MACHINE INTELLIGENCE COMMUNITY

Summary of Reinforcement Learning: An Introduction Chapter 4: Dynamic Programming

Justin Chen* chenjus@bu.edu
Tyrone Hou* tyroneh@bu.edu

January 15, 2018

Introduction

- **Dynamic Programming (DP)** - algorithms for computing optimal policies given a perfect model of the environment in the form of a Markov Decision Process (MDP)
- Assumes access to a perfect model
- DPs are computationally expensive
- Main idea in DP and RL is to use value functions to organize and structure search for good policies
- Can simplify tasks by assuming a finite MDP
  - State, action, reward $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ are all finite
  - Dynamics defined by probabilities $p(s', r|s, a)$ $\forall s \in \mathcal{S}, a \in \mathcal{A}, r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$ ($\mathcal{S}^+$ includes additional terminal state if task is episodic)
  - Can apply discrete DP methods to continuous state and action spaces by quantizing those spaces
- **Goal**: Find optimal value functions v_* and q_* that satisfy the Bellman optimal equations to obtain optimal policies
- **Bellman optimality equations**:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_*(s')\big] \\ q_*(s, a) &= \mathbb{E}\big[R_{t+1} + \gamma \max_{a'} (S_{t+1}, a' | S_t = s, A_t = a)\big] \\ &= \sum_{s',r} p(s', r|s, a)\big[r + \gamma \max_{a'} q_*(s', a')\big] \qquad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}(s), \text{ and } s' \in \mathcal{S}^+ \end{aligned}$$

4.1 Policy Evaluation (Prediction)

- Given policy π, how do we compute its state-value function v_π?
- Known as the **policy evaluation** or **prediction problem**
- Recall from Chapter 3

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big]$$

*These authors contributed equally to this work

- Assuming full knowledge of model dynamics we can solve for the values $v_\pi(s)$ for all states as a system of linear equations with $|\mathcal{S}|$ unknowns.
- Alternatively, we could iteratively improve an estimated value function v_k over time until it converges to v_π.
- **Iterative Policy Evaluation**

  1. Initialize v_0 arbitrarily for $s \in \mathcal{S}$ and let the terminal state value be 0.
  2. Use the Bellman update equation as a rule to update v_k

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \end{aligned}$$

  3. Given that v_k converges to v_π, we stop the algorithm once the change in policy values is very small.

  - $v_k = v_\pi$ is a *fixed point* for the update rule. This just means if we replace v_k with v_π the equation holds for any k due to the self-consistency condition. If $v_k \neq v_\pi$, updating it will still converge to the fixed point.
  - We call step 2 above an **expected update** because we derive v_{k+1} from the expected value of v_k given all possible next actions, states, and rewards.

Iterative Policy Evaluation Algorithm
Input π, policy to evaluate
Initialize array $V(s) = 0$, $\forall s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (small positive number)
Output $V \approx v_\pi$

- The value function can be updated with two arrays for v_{k+1} and v_k or in place as below. For the in place method some states are updated using newly calculated values v_{k+1} instead of v_k, but either method converges.

4.2 Policy Improvement

- For some state s, should we change to the policy to deterministically choose an action $a \neq \pi(s)$?

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_\pi(s')\big] \end{aligned}$$

- **Policy Improvement Theorem** - A policy is better if it produces greater expected reward over time:

$$\begin{gathered} \text{If } q_\pi(s, \pi'(s)) \geq v_\pi(s) \\ \text{Then } v_{\pi'}(s) \geq v_\pi(s) \end{gathered}$$

  - For any state s, if $q_\pi(s, \pi'(s)) > v_\pi(s)$ (strictly greater), then there is at least one state s such that $v_{\pi'}(s) > v_\pi(s)$. The same is true for $<$.

- Conditioned on the future

- Shift behavioral distribution depending on future trajectories

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(a)] \\ &= \pi'[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(a)] \\ &\leq \mathbb{E}_{\pi'}[R] \end{aligned}$$

- **Policy Improvement** - Process of making a new policy that improves on an original policy by making it greedy w.r.t. the value function of the original policy
  - Must give a strictly better policy except when the original policy is already optimal

4.3 Policy Iteration

- We can repeatedly do policy evaluation and policy improvement steps to converge towards an optimal policy

$$\pi_0 \xrightarrow{E} v_0 \xrightarrow{I} \pi_2 \xrightarrow{E} v_2 \ldots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

- Initially, we randomly choose value function estimate and policy. In evaluation fix the policy and improve the value estimate. In improvement, use the value function to create a more optimal policy.

Policy Iteration (using Iterative Policy Evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}$, $\forall s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in \mathcal{S}$
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \underset{a}{argmax} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, stop and return $V \approx v_*$ and $\pi \approx \pi_*$;
   Else go to 2

- Because we improve the policy between policy evaluation steps, the value function converges much more quickly to optimal.

4.4 Value Iteration

- Each policy evaluation step in policy iteration requires multiple sweeps (passes) through $\mathcal{S}$.
- We can speed up convergence by interleaving sweeps of evaluation and improvement instead of fully approximating v_π and then improving π.

- **Value Iteration** - A technique similar to policy iteration except we combine (truncated) policy evaluation and policy improvement into one update step

$$v_{k+1}(s) \doteq \underset{a}{argmax}\ \mathbb{E}[R_t + \gamma v_k(S_{t+1})|S_t = s, A_t = a]$$
$$= \underset{a}{argmax} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

- Note that this equation is essentially the same as the Bellman optimality equation for v_*, so that v_π converges to v_* over many iterations.
- The update equation is similar to policy evaluation, except we add a maximum over actions instead of averaging over all actions.

Value Iteration
Initialize array V arbitrarily e.g.$V(s) = 0 \forall s \in S^+$

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathcal{S}$
$v \leftarrow V(s)$
$V(s) \leftarrow \underset{a}{max} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
$\Delta \leftarrow max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (small positive number)

Output deterministic policy $\pi \approx \pi_*$ where $\pi(s) = \underset{a}{argmax} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

- You can generalize policy and value iteration by interposing multiple policy evaluation sweeps between each policy improvement step. All generalizations converge.
  - For *value iteration* the ratio is 1:1.
  - For *policy iteration* you do evaluation sweeps until $V(s)$ converges to v_π.

4.5 Asynchronous Dynamic Programming

- A single sweep can be prohibitively expensive if state space is very large
  - e.g. 10^{20} states in the game of backgammon
- **Asynchronous DP** - In-place DP that do not sweep state space
  - State value updates can occur out of order
  - Correct convergence depends on continuous updates
  - Allows flexible selection for updates
  - Some states may be updated more than other states. There could be situations in which some states may not even need to be updated so they can safely be skipped.
  - Circumventing sweeps does not guarantee improved computational efficiency - only guarantees asynchronous updates.
  - Asynchrony allows intermixing real-time interaction
    - e.g. Run iterative DP in parallel while agent is experiencing the MDP
    - Experience indicates which states to update
    - Simultaneously, DP can inform agent's decisions
    - Allows focusing on relevant regions of state space
- If $0 \leq \gamma < 1$ for value iteration, asymptotic convergence to v_* is guaranteed give only that all states occur in the sequence $\{s_k\}$ an infinite number of times
  - Sequence can be deterministic or stochastic
  - Some sequences may not converge, but these are easily avoidable

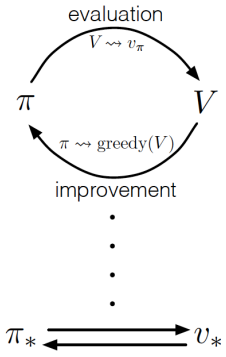

Figure 1: Generalized Policy Iteration

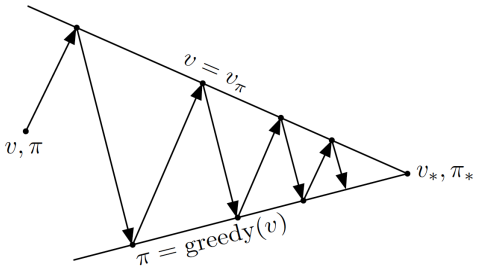

Figure 2: Policy evaluation versus Policy improvement

4.6 Generalized Policy Iteration

- **Policy Iteration** has two main processes:
  - **Policy Evaluation** - Making the value function consistent with the *current policy*
  - **Policy Improvement** - Making the policy greedy w.r.t. the *current value function*
- **Value Iteration** employs a single iteration of policy evaluation before each *policy improvement*
- Asynchronous DP interleaves policy evaluation and policy improvement unevenly
- **General Policy Iteration (GPI)** - Principle of allowing both policy improvement and policy evaluation to interact independently of each other
- All reinforcement learning methods have identifiable policies and value functions
  - Policy is continuously improved w.r.t. value function
  - Value function is continuously updated to improve the policy
  - **Value function** and **policy function** are optimal w.r.t. each other when both evaluation and improvement processes stabilize - when neither process produce changes
    - Both processes stabilize only when a policy has been found that is greedy w.r.t. its own evaluation function, which implies the Bellman optimality equations and that the policy and value function are optimal
- Continuous opposition and feedback between GPI's policy evaluation and policy improvement processes allows both to find optimal value function and optimal policy
  - Greedy policy w.r.t. value function causes the value function to be incorrect for the updated policy
  - Aligning value function with policy causes policy to be less greedy
  - Policy evaluation and policy improvement work together to reach optimality although neither attempt to achieve it directly

4.7 Efficiency of Dynamic Programming

- **Method Comparison**
  - Directly searching policy space means testing $|A|^{|S|}$ policies.
  - Dynamic programming is *polynomial* in number of states and actions.
  - Linear programming methods can be more efficient for small policy spaces, but doesn't scale nearly as well.

  - Evolutionary methods may also be less efficient for large policy spaces.

- **Ways to improve convergence speed**
  - Good initialization of value function and policy
  - Asynchronous DP methods - sometimes only a relatively few states need to be visited in optimal policies, further speeding up convergence
  - Generalized Policy Iteration

- **Curse of Dimensionality** - The state space grows exponentially as the number of state variables increase

- This is an inherent limitation of the RL problem, but DP methods are comparatively more adept at handling the curse

4.8 Summary

- Classical Dynamic Programming (DP) involves sequentially improving the value function and policy so they converge on v_* and π_*, the optimal value function and policy, using *expected updates*.

- **Generalized Policy Iteration** interleaves sweeps through the state space of evaluation and improvement steps, while **Asynchronous DP** abandons the idea of sweeping the entire state space, instead updating states in arbitrary order. Policy and value iteration are special cases of GPI.

- As long as both the value function and policy are improving, the update order doesn't really matter.

- Many RL methods, including all DP methods, use the **bootstrapping** technique - updating estimates based on other estimates

References

Sutton, Richard S., and Andrew G. Barto. "Dynamic Programming." Reinforcement Learning: An Introduction, The MIT Press, 2018, pp. 59–73.