Reinforcement Learning

Matthew Chen

September 28, 2025

Contents

1 Markov Decision Process

These notes introduce core ideas from reinforcement learning, which is the study of teaching autonomous agents to make good decisions by learning through data and experience from interacting with the environment. The source material comes from the lectures of Emma Brunskill and Sutton and Barto (2020). To begin, we define the Markov Decision Process, a mathematical model for sequential decision making.

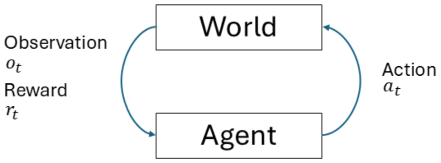

Figure 1: Framing of a MDP

At each timestep t, consider an agent that takes an *action* a_t, which the world receives and outputs an *observation* o_t and an associated *reward* r_t. The agent receives both o_t and r_t, which it uses to inform its next action. Let the *history* $h_t = (a_1, 0_1, r_1, ..., a_t, o_t, r_t)$ be the collection of all past actions, observations, and rewards up until the present. Then define the *state*, as a function of the history, $s_t = f(h_t)$. We assume that the state contains the information that determines what happens next.

More formally, the **Markov Assumption** states that s_t is a sufficient statistic of history. That is,

$$p(s_{t+1}|s_t, a_t) = p(s_{t+1}|h_t, a_t) \tag{1.1}$$

In practice, the most recent observation is often assumed to be a sufficient statistic of history $s_t = o_t$.

A Markov Decision Process involves a **Dynamics Model** P, a **Rewards Model** R, and a **Policy** π. The Dynamics Model (also known as a Transition Model) specifies the probability of the next agent state, given the current state and action.

$$p(s_{t+1} = s'|s_t = s, a_t = a) \tag{1.2}$$

The Rewards Model determines the mean immediate reward, based on the current state and the current action.

$$R(s_t = s, a_t = a) = \mathbb{E}[r_t|s_t = s, a_t = a] \tag{1.3}$$

The Policy determines how actions are chosen given the state ($\pi : S \to A$) and can either be deterministic (always take the same action for a given state)

$$\pi(s) = a \tag{1.4}$$

or stochastic

$$\pi(a|s) = P(a_t = a|s_t = s) \tag{1.5}$$

Consider a simple example with 7 possible states, with the agent starting out in s_4. The actions may include going right or going left, and the rewards are +1 in the first state and +10 in the seventh state. What policy would result in the best rewards?

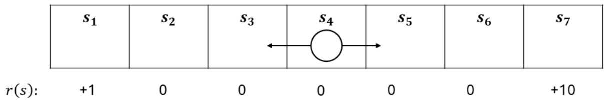

Figure 2: Tabular MDP example

1.1 Markov Process

Before discussing actions and rewards, we first introduce the Markov Process or Markov Chain, which is simply a sequence of random states that follow the Markov property (making it a memoryless random process). Let S be a finite set of states ($s \in S$) and let P be the dynamics model that specifies $p(s_{t+1} = s'|s_t = s)$. With a finite number of possible states (i.e. a tabular setting) we can express P as a matrix

$$P = \begin{pmatrix} p(s_1|s_1) & p(s_2|s_1) & ... & p(s_N|s_1) \\ p(s_1|s_2) & p(s_2|s_2) & ... & p(s_N|s_2) \\ ... & ... & ... & ... \\ p(s_1|s_N) & ... & ... & p(s_N|s_N) \end{pmatrix}$$

In the previous example, if the transition probabilities for each state are the following

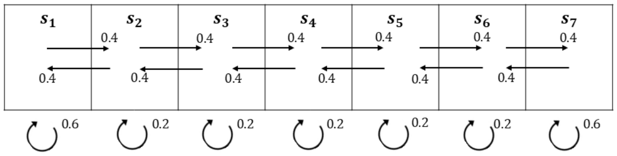

Figure 3: Tabular MDP example with transition probabilities

then the corresponding P matrix would be

$$P = \begin{pmatrix} 0.6 & 0.4 & 0 & 0 & 0 & 0 & 0 \\ 0.4 & 0.2 & 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0.2 & 0.4 & 0 & 0 & 0 \\ 0 & 0 & 0.4 & 0.2 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0.2 & 0.4 & 0 \\ 0 & 0 & 0 & 0 & 0.4 & 0.2 & 0.4 \\ 0 & 0 & 0 & 0 & 0 & 0.4 & 0.6 \end{pmatrix}$$

Note that we can sample episodes given a starting state (for example, s_4) to get sequences of states: $s_4, s_5, s_6, s_6, s_7, ...$ or $s_4, s_4, s_5, s_4, s_5, s_6, ...$ and so on.

1.2 Markov Reward Process (MRP)

Let $MRP(S, R, P, \gamma)$ be a Markov Reward Process, where as before, S is a finite set of states ($s \in S$) and P is the dynamics model that specifies $p(s_{t+1} = s'|s_t = s)$. R is a reward function that specifies the expected reward for a given state, and γ is a constant that discounts future rewards. We can think of a Markov Reward Process as a Markov Process with added rewards depending on the state (although there are no actions yet).

Let H be the horizon, or the number of timesteps in an episode (this can be infinite). Then the **Return** G_t is the discounted sum of rewards starting from timestep t.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{H-1} r_{r+H-1} \tag{1.6}$$

Note that the discount factor $\gamma \in [0, 1]$ is important since it prevents the sum from exploding in the case of $H = \infty$ (in the case of a finite H we can take $\gamma = 1$). Note that humans often act as if $\gamma < 1$, since we naturally value future returns less than immediate rewards.

Let the **State Value Function** $V(s)$ be the expected return starting in state s. From the Markov assumption, future states/rewards only depend on the previous state, so it makes sense to only condition on the starting state.

$$V(s) = \mathbb{E}\left[G_t | s_t = s\right] \tag{1.7}$$

Note that for an infinite horizon, we can take advantage of the recursive Markov structure to write

$$\begin{aligned}
V(s) &= \mathbb{E}\left[G_t | s_t = s\right] \\
&= \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{H-1} r_{r+H-1} | s_t = s] \\
&= \mathbb{E}[r_t | s_t = s] + \gamma \mathbb{E}[r_{t+1} + \gamma r_{t+2} + ... | s_t = s] \\
&= R(s) + \gamma \sum_{s' \in S} p(s'|s) \mathbb{E}[r_{t+1} + \gamma r_{t+2} + ... | s_{t+1} = s'] \\
&= R(s) + \gamma \sum_{s' \in S} p(s'|s) V(s')
\end{aligned}$$

This is the Bellman Equation for a MRP

$$V(s) = R(s) + \gamma \sum_{s' \in S} p(s'|s) V(s') \tag{1.8}$$

If we write the Bellman Equation in matrix form (representing each state), that is, $V \in \mathbb{R}^{|S|}$, $R \in \mathbb{R}^{|S|}$, $P \in \mathbb{R}^{|S| \times |S|}$, we have

$$V = R + \gamma P V \tag{1.9}$$

which has an analytic solution assuming $I - \gamma P$ is invertible

$$V = (I - \gamma P)^{-1} R \tag{1.10}$$

However, inverting a matrix is expensive ($\sim O(N^3)$). Instead, we can solve for the value function using an iterative solution which is $\sim O(N^2)$ for each iteration.

Algorithm 1.1 Iterative Algorithm for Computing MRP Value

```
Initialize $V_0(s) = 0 \ \forall s$
for $k = 1, ..$ until convergence do
    for all $s \in S$ do
        $V_k(s) = R(s) + \gamma \sum_{s' \in S} p(s'|s) V_{k-1}(s')$
    end for
end for
```

1.3 Markov Decision Process (MDP)

We can think of a Markov Decision Process (MDP) as a Markov Reward Process, but now we can take actions that affect the transition model and the rewards (see Eq. 1.2, 1.3). The policy specifies what action we take given the state (see Eq. 1.4, 1.5). If we let A be a finite set of actions ($a \in A$), we can express the MDP as $MRP(S, R^\pi, P^\pi, \gamma)$ where

$$R^\pi(s) = \sum_{a \in A} \pi(a|s) R(s,a) \tag{1.11}$$

$$P^\pi(s'|s) = \sum_{a \in A} \pi(a|s) p(s'|s,a) \tag{1.12}$$

Since the policy is a conditional distribution of actions given the state, we simply need to marginalize over all actions to reduce the MDP problem to a MRP. That is, we can use the same techniques to evaluate the value of a policy by simply solving for the MRP defined by R^π and P^π.

Let $V^\pi(s)$ be the state value function starting in s and always following policy π.

Algorithm 1.2 Iterative Algorithm for Evaluating MDP Policy

```
Initialize $V_0^\pi(s) = 0 \ \forall s$
for $k = 1, ..$ until convergence do
    for all $s \in S$ do
        $$V_k^\pi(s) = \sum_{a \in A} \pi(a|s) \left( R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_{k-1}^\pi(s') \right)$$
    end for
end for
```

Note that for a deterministc policy, the update above simplifies to

$$V_k^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} p(s'|s, \pi(s)) V_{k-1}^\pi(s')$$

1.4 Policy Iteration

Our goal for an MDP is ultimately to solve for the optimal policy as an AI planning problem.

$$\pi^*(s) = \arg\max_\pi V^\pi(s) \tag{1.13}$$

It turns out that there exists a unique optimal value function, and that the optimal policy for a MDP in an infinite horizon is deterministic, stationary (does not depend on the time step), and not necessarily unique (we can have two policies with identical values).

Further, since the number of deterministic polices is $|A|^{|S|}$, it may not be feasible to enumerate all of them. Policy Iteration and Value Iteration are two ways to solve the MDP control problem more efficiently.
We define the **state-action value** of a policy as

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^\pi(s') \tag{1.14}$$

which represents the expected total discounted reward starting in state s, taking action a first, and then following the actions provided by π for all *future* states. Note the difference with the state value function $V^\pi(s)$, which is the expected total discounted reward of starting in s, but always following π afterwards. We can marginalize out the chosen action a so that

$$V^\pi(s) = \sum_{a \in A} \pi(a|s) Q^\pi(s,a) \tag{1.15}$$

For a deterministic policy, we simply have

$$V^\pi(s) = Q^\pi(s, \pi(s)) \tag{1.16}$$

The idea in policy iteration is to iteratively compute Q^π, and update π by taking the action that maximizes Q^π. While this may seem myopic, the optimized action affects all subsequent states in the Markov chain, and future consequences under π are captured by $\gamma \sum_{s' \in S} p(s'|s,a) V^\pi(s')$. At every iteration we improve the long-term future until an optimal policy is reached.

Algorithm 1.3 Policy Iteration for Tabular MDP

```
Initialize $\pi(s)$ for all $s \in S$ randomly, $i = 0$
while $i == 0$ or $||\pi_i - \pi_{i-1}||_1 > 0$ (i.e. policy is still changing) do
    for all $s \in S$ and $a \in A$ do
        Compute the state-action value of $\pi_i$
            $$Q^{\pi_i}(s,a) = R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^{\pi_i}(s')$$
        Conduct policy improvement by computing the policy update $\pi_{i+1}$ for all $s \in S$
            $$\pi_{i+1}(s) = \arg\max_a \; Q^{\pi_i}(s,a)$$
        Update $i = i + 1$
    end for
end while
```

We can show that policy iteration in the tabular MDP (finite space and action space) results in monotonic improvement.

Proposition 1.1.

$$V^{\pi_{i+1}} \geq V^{\pi_i}$$

with strict inequality if π_i is suboptimal, where π_{i+1} is the new policy from policy improvement on π_i. Define $V^{\pi_{i+1}} \geq V^{\pi_i} : V^{\pi_{i+1}}(s) \geq V^{\pi_i}(s)\ \forall s \in S$

Proof:

$$\begin{aligned}
V^{\pi_i}(s) &= Q^{\pi_i}(s, \pi_i(s)) \\
&\leq \underset{a}{max} \; Q^{\pi_i}(s,a) \\
&= \underset{a}{max} \left(R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^{\pi_i}(s') \right) \\
&= R(s, \pi_{i+1}(s)) + \gamma \sum_{s' \in S} p(s'|s, \pi_{i+1}(s)) V^{\pi_i}(s') \\
&\leq R(s, \pi_{i+1}(s)) + \gamma \sum_{s' \in S} p(s'|s, \pi_{i+1}(s)) \, \underset{a'}{max} \; Q^{\pi_i}(s', a') \\
&= R(s, \pi_{i+1}(s)) + \gamma \sum_{s' \in S} p(s'|s, \pi_{i+1}(s)) \left(R(s', \pi_{i+1}(s')) + \gamma \sum_{s'' \in S} p(s''|s', \pi_{i+1}(s')) V^{\pi_i}(s'') \right) \\
&= ... = V^{\pi_{i+1}(s)}
\end{aligned}$$

where the last line is true because after repeating the argument we obtain a recursive expression of the Bellman equation. Since we have monotonic improvement, it will take at most $|A|^{|S|}$ iterations (the number of possible deterministic policies) to reach the optimal policy since each iteration will be better than the last, although in practice we will find the optimum much faster.

1.5 Value Iteration

The alternative to policy iteration is value iteration, where instead of iterating policies, the idea is to iterate for the optimal value function instead, and then extract the policy from the result.

Define the **Bellman Backup Operator** B, which applied to a value function, returns a new value function, improving the values if possible.

$$BV(s) := \underset{a}{max}\left(R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V(s')\right) \tag{1.17}$$

In value iteration, we start with an arbitrarily initialized value function and iteratively apply the Bellman Backup (improving the value function) until convergence.

Algorithm 1.4 Value Iteration for Tabular MDP

```
Initialize $V_0(s) = 0$ for all $s \in S$, $k = 1$
while $||V_{k+1} - V_k||_\infty > \epsilon$ do
    for all $s \in S$ do
        $$V_{k+1}(s) = \underset{a}{max}\left(R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V_k(s')\right)$$
        or in Bellman Backup notation
        $$V_{k+1} = BV_k$$
        Update $k = k + 1$
    end for
end while
To extract the optimal policy for all $s \in S$
    $$\pi(s) = \underset{a}{argmax}\left(R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V_k(s')\right)$$
```

Intuitively, value iteration works by picking the optimal value given a k-step look-ahead window. We initialize values at zero, so the first iteration is the highest immediate reward. At the second iteration, we have the optimal immediate reward plus one discounted step into the future as accounted for by our estimate of V in the first iteration. This is the optimal value if we were only allowed to take two steps. At the third iteration, we have the optimal immediate reward plus two discounted steps into the future and is the optimal value if we were only allowed to take three steps. We are finding the optimal value under an expanding horizon, and eventually the discount factor shrinks distant rewards until convergence is reached.

Mathematically, it turns out that value iteration will converge for $\gamma < 1$ since the Bellman Backup is a contraction, that is, the distance between two value functions shrink after applying the Bellman Backup to each. Intuitively, if we successively apply the Bellman Backup, we will eventually converge to a fixed point. Since the optimal value V^* is the only point where $BV^* = V^*$, the optimal value is the fixed point we will converge to. Precisely,

Proposition 1.2. *The Bellman Operator is a contraction for $\gamma < 1$, i.e.,*

$$||BV - BV'||_\infty \leq ||V - V'||_\infty$$

where V, V' are two different value functions

Proof.

$$\begin{aligned}
\|BV - BV'\|_\infty &= \left\| \underset{a}{max} \left(R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V(s') \right) - \underset{a'}{max} \left(R(s,a') + \gamma \sum_{s' \in S} p(s'|s,a')V'(s') \right) \right\|_\infty \\
&\leq \underset{a}{max} \left\| R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V(s') - R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V'(s') \right\|_\infty \\
&= \underset{a}{max} \ \gamma \left\| \sum_{s' \in S} p(s'|s,a) \left(V(s') - V'(s') \right) \right\|_\infty \\
&\leq \underset{a}{max} \ \gamma \left\| \sum_{s' \in S} p(s'|s,a) \|V - V'\|_\infty \right\|_\infty \\
&= \underset{a}{max} \ \gamma \|V - V'\|_\infty \left\| \sum_{s' \in S} p(s'|s,a) \right\|_\infty \\
&= \gamma \|V - V'\|_\infty
\end{aligned}$$

So for $\gamma < 1$ we have

$$\|BV - BV'\| \leq \|V - V'\|_\infty$$

As an aside on Bellman Backup notation, note that *for a particular policy* π, we can define the Bellman Backup Operator as

$$B^\pi V(s) = R^\pi(s) + \gamma \sum_{s' \in S} p^\pi(s'|s)V(s') \tag{1.18}$$

so that in policy evaluation, we are just applying

$$V^\pi = B^\pi B^\pi B^\pi ... V$$

until V stops changing.

2 Model Free Policy Evaluation

When analyzing a MDP, we assumed knowledge of the dynamics model P and the rewards model R (so that finding an optimal policy is simply a planning problem). However, how do we estimate the return of a policy when we do not have direct access to these models (i.e. we do not have an explicit model for how the world works)? The solution is to estimate values by gathering data and interacting with the environment.

2.1 Monte-Carlo Policy Evaluation

In MC Policy Evaluation, the idea is simply to sample many episodes and take the average of the total returns as an estimate for the expected total return under some policy. This approach is convenient because it does not require MDP dynamics and rewards, and does not assume the Markov structure holds. However, we do require that each episode terminates eventually (episodic setting).

Algorithm 2.1 First Visit Monte-Carlo Policy Evaluation

```
Initialize for all $s \in S$: $N(s) = 0$ (count times state is visited), $G(s) = 0$ (start with no returns)
for $i = 1, ...$ until convergence do
    Sample $i$th episode: $(s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, ..., s_{i,T_i}, a_{i,T_i}, r_{i,T_i})$

    for $t = 1 : T_i$ do
        if first time $t$ that state $s$ is visited in episode $i$ then
            Calculate
                $$G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + ... + \gamma^{T_i - t} r_{i,T_i}$$
            Increment counter
                $$N(s) = N(s) + 1$$
            Increment total return
                $$G(s) = G(s) + G_{i,t}$$
            Update estimate
                $$V^\pi(s) = G(s)/N(s)$$
        end if
    end for
end for
```

Algorithm 2.2 Every Visit Monte-Carlo Policy Evaluation

```
Initialize for all $s \in S$: $N(s) = 0$ (count times state is visited), $G(s) = 0$ (start with no returns)
for $i = 1, ...$ until convergence do
    Sample $i$th episode: $(s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, ..., s_{i,T_i}, a_{i,T_i}, r_{i,T_i})$

    for $t = 1 : T_i$ do
        Let $s$ be the state visited at time $t$ in episode $i$
        Calculate
            $$G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + ... + \gamma^{T_i - t} r_{i,T_i}$$
        Increment counter
            $$N(s) = N(s) + 1$$
        Increment total return
            $$G(s) = G(s) + G_{i,t}$$
        Update estimate
            $$V^\pi(s) = G(s)/N(s)$$
    end for
end for
```

Algorithm 2.3 Incremental Every Visit Monte-Carlo Policy Evaluation

```
Initialize for all $s \in S$: $N(s) = 0$ (count times state is visited), $V(s) = 0$

for $i = 1, ...$ until convergence do
    Sample $i$th episode: $(s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, ..., s_{i,T_i}, a_{i,T_i}, r_{i,T_i})$

    for $t = 1 : T_i$ do
        Let $s$ be the state visited at time $t$ in episode $i$
        Calculate
            $$G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + ... + \gamma^{T_i - t} r_{i,T_i}$$
        Increment counter
            $$N(s) = N(s) + 1$$
        Update estimate with new data, weighted by $N$
            $$\begin{aligned} V^\pi(s) &= V^\pi(s) \frac{N(s)-1}{N(s)} + \frac{G_{i,t}}{N(s)} \\ &= V^\pi(s) + \frac{1}{N(s)}(G_{i,t} - V^\pi(s)) \end{aligned}$$
        In general, we can update with some stepsize $\alpha_k$ where $k = N(s)$
            $$V^\pi(s) = V^\pi(s) + \alpha_k(s)(G_{i,t} - V^\pi(s))$$

    end for
end for
```

Recall that for a statistical model $p(x|\theta)$ parameterized by θ, with $\hat{\theta}$ providing an estimate for θ then

$$Bias_\theta(\hat{\theta}) = \mathbb{E}_{x|\theta}\hat{\theta} - \theta \tag{2.1}$$

$$Var(\hat{\theta}) = \mathbb{E}_{x|\theta}\left[(\hat{\theta} - \mathbb{E}\hat{\theta})^2\right] \tag{2.2}$$

$$MSE(\hat{\theta}) = Var(\hat{\theta}) + Bias_\theta(\hat{\theta})^2 \tag{2.3}$$

We say that an estimator is consistent, if for all $\epsilon > 0$ we have

$$\lim_{n\to\infty} P(|\hat{\theta}_n - \theta| > \epsilon) = 0 \tag{2.4}$$

Note that an unbiased estimator is not necessarily consistent, and a consistent estimator is not necessarily unbiased.

In the context of Monte Carlo Policy Evaluation, the First Visit estimator is unbiased of the true $V^\pi(s) = \mathbb{E}[G_t|s_t = s]$ and is consistent by the Law of Large Numbers. In every visit, introduces a small bias since a state may appear more than once in an episode, but often has better MSE than First Visit and is still consistent. The incremental is also biased but consistent if the step size satisfies the Robbins-Munro sequence, i.e.

$$\sum_{n=1}^{\infty} \alpha_n(s) = \infty \tag{2.5}$$

$$\sum_{n=1}^{\infty} \alpha_n^2(s) < \infty \tag{2.6}$$

In general, Monte Carlo methods have high variance. This is due to the estimates $G_{i,t}$ running from t until the end of the episode, where each time step introduces an additional level of stochasticity.

2.2 Temporal Difference Learning (TD(0))

The idea behind temporal difference learning is to update the estimate after each (s, a, r, s') tuple instead of waiting for the end of the episode for $G_{i,t}$ which has high variance. Additionally, in this case we no longer need an episodic setting, and the horizon can be infinite.

Recall that in the incremental every visit Monte Carlo strategy we updated the value estimate using

$$V^\pi(s) = V^\pi(s) + \alpha(G_{i,t} - V^\pi(s))$$

What if, instead of using $G_{i,t}$ we construct a bootstrap estimate that exploits the Markov structure. That is, we use the update rule

$$V^\pi(s_t) = V^\pi(s_t) + \alpha(r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)) \tag{2.7}$$

where $r_t + \gamma V^\pi(s_{t+1})$ is called the TD target. Here, r_t is an estimate of the expected immediate reward, and $\gamma V^\pi(s_{t+1})$ is a boostrap estimate of the expected discounted future rewards. The boostrapping occurs from using our current estimate of $V^\pi(s_{t+1})$ which may not be accurate early on.

Algorithm 2.4 TD(0) Policy Evaluation

```
Initialize for all $s \in S$: $V^\pi(s) = 0$

for $t = 0, ...$ until convergence do
    Sample next tuple in episode from $\pi$: $(s_t, a_t, r_t, s_{t+1})$
    Update estimate of $V^\pi$

        $V^\pi(s_t) = V^\pi(s_t) + \alpha(r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t))$

end for
```

TD(0) is a biased estimator, but generally has lower variance than MC evaluation. It is consistent if the stepsize α satisfies the Robbins-Munro sequence.

2.3 Certainty Equivalence

Since we have data from policy π, can we directly estimate the world models using the maximum likelihood estimator (MLE)? Specifically, after each (s, a, r, s') tuple we can update the MLE MDP models for (s, a) using

$$\hat{p}(s'|s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k-1} \mathbb{1}\{s_{k,t} = s, a_{k,t} = a, s_{k,t+1} = s'\} \tag{2.8}$$

$$\hat{r}(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k-1} \mathbb{1}\{s_{k,t} = s, a_{k,t} = a\} r_{t,k} \tag{2.9}$$

This strategy is called certainty equivalence because we use the estimated models to solve the MDP, as if they were the real models. This strategy is computationally expensive, because at each update we need to both update the MLE and solve the resulting MDP. However, the result is consistent and very data efficient.

2.4 Batch (Offline) MC/TD

Given a set of K episodes, imagine repeatedly sampling an episode from K, and then applying MC or TD(0) to the sampled episode to evaluate the value function.

Consider the following conceptual example where we observe the following batch of 8 episodes of (states, rewards):

- (A, 0), (B, 0)
- (B, 1)
- (B, 1)
- (B, 1)
- (B, 1)
- (B, 1)
- (B, 1)
- (B, 0)

What is a reasonable prediction for $V(B)$? Since 6 of the 8 B states immediately terminated with reward 1 (and the other two with reward zero), 6/8 is a reasonable. However, note that there are two reasonable predictions for $V(A)$:

1. We model the Markov process as the following:

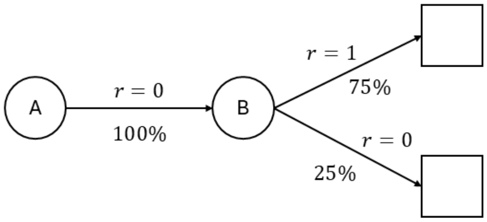

Figure 4: Batch policy evaluation conceptual example

In this case, our estimate for $V(A)$ would be 6/8, since 100% of the time, state A goes to state B, and $V(B)$ is 6/8.

2. We observe that 100% of episodes with state A results in 0 return, so we estimate $V(A) = 0$

For the latter, we are minimizing the MSE on observed data, which turns out is the same answer as batch MC. However, if the process is Markov, then we would except the first solution to be better. Even if the MSE on existing data is worse, we would expect it to be better in future data since we got our answer from modeling the world.

It turns out, that in general, batch MC converges to a solution where we minimize MSE on observed data. Batch TD(0), however, converges to V^π for the MDP with MLE model estimates. That is, it converges to the same answer as dynamic programming with certainty equivalence!

3 Model Free Control

In the previous section, we discussed how we can estimate the value function if we did not have explicit models of how the world works. In this section, we will discuss control methods in the same setting. Recall that in policy improvement, we update our policy by choosing actions that maximizes $Q^\pi(s, a)$. The problem that arises with collecting data from a deterministic policy, however, is that we will always follow $a = \pi(s)$ so how can we compute, let alone maximize, $Q(s, a)$ for any other action $a \neq \pi(s)$. We cannot learn about actions without ever trying them (i.e. exploration), but we also need to take actions that maximize rewards (i.e. exploitation). The relationship between exploration and exploitation is a fundamental idea in reinforcement learning.

Instead, consider a stochastic policy (called ϵ-greedy) that balances exploration and exploitation. An ϵ-greedy policy with respect to a state-action value $Q(s, a)$, with $|A|$ possible actions, is defined as

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|A|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \dfrac{\epsilon}{|A|} & \text{otherwise} \end{cases} \tag{3.1}$$

Sometimes, we choose to take the action that maximizes $Q(s, a)$, otherwise, we sample an action uniformly at random, which helps us to collect more data and improve the coverage and quality of the Q value estimates. Recall that policy iteration (when we are given a dynamics and rewards model) was guaranteed to monotonically improve for a deterministic policy. It turns out that the same property holds for ϵ-greedy policies under the same conditions as well!

3.1 Monte Carlo Online Control

Now we can use Monte Carlo simulations to estimate $Q(s, a)$, and then simply extract an ϵ-greedy policy from our estimated $Q(s, a)$.

Algorithm 3.1 Monte Carlo Online Control (On Policy Improvement)

```
Initialize for all $s \in S$ and all $a \in A$: $Q(s,a) = 0, N(s,a) = 0$
Initialize $\epsilon = 1$, $\pi_k = \epsilon - greedy(Q)$

for $k = 1, ...$ until convergence do
    Sample $k$th episode given $\pi_k$: $(s_{k,1}, a_{k,1}, r_{k,1}, s_{k,2}, a_{k,2}, r_{k,2}, ..., s_{k,T_k})$

    for $t = 1 : T_k$ do
        if first visit to $(s,a)$ in episode $k$ then
            Update counter
                $$N(s,a) = N(s,a) + 1$$
            Update estimate of $Q(s,a)$
                $$Q(s_t,a_t) = Q(s_t,a_t) + \frac{1}{N(s,a)}(G_{k,t} - Q(s_t,a_t))$$
        end if
    end for
    $\epsilon = 1/k$

    Extract $\epsilon$-greedy policy $\pi_{k+1} = \epsilon - greedy(Q)$
end for
```

Note how with each iteration, we are shrinking ϵ toward zero. The intuition is, as we collect more data, the better our estimate of the state-value function $Q(s,a)$ becomes, so the need for exploration becomes smaller. Formally, we say a policy is **Greedy in the Limit of Inifinite Exploration (GLIE)** if in the limit where all state-action pairs are visited an infinite number of times

$$\lim_{k \to \infty} N_k(s,a) \to \infty \tag{3.2}$$

then the behavior policy converges to the greedy policy

$$P\left(\lim_{k \to \infty} \pi(a|s) \to \underset{a}{argmax}\, Q(s,a)\right) = 1 \tag{3.3}$$

It turns out that GLIE Monte Carlo control, like ours where we are using an ϵ-greedy policy where $\epsilon \to 0$ with rate $\epsilon = 1/k$, will converge to the optimal state-action function

$$Q(s,a) \to Q^*(s,a) \tag{3.4}$$

3.2 State Action Reward Next State Next Action (SARSA)

We can also use a temporal difference alternative to updating $Q(s,a)$ rather than using MC. Consider the SARSA algorithm:

Algorithm 3.2 State Action Reward Next State Next Action (SARSA)

```
Initialize $\epsilon$-greedy policy $\pi$ randomly
Sample initial state $s_0$, take $a_0 \sim \pi(s_0)$, and observe $(r_0, s_1)$

for $t = 0, ...$ until convergence do
    Take next action in episode $a_{t+1} \sim \pi(s_{t+1})$
    Observe tuple $(r_{t+1}, s_{t+2})$
    Update estimate of $Q$ given $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$
        $Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$
    Perform policy improvement by maintaining $\pi(s) = \epsilon - greedy(Q)$
    $\epsilon = 1/t$ if $t \neq 0$
    If $s_{t+2}$ is terminal then reset the episode, sample new initial state, repeat
end for
```

SARSA, for finite state and finite action MDP will converge to the optimal state-action value, provided that the step sizes satisfy the Robbins-Munro sequence (e.g. $\alpha_t = 1/t$ will satisfy these conditions) and the policy is GLIE.

3.3 Q-learning

SARSA is an example of **on-policy learning**, since we are learning to evaluate a policy from direct experience from following that policy. In contrast, in **off-policy learning** we want to learn about a policy using experience gathered from following a different policy. For example, in Q-learning, our goal is to learn the *optimal* state-action value directly, instead of the state-action value of the policy we are currently following. Specifically in SARSA our update to Q is

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{3.5}$$

where a_{t+1} is an actual action sampled from the current policy. Instead, Q-learning uses the update

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \underset{a'}{max}\, Q(s_{t+1}, a') - Q(s_t, a_t)) \tag{3.6}$$

where $\gamma \underset{a'}{max}\, Q(s_{t+1}, a')$ is an estimate of the maximum (optimal) reward that can be obtained from s_{t+1} (note that this strategy is akin to value iteration).

Algorithm 3.3 Q-learning with ϵ-greedy Exploration

```
Initialize for all $s \in S$ and all $a \in A$: $Q(s, a) = 0$ and sample initial state $s_0$
Set $\pi = \epsilon - greedy(Q)$

for $t = 0, ...$ until convergence do
    Take action $a_t \sim \pi(s_t)$
    Observe tuple $(r_t, s_{t+1})$
    Update estimate of $Q$ given $(s_t, a_t, r_t, s_{t+1})$
        $Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \underset{a'}{max}\, Q(s_{t+1}, a') - Q(s_t, a_t))$
    Perform policy improvement by maintaining $\pi(s) = \epsilon - greedy(Q)$
    $\epsilon = 1/t$ if $t \neq 0$
    If $s_{t+1}$ is terminal then reset the episode, sample new initial state, repeat
end for
```

Q-learning with ϵ-greedy exploration will converge to the optimal state-action value Q^* if all (s, a) pairs are visited infinitely often and α_t satisfies the Robbins-Munro sequence (GLIE is not required for this, so ϵ can be kept large). However, to converge to the optimal π^* we require the algorithm to be GLIE.

4 Value Function Approximation (VFA)

So far, we have discussed the tabular case in which there are a finite number of states and actions. What if the state and action space is too large to store P, R, V, Q, π for every state and action? We would need a strategy that reduces the computation and storage needed to compute values, and hopefully also reduces the experience needed to find a good policy.

Consider an oracle $Q^\pi(s, a)$ which returns the true value of Q^π for a given state and action. What if we build a supervised learning dataset with the pairs

$$< (s, a), Q^\pi(s, a) > \tag{4.1}$$

and train an approximate representation (i.e. a neural network) parameterized by weights w

$$\hat{Q}^\pi(s, a; w) \tag{4.2}$$

If we take the square loss then the objective function is

$$J(w) = \mathbb{E}\left[(Q^\pi(s, a) - \hat{Q}^\pi(s, a; w))^2\right] \tag{4.3}$$

with gradient

$$\nabla_w J(w) = 2\mathbb{E}\left[\hat{Q}^\pi(s, a; w) - Q^\pi(s, a)\right] \nabla_w \hat{Q}^\pi(s, a, ; w) \tag{4.4}$$

and gradient descent update

$$w = w + \Delta w = w - \frac{\alpha}{2} \nabla_w J(w) \tag{4.5}$$

4.1 Policy Evaluation using VFA

Consider the Monte-Carlo return G_t which is an unbiased but noisy sample of the true expected return for $Q^\pi(s_t, a_t)$. We can substitute G_t for the oracle $Q^\pi(s_t, a_t)$ when fitting the function approximator. Then, the MC VFA problem is reduced to supervised learning on the dataset with data pairs

$$< (s_t, a_t), G_t > \tag{4.6}$$

Algorithm 4.1 Monte Carlo VFA for Policy Evaluation

```
Initialize weights $w$ for $\hat{Q}(s, a; w)$
for $k = 1, ...$ until convergence do
    Sample $k$th episode given $\pi$: $(s_{k,1}, a_{k,1}, r_{k,1}, s_{k,2}, a_{k,2}, r_{k,2}, ..., s_{k,T_k})$

    for $t = 1 : T_k$ do
        if first visit to $(s, a)$ in episode $k$ then
            Update the weights with one step of stochastic gradient descent by computing
                $\nabla_w J(w) = -2\left(G_{k,t} - \hat{Q}^\pi(s, a; w)\right) \nabla_w \hat{Q}^\pi(s, a; w)$
                $w = w + \Delta w = w - \frac{\alpha}{2} \nabla_w J(w)$
        end if
    end for
end for
```

Note that since in every iteration we are updating the weights using one sample of G_t using the gradient as an estimate of the expected stochastic gradient (i.e. Eq. 4.4), so we are using stochastic gradient descent.

The temporal difference strategy is very similar. Recall in TD(0) that $r + \gamma V^\pi(s')$ is used as a biased bootstrap estimate for $V^\pi(s)$. In the VFA context, we can substitute a function approximation $\hat{V}(s'; w)$ in the bootstrap estimator and conduct supervised learning on the data pairs

$$< s, r + \gamma \hat{V}^\pi(s' : w) > \tag{4.7}$$

Algorithm 4.2 TD(0) VFA Policy Evaluation

```
Initialize initial state $s_0$, weights $w$

**for** $t = 0, ...$ until convergence **do**
    Given $s_t$, take $a_t \sim \pi(s_t)$ and observe reward $r_t$ and next state $s_{t+1}$
    Update the weights with one step of stochastic gradient descent by computing
        $\nabla_w J(w) = -2\left(r_t + \gamma \hat{V}^\pi(s_{t+1}; w) - \hat{V}^\pi(s_t; w)\right) \nabla_w \hat{V}^\pi(s_t; w)$
        $w = w + \Delta w = w - \frac{\alpha}{2} \nabla_w J(w)$
**end for**
```

Note that while we wrote out TD(0) VFA for the state-value function, we can also easily do the same with the state-action value by taking the next action a_{t+1} and using $r_t + \gamma \hat{Q}^\pi(s_{t+1}, a_{t+1}; w)$ as the target (akin to SARSA).

4.2 Control with VFA

For control, we simply apply VFA to approximate the state-action value function $\hat{Q}^\pi(s, a; w)$, and then perform ϵ-greedy policy improvement. For example, in Monte Carlo methods we can use the weights update

$$\Delta w = \alpha(G_t - \hat{Q}(s_t, a_t; w))\nabla_w \hat{Q}(s_t, a_t; w) \tag{4.8}$$

For SARSA

$$\Delta w = \alpha(r + \gamma \hat{Q}(s', a'; w) - \hat{Q}(s, a; w))\nabla_w \hat{Q}(s, a; w) \tag{4.9}$$

For Q-learning

$$\Delta w = \alpha(r + \gamma \underset{a'}{max}\, \hat{Q}(s', a'; w) - \hat{Q}(s, a; w))\nabla_w \hat{Q}(s, a; w) \tag{4.10}$$

However, multiple levels of the following

- Bootstrapping (i.e. using $r + \gamma \hat{Q}(s', a'; w)$ as the TD target in SARSA)
- Function Approximation (i.e. fitting $\hat{Q}$)
- Off-policy learning (i.e. using the related TD target $r + \gamma \underset{a'}{max}\, \hat{Q}(s', a'; w)$ in Q-learning)

can lead to oscillations or a lack of convergence. Technically, while the Bellman Operator is a contraction, fitting VFA can turn out to be an expansion. These combinations of factors are known as the ”Deadly Triad.”

4.3 Deep Q-learning (DQN)

Since Q-learning with VFA can diverge, we introduce two strategies to help stabilize it. First is *experience replay*, which seeks to remove correlations between samples. Supervised learning typically assumes iid samples, but this assumption is broken since consecutive tuples (s, a, r, s') are highly correlated with each other and can lead to poor generation in the fitted function. Instead, we store tuples from all prior experience and resample a mini-batch of them at each iteration. This not only improves the correlations between samples, but also allows data to be reused which improves data efficiency. Second are *fixed Q-targets*. Since weights are updated in every iteration, this also means that we are trying to predict moving targets. Instead, we fix the target weights for several iterations (apart from the weights that are being updated at every iteration) to stabilize training.

Algorithm 4.3 Deep Q-learning

```
Input constant $C$, $\alpha$
Initialize weights $w$, set $\bar{w} = w$, sample initial state $s_0$, and initialize replay buffer $\mathcal{D} = \{\}$
Set $\pi = \epsilon - greedy(\hat{Q})$

for $t = 0, ...$ until convergence do
    Take action given $\epsilon$-greedy policy w.r.t current $\hat{Q}(s_t, a; w) : a_t \sim \pi(s_t)$
    Observe tuple $(r_t, s_{t+1})$

    Store transition tuple $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
    Sample mini-batch of tuples $(s_i, a_i, r_i, s_{i+1})$ from $\mathcal{D}$

    for each mini-batch in sample do
        if episode terminated at $i + 1$ then $y_i = r_i$
        else
            $y_i = r_i + \gamma \underset{a'}{max} \hat{Q}(s_{i+1}, a'; \bar{w})$
        Perform stochastic gradient descent step with weight update
            $\Delta w = \alpha(y_i - \hat{Q}(s_i, a_i; w))\nabla_w \hat{Q}(s_i, a_i; w)$
            $w = w + \Delta w$
    end for
    if mod(t, C) == 0 then $\bar{w} = w$
    $\epsilon = 1/t$ if $t \neq 0$
end for
```

5 Policy Search

Previously, we parameterized the state-action value function, and extracted the learned policy directly from the function approximation (for example, using ϵ-greedy). What if instead we parameterize the policy directly, and then optimize it directly to find the policy with the best value? In general, these are two fundamental approaches to reinforcement learning: value based (the former) and policy based (the latter). Let θ be the parameters of the policy, i.e.

$$\pi_\theta(s, a) = p(a|s; \theta) \tag{5.1}$$

Then we can measure the quality of a particular policy π_θ by using the policy value at the start state $V(s_0, \theta)$ and optimize for θ with the highest value. In *direct policy search*, we can use gradient free optimization such as hill climbing, genetic algorithms, cross-entropy method, covariance matrix adaptation, etc. which can work surprisingly well, although they ignore any temporal structure and is often less sample efficient.

The advantage of the policy-based approach is that we can parameterize a stochastic policy (whereas value based learning such as ϵ-greedy gives us a near deterministic policy) and/or easily represent a continuous action space. A stochastic policy can be advantageous in partially observable (i.e. different hidden states consistent with the same observation may demand different actions) or non-stationary (i.e. what is optimal now may not be optimal later) environment, although note that both these cases break the Markov assumption. Further, in a multi-agent game a deterministic strategy can be exploited by the oppoenent; higher rewards may be available from being unpredictable.

5.1 Policy Gradient Method

We can solve the policy search problem using gradient based optimization. Define $V^{\pi_\theta} = V(s_0, \theta)$ be the state value function at the starting state s_0 following policy π_θ. Our goal is to seach for the local max of $V(s_0, \theta)$ by ascending the gradient of the policy with respect to θ. This method has better convergence properties and is effective in high-dimensional or continuous action space. However, we can only guarantee convergence to a local optima, and evaluating a policy can have high variance.

In the episodic case, define the trajectory $\tau = (s_0, a_0, r_0, ..., s_{T-1}, a_{T-1}, r_{T-1}, s_T)$, and define $V(\theta) = V(s_0, \theta)$. Let $p(\tau, \theta)$ be the probability over all trajectories resulting from executing policy π_θ starting in s_0, and $R(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t)$ be the sum of rewards for trajectory τ. Then we can express $V(\theta)$ as

$$V(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} R(s_t, a_t)\right] = \sum_\tau p(\tau; \theta) R(\tau) \tag{5.2}$$

where our goal is to find

$$\underset{\theta}{argmax}\, V(\theta) = \underset{\theta}{argmax} \sum_\tau p(\tau; \theta) R(\tau) \tag{5.3}$$

We can find the policy gradient $\nabla_\theta V(\theta)$ by using the *likelihood ratio* trick, noting that

$$\nabla_\theta \log p(\tau; \theta) = \frac{\nabla_\theta p(\tau; \theta)}{p(\tau; \theta)} \tag{5.4}$$

So we have

$$\begin{aligned}
\nabla_\theta V(\theta) &= \nabla_\theta \sum_\tau p(\tau; \theta) R(\tau) \\
&= \sum_\tau R(\tau) \nabla_\theta p(\tau; \theta) \\
&= \sum_\tau R(\tau) \frac{p(\tau; \theta)}{p(\tau; \theta)} \nabla_\theta p(\tau; \theta) \\
&= \sum_\tau R(\tau) p(\tau; \theta) \nabla_\theta \log p(\tau; \theta)
\end{aligned}$$

We can approximate the expectation over τ using m sample paths under the policy π_θ

$$\nabla_\theta V(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \nabla_\theta \log p(\tau^{(i)}; \theta)$$

Decomposing the log probability of the observed trajectories into states and actions using the policy and the dynamics model, and letting $\mu(s_0)$ be the initial state distribution, we have

$$\begin{aligned}\nabla_\theta \log p(\tau^{(i)};\theta) &= \nabla_\theta \log \left[\mu(s_0)\prod_{t=0}^{T-1}\pi_\theta(a_t^{(i)}|s_t^{(i)})p(s_{t+1}^{(i)}|s_t^{(i)},a_t^{(i)})\right] \\ &= \nabla_\theta\left[\log\mu(s_0)+\sum_{t=0}^{T-1}\Big(\log\pi_\theta(a_t^{(i)}|s_t^{(i)})+\log p(s_{t+1}^{(i)}|s_t^{(i)},a_t^{(i)})\Big)\right] \\ &= \sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t^{(i)}|s_t^{(i)})\end{aligned}$$

where $\nabla_\theta \log\pi_\theta(a_t^{(i)}|s_t^{(i)})$ is called the *score function.* Then the estimate of the policy gradient $\hat{g}$ becomes

$$\nabla_\theta V(\theta) \approx \hat{g} := \frac{1}{m}\sum_{i=1}^{m}\left(R(\tau^{(i)})\sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t^{(i)}|s_t^{(i)})\right) \tag{5.5}$$

Importantly, this estimate does not require knowledge of the dynamics model! Intuitively, if we go in the direction of $\hat{g}$, we push up the log-probability of the sample proportional to its quality, as measured by $R(\tau)$. However, while $\hat{g}$ as formulated is unbiased, it is also very noisy.

5.2 REINFORCE

We can improve on (i.e. reduce the variance of) Eq. 5.5 by incorporating the temporal structure. Consider repeating the same argument as before, but for a single reward term at time t', $r_{t'}$. Notice that actions and states beyond t' have no impact on the reward at t' (i.e. the reward at time t' cannot be influenced by a decision made at $t'+1$). This reduces the number of timesteps we need to sum the score function over so that the variance is reduced.

$$\nabla_\theta \mathbb{E}_\tau r_{t'} = \mathbb{E}_\tau\left[r_{t'}\sum_{t=0}^{t'}\nabla_\theta\log\pi_\theta(a_t|s_t)\right] \tag{5.6}$$

Summing over all individual rewards we obtain

$$\nabla_\theta V(\theta) = \nabla_\theta\mathbb{E}_\tau R = \mathbb{E}_\tau\left[\sum_{t'=0}^{T-1}\left(r_{t'}\sum_{t=0}^{t'}\nabla_\theta\log\pi_\theta(a_t|s_t)\right)\right]$$

Reorganizing the indices and substituting $G_t = \sum_{t'=t}^{T-1} r_{t'}$ we have

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

As before, we can estimate the expectation using m trajectory samples to finally obtain

$$\nabla_\theta V(\theta) \approx \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t,s_t)G_t^{(i)} \tag{5.7}$$

which improves on Eq. 5.5 by taking advantage of the temporal structure and reducing variance. This gives us the REINFORCE algorithm.

Algorithm 5.1 REINFORCE (Monte-Carlo Policy Gradient)

```
Initialize policy parameters $\theta$

for each episode $(s_0, a_0, r_0, ..., s_{T-1}, a_{T-1}, r_{T-1}, s_T)$ sampled from $\pi_\theta$ until convergence do
    for $t = 0, ..., T-1$ do
        $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G_t$
    end for
end for
```

5.3 Actor-Critic Methods

The REINFORCE algorithm still relies on Monte-Carlo sampling which we know is higher variance - can we fold in temporal-difference style updates into the policy gradient algorithm? To do so, we introduce the **Actor-Critic Method**, where there are explicit representations of both the policy (i.e. the *actor*, parameterized by θ) and the value function (i.e. the *critic*, parameterized by w). We can begin by introducing a baseline term in our current policy gradient estimate.

Proposition 5.1. *The policy gradient estimator below remains unbiased for any choice of baseline $b(s)$ as a function of state as long as it does not depend on the action.*

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_\tau R = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t;\theta)(G_t - b(s_t)) \right] \tag{5.8}$$

Proof.

$$\begin{aligned}
\mathbb{E}_\tau \left[\nabla_\theta \log \pi(a_t|s_t;\theta) b(s_t) \right] &= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[\mathbb{E}_{s_{t+1:T}, a_{t:T-1}} \nabla_\theta \log \pi(a_t|s_t;\theta) b(s_t) \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[b(s_t) \mathbb{E}_{s_{t+1:T}, a_{t:T-1}} \nabla_\theta \log \pi(a_t|s_t;\theta) \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[b(s_t) \mathbb{E}_{a_t} \nabla_\theta \log \pi(a_t|s_t;\theta) \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[b(s_t) \sum_{a_t \in A} \pi_\theta(a_t|s_t) \nabla_\theta \log \pi(a_t|s_t;\theta) \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[b(s_t) \sum_{a_t \in A} \pi_\theta(a_t|s_t) \frac{\nabla_\theta \pi(a_t|s_t;\theta)}{\pi_\theta(a_t|s_t)} \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[b(s_t) \sum_{a_t \in A} \nabla_\theta \pi(a_t|s_t;\theta) \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[b(s_t) \nabla_\theta \sum_{a_t \in A} \pi(a_t|s_t;\theta) \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:t-1}} \left[b(s_t) \nabla_\theta 1 \right] = 0
\end{aligned}$$

It turns out that the state-value function $V^\pi(s_t)$ serves as a great baseline. Define

$$A(s, a) = Q(s, a) - V(s) \tag{5.9}$$

as the **advantage function**. Then the MC return subtracted by the baseline $G_t - V(s_t)$ is an estimate for the advantage $A(s_t, a_t)$. Intuitively, by using the advantage estimate instead of the returns directly, we are increasing the log probability of action a_t proportionally to the amount of returns G_t are *better than expected*. Combining both temporal structure and the baseline term, we can construct a ”vanilla” policy gradient algorithm.

Algorithm 5.2 Vanilla Policy Gradient

```
Initialize policy parameters $\theta$, baseline $b$

for iteration $k = 1, 2, ...$ until convergence do
    Sample a set of $m$ trajectories $\tau_i$ by executing current policy $\pi_\theta$
    for $t = 0, ..., T-1$ in each trajectory $\tau_i$ do
        Calculate return
            $$G_t^i = \sum_{t'=t}^{T-1} r_{t'}^i$$
        Calculate advantage estimate
            $$\hat{A}_t^i = G_t^i - b(s_t^i)$$
    end for
    Refit the baseline (using Monte-Carlo value function approximation) by minimizing
        $$\sum_{i=1}^{m} \sum_{t=0}^{T-1} \left(b(s_t^i) - G_t^i\right)^2$$
    Update the policy with SGD by using the policy gradient estimate $\hat{g}$
        $$\nabla_\theta V(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t;\theta) \hat{A}_t^i$$
end for
```

Note how in the above algorithm, we need to maintain a separate network to estimate the state-values. Using the critic network, we can estimate the advantage using any blend of TD or MC estimators, striking a balance between bias and variance. These are called **N-step estimators**.

$$\hat{A}_t^{(1)} = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{5.10}$$

$$\hat{A}_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t) \tag{5.11}$$

$$...$$

$$\hat{A}_t^{(N)} = \sum_{k=0}^{N-1} \gamma^k r_{t+k} + \gamma^N V(s_{t+N}) - V(s_t) \tag{5.12}$$

$$...$$

$$\hat{A}_t^{(\infty)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... - V(s_t) = G_t - V(s_t) \tag{5.13}$$

At $N = 1$ we are using a TD(0) approach, and at $N = \infty$ we are using a fully Monte-Carlo estimate.

5.4 Proximal Policy Optimization (PPO)

The vanilla policy gradient method faces two main challenges. First, distance in the parameter space does not equal distance in the policy space

$$\Pi = \{\pi : \pi \in \mathbb{R}^{|A| \times |S|}, \sum_a \pi_{sa} = 1, \pi_{sa} \geq 0\} \tag{5.14}$$

Practically, this means that we may be too sensitive to step size. A small change in parameters θ could result in a very large change in policy, which can make training unstable and can collapse performance. Second, there is very poor sample efficiency since each batch of data is discarded after only one gradient step. Ideally, we would want a method that directly respects the policy space and can take multiple gradient steps before gathering more data. We can begin by exploiting the relationship between the performance of two different policies. The *Performance Difference Lemma* defines the relative performance of policy π' over π in terms of advantages from π.

Proposition 5.2. ***Performance Difference Lemma***.

$$\begin{aligned} J(\pi') - J(\pi) &= \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi}(s_t,a_t)\right] && (5.15)\\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi'},a\sim\pi'}\left[A^{\pi}(s,a)\right] && (5.16)\end{aligned}$$

where $d^{\pi}(s)$ is the discounted future state distribution and $p(s_t = s|\pi)$ is the likelihood of being in state s at time t when following policy π

$$d^{\pi}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t p(s_t = s|\pi) \tag{5.17}$$

and $J(\pi)$ be the value of a policy under initial state distribution μ

Proof.

$$\begin{aligned} J(\pi') - J(\pi) &= \mathbb{E}_{s_0\sim\mu}\left[V^{\pi'}(s_0) - V^{\pi}(s_0)\right]\\ &= \mathbb{E}_{s_0\sim\mu,a_0\sim\pi'}\left[Q^{\pi'}(s_0,a_0) - Q^{\pi}(s_0,a_0) + Q^{\pi}(s_0,a_0) - V^{\pi}(s_0)\right]\\ &= \mathbb{E}_{s_0\sim\mu,a_0\sim\pi'}\left[Q^{\pi'}(s_0,a_0) - Q^{\pi}(s_0,a_0) + A^{\pi}(s_0,a_0)\right]\\ &= \mathbb{E}_{s_0\sim\mu,a_0\sim\pi'}\left[r(s_0,a_0) + \gamma\mathbb{E}_{s_1\sim\pi'}V^{\pi'}(s_1) - r(s_0,a_0) + \gamma\mathbb{E}_{s_1\sim\pi'}V^{\pi}(s_1) + A^{\pi}(s_0,a_0)\right]\\ &= \mathbb{E}_{s_0\sim\mu,a_0,s_1\sim\pi'}\left[\gamma\left(V^{\pi'}(s_1) - V^{\pi}(s_1)\right) + A^{\pi}(s_0,a_0)\right]\\ &= \mathbb{E}_{s_0\sim\mu,a_0,s_1,a_1,s_2\sim\pi'}\left[\gamma^2\left(V^{\pi'}(s_2) - V^{\pi}(s_2)\right) + \gamma A^{\pi}(s_1,a_1) + A^{\pi}(s_0,a_0)\right]\\ &= \mathbb{E}_{s_0\sim\mu,a_0,s_1,a_1,s_2,a_2,s_3\sim\pi'}\left[\gamma^3\left(V^{\pi'}(s_3) - V^{\pi}(s_3)\right) + \gamma^2 A^{\pi}(s_2,a_2) + \gamma A^{\pi}(s_1,a_1) + A^{\pi}(s_0,a_0)\right]\end{aligned}$$

Repeating the argument indefinitely yields

$$J(\pi') - J(\pi) = \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi}(s_t,a_t)\right]$$

We complete the result by rewriting the expectation over τ

$$\begin{aligned} J(\pi') - J(\pi) &= \mathbb{E}_{s\sim d^{\pi'},a\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi}(s,a)\right]\\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi'},a\sim\pi'}[A^{\pi}(s,a)]\end{aligned}$$

For intuition about the discounted future state distribution $d^{\pi}(s)$, consider our goal of rewriting

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\right] = \sum_s\sum_a d^{\pi}(s)\pi(a|s)r(s,a)$$

which only works if d^{π} is discounted.

Our idea is to use the performance difference lemma for policy improvement. That is, what if we iteratively take

$$\max_{\pi'} J(\pi') = \max_{\pi'} J(\pi') - J(\pi) = \max_{\pi'} \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^\pi(s_t, a_t)\right] \tag{5.18}$$

The problem is the expectation requires trajectories sampled from π' and not π. We can rewrite the relative policy performance as follows (note that this is an instance of *importance sampling*)

$$\begin{aligned} J(\pi') - J(\pi) &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi'}, a\sim\pi'}[A^\pi(s,a)] \\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi'}}\left[\sum_a \pi'(a|s)A^\pi(s,a)\right] \\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi'}}\left[\sum_a \frac{\pi'(a|s)}{\pi(a|s)}\pi(a|s)A^\pi(s,a)\right] \\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi'}}\mathbb{E}_{a\sim\pi}\left[\frac{\pi'(a|s)}{\pi(a|s)}A^\pi(s,a)\right] \end{aligned}$$

but this is still not practically usable because it requires $s \sim d^{\pi'}$. What if we simply approximated $d^\pi \approx d^{\pi'}$? We hope this is a reasonable approximation if the policies are not too different from each other. Let such a quantity be $\mathcal{L}_\pi(\pi')$ which we define as

$$\begin{aligned} \mathcal{L}_\pi(\pi') &:= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^\pi}\mathbb{E}_{a\sim\pi}\left[\frac{\pi'(a|s)}{\pi(a|s)}A^\pi(s,a)\right] && (5.19) \\ &= \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t \frac{\pi'(a|s)}{\pi(a|s)}A^\pi(s_t,a_t)\right] && (5.20) \\ &\approx J(\pi') - J(\pi) && (5.21) \end{aligned}$$

It turns out, we can quantify how tight the approximation is using the KL-divergence between π' and π and a constant factor C

$$|J(\pi') - J(\pi) - \mathcal{L}_\pi(\pi')| \leq C\sqrt{\mathbb{E}_{s\sim d^\pi}\left[D_{KL}(\pi'||\pi)[s]\right]} \tag{5.22}$$

That is, the closer π' is to π (KL-divergence close to zero), the better the approximation. Recall that the KL-divergence between discrete probability distributions P and Q is defined as

$$D_{KL}(P||Q) = \sum_x P(x)\log\frac{P(x)}{Q(x)} \tag{5.23}$$

Between policies π' and π we can define

$$D_{KL}(\pi'||\pi)[s] = \sum_a \pi'(a|s)\log\frac{\pi'(a|s)}{\pi(a|s)} \tag{5.24}$$

Proximal Policy Optimization (PPO) is a family of methods that do policy improvement on $\mathcal{L}_\pi(\pi')$ but also penalize policies from changing too much between updates (i.e. trying to enforce the KL-divergence constraint).

Algorithm 5.3 PPO with Adaptive KL Penalty

```
Initialize policy parameters $\theta_0$, initial KL penalty $\beta_0$, target KL divergence $\delta$

for iteration $k = 0, 1, ...$ until convergence do
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

    Compute the policy update by taking $K$ steps of minibatch SGD

        $\theta_{k+1} = \underset{\theta}{argmax}\, \mathcal{L}_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}(\theta||\theta_k)$
        $\bar{D}_{KL}(\theta||\theta_k) = \mathbb{E}_{s\sim d^{\pi_k}}[D_{KL}(\pi(\theta)||\pi(\theta_k))[s]]$

    If $\bar{D}_{KL}(\theta||\theta_k) \geq 1.5\delta$ then $\beta_{k+1} = 2\beta_k$
    Else If $\bar{D}_{KL}(\theta||\theta_k) \leq \delta/1.5$ then $\beta_{k+1} = \beta_k/2$
end for
```

Note that in contrast to the vanilla policy gradient algorithm, we can take multiple optimization steps in PPO before collecting more trajectories under the new policy. It is safe to do so since the $\beta_k \bar{D}_{KL}(\theta||\theta_k)$ term penalizes the objective when the distance between policies becomes too large. Some iterations in practice may violate the KL-constraint, but most do not. Also note that the initial choice of the KL penalty is not important since it adapts quickly.

In the variant of PPO with objective clipping, we clip the ratio $\pi_\theta(a_t|s_t)/\pi_{\theta_k}(a_t|s_t)$ which prevents the new policy π_θ from having incentive to go far away from the current policy π_{θ_k}. Specifically, we define the clipped objective function with hyperparameter ϵ as

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathbb{E}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T-1} min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}\hat{A}_t^{\pi_k}, clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k}\right)\right] \tag{5.25}$$

Algorithm 5.4 PPO with Clipped Objective

```
Initialize policy parameters $\theta_0$, clipping threshold $\epsilon$

for iteration $k = 0, 1, ...$ until convergence do
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

    Compute the policy update by taking $K$ steps of minibatch SGD

        $\theta_{k+1} = \underset{\theta}{argmax}\, \mathcal{L}_{\theta_k}^{CLIP}(\theta)$

    where

        $\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathbb{E}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T} min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}\hat{A}_t^{\pi_k}, clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k}\right)\right]$
end for
```

In practice, clipping seems to work at least as well as the KL-penalty, but is easier to implement.

5.5 Generalized Advantage Estimators

Recall the N-step advantage estimator

$$\hat{A}_t^{(N)} = \sum_{k=0}^{N-1} \gamma^k r_{t+k} + \gamma^N V(s_{t+N}) - V(s_t)$$

If we define

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{5.26}$$

then we can rewrite the N-step advantage estimators as

$$\begin{aligned}
\hat{A}_t^{(1)} &= \delta_t^V && (5.27)\\
\hat{A}_t^{(2)} &= \delta_t^V + \gamma\delta_{t+1}^V && (5.28)\\
&\dots\\
\hat{A}_t^{(N)} &= \sum_{\ell=0}^{N-1} \gamma^\ell \delta_{t+\ell}^V && (5.29)
\end{aligned}$$

The **Generalized Advantage Estimator (GAE)** is an exponentially-weighted average of all N-step estimators. That is,

$$\begin{aligned}
\hat{A}_t^{GAE(\gamma,\lambda)} &= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots\right)\\
&= (1-\lambda)\left(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \dots\right)\\
&= (1-\lambda)\left(\delta_t^V(1+\lambda+\lambda^2+\dots) + \gamma\delta_{t+1}^V(\lambda+\lambda^2+\dots) + \gamma^2\delta_{t+2}^V(\lambda^2+\lambda^3+\dots) + \dots\right)\\
&= (1-\lambda)\left(\frac{\delta_t^V}{1-\lambda} + \frac{\gamma\lambda\delta_{t+1}^V}{1-\lambda} + \frac{\gamma^2\lambda^2\delta_{t+2}^V}{1-\lambda} + \dots\right)\\
&= \sum_{\ell=0}^{\infty} (\gamma\lambda)^\ell \delta_{t+\ell}^V && (5.30)
\end{aligned}$$

Note how setting $\lambda = 0$ is the pure TD(0) estimate (high bias, low variance). In general, we want to choose $\lambda \in (0,1)$ to balance both bias and variance.

In PPO, we can use a truncated version of GAE, which is more practical because we only have to run a policy in the environment for T timesteps before updating the advantage estimate.

$$\hat{A}_t = \sum_{\ell=0}^{T-t-1} (\gamma\lambda)^\ell \delta_{t+\ell}^V \tag{5.31}$$

6 Imitation Learning

Imitation Learning is useful when it is easier for an expert to demonstrate the desired behavior rather than specifying a reward function or specifying a policy directly that mimics such behavior. For example, it may be easier for a human to demonstrate how to drive a car than to explicitly write down a reward function that results in smooth and safe driving. As before we interact with an environment with some state space, action space, and some dynamics/transition model. However, we have no explicitly provided reward function; instead, we have a set of teacher/expert demonstrations $(s_0, a_0, s_1, a_1, \dots)$ drawn from the expert policy π^*. In **behavior cloning**, the goal is to directly learn the expert policy using supervised learning. Alternatively, we can learn reward signals from the expert (**reward learning**). For example, in **inverse reinforcement learning**, the goal is to recover a reward function R from the expert. In **apprenticeship learning via inverse RL**, the goal is to use the recovered reward function to generate a good policy.

6.1 Behavior Cloning

In behavior cloning, we simply reduce the problem to standard supervised learning. That is, we fit a policy from the expert training examples

$$< s_t, a_t > \tag{6.1}$$

We can also use recurrent neural network, for example, using s_0, a_0, s_1, a_1, s_2 to predict a_2 and so on.

The main challenge that behavior cloning suffers from are compounding errors. While supervised learning assumes iid (s, a) training pairs which ignores temporal structure and assumes independent in time errors. However, this assumption is not true. Errors often can compound over time, and take the agent to a different part of the state space where the expert would have not gone. For example, imagine an agent learning to drive on a racetrack from a human driver. Small errors in each turn can compound which can take the agent off the track, but since the human stayed on the track the whole time, there are no demonstrations of what to do in this state. With no relevant training, there is no guarantee the agent will do anything reasonable (such as attempting to return to the track). This is a distribution mismatch and a fundamental problem in behavior cloning. That is, the state distribution visited in training (i.e. by the expert) follow $s_t \sim \mathcal{D}_{\pi^*}$ but the state distribution visited by the agent in testing $s_t \sim \mathcal{D}_{\pi_\theta}$ can be different! In supervised learning, we assume that the train and test data draws from the same distribution.

The DAgger algorithm helps solve the distribution mismatch problem by providing expert feedback during training, and giving a change for the agent to learn from its mistakes. We want additional labels of the expert action along the path that was taken by the agent. For example, in the racetrack example using DAgger, the expert will provide additional guidance in what do in the off-track state.

Algorithm 6.1 DAgger: Dataset Aggregation

```
Initialize $\mathcal{D} = \{\}$, policy $\hat{\pi}_1$
for $i = 1, 2, ..., N$ until convergence do
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$ where $\beta_i$ is a decreasing coefficient
    Sample $T$-step trajectories using $\pi_i$
    Get dataset $\mathcal{D}_i = \{(s, \pi^*(s)\}$ of states visited by $\pi_i$ and actions given by the expert
    Aggregate dataset $\mathcal{D} = \mathcal{D} \cup \mathcal{D}_i$ and train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$
end for
return Best $\hat{\pi}_i$ on validation
```

This obtains a stationary and deterministic policy with good performance under its induced state distribution. However, the expert is fully in the training loop and needs to constantly provide feedback, which is a key limitation of this method.

6.2 Reward Learning

Consider when a reward is linear over features, that is

$$R(s) = \phi^T f(s) \tag{6.2}$$

where $\phi \in \mathbb{R}^n$ and $f : S \to \mathbb{R}^n$ is some feature representation. Then the resulting value function for a policy π is

$$V^\pi(s_0) = \mathbb{E}_{s\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0\right] = \mathbb{E}_{s\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \phi^T f(s_t) \mid s_0\right] = \phi^T \mu(\pi) \tag{6.3}$$

where $\mu(\pi)$ is the discounted weighted frequency of state features under π starting in s_0.

$$\mu(\pi) = \mathbb{E}_{s\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t f(s_t) \mid s_0\right] \tag{6.4}$$

Since the optimal policy has the highest value, we have that

$$\phi^T\mu(\pi^*) \geq \phi^T\mu(\pi)\ \forall\pi \tag{6.5}$$

Abbeel and Ng (2004) provides the result that for a policy π to be guaranteed to perform as well as the expert π^*, it is sufficient if its $\mu(\pi)$ matches the expert $\mu(\pi^*)$ (this is called feature matching). More precisely, if $||\mu(\pi)-\mu(\pi^*)||_1 \leq \epsilon$ then for all ϕ satisfying $||\phi||_\infty \leq 1$ we have $|\phi^T\mu(\pi)-\phi^T\mu(\pi^*)| \leq \epsilon$. However, there are infinitely many policies that can match feature counts. For example, let's say an optimal taxi driver crosses on average 3 bridges per trip. Then to match features, we would want our modeled policy to also cross three bridges per trip - but there can be many different ways to do this, so which one should we choose? In addition, there are also infinitely many reward functions that result in the same optimal policy (imagine for instance, simply scaling up the reward function by a constant factor, or when $\phi = 0$ and all policies have the same zero reward). This is an ambiguous and ill-posed problem.

One way we can approach this is **Maximum Entropy Inverse Reinforcement Learning**. Recall that a policy induces a distribution over trajectories, which broken into its states and actions can be represented by

$$p(\tau) = p(s_0)\prod_t p(a_t|s_t)p(s_{t+1}|s_t,a_t) \tag{6.6}$$

The idea is to choose a policy that induces a distribution over trajectories with the maximum entropy while still being consistent with what we know about the observed expert data. This follows from the principle of max entropy which states that the probability distribution which best represents the current state of knowledge is the one with the largest entropy, given the constraints of precisely state prior data. Intuitively, we are being honest with what we know, and choosing to maximize uncertainty for anything beyond our stated constraints (which in this case is being consistent with the expert data).

Recall that the entropy of a distribution $p(s)$ is

$$-\sum_{s'} p(s=s')\log p(s=s') \tag{6.7}$$

For a general reward function r_ϕ (not necessarily linear), we want to find a policy π that induces a distribution over trajectories $p(\tau)$ which has the *same expected reward* as the expert's demonstrations $\hat{P}(\tau)$ and is a valid probability distribution (sums to one). Maximizing entropy given these constraints gives

$$\underset{p(\tau)}{max} -\sum_\tau p(\tau)\log p(\tau)\ s.t. \sum_\tau p(\tau)r_\phi(\tau) = \sum_\tau \hat{P}(\tau)r_\phi(\tau),\ \sum_\tau p(\tau) = 1 \tag{6.8}$$

Writing the Langrage multiplier for this constrained optimization problem

$$\mathcal{L}(p,\lambda) = \sum_\tau p(\tau)\log p(\tau) + \lambda_1\left(\sum_\tau \hat{P}(\tau)r_\phi(\tau) - p(\tau)r_\phi(\tau)\right) + \lambda_0\left(\sum_\tau p(\tau) - 1\right) \tag{6.9}$$

Taking the derivative w.r.t. a specific $p(\tau)$ and setting it to zero we have

$$\frac{\partial\mathcal{L}}{\partial p(\tau)} = \log p(\tau) + p(\tau)\frac{1}{p(\tau)} - \lambda_1 r_\phi(\tau) + \lambda_0 = 0 \tag{6.10}$$

Rearranging

$$p(\tau) = e^{-1+\lambda_1 r_\phi(\tau)-\lambda_0} \tag{6.11}$$

So now we know the structure of the functional form of the distribution over trajectories that maximizes entropy is $p(\tau) \propto e^{r_\phi(\tau)}$. Adding a normalizing constant so the distribution sums to one we finally obtain

$$p(\tau_i|\phi) = \frac{1}{\sum_\tau e^{r_\phi(\tau)}}e^{r_\phi(\tau_i)} \tag{6.12}$$

Jaynes (1957) provides that maximizing the entropy of the distribution over the paths subject to the feature constraints from observed (expert) data implies we maximize the likelihood of the observed data under the maximum entropy (exponential family) distribution. That is, we can find ϕ using maximum likelihood.

Let $\tau* \in \mathcal{D}$ be the observed trajectories over the expert data

$$\begin{aligned} \underset{\phi}{argmax} \log \prod_{\tau^* \in \mathcal{D}} p(\tau^*|\phi) &= \underset{\phi}{argmax} \sum_{\tau^* \in \mathcal{D}} \log \left(\frac{e^{r_\phi(\tau^*)}}{\sum_\tau e^{r_\phi(\tau)}} \right) \\ &= \underset{\phi}{argmax} \sum_{\tau^* \in \mathcal{D}} \left(r_\phi(\tau^*) - \log \sum_\tau e^{r_\phi(\tau)} \right) \\ &= \underset{\phi}{argmax} \sum_{\tau^* \in \mathcal{D}} r_\phi(\tau^*) - |\mathcal{D}| \log \sum_\tau e^{r_\phi(\tau)} \end{aligned} \tag{6.13}$$

Let the objective be $J(\phi)$. Then the gradient (to use in gradient based optimization) is

$$\begin{aligned} \nabla_\phi J(\phi) &= \sum_{\tau^* \in \mathcal{D}} \nabla_\phi r_\phi(\tau^*) - |D| \frac{1}{\sum_\tau e^{r_\phi(\tau)}} \sum_\tau e^{r_\phi(\tau)} \nabla_\phi r_\phi(\tau) \\ &= \sum_{\tau^* \in \mathcal{D}} \nabla_\phi r_\phi(\tau^*) - |D| \sum_\tau p(\tau|\phi) \nabla_\phi r_\phi(\tau) \end{aligned} \tag{6.14}$$

We can rewrite this using states instead of trajectories for some dynamics model $\mathcal{T}$ and state visitation frequency $p(s|\phi, \mathcal{T})$

$$\nabla_\phi J(\phi) = \sum_{s \in \tau^* \in \mathcal{D}} \nabla_\phi r_\phi(s) - |\mathcal{D}| \sum_s p(s|\phi, \mathcal{T}) \nabla_\phi r_\phi(s) \tag{6.15}$$

For a linear reward $r_\phi(s) = \phi^T f(s)$ the gradient simplifies to

$$\nabla_\phi J(\phi) = \sum_{s \in \tau^* \in \mathcal{D}} f(s) - |\mathcal{D}| \sum_s p(s|\phi, \mathcal{T}) f(s) \tag{6.16}$$

The initial formulation of the max entropy approach (which assumed a linear reward) by Ziebart et al. (2008) calculates the state visitation frequency in the tabular MDP case given the policy π by

- Setting $\mu_1(s) = p(s_{initial})$
- For $t = 1, ..., T$ compute $\mu_{t+1}(s) = \sum_a \sum_{s'} \mu_t(s') \pi(a|s') p(s|s', a)$
- Summing over timesteps $p(s|\phi, \mathcal{T}) = \sum_t \mu_t(s)$

where $\mu_t(s)$ is the state visitation probability at time t, and $p(s|s', a)$ is the transition probability given by $\mathcal{T}$.

To summarize, we can learn a reward function from expert data and fit a policy using the max entropy approach (assuming known dynamics model and linear rewards) by following

Algorithm 6.2 Inverse RL with Max Entropy

```
Input expert demonstrations $\mathcal{D}$, dynamics model $\mathcal{T}$
Initialize $\phi$ for linear rewards model $r_\phi = \phi^T f(s)$
for $i = 1, 2, ...$ until convergence do
    Compute optimal policy $\pi_i(a|s)$ given $r_\phi$ (e.g. using value iteration)
    Compute state visitation frequencies $p(s|\phi, \mathcal{T})$
    Compute the gradient on the reward model
        $\nabla_\phi J(\phi) = \sum_{s \in \tau^* \in \mathcal{D}} f(s) - |\mathcal{D}| \sum_s p(s|\phi, \mathcal{T}) f(s)$
    Update $\phi$ using one gradient step
end for
```

The maximum entropy approach has been hugely influential. Since its initial formulation, Finn et al. (2016) showed how to use general reward functions (non-linear) and removed the need to know the dynamics model.

6.3 Reinforcement Learning from Human Feedback (RLHF)

Consider the problem of encoding human preferences into a reward function. For example, when finetuning a large language model, how do we train it to produce responses that people find useful? It will be difficult to hand write a reward function that describes what a good response looks like. Instead, what if we can learn a reward function that encodes human preference by training on pairwise comparisons. Pairwise comparisons are useful because it is often easier for people to compare two different options than to provide a scalar reward (i.e. rating from one to ten). Scalar rewards from humans are also poorly calibrated to each other - ratings from one to ten could mean different things to different people.

We can model pairwise comparisons probabilistically using the **Bradley-Terry Model**. Let $\mathcal{D} = \{x_i, y_i^w, y_i^\ell\}$ be a labeled dataset where x are some state variables (i.e. a prompt to a LLM), y^w is the preferred or winning response (action), and y^ℓ is the dispreferred or losing response (action). Then we model the probability of preferring the winning response using a latent (unobserved) reward function $r_\phi(x, y)$ parameterized by ϕ.

$$p(y^w > y^\ell|x) = \frac{\exp(r_\phi(x, y^w))}{\exp(r_\phi(x, y^w)) + \exp(r_\phi(x, y^\ell))} = \sigma\left(r_\phi(x, y^w) - r_\phi(x, y^\ell)\right) \tag{6.17}$$

where $\sigma(x) = 1/(1+\exp(-x))$ is the sigmoid/logistic function. We can fit the reward function by minimizing the negative log likelihood, that is

$$\hat{\phi} = \underset{\phi}{argmin} \sum_{(x_i, y_i^w, y_i^\ell) \in \mathcal{D}} - \log p\left(y_i^w > y_i^\ell|x\right) = \underset{\phi}{argmin} \sum_{(x_i, y_i^w, y_i^\ell) \in \mathcal{D}} - \log \sigma\left(r_\phi(x_i, y_i^w) - r_\phi(x_i, y_i^\ell)\right) \tag{6.18}$$

We can also do this by comparing pairwise trajectories. Consider two trajectories τ^w and τ^ℓ. We can model the probability that a human prefers $\tau^w > \tau^\ell$ as

$$p(\tau^w > \tau^\ell) = \frac{\exp\left(\sum_{i=0}^{T-1} r_\phi(s_i^w, a_i^w)\right)}{\exp\left(\sum_{i=0}^{T-1} r_\phi(s_i^w, a_i^w)\right) + \exp\left(\sum_{i=0}^{T-1} r_\phi(s_i^\ell, a_i^\ell)\right)} \tag{6.19}$$

Next, we can simply use PPO to train a policy π_θ that optimizes for our learned reward model. Note in the case of finetuning a LLM, we may want to add an additional penalty term so that the finetuned model π_θ stays relatively close (in KL-divergence) to the original (i.e. pretrained) model π_{ref}. In this case, the objective becomes for some hyperparameter β

$$\underset{\pi_\theta}{max}\, \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta}[r_\phi(x, y)] - \beta D_{KL}[\pi_\theta(y|x)||\pi_{ref}(y|x)] \tag{6.20}$$

We want to keep the KL-divergence to the reference model small since the reward model is trained on pairwise comparisons using the outputs of the original reference model. If we drift too far (the model changes too much), the reward model could be too far out of distribution and no longer reliable without refitting. The following figure summarizes the RLHF pipeline for finetuning a large language model from Ouyang et al. (2022).

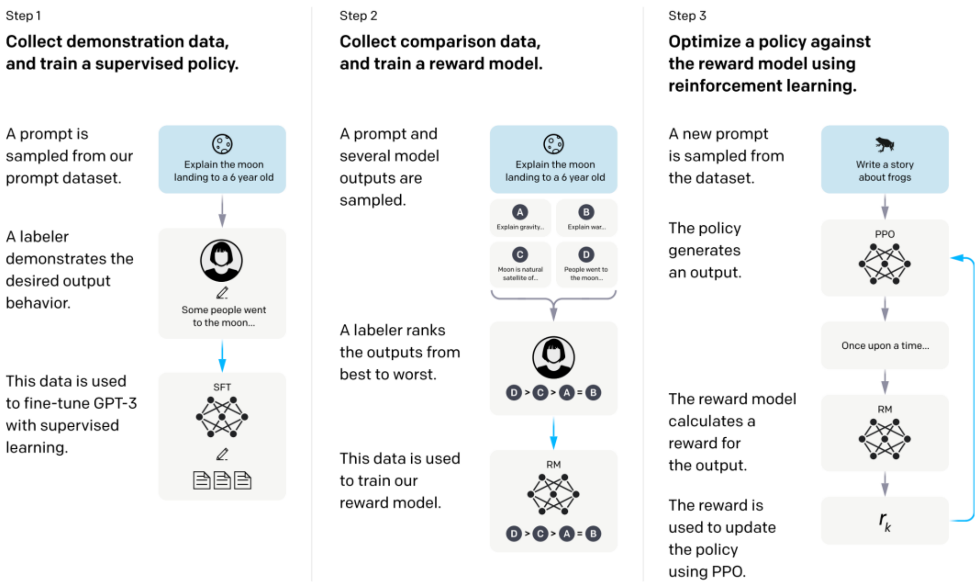
Step 1
Collect demonstration data, and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain the moon landing to a 6 year old
A labeler demonstrates the desired output behavior.
Some people went to the moon...
This data is used to fine-tune GPT-3 with supervised learning.
SFT
Step 2
Collect comparison data, and train a reward model.
A prompt and several model outputs are sampled.
Explain the moon landing to a 6 year old
A
Explain gravity...
B
Explain war...
C
Moon is natural satellite of...
D
People went to the moon...
A labeler ranks the outputs from best to worst.
D > C > A = B
This data is used to train our reward model.
RM
D > C > A = B
Step 3
Optimize a policy against the reward model using reinforcement learning.
A new prompt is sampled from the dataset.
Write a story about frogs
The policy generates an output.
PPO
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Figure 5: Training language models to follow instructions with RLHF

6.4 Direct Preference Optimization (DPO)

Direct Preference Optimization is an alternative to RLHF by Rafailov et al. (2024) that encodes human preferences using an implicit reward model. It turns out that the optimal policy from the RLHF objective as stated in Eq. 6.20 has a closed-form solution

$$\pi^*(y|x) = \frac{1}{Z(x)}\pi_{ref}(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right) \tag{6.21}$$

where

$$Z(x) = \sum_y \pi_{ref}(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right) \tag{6.22}$$

This is computationally infeasible (for a LLM), since $Z(x)$ requires a sum over all possible responses y given x. Instead, we can rearrange Eq. 6.22 to write the reward function as a function of the optimal policy

$$r(x,y) = \beta\log\frac{\pi^*(y|x)}{\pi_{ref}(y|x)} + \beta\log Z(x) \tag{6.23}$$

Here, the reward is higher when the optimal policy prefers a response more than the reference model $\pi^*(y|x) > \pi_{ref}(y|x)$. What if we borrow this structure for the policy we are training (π_θ) rather than the optimal policy π^*? Let

$$r_{\pi_\theta}(x,y) = \beta\log\frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)} + \beta\log Z(x) \tag{6.24}$$

Recall that in the Bradley-Terry Model, we fit the reward function using a negative log likelihood loss

$$\mathcal{L}_R(r) = -\mathbb{E}_{(x,y^w,y^\ell)\in\mathcal{D}}[\log\sigma\left(r(x,y^w) - r(x,y^\ell)\right)] \tag{6.25}$$

Substituting Eq. 6.24 in 6.25 yields a loss function on policies.

$$\mathcal{L}_{DPO}(\pi_\theta;\pi_{ref}) = -\mathbb{E}_{(x,y^w,y^\ell)\in\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y^w|x)}{\pi_{ref}(y^w|x)} - \beta\log\frac{\pi_\theta(y^\ell|x)}{\pi_{ref}(y^\ell|x)}\right)\right] \tag{6.26}$$

In the spirit of Eq. 6.23, the loss is lower when π_θ prefers the winning response more than the reference model, and prefers the losing response less than the reference model. When we optimize π_θ based on $\mathcal{L}_{DPO}$

we are pushing the model in this direction, which is directly optimizing the policy based on human preferences. Since the reward function is implicit in the DPO objective, we remove the need to train a separate reward model which saves computation. In the case of finetuning an LLM, the authors show that DPO performs as well as classic RLHF with PPO.

7 More Exploration

In this section, we will discuss alternative strategies to exploration, with the larger goal of performing RL with greater data efficiency. In particular, we introduce the approach of optimism under uncertainty and choosing actions by constructing upper confidence bounds (UCB), as well as the Bayesian approach via Thompson Sampling.

7.1 Multi-armed Bandits

We begin by discussing the simpler setting of multi-armed bandits, where an agent can take a known set of actions from the action space A. For each arm, there is some unknown probability distribution over rewards $\mathcal{R}^a(r) = p(r|a)$ (unlike MDP's, there is no sequential dependence and rewards only depend on the action taken and not the state). At each timestep t the agent selects an action $a_t \in A$, with the goal of maximizing cumulative reward $\sum_{t=1}^{T} r_t$. Let the action-value be the expected reward for a given action,

$$Q(a) = \mathbb{E}[r|a] \tag{7.1}$$

and consider algorithms that estimate $Q(a)$ using Monte-Carlo evaluation where $N_t(a)$ is the number of times action a has been taken up until time t.

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{i=1}^{t} r_i \mathbb{1}(a_i = a) \tag{7.2}$$

The greedy algorithm will select the action with the highest value $\underset{a\in A}{argmax}\ \hat{Q}_t(a)$.

Consider a toy example of treating broken toes with 3 possible actions: 1. surgery a^1, 2. bandaging a^2, 3. doing nothing a^3. We observe a Bernoulli reward, that is, we get +1 reward if the broken toe is healed, and +0 if it is not. Each arm corresponds to a Bernoulli parameter $\theta_1 = 0.95$ for surgery, $\theta_2 = 0.9$ for bandaging, and $\theta_3 = 0.1$ for doing nothing (so surgery is the optimal action). Note how a multi-armed bandit is a better fit to this problem compared to a MDP since there is no sequential time dependence - only actions taken. Let's say after we sampled each arm once we get $\hat{Q}(a^1) = 0$ (the patient did not get better after surgery), $\hat{Q}(a^2) = 1$, and $\hat{Q}(a^3) = 0$. The greedy algorithm will continue to select a^2 since it has the highest estimate $\hat{Q}$, but in doing so, it misses the optimal solution of taking arm a^1. In this way, the greedy approach can get stuck at a suboptima.

Let $V^* = Q(a^*) = \underset{a\in A}{max}\, Q(a)$ be the optimal value. Then **regret** is defined as the opportunity loss for one timestep

$$I_t = \mathbb{E}[V^* - Q(a_t)] \tag{7.3}$$

and let the **total regret** by the cumulative opportunity loss (note that our goal of maximizing cumulative reward is the same as minimizing total regret).

$$L_T = \mathbb{E}\left[\sum_{t=1}^{T} V^* - Q(a_t)\right] \tag{7.4}$$

Let the **gap** Δ_a for some action a be the difference in value between action a and optimal action a^* (this is akin to the advantage)

$$\Delta_a = V^* - Q(a) \tag{7.5}$$

Note that total regret is a function of gaps and counts

$$\begin{aligned} L_T &= \mathbb{E}\left[\sum_{t=1}^{T} V^* - Q(a_t)\right] \\ &= \sum_{a \in A} \mathbb{E}[N_T(a)](V^* - Q(a)) \\ &= \sum_{a \in A} \mathbb{E}[N_T(a)]\Delta_a \end{aligned} \tag{7.6}$$

In the worst case, a greedy method can be linear in regret when it is stuck at a suboptima (i.e. making a mistake at every timestep). How about an $\epsilon - greedy$ approach? Recall that an algorithm is $\epsilon - greedy$ when it takes the greedy action $\underset{a \in A}{argmax}\ \hat{Q}_t(a)$ with probability $1-\epsilon$, or an action uniformly at random with probability ϵ? With fixed ϵ, we can make a suboptimal decision (in exploration) ϵ fraction of timesteps, so regret is still scaling linearly, although at a slower rate than the pure greedy approach. This highlights the importance of decaying ϵ with timestep as we saw in Q-learning, which results in sublinear regret.

Lai and Robbins (1985) provides that the lower bound of total regret is asymptotically at least logarithmic in timesteps, and is function of the gap Δ_a and the similarity between reward distributions $D_{KL}(\mathcal{R}^a || \mathcal{R}^{a^*})$.

$$\lim_{T \to \infty} L_T \geq \log T \sum_{a | \Delta_a > 0} \frac{\Delta_a}{D_{KL}(\mathcal{R}^a || \mathcal{R}^{a^*})} \tag{7.7}$$

Consider an alternative approach: optimism in the face of uncertainty. The basic idea is to choose actions that we think might have a high value, because we either actually get a high reward, or if we are wrong, we will learn something from it (i.e. reduce the uncertainty of our estimate). For example, consider estimating an upper confidence bound $U_t(a)$ for each action, such that $Q(a) \leq U_t(a)$ with high probability. We can select the action that maximizes the **Upper Confidence Bound (UCB)**

$$a_t = \underset{a \in A}{argmax}\ U_t(a) \tag{7.8}$$

For motivation, consider **Hoeffding's Inequality** which states that for $X_1, ..., X_n$ iid random variables in $[0, 1]$ and sample mean $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ we have

$$P(|\mathbb{E}X - \bar{X}_n| > u) \leq 2\exp(-2nu^2) \tag{7.9}$$

If we set $\delta = 2\exp(-2nu^2)$ then u can be rewritten as $u = \sqrt{\frac{\log(2/\delta)}{n}}$ so we have

$$P\left(|\mathbb{E}X - \bar{X}_n| > \sqrt{\frac{\log(2/\delta)}{n}}\right) \leq \delta \tag{7.10}$$

which we can rearrange into a confidence interval containing the true mean

$$\bar{X}_n - \sqrt{\frac{\log(2/\delta)}{n}} \leq \mathbb{E}X \leq \bar{X}_n + \sqrt{\frac{\log(2/\delta)}{n}} \tag{7.11}$$

with probability at least $1-\delta$. Note how this interval shrinks for increasing n (sampling more from an arm reduces uncertainty). We can apply the same idea of constructing an upper confidence bound (UCB) to our estimates of the action value. For example, we can select

$$a_t = \underset{a \in A}{armax}\left(\hat{Q}(a) + \sqrt{\frac{2\log(\frac{1}{\delta})}{N_t(a)}}\right) \tag{7.12}$$

For this UCB algorithm, it turns out that any suboptimal arm $a \neq a^*$ is pulled for some constant C at most

$$\mathbb{E}N_T(a) \leq C\frac{\log(\frac{1}{\delta})}{\Delta_a^2} + \frac{\pi^2}{3} + 1 \tag{7.13}$$

with regret bounded by

$$L_T = \sum_a \mathbb{E}[N_T(a)]\Delta_a \leq \sum_a C\frac{\log T}{\Delta_a} + |A|\left(\frac{\pi^2}{3} + 1\right) \tag{7.14}$$

So UCB achieves regret upper bounded logarithmically with timestep (and matching the lower bound), rather than the worst linear case compared to the greedy or $\epsilon - greedy$ approach! As actions are visited more, the uncertainty we have in value estimate is reduced and the upper bound shrinks, which encourages us to explore areas with potentially greater uncertainty that have higher upper bounds. Eventually, we will converge to the optimal action after sufficient exploration.

As a subtly, note that for a fixed number of time steps T and confidence level δ we can set

$$\delta = \frac{\delta}{T|A|} \tag{7.15}$$

noting that for some arbitrary events E_i we have the union bound. This is akin to the Bonferroni correction for multiple testing.

$$P(\cup E_i) \leq \sum_i P(E_i) \tag{7.16}$$

7.2 Bayesian Exploration

In the Bayesian approach, we can maintain a prior over the unknown parameters on the reward distribution, and iteratively update the prior as observed reward data come in. Let θ_a parameterize the reward distribution for arm a, and let $p(\theta_a)$ be the prior over θ_a. If we pull arm a and observe reward r_a, we can apply Bayes Rule to obtain the posterior update to $p(\theta_a)$.

$$p(\theta_a|r_a) = \frac{p(r_a|\theta_a)p(\theta_a)}{p(r_a)} = \frac{p(r_a|\theta_a)p(\theta_a)}{\int_{\theta_a} p(r_a|\theta_a)p(\theta_a)d\theta_a} \tag{7.17}$$

For each iteration, we can sample θ_a from the prior for each arm, compute the action-value function for the selected θ_a, and select the action with the highest action value. This approach is called **Thompson Sampling**. As more data is collected, the uncertainty in the posterior is reduced and Thompson Sampling naturally becomes more greedy, similar to the shrinking confidence bounds in UCB.

Algorithm 7.1 Thompson Sampling

```
Initialize prior over each arm $p(\mathcal{R}_a)$
for $t = 0, ...$ until convergence do
    For each arm, sample a reward distribution $\mathcal{R}_a$
    Based on the sampled reward distributions, choose action $a_t = argmax_{a \in A}\ Q(a)$
    Observe reward $r_t$
    Update posterior $p(\mathcal{R}_a|r_t)$ using Bayes Rule
end for
```

It turns out that Thompson Sampling is approximating probability matching, where the policy is defined by the posterior probability that an action a is the optimal action. For some history h_t

$$\pi(a|h_t) = p(Q(a) > Q(a'),\ \forall a' \neq a|h_t) \tag{7.18}$$

This is often difficult to compute analytically, so Thompson Sampling approximates this by sampling from the posterior instead noting that

$$p(Q(a) > Q(a'),\ \forall a' \neq a | h_t) = \mathbb{E}_{\mathcal{R}|h_t}\left[\mathbb{1}\{a = \underset{a \in A}{argmax}\, Q(a)\}\right] \tag{7.19}$$

Empirically, Thompson Sampling performs well. Additionally, in cases where rewards are delayed and not immediately observed, Thompson Sampling may outperform UCB. For UCB in this scenario, the upper confidence cannot be updated until the rewards are observed, so each step can be stuck at a suboptimal action until the reward is observed. In contrast, Thompson Sampling naturally handles uncertainty when sampling from the posterior, and can promote stochastic exploration while waiting for the reward signal.

7.3 UCB for MDP's

We can extend the UCB approach from multi-armed bandits to the tabular MDP setting, for example, in Model-Based Interval Estimation with Exploration Bonus (MBIE-EB) by Strehl and Littman (2008). Note how this algorithm resembles Q-learning, except there is an exploration bonus term $\beta/\sqrt{n_{sa}(s,a)}$ which is on the order of $\log(1/\delta)$ (just like our UCB algorithm in the multi-armed bandit case) that promotes exploration for state action pairs with higher uncertainty.

Algorithm 7.2 MBIE-EB

```
Given $\epsilon, \delta, m, s_0$
$\beta = (1/(1-\gamma))\sqrt{0.5\log(2|S||A|m/\delta)}$
$n_{sas}(s,a,s') = 0, n_{sa}(s,a) = 0\ \forall s, s' \in S, a \in A$
$\hat{R}(s,a) = 0, \hat{Q}(s,a) = 1/(1-\gamma)\ \forall s \in S, a \in A$

for $t = 0, ...$ until convergence do
    Choose action
        $a_t = \underset{a \in A}{argmax}\ \hat{Q}(s_t, a)$
    Observe reward $r_t$ and next state $s_{t+1}$
    Update counts
        $n_{sa}(s_t,a_t) = n_{sa}(s_t,a_t) + 1, n_{sas}(s_t,a_t,s_{t+1}) = n_{sas}(s_t,a_t,s_{t+1}) + 1$
    Update models
        $\hat{R}(s_t,a_t) = \frac{\hat{R}(s_t,a_t)(n_{sa}(s_t,a_t)-1) + r_t}{n_{sa}(s_t,a_t)},\ \hat{p}(s'|s_t,a_t) = \frac{n_{sas}(s_t,a_t,s')}{n_{sa}(s_t,a_t)} \forall s' \in S$
    Update $\hat{Q}(s,a)$ for all $(s,a)$
        $\hat{Q}(s,a) = \hat{R}(s,a) + \gamma\sum_{s' \in S}\hat{p}(s'|s,a)\max_{a'}\ \hat{Q}(s',a') + \frac{\beta}{\sqrt{n_{sa}(s,a)}} \forall s \in S, a \in A$
end for
```

7.4 Thompson Sampling for MDP's

Similarly, we can extend the Thompson Sampling approach to tabular MDP's. Osband et al. (2013) provides Posterior Sampling for Reinforcement Learning (PSRL) which is as follows

Algorithm 7.3 Posterior Sampling for Reinforcement Learning (PSRL)

```
Initialize prior over dynamics and rewards models for all $(s, a)$ pairs, $p(\mathcal{R}_{sa}), p(\mathcal{T}(s'|s, a))$

for $k = 1, ..., K$ episodes do
    Sample a MDP $\mathcal{M}$:
    for each $(s, a)$ pair do
        Sample a dynamics model $\mathcal{T}(s'|s, a)$
        Sample a reward model $\mathcal{R}(s, a)$
    end for

    Compute optimal value (i.e. via value iteration) $Q^*_{\mathcal{M}}$ for the sampled MDP $\mathcal{M}$

    for $t = 1, ..., T$ do
        Choose action
            $$a_t = \underset{a \in A}{argmax}\ Q^*_{\mathcal{M}}(s_t, a)$$
        Observe reward $r_t$ and next state $s_{t+1}$
        Update posterior using Bayes rule
            $$p(\mathcal{R}_{s_t a_t}|r_t),\ p(\mathcal{T}(s'|s_t, a_t)|s_{t+1})$$
    end for
end for
```

References

Abbeel, P., & Ng, A. Y. (2004). *Apprenticeship learning via inverse reinforcement learning* (tech. rep.).

Finn, C., Levine, S., & Abbeel, P. (2016). *Guided cost learning: Deep inverse optimal control via policy optimization* (tech. rep.).

Jaynes, E. T. (1957). Information theory and statistical mechanics. *Physical Review*, *106*, 620–630. https://doi.org/10.1103/PhysRev.106.620

Lai, T., & Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, *6*, 4–22. https://doi.org/10.1016/0196-8858(85)90002-8

Osband, I., Roy, B. V., & Russo, D. (2013). *(more) efficient reinforcement learning via posterior sampling* (tech. rep.).

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. http://arxiv.org/abs/2203.02155

Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., & Finn, C. (2024). Direct preference optimization: Your language model is secretly a reward model. http://arxiv.org/abs/2305.18290

Strehl, A. L., & Littman, M. L. (2008). An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, *74*, 1309–1331. https://doi.org/10.1016/j.jcss.2007.08.009

Sutton, R. S., & Barto, A. G. (2020). *Reinforcement learning : An introduction*. The MIT Press.

Ziebart, B. D., Maas, A., Bagnell, J. A., & Dey, A. K. (2008). *Maximum entropy inverse reinforcement learning* (tech. rep.). www.aaai.org