Monte-Carlo Planning: Policy Improvement

Alan Fern
Monte-Carlo Planning Outline

- Single State Case (multi-armed bandits)
  - A basic tool for other algorithms

- Monte-Carlo Policy Improvement
  - Policy rollout
  - Policy Switching

- Monte-Carlo Tree Search
  - Sparse Sampling
  - UCT and variants
Policy Improvement via Monte-Carlo

• Now consider a very large multi-state MDP.
• Suppose we have a simulator and a non-optimal policy
  • E.g. policy could be a standard heuristic or based on intuition
• Can we somehow compute an improved policy?

World Simulator

+ Base Policy

Real World

State + reward

action
Recall: Policy Improvement Theorem

\[ Q_\pi(s, a) = R(s, a) + \beta \sum_{s'} T(s, a, s') \cdot V_\pi(s') \]

- The Q-value function of a policy gives expected discounted future reward of starting in state \( s \), taking action \( a \), and then following policy \( \pi \) thereafter.

- Define: \( \pi'(s) = \arg \max_a Q_\pi(s, a) \)

- **Theorem [Howard, 1960]**: For any non-optimal policy \( \pi \) the policy \( \pi' \) is a strict improvement over \( \pi \).

- Computing \( \pi' \) amounts to finding the action that maximizes the Q-function of \( \pi \)
  - Can we use the bandit idea to solve this?
Policy Improvement via Bandits

- **Idea**: define a stochastic function $\text{SimQ}(s, a, \pi)$ that we can implement and whose expected value is $Q_\pi(s, a)$

- Then use Bandit algorithm to select (approx) best action

How to implement SimQ?
Q-value Estimation

- SimQ might be implemented by simulating the execution of action \( a \) in state \( s \) and then following \( \pi \) thereafter
  - But for infinite horizon problems this would never finish
  - So we will approximate via finite horizon

- The \( h \)-horizon Q-function \( Q_\pi(s,a,h) \) is defined as:
  expected total discounted reward of starting in state \( s \), taking action \( a \), and then following policy \( \pi \) for \( h-1 \) steps

- The approximation error decreases exponentially fast in \( h \)

\[
\left| Q_\pi(s,a) - Q_\pi(s,a,h) \right| \leq \beta^h V_{\text{max}}
\]

\[
V_{\text{max}} = \frac{R_{\text{max}}}{1 - \beta}
\]
Policy Improvement via Bandits

- **Refined Idea**: Define a stochastic function $\text{SimQ}(s, a, \pi, h)$ that we can implement, whose expected value is $Q_\pi(s, a, h)$

- Use Bandit algorithm to select (approx) best action

**How to implement SimQ?**
Policy Improvement via Bandits

SimQ(s,a,\pi,h)

\[
\begin{align*}
    r &= R(s,a) \\
    s &= T(s,a) \\
    \text{for } i = 1 \text{ to } h-1 \\
    r &= r + \beta^i R(s, \pi(s)) \\
    s &= T(s, \pi(s))
\end{align*}
\]

Return r

- Simply simulate taking a in s and following policy for h-1 steps, returning discounted sum of rewards
- Expected value of SimQ(s,a,\pi,h) is Q_{\pi}(s,a,h) which can be made arbitrarily close to Q_{\pi}(s,a) by increasing h
Policy Improvement via Bandits

SimQ(s,a,π,h)

\[
\begin{align*}
  r &= R(s,a) \\
  s &= T(s,a) \\
  \text{for } i = 1 \text{ to } h-1 \\
  &\quad r = r + \beta^i R(s, \pi(s)) \\
  s &= T(s, \pi(s)) \\
\end{align*}
\]

Return r

Simulate a in s

Simulate h-1 steps of policy

Trajectory under \( \pi \)

Sum discount rewards = SimQ(s,a_1,\pi,h)

Sum discount rewards = SimQ(s,a_2,\pi,h)

Sum discount rewards = SimQ(s,a_k,\pi,h)
Policy Improvement via Bandits

- **Refined Idea**: define a stochastic function $\text{SimQ}(s,a,\pi,h)$ that we can implement, whose expected value is $Q_\pi(s,a,h)$

- Use Bandit algorithm to select (approx) best action

Which bandit objective/algorithm to use?
Traditional Approach: Policy Rollout

UniformRollout[π,h,w](s)

1. For each $a_i$ run $\text{SimQ}(s,a_i,\pi,h)$ $w$ times
2. Return action with best average of SimQ results

$\text{SimQ}(s,a_i,\pi,h)$ trajectories

Each simulates taking action $a_i$ then following $\pi$ for $h-1$ steps.

Samples of $\text{SimQ}(s,a_i,\pi,h)$

$q_{11} \quad q_{12} \ldots q_{1w} \quad q_{21} \quad q_{22} \ldots q_{2w} \quad q_{k1} \quad q_{k2} \ldots q_{kw}$
Executing Rollout in Real World

Real world state/action sequence

Simulated experience

run policy rollout

a₁ a₂ aₖ

s → a₂ → ... → aₖ

Simulated experience

run policy rollout

a₁ a₂ aₖ
Uniform Policy Rollout: # of Simulator Calls

- For each action \( w \) calls to SimQ, each using \( h \) sim calls
- Total of \( kwh \) calls to the simulator

SimQ(\( s, a_i, \pi, h \)) trajectories
Each simulates taking action \( a_i \) then following \( \pi \) for \( h-1 \) steps.
Let $a^*$ be the action that maximizes the true Q-function $Q_\pi(s,a)$.  

Let $a'$ be the action returned by $\text{UniformRollout}[\pi,h,w](s)$.  

Putting the PAC bandit result together with the finite horizon approximation we can derive the following:

$$w \geq \left( \frac{R_{\text{max}}}{\varepsilon} \right)^2 \ln \frac{k}{\delta} \text{ then with probability at least } 1 - \delta$$

$$\left| Q_\pi(s,a^*) - Q_\pi(s,a') \right| \leq \varepsilon + \beta^h V_{\text{max}}$$

But does this guarantee that the value of $\text{UniformRollout}[\pi,h,w](s)$ will be close to the value of $\pi'$?
Policy Rollout: Quality

- How good is UniformRollout[$\pi, h, w$] compared to $\pi'$?

- **Bad News.** In general for a fixed $h$ and $w$ there is always an MDP such that the quality of the rollout policy is arbitrarily worse than $\pi'$.

- The example MDP is somewhat involved, but shows that even small error in Q-value estimates can lead to large performance gaps compared to $\pi'$.

  - But this result is quite pathological
Policy Rollout: Quality

- How good is UniformRollout[\(\pi, h, w\)] compared to \(\pi'\)?
- **Good News.** If we make an assumption about the MDP, then it is possible to select \(h\) and \(w\) so that the rollout quality is close to \(\pi'\).
  - This is a bit involved.
  - Assume a lower bound on the difference between the best Q-value and the second best Q-value.
- **More Good News.** It is possible to select \(h\) and \(w\) so that Rollout[\(\pi, h, w\)] is (approximately) no worse than \(\pi\) for any MDP.
  - So at least rollout won’t hurt compared to the base policy.
  - At the same time it has the potential to significantly help.
Non-Uniform Policy Rollout

- Should we consider minimizing cumulative regret?

No! We really only care about finding an (approx) best arm.
Non-Uniform Policy Rollout

**PAC Setting:** use **MedianElimination**

(parameterized by $\epsilon$ and $\delta$ instead of $w$)

- Often we are given a budget on number of samples (i.e. time per decision).
- MedianElimination not applicable.
Non-Uniform Policy Rollout

Simple Regret: use \( \varepsilon \)-Greedy

(parameterized by budget \( n \) on # of pulls)

- Call this \( \varepsilon \)-Rollout\([\pi, h, n]\)
- \( n \) is number of samples per step
- For \( \varepsilon = 0.5 \) we might expect it to be better than UniformRollout for same # of total samples.
Multi-Stage Rollout

• In what follows we will use the notation $\text{Rollout}[\pi]$ to refer to either UniformRollout[$\pi$, $h$, $w$] or $\epsilon$-Rollout[$\pi$, $h$, $n$].

• A single call to $\text{Rollout}[\pi](s)$ approximates one iteration of policy iteration initialized at policy $\pi$
  ▲ But only computes the action for state $s$ rather than all states (as done by full policy iteration)!

• We can use more computation time to approximate multiple iterations of policy iteration via nesting calls to Rollout.

• Gives a way to use more time in order to improve performance.
Each step requires $khw$ simulator calls for Rollout policy

Trajectories of $\text{SimQ}(s,a_i,\text{Rollout}[\pi], h)$

- Two stage: compute rollout policy of “rollout policy of $\pi$”
- Requires $(khw)^2$ calls to the simulator for 2 stages
- In general exponential in the number of stages
Rollout Summary

• We often are able to write simple, mediocre policies
  ▶ Network routing policy
  ▶ Policy for card game of Hearts
  ▶ Policy for game of Backgammon
  ▶ Solitaire playing policy

• Policy rollout is a general and easy way to improve upon such policies given a simulator

• Often observe substantial improvement, e.g.
  ▶ Compiler instruction scheduling
  ▶ Backgammon
  ▶ Network routing
  ▶ Combinatorial optimization
  ▶ Game of GO
  ▶ Solitaire
Example: Rollout for Solitaire [Yan et al. NIPS’04]

<table>
<thead>
<tr>
<th>Player</th>
<th>Success Rate</th>
<th>Time/Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Expert</td>
<td>36.6%</td>
<td>20 min</td>
</tr>
<tr>
<td>(naïve) Base Policy</td>
<td>13.05%</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>1 rollout</td>
<td>31.20%</td>
<td>0.67 sec</td>
</tr>
<tr>
<td>2 rollout</td>
<td>47.6%</td>
<td>7.13 sec</td>
</tr>
<tr>
<td>3 rollout</td>
<td>56.83%</td>
<td>1.5 min</td>
</tr>
<tr>
<td>4 rollout</td>
<td>60.51%</td>
<td>18 min</td>
</tr>
<tr>
<td>5 rollout</td>
<td>70.20%</td>
<td>1 hour 45 min</td>
</tr>
</tbody>
</table>

- Multiple levels of rollout can payoff but is expensive
Monte-Carlo Planning Outline

• Single State Case (multi-armed bandits)
  ▲ A basic tool for other algorithms

• Monte-Carlo Policy Improvement
  ▲ Policy rollout
  ▲ Policy Switching

• Monte-Carlo Tree Search
  ▲ Sparse Sampling
  ▲ UCT and variants
Another Useful Technique: Policy Switching

• Sometimes policy rollout can be too expensive when the number of actions is large (time scales linearly with number of actions)

• Sometimes we have multiple base policies and it is hard to pick just one to use for rollout.

• Policy switching helps deal with both of these issues.
Another Useful Technique: Policy Switching

• Suppose you have a set of base policies \( \{\pi_1, \pi_2, \ldots, \pi_M\} \)

• Also suppose that the best policy to use can depend on the specific state of the system and we don’t know how to select.

• Policy switching is a simple way to select which policy to use at a given step via a simulator
Another Useful Technique: Policy Switching

- The stochastic function $\text{Sim}(s, \pi, h)$ simply samples the $h$-horizon value of $\pi$ starting in state $s$
- Implement by simply simulating $\pi$ starting in $s$ for $h$ steps and returning discounted total reward
- Use Bandit algorithm to select best policy and then select action chosen by that policy
Uniform Policy Switching

UniformPolicySwitch[\{\pi_1, \pi_2, \ldots, \pi_M\}, h, w](s)

1. For each \pi_i run Sim(s, \pi_i, h) w times
2. Let i* be index of policy with best average result
3. Return action \pi_i^*(s)

Sim(s, \pi_i, h) trajectories
Each simulates following \pi_i for h steps.

Discounted cumulative rewards

\begin{align*}
V_1^1 & \quad V_1^2 & \quad \ldots & \quad V_1^w \\
V_2^1 & \quad V_2^2 & \quad \ldots & \quad V_2^w \\
\vdots & \quad \vdots & \quad \ddots & \quad \vdots \\
V_M^1 & \quad V_M^2 & \quad \ldots & \quad V_M^w
\end{align*}
Executing Policy Switching in Real World

Real world state/action sequence

Simulated experience

\( \pi_2(s) \)
\( \pi_k(s') \)

run policy rollout

run policy rollout
Uniform Policy Switching: Simulator Calls

For each policy use \( w \) calls to Sim, each using \( h \) simulator calls

Total of \( Mhw \) calls to the simulator

Does not depend on number of actions!
\(\epsilon\)-Greedy Policy Switching

- Similar to rollout we can have a non-uniform version that takes a total number of trajectories \(n\) as an argument.

\[
\epsilon\text{-PolicySwitch}[\{\pi_1, \ldots, \pi_M\}, h, n]
\]

Use \(\epsilon\)-Greedy as the bandit algorithm for \(n\) pulls and return best arm/policy.
Policy Switching: Quality

• Let $\pi_{ps}$ denote the ideal switching policy
  ▲ Always pick the best policy index at any state

Theorem: For any state $s$, $\max_i V_{\pi_i}(s) \leq V_{\pi_{ps}}(s)$.

• The value of the switching policy is at least as good as the best single policy in the set
  ▲ It will often perform better than any single policy in set.
  ▲ For non-ideal case, were bandit algorithm only picks approximately the best arm we can add an error term to the bound.
**Proof**

**Theorem:** For any state $s$, $\max_i V_{\pi_i}(s) \leq V_{\pi_{ps}}(s)$.

We’ll use the following property.

**Proposition:** For any policy $\pi$ and value function $V$, if $V \leq B_\pi[V]$, then $V \leq V_\pi$.

Recall $B_\pi[V](s) = R(s) + \sum_s T(s, \pi(s), s') \cdot V(s')$ is the restricted Bellman backup.

So all we need to do is prove that $\max_i V_{\pi_i} \leq B_{\pi_{ps}} \left( \max_i V_{\pi_i} \right)$ since this will imply that $\max_i V_{\pi_i} \leq V_{\pi_{ps}}$ as desired.
Proof \( (to\ simply\ notation\ and\ without\ loss\ of\ generality,\ assume\ rewards\ only\ depend\ on\ state\ and\ are\ deterministic)\)

Prove that \(\max_i V_{\pi_i} \leq B_{\pi_{ps}} \left[\max_i V_{\pi_i}\right]\)

Let \(i^*\) be the index of the best policy in state \(s\).

\[
B_{\pi_{ps}} \left[\max_i V_{\pi_i}\right](s) = R(s) + \sum_{s'} T(s, \pi_{ps}(s), s') \cdot \max_i V_{\pi_i}(s')
\]

\[
\geq R(s) + \max_i \sum_{s'} T(s, \pi_{i^*}(s), s') \cdot V_{\pi_i}(s')
\]

\[
= \max_i \left[ R(s) + \sum_{s'} T(s, \pi_{i^*}(s), s') \cdot V_{\pi_i}(s') \right]
\]

\[
\geq \max_i \left[ R(s) + \sum_{s'} T(s, \pi_i(s), s') \cdot V_{\pi_i}(s') \right]
\]

\[
= \max_i V_{\pi_i}(s)
\]
Policy Switching Summary

• Easy way to produce an improved policy from a set of existing policies.
  ▲ Will not do any worse than the best policy in your set.

• Complexity does not depend on number of actions.
  ▲ So can be practical even when action space is huge, unlike policy rollout.

• Can combine with rollout for further improvement
  ▲ Just apply rollout to the switching policy.