apply them to our problem of Q-learning. As we mentioned, we will replace our table of Q-values with a parameterized $Q_\theta(s, a)$ that takes the form of a neural network. And we saw that each sample generated by the agent will generate a target target(s') that is mixed with the previous value for Q_θ. Our Q-network is then updated using a loss such as mean squared error. To avoid overfitting, we may batch our gradient updates.

2.2.1 Atari DQN

Let's start with the algorithm used in the 2013 Atari paper, shown in Algorithm 3. This has a few different features than we have seen before. First, it relies on a replay memory to store previous results. And second, it actually introduces a second Q-function (the target Q-function) that represents the same Q-function but is lagging from the other Q-function. As we will see, this helps to stabilize learning. Finally, the other major difference is that the state used in our MDP is different from the observation of the agent. The preprocessing ϕ maps an observation onto a state. In the case of the Atari games, this was a sequence of frames (observations) that we stacked to form a state that has more information (such as velocities).

Algorithm 3: Q-learning with experience replay

```
Initialize replay memory $D$ with capacity $N$;
Initialize function $Q$ with random weights $\theta$;
Initialize our target function $\hat{Q}$ with weights $\theta^- = \theta$;
for episode ← 1 to $M$ do
    Initialize sequence $s_1 = \{x_1\}$; Compute preprocessed sequence $\phi_1 = \phi(s_1)$;
    for $t \leftarrow 1$ to $T$ do
        With probability $\epsilon$ select random action $a_t$;
        Otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$;
        Execute action $a_t$, receiving reward $r_t$ and new image $x_{t+1}$;
        $s_{t+1} \leftarrow (s_t, a_t, x_{t+1})$;
        Preprocess: $\phi_{t+1} \leftarrow \phi(s_{t+1})$;
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$;
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from past experiences in $D$;
        Set target value
          $y_j \leftarrow \begin{cases} r_j & \text{episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{else} \end{cases}$;
        Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ wrt $\theta$;
        Every $C$ steps reset $\hat{Q} = Q$ for stabilization;
    end
end
```

One final detail of the original DQN algorithm is the use of Huber loss rather than squared loss when computing the Bellman error. This is defined as a quadratic loss function for small deviations and a linear loss for large deviations. This prevents any single target from causing too large of an update in the neural network weights and makes the network more robust to outliers.

2.2.2 Modern DQN Improvements

While the DQN used by Atari was able to achieve very impressive results, it has benefitted from a number of improvements since its inception. The first is this idea of double DQN. With a single DQN, we introduce a bias in using $\max_a Q(s, a; \theta)$, since this will overestimate our targets. Since we already have two Q-networks, we can reduce the bias using Q_θ for selecting the best action but Q_{θ^-} for evaluating the best action. This introduces some independence between how the action is chosen and how it is evaluated. In terms of our loss function, it will now take the form

$$L_i(\theta_i) = E_{(s,a,s',r)\sim D}\big(r + \gamma Q(s', \operatorname{argmax}_{a'} Q(s', a'; \theta); \theta_i^-) - Q(s, a; \theta_i)\big)^2 \tag{2.12}$$

Empirically, this leads to much faster learning.

Another important idea comes from using prioritized experience replay. Recall that we have some buffer storing all of our past experiences. Replaying all transitions with equal probability is highly suboptimal, since not all data is equally valuable. The idea with prioritized experience replay is to keep track of the Bellman error. By keeping track of the difference between the observed and predicted target values, we can give higher priority to these samples that we can better learn from. This too has been shown to lead to much faster learning in practice.

There are many other improvements in addition to these two, including dueling DQNs, distribution DQN (in which the reward distribution rather than just the expected value is learned), and noisy DQN (which introduces more randomness in the actions). There is a famous paper named "Rainbow DQN" that combines all of these modern improvements, and this is a natural place to start when using any sort of DQN today.

Chapter 3

Lecture 3: Policy Gradients, Advantage Estimation

3.1 Why Policy Optimization

In this lecture, we will look at using both policy gradients and advantage estimation to solve large MDPs. A valid initial question is why we would bother with these methods after we have learned Q-learning. In the current deep reinforcement learning landscape, there are multiple types of methods that come with their own preferred use cases. For DQN methods, for instance, we have very data-efficient methods but often are less stable than alternative approaches. If data efficiency is not our bottleneck (for instance, because we have access to a fast simulator), we might want to choose to have more simulation than Q-learning updates. When our data can be collected quickly, "on-policy" methods may be preferred. In this class of methods, $Q(s, a)$ is learned from actions taken using the current policy $\pi(a|s)$. These can be faster in terms of wall-clock time when simulation is cheap.

First, to orient ourselves, let's return to the MDP picture of an agent making actions in an environment. With policy methods, our agent is really a policy $\pi_\theta(a|s)$. For us, this policy will be given by a neural network that will match a state to an action. Our goal is to learn the proper weights of the network to maximize our expected reward. Mathematically, we wish to find

$$\max_\theta E\left[\sum_{t=0}^{H} R(s_t|\pi_\theta)\right] \tag{3.1}$$

Typically, this network will output a distribution over actions. This means that we have a stochastic policy class, and this helps to smooth out the optimization landscape. If our policies are deterministic, there is no continuum between two candidate policies. This makes the optimization more challenging. By adding in a stochastic interpolation between these two policies, we can arrive at a smoother optimization. In addition, the data collected by the agent requires some exploration. Stochasticity can help with this exploration. So our network will represent $\pi(a|s)$, the probability of action a in state s.

We have not yet addressed why we perform policy optimization. In many cases, π can be simpler than either Q or V. For example, consider a robotics problem in which we want a hand to grasp an object. We want the hand to move towards some object and close the mechanical fingers to pick up an object. Without a quantitative knowledge of the time required to perform this action or a metric to evaluate grasp quality, thinking of Q or V may be challenging. But thinking of the correct strategy may be more simple and thus faster to learn. There is also an inherent shortcoming in V: it does not prescribe actions. We need a dynamics model of the world (provided or learned) to compute Bellman updates. While a Q function overcomes this limitation, recall that we need to be able to efficiently solve $\operatorname{argmax}_a Q_\theta(s,a)$. This becomes challenging for action spaces that are continuous or high-dimensional. This becomes an optimization problem of itself. If we had a policy, however, we could quickly read off the action for a given state.

3.2 Policy Gradient Derivation

To compute our policy gradients, we will work with the likelihood ratio. First, let's define τ to denote a state-action sequence $(s_0, a_0), \ldots, (s_H, a_H)$. We then define

$$R(\tau) = \sum_{t=0}^{H} R(s_t, a_t), \tag{3.2}$$

so that our objective can be expressed as

$$\begin{aligned} U(\theta) &= E_{\pi_\theta}\left[\sum_{t=0}^{H} R(s_t, a_t)\right] && (3.3) \\ &= \sum_{\tau} P(\tau;\theta) R(\tau) && (3.4) \end{aligned}$$

And our goal is obviously to find θ satisfying $\max_\theta U(\theta)$. We again begin with gradient-based optimization, so that we want to calculate

$$\begin{aligned} \nabla_\theta U(\theta) &= \nabla_\theta \sum_{\tau} P(\tau;\theta) R(\tau) && (3.5) \\ &= \sum_{\tau} R(\tau) \nabla_\theta P(\tau;\theta) && (3.6) \end{aligned}$$

This is no longer a weighted sum, so let's do a trick to fix this:

$$\begin{aligned} &= \sum_{\tau} \frac{P(\tau;\theta)}{P(\tau;\theta)} R(\tau) \nabla_\theta P(\tau;\theta) && (3.7) \\ &= \sum_{\tau} P(\tau;\theta) \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} R(\tau) && (3.8) \end{aligned}$$

This now permits a sample-based method, since we have expressed this as an expectation again. To simplify our result, we can write

$$\nabla_\theta U(\theta) = \sum_{\tau} P(\tau;\theta) \nabla_\theta \log P(\tau;\theta) R(\tau) \tag{3.9}$$

which can be empirically estimated by sampling m paths under policy θ according to

$$\begin{aligned} \nabla_\theta U(\theta) &\simeq \hat{g} && (3.10) \\ &= \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P\left(\tau^{(i)};\theta\right) R\left(\tau^{(i)}\right). && (3.11) \end{aligned}$$

In practice, this means that we can use our current parameter values to perform a large number of roll-outs. For each roll-out, we compute the gradient of the probability under the current policy multiplied by the rewards collected along that trajectory. This likelihood ratio gradient that we have formed is very interesting for two reason. First, it is valid even if R is discontinuous or even unknown, since we aren't taking derivatives with respect to the reward function. Second, it is valid even when our sampled space of paths is a discrete set.

Let's look at a little intuition behind this equation. The likelihood ratio gradient tries to increase the probability of paths with a positive R and decrease the probability of paths with negative R. We are effectively trying to push up the probability of trajectories with a large R and shift probability mass away from trajectories with a negative R.

3.3 Temporal Decomposition

So far, we have thought about entire trajectories through our state-action space. Often times, though, rewards are more localized. This is a reason to not perform a shift in probability mass in terms of an entire path through our state-action space. For this reason, let's decompose our path into both states and actions. We can write

$$\boldsymbol{\nabla}_\theta \log P(\tau^{(i)};\theta) = \boldsymbol{\nabla}_\theta \log \left[\prod_{t=0}^{H} \underbrace{P\left(s_{t+1}^{(i)} | s_t^{(i)}, a_t^{(i)}\right)}_{\text{dynamics model}} \cdot \underbrace{\pi_\theta\left(a_t^{(i)} | s_t^{(i)}\right)}_{\text{policy}} \right] \tag{3.12}$$

Using the properties of logarithms, we can instead write

$$\boldsymbol{\nabla}_\theta \log P(\tau^{(i)};\theta) = \boldsymbol{\nabla}_\theta \left[\sum_{t=0}^{H} \log P\left(s_{t+1}^{(i)} | s_t^{(i)}, a_t^{(i)}\right) + \sum_{t=0}^{H} \log \pi_\theta\left(a_t^{(i)} | s_t^{(i)}\right) \right] \tag{3.13}$$

Since the first term has no dependence on θ, this becomes

$$\boldsymbol{\nabla}_\theta \log P(\tau^{(i)};\theta) = \sum_{t=0}^{H} \boldsymbol{\nabla}_\theta \log \pi_\theta\left(a_t^{(i)} | s_t^{(i)}\right). \tag{3.14}$$

Intuitively, since the model dynamics aren't impacted by θ, we do not need them to take this gradient. This is immensely helpful, since we dont' require any dynamics model to compute this gradient. Since the path $\tau^{(i)}$ consists of both the dynamics and policy, being able to increase the probability of a given trajectory without access to the dynamics model is not trivial.

Now let's orient ourselves to the problem at hand again. We found that we can compute an unbiased estimate of the gradient of our utility function using

$$\hat{g} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\nabla}_\theta \log P(\tau^{(i)};\theta) R\left(\tau^{(i)}\right) \tag{3.15}$$

and we just found that we can write

$$\boldsymbol{\nabla}_\theta \log P(\tau^{(i)};\theta) = \sum_{t=0}^{H} \boldsymbol{\nabla}_\theta \log \pi_\theta\left(a_t^{(i)} | s_t^{(i)}\right), \tag{3.16}$$

so that no dynamics model is required. In some sense, we could be finished here—we can apply roll-outs of our current policy, collect the rewards along the trajectory, and perform backpropogation by computing the grad log probability of the action given the state.

However, as we have formulated our approach thus far, out estimator would be unbiased but very noisy. Recall that for an unbiased estimator, we require that

$$E[\hat{g}] = \boldsymbol{\nabla}_\theta U(\theta). \tag{3.17}$$

In the real-world, we want reduce some of this sampling noise to have a practical approach. Possible fixes include

- Adding a baseline subtraction
- Incorporating temporal structure
- Calculating a trust region / natural gradient (next lecture)

3.4 Baseline Subtraction

To understand the utility of baseline subtraction, let's return to the intuition behind calculating

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{\nabla}_\theta \log P(\tau^{(i)};\theta) R\Big(\tau^{(i)}\Big). \tag{3.18}$$

As we saw, this would increase the probability of trajectories with a large reward and decrease the probability of trajectories with a negative reward. And of course, there is a little additional subtlety here, since the probability can simply be increased for one trajectory without renormalizing. If we think about what we really want, though, we would like to increase the probability not of trajectories with $R^{(i)} > 0$ but those with $R^{(i)} > \langle R \rangle$ (and likewise decrease probability for trajectories with $R^{(i)} < \langle R \rangle$). We would require a lot of averaging effects in the naive formulation for trajectories with large rewards to increase over those with smaller positive rewards.

This motivates thinking about adding in some baseline b. With this in place, we could write

$$\boldsymbol{\nabla}_\theta U(\theta) \simeq \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{\nabla}_\theta \log P(\tau^{(i)};\theta) \Big[R\Big(\tau^{(i)}\Big) - b\Big] \tag{3.19}$$

We find that this modified expression for $\hat{g}$ is still an unbiased estimator of $\boldsymbol{\nabla} U(\theta)$. This is because

$$E[\boldsymbol{\nabla}_\theta \log P(\tau;\theta) b] = \sum_\tau P(\tau;\theta) \boldsymbol{\nabla}_\theta \log P(\tau;\theta) b \tag{3.20}$$

$$= \sum_\tau \boldsymbol{\nabla}_\theta P(\tau;\theta) b \tag{3.21}$$

$$= b \boldsymbol{\nabla}_\theta \left(\sum_\tau P(\tau;\theta) \right) \tag{3.22}$$

as long as our baseline doesn't depend on the action occurring in the log probability in our action. Since we require $\sum_\tau P(\tau) = 1$, this simplifies to

$$E[\boldsymbol{\nabla}_\theta \log P(\tau;\theta) b] = b \times 0, \tag{3.23}$$

proving that our estimator is still unbiased upon adding in a baseline b. However, although adding in b does not affect the estimate in expectation, the estimate that we accumulate benefits from a reduction in variance with finite samples. This allows for a better gradient estimate with the proper choice of baseline.

3.5 Value Function Estimation

We already performed a temporal decomposition for the trajectory probability $P(\tau^{(i)};\theta)$, which resulted in the grad log probability for our policy given the state. But our rewards also have a temporal element, so we can perform a similar temporal decomposition of our reward. We can write

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P\left(\tau^{(i)};\theta\right)\left[R\left(\tau^{(i)}\right) - b\right] \tag{3.24}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(\sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right)\left(\sum_{t=0}^{H-1} R\left(s_t^{(i)}, a_t^{(i)}\right) - b\right) \tag{3.25}$$

We can now split up our reward sum into two parts to write

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\left(\sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right)\left[\sum_{k=0}^{t-1} R\left(s_k^{(i)}, a_k^{(i)}\right) + \sum_{k=t}^{H-1} R\left(s_k^{(i)}, a_k^{(i)}\right) - b\right] \tag{3.26}$$

Now when we think about grad log probability of an action given a state, would it be reasonable for it to be multiplied with a reward from the past? Since actions we take now can only influence the future, the answer to this question is no. This is the motivation behind splitting our reward sum above, since the first sum corresponds to rewards from the past for a given value of t and the second sum corresponds to rewards from the future. Intuitively, we know that rewards from the past should not appear in our estimate, and one can show with careful that the expected value of including past rewards is zero.

By removing terms that don't depend on our current action, we can lower the variance associated with our estimator. Then, we are left with

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\left(\sum_{k=t}^{H-1} R(s_k^{(i)}, a_k^{(i)}) - b(s_t^{(i)})\right) \tag{3.27}$$

Additionally, we know that our baseline b cannot depend on the action $a_t^{(i)}$ in order for our estimate to remain unbiased, but it is ok to depend on our state $s_t^{(i)}$. So we can safely use rewards for a given state accumulated from some time onwards as a meaningful baseline to be subtracted.

We now have a practical expression to perform policy gradients. We can perform m roll-outs, each that contains a number of different steps. We accumulate the grad log probability of the action we took for a given state and multiply this by the reward achieved from then onwards minus the baseline. We then shift probability mass for each action specifically according to whether it lead to above or below average rewards from that state moving forward.

So far, we have been loosely referring to "average" when talking about b. There are a number of choices that researchers use

- Constant baseline

$$b = E[R(\tau)] \tag{3.28}$$

$$\simeq \frac{1}{m}\sum_{i=1}^{m} R(\tau^{(i)}) \tag{3.29}$$

- Optimal constant baseline

$$b = \frac{\sum_i \left(\nabla_\theta \log P(\tau^{(i)};\theta)\right)^2 R(\tau^{(i)})}{\left(\nabla_\theta \log P(\tau^{(i)};\theta)\right)^2}, \tag{3.30}$$

  which can be shown to lead to the minimal variance through a formal derivation. However, this is not commonly used in practice.

- Time-dependent baseline

$$b_t = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=t}^{H-1} R(s_k^{(i)}, a_k^{(i)}), \tag{3.31}$$

  which have become very popular recently. This nicely captures the fact that there may be less reward remaining at some time for an environment with a finite horizon.

- State-dependent expected return

$$b(s_t) = E[r_t + r_{t+1} + \ldots + r_{H-1}], \tag{3.32}$$

  which is really just equivalent to the value function $V^\pi(s_t)$ that we saw in Lecture 1. This offers the highest degree of precision since it depends on the specific state and time, although estimating this becomes almost a separate problem (as we will see).

3.6 Value Function Estimation

Let's move forward with this final choice of baseline,

$$b(s_k^{(i)}) = V^\pi(s_k^{(i)}), \tag{3.33}$$

which requires us to estimate the value function.

3.6.1 Monte Carlo Estimation

One way to estimate this value function is through Monte Carlo estimation:

1. Initialize $V_{\phi_0}^\pi$
2. Collect trajectories $\{\tau_1, \ldots, \tau_m\}$
3. Regress against the empirical mean:

$$\phi_{i+1} \leftarrow \text{argmin}_\phi \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\left(V_\phi(s_t^{(i)}) - \sum_{k=t}^{H-1} R\Big(s_k^{(i)}, a_k^{(i)}\Big)\right)^2 \tag{3.34}$$

This essentially becomes a supervised learning problem, in which we collect the true accumulated rewards and try to minimize the deviation from those values. This is likely the first thing one would do in practice, since it is easy and straightforward to implement.

3.6.2 Bootstrap Estimation

Alternatively, we could also try to bootstrap estimate V^π. Recall that the value function satisfies the Bellman equation

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V^\pi(s')\big]. \tag{3.35}$$

This is something that we solved exactly in Lecture 1, but we can also perform this in an approximation. In this case, our routine would look something like

1. Initialize $V^\pi_{\phi_0}$
2. Collect data (s, a, s', r) from our roll-outs
3. Perform fitting value iteration

$$\phi_{i+1} \leftarrow \min_\phi \sum_{s,a,s',r} \left\| r + V^\pi_{\phi_i}(s') - V_\phi(s) \right\|_2^2 + \lambda \|\phi - \phi_i\|_2^2 \tag{3.36}$$

where we have added some regularization on the network parameters ϕ.

3.6.3 Policy Gradient

With either choice of estimation, we can now form a complete algorithm to perform vanilla policy gradient (Algorithm 4). Note that the bootstrap estimates are typically seen as being more sample efficient but less stable. So in practice, the recommendation would be to start with Monte Carlo estimation, and then you can try the bootstrap version and see if that improves the learning.

Algorithm 4: Vanilla policy gradient

```
Initialize policy network with weights $\theta$;
Initialize baseline $b$;
for iteration $\leftarrow 1$ to $N$ do
    Collect a set of trajectories by running the current policy;
    for timestep $t$ in trajectory do
        Calculate the return $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}$;
        Calculate the advantage estimate $\hat{A}_t = R_t - b(s_t)$;
    end
    Re-fit the baseline by minimizing $\|b(s_t) - R_t\|_2^2$, summed over all trajectories and timesteps;
    Update the policy using the policy estimate $\hat{g}$, calculated by summing $\nabla_\theta \log \pi(a_t|s_t;\theta)\hat{A}_t$;
end
```

3.7 Advantage Estimation (A2C/A3C and GAE)

We will finally look at some methods for making the advantage estimation more efficient. Recall that our full likelihood ratio policy gradient estimator is given by

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta\Big(a_t^{(i)}|s_t^{(i)}\Big)\left(\sum_{k=t}^{H-1} R(s_k^{(i)}, a_k^{(i)}) - V^\pi(s_k^{(i)})\right) \tag{3.37}$$

Our first goal will be to improve this by doing something better than a single sample estimate for $R(s_k^{(i)}, a_k^{(i)})$. If we were to estimate Q for a single roll-out, we would look at

$$Q^{\pi}(s,a) = E[r_0 + r_1 + \dots | s_0 = s, a_0 = a], \tag{3.38}$$

which will have a high variance per sample and poor generalization. One we to improve this is to reduce the variance by discounting, so that we instead estimate

$$Q^{\pi,\gamma}(s,a) = E\big[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots | s_0 = s, a_0 = a\big]. \tag{3.39}$$

At first, this seems a little weird, since we have already stated in our problem definition that an MDP is defined by a discount factor. It turns out that algorithmically, treating this hyperparameter as tuneable can help improve our estimation. If we think about what we are doing, we are trying to evaluate how good an action is by looking at the grad log probability of a given state. Perhaps that action has more influence to nearby actions, motivating a temporal weighting incorporated by a discount factor. So now the discount factor is not motivated by the economics of our solution, as it was in the formulation of the general MDP problem, but instead to represent the idea that actions have more influence to nearby states for many problems of interest. This is essentially putting in a prior that the effect of actions should decay with time to get a lower variance estimate of our policy gradients. Of course, our baseline will also require a similar discount factor to allow comparison.

And another way to reduce the variance is to use function approximation. Sticking with the discounted rewards idea, we wish to calculate

$$\begin{aligned} Q^{\pi,\gamma}(s,a) &= E\big[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots | s_0 = s, a_0 = a\big] &\quad (3.40)\\ &= E[r_0 + \gamma V^{\pi}(s_1) | s_0 = s, a_0 = a], &\quad (3.41)\end{aligned}$$

where we use a one-step look-ahead. Of course, this choice is arbitrary, and we could instead use something like

$$Q^{\pi,\gamma}(s,a) = E\big[r_0 + +\gamma r_1 + \gamma^2 V^{\pi}(s_2) | s_0 = s, a_0 = a\big], \tag{3.42}$$

or

$$Q^{\pi,\gamma}(s,a) = E\big[r_0 + +\gamma r_1 + \gamma^2 r_2 + \gamma^3 V^{\pi}(s_3) | s_0 = s, a_0 = a\big]. \tag{3.43}$$

There are many such options that would be reasonable here. This is the idea behind the Asynchronous Advantage Actor Critic (A3C) algorithm, in which $\hat{Q}$ uses one of the above choices. The number of steps to look ahead becomes a hyperparameter, and the original paper relied largely on $k = 5$ steps to look ahead.

Why would we want to use this mixed approach? If we use a $k = 0$ approach (and don't rely on a value function estimation at all), we know that this is exact. Once we start relying on value estimates, we may be introducing some error. But the benefit is that we are reducing variance, since it is an estimate based on many past experiences. So there is essentially a trade-off between a zero bias estimate ($k = 0$) with high variance to a choice with very low variance ($k = 1$, for instance) but with very high bias. Somewhere between these two extremes, we likely find the optimal estimation of the advantage for maximally fast learning.

The Generalized Advantage Estimation (GAE) algorithm works similarly but uses a lambda exponentially weighted average of all possible values of k in forming $\hat{Q}$. For some problems, this can lead to better results than instead relying on one specific choice of k. This idea is closely related to TD lambda and eligibility traces developed by Sutton and Barton long ago.

Regardless of our choice of forming $\hat{Q}$, it is worth writing down the full algorithm for our modified policy gradient approach. In Algorithm 5, we see how we can leverage both a policy and value network for our policy gradients. Note that sometimes these are two separate networks, and sometimes this may be a single network with two different heads. In addition to the choice between how $\hat{Q}_i$ is formed, there is also flexibility in the choice in the number of steps to look-ahead in the value function. For example, we could use a one-step look ahead update for V, given by

$$\phi_{i+1} \leftarrow \min_{\phi} \sum_{(s,a,s',r)} \left\| r + V^{\pi}_{\phi_i}(s') - V^{\pi}_{\phi}(s) \right\|_2^2 + \kappa \|\phi - \phi_i\|_2^2, \tag{3.44}$$

and a full roll-out for π:

$$\theta_{i+1} \leftarrow \theta_i + \alpha \frac{1}{m} \sum_{k=1}^{m} \sum_{t=0}^{H-1} \boldsymbol{\nabla}_\theta \log \pi_{\theta_i}(a_t^{(k)} | s_t^{(k)}) \left(\sum_{t'=t}^{H-1} r_{t'}^{(k)} - V^{\pi}_{\phi_i}(s_t^{(k)}) \right) \tag{3.45}$$

to remove the Monte Carlo estimates. We have a large number of these variations, but we have covered the main intuition behind the policy gradient algorithms. Once they begin to use sophisticated advantages, they fall into the name "actor-critic" algorithms, where we have both an actor (policy network) and critic (value network).

Algorithm 5: A3C/GAE policy gradient

```
Initialize policy network with weights $\theta_0$;
Initialize value network for current policy with weights $\phi_0$;
Collect roll-outs $(s, a, s', r)$ and form estimates of $\hat{Q}_i(s, a)$ from roll-outs;
Update the value function using $\phi_{i+1} \leftarrow \min_\phi \sum_{(s,a,s',r)} \left\| \hat{Q}_i(s,a) - V^{\pi}_{\phi}(s) \right\|_2^2 + \kappa \|\phi - \phi_i\|_2^2$;
Update the policy network using
  $\theta_{i+1} \leftarrow \theta_i + \alpha \frac{1}{m} \sum_{k=1}^{m} \sum_{t=0}^{H-1} \boldsymbol{\nabla}_\theta \log \pi_{\theta_i}(a_t^{(k)} | s_t^{(k)}) \left( \hat{Q}_i(s_t^{(k)}, a_t^{(k)}) - V^{\pi}_{\phi_i}(s_t^{(k)}) \right)$;
```

Chapter 4

Lecture 4: TRPO, PPO

4.1 Surrogate Loss

We are going to rederive the policy gradient equation starting from importance sampling. We have a utility

$$U(\theta) = E_{\tau \sim \theta_{\text{old}}} \left[\frac{P(\tau|\theta)}{P(\tau|\theta_{\text{old}})} R(\tau) \right] \tag{4.1}$$

so that the gradient is given by

$$\boldsymbol{\nabla}_\theta U(\theta) = E_{\tau \sim \theta_{\text{old}}} \left[\boldsymbol{\nabla}_\theta \frac{P(\tau|\theta)}{P(\tau|\theta_{\text{old}})} R(\tau) \right]. \tag{4.2}$$

This allows us to collect data from our old policy and use this expectation to improve θ.

Note that this equation holds true for any value of θ. If we choose to look at $\boldsymbol{\nabla}_\theta U(\theta)|_{\theta=\theta_{\text{old}}}$, this reduces to a standard policy gradient. But it is a valid equation for any choice of θ. When we have θ close to θ_{old}, the policy used to collect the data will be close to the new policy, so that the data can be used efficiently. However, if θ is very far from θ_{old}, we likely have considerable variance to deal with.

If we look at our loss again, we see that we are taking steps to minimize our loss, and the gradient allows us to take a first-order approximation for our loss function. But what if we moved beyond a first-order approximation? We will refer to

$$U(\theta) = E_{\tau \sim \theta_{\text{old}}} \left[\frac{P(\tau|\theta)}{P(\tau|\theta_{\text{old}})} R(\tau) \right] \tag{4.3}$$

as our "surrogate loss," and we will look at doing something more advanced than just taking a first-order approximation using the gradient.

4.2 Step-Sizing and Trust Region Policy Optimization

With this loss function in mind, let's think about our choice of step size. The gradient only gives us a first-order approximation that is good locally, so we don't want to step infinitely far in that direction. How can we go about choosing a good step size? Recall in supervised learning, if we step too far, we can count on the next update to correct for that mistake. So terrible step sizes are still an issue, but not so great step sizes are not really important since the data will provide a correction. In reinforcement learning, however, a bad step size can result in a terrible policy, and this terrible policy will give us terrible data. Then we cannot depend on the data gathered by this policy to provide a correction. In the worst case scenario, this new data is no longer informative and all of our learning is lost.

Ideally, we would have some way of estimating a good step size so that our agent is always collecting good data and the policy is constantly improving. A simple step-sizing approach would be to perform a line search in the direction of the gradient. This would look like taking a certain step size, performing a number of roll-outs, and then evaluating these roll-outs. Although this approach is simple, it is very expensive since we require many evaluations along this line. This method is a bit naive, since it ignores information about where the approximation is good or bad.

Instead of taking gradient steps, there is an alternative approach named trust region policy optimization using our surrogate loss. Instead of just computing an objective from this, we will set up a loss function and optimize this loss according to

$$\max_{\pi} L(\pi) = E_{\pi_{\text{old}}}\left[\frac{\pi(a|s)}{\pi_{\text{old}}(a|s)} A^{\pi_{\text{old}}}(s,a)\right], \tag{4.4}$$

where the advantage is estimated from data collected under π_{old}. Because this advantage is estimated under the old policy, we need to be careful while optimizing our loss that we don't move too far from the old policy, or else the estimated advantage may not be very predictive. We will enforce a constraint on the steps that we take to prevent our distance from the old policy from growing too large, according to

$$E_{\pi_{\text{old}}}[\text{KL}(\pi \| \pi_{\text{old}})] \leq \epsilon. \tag{4.5}$$

This is now a constrained optimization problem that we can turn into an algorithm for TRPO, shown in Algorithm 6. A specific instantiation of TRPO would be to use a first-order approximation of the surrogate loss and perform conjugate-gradient based on the second order approximation of the constraint. This particular choice is rather popular, but there are other variations within TRPO as well.

Algorithm 6: TRPO

```
for iteration ← 1 to $N_{step}$ do
    Run the policy for $T$ timesteps or $N$ trajectories;
    Estimate advantage function at all timesteps;
    Compute policy gradient $g$;
    Use CG (with Hessian-vector products) to compute $F^{-1}g$;
    Perform line search on surrogate loss and KL constraint;
end
```

Let's now think about how to evaluate the KL divergence. Recall that we can decompose our trajectory according to

$$P(\tau;\theta) = P(s_0)\prod_{t=0}^{H-1} \pi_\theta(a_t|s_t)P(s_{t+1}|s_t,a_t). \tag{4.6}$$

So when we compute the KL divergence between two distributions of trajectories over two different policies, we can write

$$\mathrm{KL}(P(\tau;\theta)\|P(\tau;\theta+\delta\theta)) = \sum_{\tau} P(\tau;\theta)\log\frac{P(\tau;\theta)}{P(\tau;\theta+\delta\theta)} \tag{4.7}$$

$$= \sum_{\tau} P(\tau;\theta)\frac{P(s_0)\prod_{t=0}^{H-1}\pi_\theta(a_t|s_t)P(s_{t+1}|s_t,a_t)}{P(s_0)\prod_{t=0}^{H-1}\pi_{\theta+\delta\theta}(a_t|s_t)P(s_{t+1}|s_t,a_t)} \tag{4.8}$$

The dynamics again cancel out, leaving us with

$$\mathrm{KL}(P(\tau;\theta)\|P(\tau;\theta+\delta\theta)) = \sum_{\tau} P(\tau;\theta)\log\frac{\prod_{t=0}^{H-1}\pi_\theta(a_t|s_t)}{\prod_{t=0}^{H-1}\pi_{\theta+\delta\theta}(a_t|s_t)} \tag{4.9}$$

which now looks like an expectation that we can estimate using roll-outs

$$\mathrm{KL}(P(\tau;\theta)\|P(\tau;\theta+\delta\theta)) \simeq \frac{1}{M}\sum_{(s,u)\text{ in roll-outs under }\theta}\log\frac{\pi_\theta(a|s)}{\pi_{\theta+\delta\theta}(a|s)} \tag{4.10}$$

4.3 Proximal Policy Optimization

TRPO is nice because it captures a lot of the intuition that we want and it allows us to extend beyond a first-order approximation of our surrogate loss while the KL divergence keeps us close to the policy collecting the data. However, this KL divergence also creates a second-order optimization problem that we need to solve. Is it possible to create a version of TRPO where everything is first-order? This would permit use of existing deep learning frameworks and likely scales better to larger networks. Note that it is also not easy to enforce a trust region constraint for complex policy networks. This could be due to network stochasticity introduced through dropout or parameter sharing between the policy and value function. Also, the conjugate-gradient implementation is both complex and fails to take advantage of the good first-order optimizers used for modern networks.

4.3.1 Version 1

The idea behind the first version of PPO is to move the constraint (the KL divergence) into our objective. That is, our goal is now to solve

$$\max_\theta E_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] - \beta(E_t[\mathrm{KL}(\pi_{\theta_{\text{old}}}(\cdot|s_t)\|\pi_\theta(\cdot|s_t))] - \epsilon), \tag{4.11}$$

where we have added some weighting factor β. With a proper choice of β, this problem becomes equivalent to the TRPO problem but we are left with just an optimization problem (and no longer any constraint). This permits the use of gradient descent to improve our policy. Our pseudocode for this version of PPO would look something like Algorithm 7. Here "dual descent" refers to a specific procedure for updating β according to the value of the KL divergence. If the KL divergence is close to ϵ, we are happy with our choice. But if our KL divergence is generally larger than ϵ, we want to increase our β to increase the weight given to the KL divergence term. And if our KL divergence is much smaller than ϵ, we can decrease our β so the next optimization pays less attention to the KL divergence term.

Algorithm 7: Version 1 of PPO

```
for iteration ← 1 to $N_{step}$ do
    Run the policy for $T$ timesteps or $N$ trajectories;
    Estimate advantage function at all timesteps;
    Perform SGD on objective for $M$ epochs;
    Perform dual descent update for $\beta$;
end
```

This version of PPO is a natural representation of the familiar TRPO problem, and it captures much of the intuition that we want. However, there is another version that is much more popular that we will cover now.

4.3.2 Version 2

Let's look at our policy ratio, given by

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}. \tag{4.12}$$

Recall that the other term that appeared in our objective for version 1 of PPO was a KL divergence term meant to ensure that this ratio was valid. We can perhaps simplify this objective further by looking carefully at this ratio. For the second version of PPO, we will directly do the trust region inside of the objective using some clipping. To ensure that our ratio does not change by too much, we can look at

$$\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \tag{4.13}$$

to ensure that the new policy cannot deviate too much from the old policy. Finally, we compare this clipped version to that obtained originally, giving us an objective

$$L^{\text{clip}}(\theta) = E_t\Big[\min\Big(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\Big)\Big]. \tag{4.14}$$

We take the minimum to be pessimistic, only taking the worse of the two. This allows us to not trust the original if the clipped version is more pessimistic. And the clipping prevents θ from changing our result if we go past an ϵ-distance from the original policy. So we cannot influence the optimization beyond a certain point for a single state-action pair. This is a different way of definining a trust region that comes directly from the objective. A benefit of this approach is that the math is much simpler, since there is no calculation of KL divergence. This has led to this becoming one of the most popular reinforcement learning algorithms today. This algorithm is especially popular when data collection is efficient, since it optimizes for wallclock time rather than sample efficiency, which benefits from off-policy methods like DQN and those we will see in the next lecture.

Chapter 5

Lecture 5: DDPG, SAC

If we are not concerned about sample efficiency, PPO is likely the best algorithm for a reinforcement learning problem. For scenarios in which sample efficiency is important, however, we likely want to consider these two methods that we will discuss in this lecture. They reuse data from the past more often, so that each piece of data contributes to more gradient updates. This allows the network to extract more utility from less data.

5.1 Deep Deterministic Policy Gradient

Under DDPG, there are a number of tasks associated with each iteration:

1. Roll-outs: execute roll-outs under the current policy (plus some exploration noise)
2. Q-function update: we have estimates of our Q-function from the roll-outs collected:

   $$\hat{Q}(s_t, a_t) = r_t + \gamma Q_\phi(s_{t+1}, a_{t+1}). \tag{5.1}$$

   Note that as we have seen, we have some flexibility in choosing how exactly we specify our target $\hat{Q}$. We use these to update our Q-function according to

   $$g \propto \boldsymbol{\nabla}_\phi \sum_t \Big(Q_\phi(s_t, a_t) - \hat{Q}(s_t, a_t) \Big)^2. \tag{5.2}$$

3. Policy update: in addition to the Q-function, we also have a policy that is optimized by looking at the Q-function for each state, which depends on the state as well as the action taken under the current policy. We want to optimize the policy such that applying the policy at states where we have samples will achieve a high value using our Q-function. This leads to

   $$g \propto \sum_t \boldsymbol{\nabla}_\theta Q_\phi(s_t, \pi_\theta(s_t, v_t)). \tag{5.3}$$

   In this way, we optimize our policy to shift probability mass towards actions resulting in large Q-values (and vice versa). Because our policy gradient goes through the Q-function, our policy could be deterministic (if desired).

There are a few improvements we can make over this basic idea. Frist, we want to add noise for exploration, especially if our policy is deterministic. We can also incorporate both the replay buffer and target network concepts from DQN to improve learning stability. Often people also use a lagged version of Q_ϕ and π_θ for the target values $\hat{Q}_t$:

$$\hat{Q}_t = r_t + \gamma Q_{\phi'}(s_{t+1}, \pi_{\theta'}(s_{t+1})), \tag{5.4}$$

This is similar to what we saw in DQN where we used older versions of the networks to stabilize learning.

In practice, DDPG is very sample efficient thanks to the off-policy updates. However, it is often unstable even with these improvements. This has led to the popularity of soft actor critic, which adds the entropy of the policy to the objective to ensure better exploration and less overfitting as a result of biases in our Q-function. For these reasons, SAC is often the algorithm of choice for off-policy reinforcement learning.

5.2 Soft Actor Critic

At a high level, this can be thought of as the high entropy version of DDPG. We start with a soft policy iteration approach. For each iteration, we will perform

1. Soft policy evaluation: for a fixed policy, we apply soft Bellman updates until convergence:

$$Q(s, a) \leftarrow r(s, a) + E_{s'\sim p_s, a'\sim\pi}\big[Q(s', a') - \log \pi(a'|s')\big] \tag{5.5}$$

   to converge to Q^π. Here we have added the entropy term to our objective (like in Lecture 1 with maximum entropy) with weighting $\beta = 1$ for convenience.

2. Soft policy improvement: the plicy is updated through information projection

$$\pi_{\text{new}} = \text{argmin}_{\pi'} D_{\text{KL}}\left(\pi'(\cdot|s) \middle\| \frac{1}{Z} Q^{\pi_{\text{old}}}(s, \cdot)\right) \tag{5.6}$$

   so that the policy is optimized using the exponentiated Q-values (like in Lecture 1). This new policy will have $Q^{\pi_{\text{new}}} \geq Q^{\pi^{\text{old}}}$.

In soft actor-critic, these optimizations are not performed exactly but instead are performed iteratively. First, a stochastic gradient step is taken to minimize the soft Bellman residual. And for the second step, a stochastic gradient step is taken to minimize the KL divergence.

Chapter 6

Lecture 6: Model-Based RL

All of the previous algorithms fall under model-free reinforcement learning. This means that when data is collected by the agent, a value function or a policy will be learned from that data. However, in principle, there is something else that the data could be used to do. The agent could alternatively learn a model of the world. This dynamics model would permit simulation, so that a policy could be learned from this world model. This could allow learning without requiring additional data collection in the real environment. Perhaps this would allow a great increase in sample efficiency. The idea is that real world $\rightarrow$ data $\rightarrow$ model $\rightarrow$ value function/policy. In practice, we may need to return to the real world periodically, but this is the general idea of model-based reinforcement learning.

Note that unlike model-free reinforcement learning, the field of model-based reinforcement learning is more open-ended in the sense that there do not exist clear guidelines for when a particular algorithm is the proper choice. There are also less mature algorithms, so there are more open questions here. We will focus on two particular ideas, but there remains considerable variation in how model-based reinforcement learning is approached.

6.1 Model-Based RL Framework

If we want to think of a vanilla model-based reinforcement learning algorithm, for each iteration we would. . .

1. Collect data under the current policy
2. Learn dynamics model from past data
3. Improve policy using our dynamics model (either achieved using backpropogation-through-time in the learned model or using the learned model as a simulation in which to run reinforcement learning)

Note that this routine requires iteration. If we only optimize our model within the simulator, it may perform very well within the learned dynamics model, but this is an imperfect model with limited data collected under the initial policy. The new policy may better exploit the system to produce more interesting data that needs to be incorporated for our policy to perform well in the real world.

As we hinted at previously, a model-based approach to reinforcement learning may lead to high data efficiency. Since we are able to form a model from the collected data, we can get more policy updates per data than just using a policy gradient. Additionally, by learning a model, we have something that can be used for other tasks, since it is not specific to the rewards we are dealing with. If the reward changed, for instance, the dynamics model would not.

Since a model-based approach can have these desirable properties, why is not this approach used all of the time? In practice, this is one of the least commonly used approaches to reinforcement learning. One reason for this is that it is less mature and requires more work. Also, if we have access to a simulator, why do we choose to run dynamics in a learned simulator? This leads to training instability, which is a major downside. Also in practice, we rarely achieve the same asymptotic performance as model-free methods.

6.2 Robust Model-Based RL: Ensemble TRPO

This aims to improve the training instability that has traditionally limited the applicability of model-based reinforcement learning. This uses TRPO since it was the state-of-the-art method when it was developed, but if redone today, this would probably be ME-PPO. Regardless of this slight difference, let's return to overfitting in model-based reinforcement learning. First, let's think about standard overfitting in supervised learning. In this case, a network performs well on training data but poorly on test data. With regularization, dropout, hold-out data, and other approaches, we have solutions to avoid overfitting.

while this type of overfitting may be present in model-based reinforcement learning, there is an additional manifestation of overfitting for model-based reinforcement learning specifically. We know that our policy optimization step tends to exploit regions where insufficient data is available to train the model. This is termed "model bias," often leading to catastrophic failures. This is because the policy will concentrate on regions where it expects a high reward in the simulator that does not exist in the real system. In this way, we overfit our policy to our simulator by concentrating on regions of parameter space that do not exist in the real world. The proposed fix in ME-TRPO is to use multiple models for the dynamics, with the details shown in Algorithm 8.

Algorithm 8: Model-ensemble trust-region policy optimization

```
Initialize a policy $\pi_\theta$ and models $\hat{f}_{\phi_1}, \hat{f}_{\phi_2}, \ldots, \hat{f}_{\phi_K}$;
Initialize an empty dataset $\mathcal{D}$;
for $i_{env} \leftarrow 1$ to $N_{env}$ do
    Collect samples from real system $f$ using $\pi_\theta$ and add them to $\mathcal{D}$;
    Train all models using $\mathcal{D}$;
    for $j_{sim} \leftarrow 1$ to $N_{sim}$ do
        Collect ficitious samples $\{\hat{f}_{\phi_i}\}_{i=1}^{K}$ using $\pi_\theta$;
        Update policy using TRPO on the fictitious samples;
        Estimate the performances $\hat{\eta}(\theta; \phi_i)$;
    end
end
```

As we can see, we now estimate performance across multiple members of the ensemble models for the dynamics. As we train an ensemble of dynamics models, we expect agreement between the models whenever there is sufficient data to support accurate dynamics. When there is not enough data, however, the predictions from these models will disagree. This tells us that we are outside of the region of trust for our simulator, and this is what is leveraged to determine where new data needs to be collected.

6.3 Adaptive Model-Based RL: MB-MPO

We just saw how using an ensemble of models can make our model-based reinforcement learning approach more robust. But can we do more? We know in the real world, for instance, there is not an ensemble of simulators but a single simulator. Can we learn something that can quickly adapt to the real world rather than something robust, which may mean that it is more conservative as a result. We now look into model-based meta-policy optimization, which aims to match the asymptotic performance of model-free reinforcement learning approaches by acting less conservatively.

The obvious fix is to simply learn a better dynamics model. But the problem is that this is very challenging. Instead, we apply model-based reinforcement learning using meta-policy optimization. The key idea is that we learn an adaptive policy that can quickly adapt from a learned model in which it was trained to the real world. To accomplish this, we still learn an ensemble of models for the world dynamics. But now we learn an adaptive policy that, when deployed within any of these models, can quickly adapt.

Algorithm 9: Model-based meta-policy optimization

```
Initialize a policy $\pi_\theta$ and models $\hat{f}_{\phi_1}, \hat{f}_{\phi_2}, \ldots, \hat{f}_{\phi_K}$;
Initialize an empty dataset $\mathcal{D}$;
for $i_{env} \leftarrow 1$ to $N_{env}$ do
    Collect samples from real system $f$ using adapted policies $\pi_{\theta'_1}, \pi_{\theta'_2}, \ldots, \pi_{\theta'_K}$ and add
     them to $\mathcal{D}$;
    Train all models using $\mathcal{D}$;
    for $k \leftarrow 1$ to $K$ do
        Sample trajectories $\mathcal{T}_k$ from $\{\hat{f}_{\phi_i}\}_{i=1}^K$ using $\pi_\theta$;
        Compute adapted parameters $\theta'_k = \theta + \alpha \nabla_\theta J_k(\theta)$ using $\mathcal{T}_k$;
        Sample imaginary trajectories $\mathcal{T}'_k$ from $\hat{f}_{\phi_k}$ using adapted policy $\pi_{\theta'_k}$;
    end
    Update $\theta \rightarrow \theta - \beta \frac{1}{K} \sum_k \nabla_\theta J_k(\theta'_k)$ using $\{\mathcal{T}'_k\}$;
end
```