Model-based Value methods

We will study two common Model-based, Dynamic Programming type methods

- Value Iteration
- Policy Iteration

and their associated Bellman equations.

Value iteration method

The simplest method is to

- iteratively update the Value function
    - until convergence
- derive the *final* Policy from the Value function
    - chose the action leading to highest return
    - based on the Value function

The estimate of the value of a state $\state$ is improved iteratively.

Let

$$\statevalfun_k(\state)$$

denote the *estimated* value of state $\state$ after k iterations.

In iteration $k+1$, the value of *each* state $\state \in \States$ is updated via the Bellman update

$$\statevalfun_{k+1}(\state) = \max \act \sum_{\state'} \transp(\state' \mid \state, \act)\big(\rew(\state, \act, \state') + \gamma \statevalfun_k(\state')\big).$$

where

$$\rew(\state, \act, \state')$$

denotes the reward returned by the environment when action $\act$ in state $\state$ results in successor state $\state'$

That is

- $\statevalfun_{k+1}(\state)$
- learns the *previous iteration's* (k) estimate

  $$\statevalfun_{k}(\state')$$

  of each successor state $\state'$
- and chooses the action leading to highest return

The backup in each iteration

- moves information about potential future (successor) states
- to the current state

The previous iteration's estimate of the successors of $\state$
$\statevalfun_k(\state')$

in turn, are based on the iteration $(k-1)$ values of the successors of $\state'$

So we need i iterations

- in order for information of states i steps ahead
- to reach state s

Hence the need to repeat until convergence.

Subtlety

Prior to convergence

- $\statevalfun_k(\state)$ is a *non-random* variable
    - $\transp(\state, \act)$ is *known* for all $\state$, $\act$ since method is model-based
- the estimate is *biased*
    - it is non-random, but not yet correct

We will subsequently contrast this to the iterative estimates produced by model-free methods.

Pseudo code for Value Iteration

Here is some pseudo-code

Value iteration

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Attribution: http://incompleteideas.net/book/RLbook2020.pdf#page=105 (http://incompleteideas.net/book/RLbook2020.pdf#page=105)

Value iteration

Initialize value function V(s) arbitrarily (e.g., zero for all states) Repeat: delta = 0 For each state s: old_value = V(s) V(s) = max over a [R(s, a) + γ * sum over s' [P(s' | s, a) * V(s')]] delta = max(delta, |old_value - V(s)|) Until delta < threshold # Derive policy after value function converges For each state s: π(s) = argmax over a [R(s, a) + γ * sum over s' [P(s' | s, a) * V(s')]] Return π, V

Policy iteration method

The Value Iteration method has one expensive step

- the $\max \act$ is a search over all possible actions $\act \in \Actions$

$$\statevalfun_{k+1}(\state) = \max \act \sum_{\state'} \transp(\state' \mid \state, \act)\big(\rew(\state, \act, \state') + \gamma \statevalfun_k(\state')\big).$$

Policy Iteration is a less computationally-expensive way to reach the optimal $\statevalfun$.

Rather than updating the Policy once (after Value function convergence)

- we introduce a method that periodically updates the Policy.

Policy iteration is an algorithm that improves π_p to π_{p+1} by alternating two steps during round p

The algorithm alternates between

- Policy evaluation
  - update the estimate of $\statevalfun_{\pi_p}$ to $\statevalfun_{\pi_{p+1}}$
- Policy improvement
  - update π_p to π_{p+1} using the newly updated $\statevalfun_{\pi_{p+1}}$

It uses the Bellman update

$$\statevalfun_{\pi,k+1}(\state) = \sum_{\state'} \transp(\state' \mid \state, \pi(\state))\big(\rew(\state, \pi(\state), \state') + \gamma \statevalfun_{\pi,k}(\state')\big).$$

where

- we add an explicit subscript π
- to the value function
$$\statevalfun_{\pi,k+1}(\state)$$
- to indicate the dependence of the update on the *current policy* π

Rather than

- evaluating *all* actions $\act \in \Actions$
- it chooses a single action $\state, \pi(\state)$ according to the current policy

Here is some pseudo-code

Policy iteration

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(\textit{terminal}) \doteq 0$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

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

Attribution: http://incompleteideas.net/book/RLbook2020.pdf#page=102 (http://incompleteideas.net/book/RLbook2020.pdf#page=102)

Pseudo code for Policy Iteration

Policy iteration

Initialize policy π arbitrarily (e.g., random policy) Initialize value function V(s) arbitrarily (e.g., zero for all states) Repeat: # Policy Evaluation (compute V for current policy π) Repeat: For each state s: V(s) = R(s, π(s)) + γ * sum over s' [P(s' | s, π(s)) * V(s')] Until V(s) converges (changes smaller than threshold) # Policy Improvement (update policy based on current V) policy_stable = True For each state s: old_action = π(s) π(s) = argmax over a [R(s, a) + γ * sum over s' [P(s' | s, a) * V(s')]] if old_action != π(s): policy_stable = False Until policy_stable is True Return π, V

Both Value Iteration and Policy Iteration will converge to the same policy.

The advantage of alternating between Policy Evaluation and Policy Improvement

- faster convergence
    - the Value function under the current policy is fully evaluated
    - before the Policy is updated
- more stable convergence
    - Policy doesn't change until Value function (under current policy) is fully known

Finding the best action in a Value-based method

The Value-based methods don't directly give you a policy

- the Value function gives you the best successor state $\state'$ from current state $\state$
- **but** it doesn't directly tell you the action $\act$ that leads to $\state'$

In order to find $\act$ you either

- need a model
    - search over all possible actions, using the model to determine the value of action's successor state
- use actual experience to estimate the effect of performing action $\act$ in state $\state$

Here is some pseudo-code that uses a model to determine the best action:

```
# For each possible action a in current state s:
for each action a in actions:
    # Take action a from state s in the environment
    observe next state s', reward r
    # Estimate value for taking action a from s
    A[a] = r + gamma * V[s']
# Find the maximum estimated action value
A_max = max over a of A[a]

# Update the value function for state s
V[s] = V[s] + alpha * (A_max - V[s])
```

Q-learning: From Value function to State-Action function

We can more readily identify the optimal *action* to take from state $\state$

- the one leading to $\state'$ with maximal $\statevalfun(\state')$

with slightly more detailed bookkeeping.

Define a *State Value function* $\backslash\mathrm{actvalfun}_{\pi}(\backslash\mathrm{state}, \backslash\mathrm{act})$

$$\backslash\mathrm{actvalfun}_{\pi} : \backslash\mathrm{States} \times \backslash\mathrm{Actions} \rightarrow \backslash\mathrm{Reals}$$

to map a state/action pair $(\backslash\mathrm{state}, \backslash\mathrm{act})$ to the expected value of the successor state.

Re-write the Value function backup equation

$$\statevalfun_{k+1}(\state) = \max \act \sum_{\state'} \transp(\state' \mid \state, \act)\big(\rew(\state, \act, \state') + \gamma \statevalfun_k(\state')\big).$$

as

$$\statevalfun_{k+1}(\state) = \max \act \actvalfun_k(\state, \act)$$

where

$$\actvalfun(\state, \act) = \sum_{\state'} \transp(\state' \mid \state, \act)\big(\rew(\state, \act, \state') + \gamma \statevalfun_k(\state')\big)$$

Key Differences Between Value Iteration and Policy Iteration

Here is a comparison of the Value and Policy Iteration methods.

Feature	Value Iteration	Policy Iteration
Approach	Updates value function until convergence	Alternates between value evaluation and improvement
Convergence	When value function $V(s)$ stabilizes	When policy $\pi(s)$ stops improving
Computational Cost	Higher per iteration, simpler logic	Lower per iteration, more complex
Speed	Requires more iterations	Fewer iterations; often faster
Policy Output	Extracted after value convergence	Updated during each iteration

Feature	Value Iteration	Policy Iteration
Approach	Updates value function iteratively using the Bellman Optimality Equation in one step	Alternates between full policy evaluation and policy improvement steps
Policy Handling	Policy is implicitly updated after value convergence	Explicitly maintains and updates policy each iteration
Initialization	Starts with an initial value function	Starts with an initial policy
Iteration Steps	Single step combining evaluation and improvement	Two-step process: separate evaluation and improvement
Convergence Criterion	Value function converges (changes below a threshold)	Policy stabilizes (no change between iterations)
Computation per Iteration	Potentially more expensive per iteration (max over all actions for each state)	More computationally intensive due to full policy evaluation, but fewer total iterations needed
Number of Iterations	Typically more iterations	Usually fewer iterations
Complexity	Simpler to implement	More complex implementation
Suitability	Suitable for smaller state spaces or when full policy evaluation is expensive	Can handle larger state spaces more efficiently when full evaluation is feasible
Policy Updates	Policy derived after convergence of value function	Policy updated after each evaluation phase

Model-based Value methods: Classical vs Deep Learning

In "Classical" Model-base Value methods

- the state and action spaces are finite
- leading to methods based on tables

Dynamic Programming style solutions are efficient methods for exploring the finite possibilities.

But non-finite action and state spaces may be accomodated

- by using Neural Networks (NN)
- to *learn*
    - an approximation of the model
    - the value function

Aspect	Classical Model-based RL (Tabular)	Model-based Value Method with NN
State/action space	finite	continuous/infinite (via approximation)
Model representation	table or analytic function	neural network (NN), e.g.,fϕf_\phifϕ
Value function representation	table	NN:Vθ(s)V*theta(s)Vθ(s),Qθ(s,a)Q*θ(s,a)Q\theta(s,a)Qθ(s,a)
Planning/Improvement	Bellman backup over all states	Rollouts/approximate backups, often via sampling
Example use cases	gridworlds, small MDPs	robotics, control, games with pixel input

In [2]:
```
print("Done")
```

```
Done
```