Reinforcement Learning

Tian-Li Yu

National Taiwan University
Department of Electrical Engineering
tianliyu@ntu.edu.tw

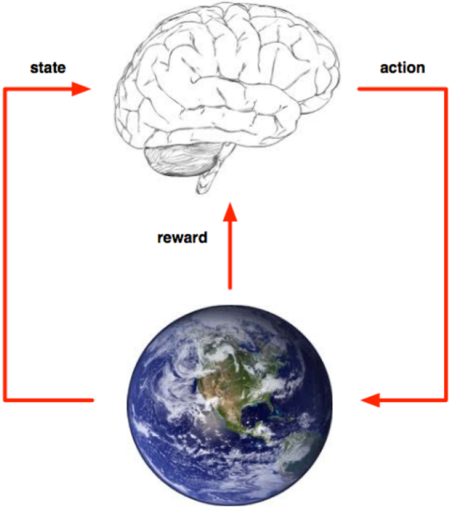
Readings: AIMA Sections 17.1∼2, 21.1, 21.3∼4, ML 13.5

Outlines

Sequential Decisions

At each time frame

- The agent
  - Receives a state
  - Receives a reward
  - Executes an action
- The environment
  - Receives an action
  - Emits a state
  - Emits a reward

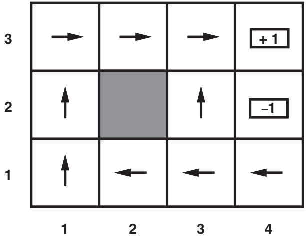


Sequential Decisions

- Assumptions:
  - Fully observable: The agent always knows where it is.
  - Nondeterministic actions.
- A Markov decision process (MDP):
  - Transition model is probabilistic: $P(s'|s,a)$
  - Reward function: $R(s)$
  - We use the shorthand $A(s)$ for $\textsc{Action}(s)$

Policy

- Any fixed action sequence won't solve the problem.
- A solution must specify what the agent should do for any reachable states.
- A solution of this kind is called a policy: π
- $\pi(s)$: the action recommended by the policy π for the state s.
- π^*: the optimal policy that yields the highest expected utility.



- π^* for the previous MDP.
- The action for $(3, 1)$ is conservative.

Utility

- With no time limit, the optimal action depends only on the current state, and the optimal policy is stationary.
- The next question is for a state sequence $[s_0, s_1, s_2, \cdots]$, how do we decide its utility U?
- Here we make a preference-independence assumption:
    - If $U([s_0, s_1, s_2, \cdots]) \geq U([s_0, s_1', s_2', \cdots])$, then $U([s_1, s_2, \cdots]) \geq U([s_1', s_2', \cdots])$.
- The above assumption seems innocuous, but the only form that satisfies it is
    - $U([s_0, s_1, s_2, \cdots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$, where $0 \leq \gamma \leq 1$.
    - For $\gamma = 1$, we call it additive rewards.
    - For $\gamma < 1$, we call it discounted rewards.

Optimal Policies and Utilities of States

- Define S_t (a random variable) to be the state that the agent reaches at time t.
- The expected utility obtained by executing π starting at s:
$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$
- Overall policies starting at s, there exist one policy such that
$$\pi_s^* = \operatorname{argmax}_{\pi} U^{\pi}(s)$$
- Given our assumption, all π_s^* are the same. So we denote the optimal policy simply as π^*.
- The true utility is then given by $U^{\pi^*}(s)$, which is shortened as $U(s)$.
- The optimal action for any given state is given by
$$\pi^*(s) = \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

Bellman Equation

$$\begin{aligned} U([s_0, s_1, s_2, \cdots]) &= R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \\ &= R(s_0) + \gamma U([s_1, s_2, \cdots]) \end{aligned}$$

- Therefore, the utility of a state is given by

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U(s')$$

- This is given by Richard Bellman (1957).
- Also known as dynamic programming equation.
- A necessary condition for optimality associated with dynamic programming.

Value Iteration Algorithm

- Bellman update: $U_{i+1}(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$
- If the update is applied simultaneously to all states, equilibrium is reached eventually.

VALUE-ITERATION(ϵ)

U' initially zeros.
repeat
    $U = U'$
    $\delta = 0$
    for each state s in S
        $U'[s] = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U[s']$
        $\delta = \max(\delta, |U'[s] - U[s]|)$
until $\delta < \epsilon(1-\gamma)/\gamma$
return U

Convergence of Value Iteration

- Define $\mathbb{B}$ as the Bellman updater: $U_{i+1} = \mathbb{B}U_i$.
- Define the max norm $||U|| = \max_s |U(s)|$.

Lamma: $||\mathbb{B}U_{N-1} - \mathbb{B}U|| \leq \gamma ||U_{N-1} - U||$.

$$
\begin{aligned}
& \gamma \max_s \left| \max_{a\in A(s)} \sum_{s'} P(s'|s,a)U_{N-1}(s') - \max_{a\in A(s)} \sum_{s'} P(s'|s,a)U(s') \right| \\
\leq \; & \gamma \max_s \left| \sum_{s'} P(s'|s,a^*)U_{N-1}(s') - \sum_{s'} P(s'|s,a^*)U(s') \right| \\
= \; & \gamma \max_s \left| \sum_{s'} P(s'|s,a^*)\left(U_{N-1}(s') - U(s')\right) \right| \\
\leq \; & \gamma \max_{s^*} |U_{N-1}(s^*) - U(s^*)| = \gamma ||U_{N-1} - U||
\end{aligned}
$$

$\square$

Convergence of Value Iteration

Proof.

1. For any given $0 < \epsilon$, we desire N exists such that $||U_N - U|| \leq \epsilon$.
2. $||U_N - U|| = ||\mathbb{B}U_{N-1} - \mathbb{B}U|| \leq \gamma ||U_{N-1} - U|| \leq \cdots \leq \gamma^N ||U_0 - U||$.
3. Utility of any infinite sequence is finite:
   $U([s_0, s_1, \cdots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$
   $\Rightarrow -\frac{R_{max}}{1-\gamma} \leq U \leq \frac{R_{max}}{1-\gamma} \Rightarrow ||U_0 - U|| \leq \frac{2R_{max}}{1-\gamma}$
4. Desire $\gamma^N \frac{2R_{max}}{1-\gamma} \leq \epsilon \Rightarrow N = \log(\frac{2R_{max}}{\epsilon(1-\gamma)}) / \log(1/\gamma)$ iterations suffice.
5. Finally, $||U_N - U|| \leq \gamma^N ||U_0 - U|| \leq \gamma^N \frac{2R_{max}}{1-\gamma} \leq \epsilon$.

□

Policy Iteration

It is possible to get an optimal policy even when the utility estimation is inaccurate.

- Policy Evaluation: Given policy π_i, calculate $U_i = U^{\pi_i}$.
- Policy Improvement: Calculate new policy π_{i+1}, using one-step look-ahead based on U_i.

The above two steps iterate until policy remains fixed.
How to execute policy evaluation? At each state s, π_i specifies the action $\pi_i(s)$. So the Bellman equation is simplified:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

Correctness and Efficiency of Policy Iteration

- Correctness:

  Policy reaches fixed-point $\rightarrow$ Utility reaches fixed-point $\rightarrow$ True utility $\rightarrow$ Optimal policy.

- Efficiency:
  - By removing the max in Bellman updater, policy evaluation can be done in $O(|S|^3)$ ($|S|$ unknowns with $|S|$ equations).
  - When $|S|$ is large, usually some fixed number of value iterations is enough.
  - Asynchronous policy iteration only updates a subset of states at each iteration. Optimality can still be guaranteed under certain conditions.

Reinforcement Learning

- Value/policy iteration assumes $R(s)$ and $P(s'|s,a)$ are known in advance.
- In many real-world applications, the agent does not know the reward $R(s)$ until it is in the state s.
- Reinforcement learning aims to learn how to behave in this manner.

Specifically, RL cares about

- Value ($U(s)$ or $Q(s,a)$): Agent's estimation of how good each state and/or action is.
- Policy ($\pi(s)$): Agent's behavior function.
- Model ($P(s'|s,a)$): Agent's representation of the environment.

Different Aspects of Reinforcement Learning

- Passive learning: The agent's policy is fixed, and the task is to learn the true utility.
- Active learning: The agent needs to learn the true utility and optimal policy.

- Off-policy learning: The agent learns the optimal policy independently of the agent's actions.
- On-policy learning: The agent learns the policy being carried out by the agent including the exploration steps.

Learning Utilities

- Two approaches:
  1. Adaptive dynamic programming (ADP)
  2. Temporal difference (TD).

- An ADP agent uses the Bellman equation directly:

$$U(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s').$$

- A TD agent updates $U(s)$ when a transition occurs from s to s':

$$U(s) \leftarrow U(s) + \alpha \left(R(s) + \gamma U(s') - U(s) \right),$$

  where α is the learning rate.

- TD can be viewed as an approximation of ADP, but very importantly, TD does not need a transition model, which is called model-free learning.

Temporal Difference

- Suppose we desire the average of several rewards:

$$\bar{R}_k = \frac{1}{k}\sum_{i=1}^{k} R_i$$

- However, we do not want to record every reward.

$$\begin{aligned}\bar{R}_k &= \frac{1}{k}\left(\sum_{i=1}^{k-1} R_i + R_k\right) \\ &= \frac{1}{k}\left((k-1)\bar{R}_{k-1} + R_k\right) \\ &= \frac{1}{k}\left(k\bar{R}_{k-1} + R_k - \bar{R}_{k-1}\right) \\ &= \bar{R}_{k-1} + \frac{1}{k}\left(R_k - \bar{R}_{k-1}\right)\end{aligned}$$

Q-Learning

- Recall Bellman equation:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U(s').$$

- Instead of learning utility, Q-learning learns the Q-values: $Q(s,a)$ for doing action a at state s.
- Q-values are directly related to utility
  - $U(s) = \max_{a \in A(s)} Q(s,a)$
  - $Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) U(s')$
- When Q-values are correct, the following equilibrium must hold:

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in A(s')} Q(s',a')$$

- The above equation can be used directly as the updater for Q-learning (ADP, off-policy).

Q-Learning

- ADP, off-policy

$$Q(s,a) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in A(s')} Q(s',a')$$

- TD, off-policy

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

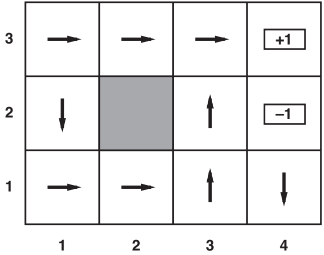
- SARSA (TD, on-policy):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R(s) + \gamma Q(s',a') - Q(s,a) \right)$$

- The above updater is performed whenever action a executed in state s leading to state s'.

Learning Policy

- If an agent always chooses the best action suggested by the Q-value, we call it a greedy agent.
- A greedy agent seldom converges to the optimal policy due to lack of exploration.



- The ADP greedy agent converges to a suboptimal policy.
- It will never discover the optimal policy since it always goes down at (1,2).

Exploration

- It's generally difficult to obtain an optimal exploration scheme, but easy to come up with a reasonable one that eventually lead to the optimal policy.
- Such a scheme is greedy in the limit of infinite exploration (GLIE).
- A GLIE scheme tries each action in each state an unbounded number of times to eliminate the probability of missing the optimal action.

- **A simple GLIE scheme:** choose a random action a fraction $1/t$ if the time; follow the greedy policy otherwise.
- **More sensible approach:** Using the multi-armed bandit techniques such as UCB (the GLIE scheme corresponds to the zero-regret strategy).

TD(λ) Algorithm

- Q-learning uses one-step lookahead:

$$Q^{(1)}(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$

- Why not two-step lookahead?

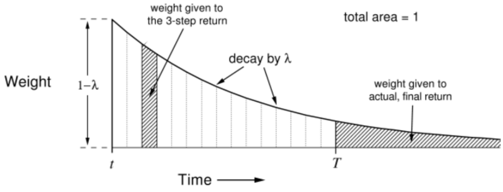
$$Q^{(2)}(s,a) = R(s) + \gamma R(s') + \gamma^2 \max_{a''} Q(s'',a'')$$

- In general, if we have many such estimators, why not use them all?

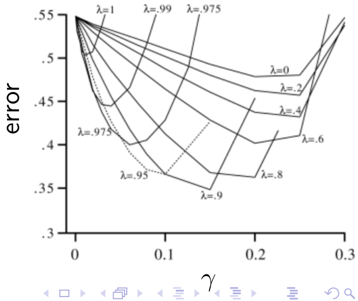
$$\begin{aligned} Q^{\lambda}(s,a) &= (1-\lambda)\left(Q^{(1)}(s,a) + \lambda Q^{(2)}(s,a) + \lambda^2 Q^{(3)}(s,a) + \cdots\right) \\ &= R(s) + \gamma\left((1-\lambda)\max_{a'} Q(s',a') + \lambda Q^{\lambda}(s',a')\right) \end{aligned}$$

- When Q-values are accurate, the equation is identical for any $0 \le \lambda \le 1$.

Other Temporal Difference Methods



- TD(λ) with $\lambda = 0$ reduces to the original Q-learning.
- TD(λ) with $\lambda = 1$ considers only the observed rewards.
- $Q(\lambda)$: limited lookahead (called λ-return) and adjustable λ.



Generalization

- Reinforcement learning needs a table of size $|S|$; Q-learning needs a table of size $|S| \times |A|$. Impractical for many problems.
- Use an evaluation function instead of table look-up:

  $\hat{U}_\theta(s) = \sum_i \theta_i f_i(s)$, where θ_i are weights and f_i are features.
- The goals of reinforcement learning is to learn θ_i.
- Define $u_j(s)$ are the observed total reward from state s in the j^{th} trial.
  - We want to minimize $E_j(s) = \left(\hat{U}_\theta(s) - u_j(s)\right)^2/2$
  - $\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$
  - This is called the Widrow-Hoff rule, or the delta rule.

Generalization

- The above idea can be easily applied to TD learning.
- For general reinforcement learning:

$$\theta_i \leftarrow \theta_i + \alpha \left(R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

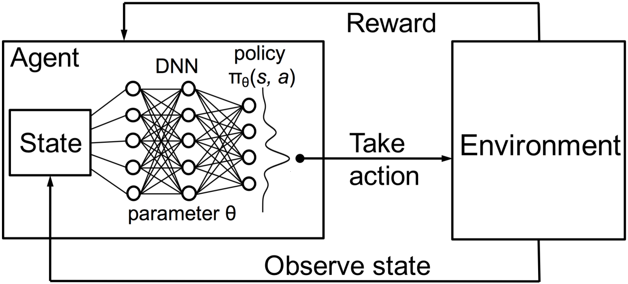
- For Q-learning:

$$\theta_i \leftarrow \theta_i + \alpha \left(R(s) + \gamma \max_{a' \in A(s')} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a) \right) \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

- If $\hat{U}_\theta(s) = \sum_i \theta_i f_i(s)$, $\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} = f_i(s)$ can be used to simplify the above updaters.

Deep Reinforcement Learning

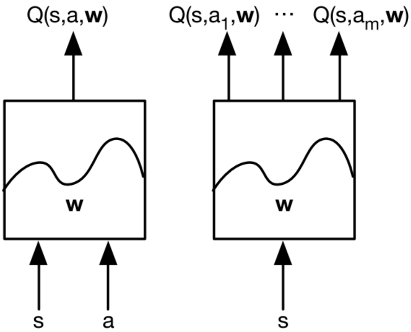
- Use deep neural networks to represent
  - Value function
  - Policy
  - Model
- Minimize loss function by stochastic gradient descent (SGD).



Deep Q-Learning

- Represent value function by Q-network with weights $\vec{w}$.

$$\hat{Q}_{\vec{w}}(s,a) \simeq Q(s,a)$$



Deep Q-Learning

- The optimal Q-values follows Bellman equation.
- Recall one-step look-ahead estimation (SARSA):

$$Q(s,a) \simeq R(s) + \gamma \max_{a'} Q(s',a')$$

- Minimize the MSE loss by SGD

$$\ell(\vec{w}) = \left(R(s) + \gamma \max_{a'} \hat{Q}_{\vec{w}}(s',a') - \hat{Q}_{\vec{w}}(s,a) \right)^2$$

- Basically converges to Q, but
  - Correlations between samples
  - Non-stationary targets

DQN

- Remove correlations by experience replay.

s_1, a_1, R_1, s_2, a_2
s_2, a_2, R_2, s_3, a_3
s_3, a_3, R_3, s_4, a_4
...
$s_t, a_t, R_t, s_{t+1}, a_{t+1}$

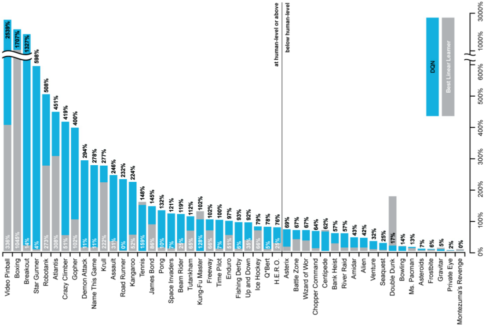
$\rightarrow s, a, R, s', a'$

$s_t, a_t, R_t, s_{t+1}, a_{t+1} \rightarrow$

- To deal with non-stationary, target parameters $\vec{w}'$ are held fixed.

$$\ell(\vec{w}) = \left(R(s) + \gamma \max_{a'} \hat{Q}_{\vec{w}'}(s', a') - \hat{Q}_{\vec{w}}(s, a) \right)^2$$

DQN on Atari (Google Nature Paper 2014)



Game	Linear	Deep netowrk	DQN with fixed Q	DQN with replay	DQN with replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Sequest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

double DQN

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
  - never on the same time steps (Q_1 and Q_2 are independent)
  - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1, use Q_2 for the value of the next state:

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha \left(R(s) + Q_2(s', \underset{a'}{\operatorname{argmax}} Q_1(s', a')) - Q_1(s,a) \right)$$

- Action selections are ϵ-greedy (or other MAB techniques) with respect to the sum of Q_1 and Q_2.

Other Improvements Since Natural DQN

- Prioritized replay
  - Weight experience according to surprise.
  - Store experiences in priority queue according to DQN error.

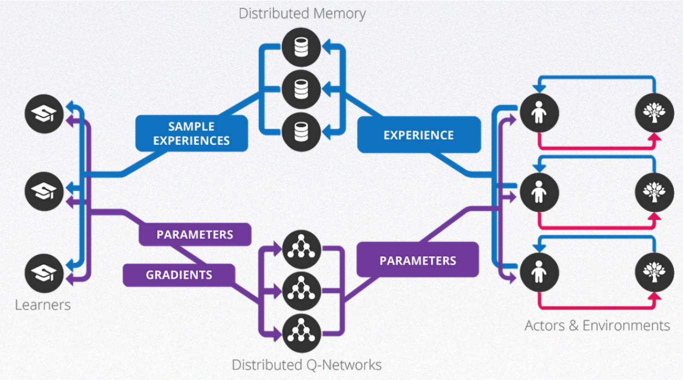
$$\left| R(s) + \gamma \max_{a'} \hat{Q}_{\vec{w}'}(s', a') - \hat{Q}_{\vec{w}}(s, a) \right|$$

- Dueling network: Split Q-network into two channels
  - Action-independent value network: $V_{\vec{v}}(s)$
  - Action-dependent advantage network: $A_{\vec{w}}(s, a)$

$$Q(s, a) \simeq V_{\vec{v}}(s) + A_{\vec{w}}(s, a)$$

- Combined with double DQN, 3x mean Atari score than Nature DQN.

Gorila (General Reinforcement Learning Architecture)



- 10x faster than Nature DQN on 38 out of 49 Atari games.
- Applied to recommender systems within Google.

Deep Policy Networks

- Represent stochastic policy by deep network with weights $\vec{u}$

$$a = \pi(a|s) \simeq \hat{\pi}_{\vec{u}}(a|s)$$

- Define objective function (maximization) as total rewards

$$L(\vec{u}) = \mathbb{E}\left[R_0 + \gamma R_1 + \gamma^2 R_2 + \ldots \mid \hat{\pi}_{\vec{u}}\right]$$

- Maximizing expected reward by adjusting policy parameters with SGD.

$$\frac{\partial L(\vec{u})}{\partial \vec{u}} = \mathbb{E}\left[\frac{\partial \log \hat{\pi}_{\vec{u}}(a|s)}{\partial \vec{u}} Q^{\hat{\pi}_{\vec{u}}}(s,a)\right]$$

Asynchronous Advantage Actor-Critic (A3C) 4x Atari score

- Estimate state-value function

$$V_{\vec{v}}(s_t) \simeq \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid s\right]$$

- Q-value estimated by an n-step sample

$$\hat{Q}_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V_{\vec{v}}(s_{t+n})$$

- Actor is updated towards target (maximizing)

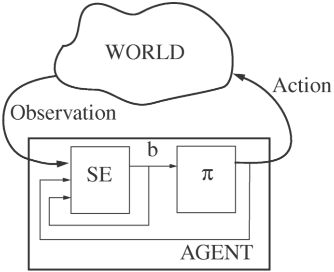
$$\frac{\partial L(\vec{u})}{\partial \vec{u}} = \frac{\partial \log \hat{\pi}_{\vec{u}}(a|s)}{\partial \vec{u}} (\hat{Q}_t - V_{\vec{v}}(s_t))$$

- Critic is updated to minimize MSE

$$l(\vec{v}) = \left(\hat{Q}_t - V_{\vec{v}}(s_t)\right)^2$$

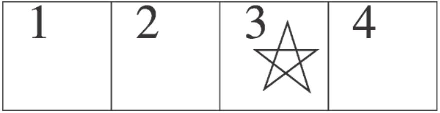
Partially Observable MDP (POMDP)

- A POMDP agent can be decomposed into a state estimator (SE) and policy (π)
- Observation: o
- Belief state: b



POMDP Example

0.9 to the directed action, 0.1 to the opposite direction.



- Initially, seeing no goal. $[0.33, 0.33, 0.0, 0.33]$
- After go east, if still no goal. $[0.10, 0.45, 0.0, 0.45]$
- After another east, if still no goal. $[0.10, 0.16, 0.0, 0.74]$

Computing Belief States

- The state estimator needs to compute a new belief state given an old belief state, an action, and an observation.

$$
\begin{aligned}
b'(s') &= P(s' \mid o, a, b) \\
&= P(o \mid s', a, b) \frac{P(s' \mid a, b)}{P(o \mid a, b)} \\
&= P(o \mid s', a) \frac{\sum_s P(s' \mid s, a, b) b(s)}{P(o \mid a, b)} \\
&\propto P(o \mid s', a) \sum_s P(s' \mid s, a) b(s)
\end{aligned}
$$

Policy and Value for POMDP

- Now everything is very alike the ordinary MDP.
- Except for doing things on $P(s'|\ s, a)$, now use $P(b'|\ b, a)$ instead.
- With little information (indistinguishable observations), RL is difficult to converge.
- Remedy: more information! (trivial, but no other way around).

Summary

- With perfect information of rewards, value iteration and policy iteration can be used to to find to true utility and optimal policy off-line.
- Reinforcement learning learns utility on-line by two approaches: ADP and TD, where TD learning does not need the information of the transition model.
- Instead of learning utility, Q-learning learns the action-utility function.
- For large state spaces, reinforcement learning adopts generalization to trade accuracy.
- By using deep networks for generalization, deep RL shines recent AI research.
- Bayesian learning and belief states are keys to solve POMDP.