Dynamic Programming Based Reinforcement Learning Methods

Reinforcement Learning

In reinforcement learning, the agent aims to maixmize its cumulative reward:

$$\max \sum_{t=1}^{T} \mathbb{E}_{a_t\sim\pi(s_t), s_{t+1}\sim p(s_{t+1}|s_t,a_t), s_t\sim p(s)} \left[\gamma^{t-1} r(s_t, a_t)\right]$$

From the perspective of Bellman equation, the caculation of cumulative reward can be also formulated as:

$$V(s_t) = \mathbb{E}_{a\sim\pi(s_t)}\left[r(s_t, a_t) + \gamma V(s_{t+1})\right]$$

Policy Iteration Learning

Once a policy, π, has been improved using $v\pi$ to yield a better policy, π', we can then compute $v\pi'$ and improve it again to yield and even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_\star \xrightarrow{\text{E}} v_\star$$

where $\xrightarrow{\text{E}}$ denotes a policy *evaluation*, and $\xrightarrow{\text{I}}$ denotes a policy improvement. This way of finding an optimal policy is called *policy iteration*.

Pesudo of Policy Iteration Learning

1. **Initialization**

   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
2. **Policy Evaluation**

   Loop:

   > $\Delta \leftarrow 0$
   > Loop for each $s \in \mathcal{S}$:
   >
   > > $v \leftarrow V(S)$
   > > $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   > > $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

   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* $\leftarrow \pi(s)$
   > $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   > If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*

   If *policy-stable*, then stop and return $V \sim v_\star$ and $\pi \sim \pi_\star$; else go to 2

```
def policy_eval(env, values, policies, upper_bound):
def policy_improve(env, values, policies):
```

Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. Must we wait for exact convergence, or can we stop short of that?

Pesudo of Value Iteration Learning

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, $\pi \approx \pi_\star$, such that $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma V(s')\Big]$

```
def value_iter(env, values, upper_bound):
```

A Simple Environment

MatrixEnv

A simple maze game, the agent needs to learn walk from the start point to the destination (goal)

```
class Env:
class MatrixEnv(Env):
```

Basic Data Structure for Learning

ValueTable

Class `ValueTable` maintains the state value function which map state space $S \in R^2$ to real number space R.

Methods:

- `update(state, value)`: update state value with given value
- `get(state)`: return state value with given state or states

Policies

Class `Policies` maintains the polcies of each states

Methods:

- `sample(state)`: sample action, return action index
- `retrieve(state)`: retrieve policy of given state
- `update(state, policy)`: update policy of state with given policy

Homework

Solving Matrix Game via Policy Iteration Learning

Solving Matrix Game via Value Iteration Learning