Reinforcement Learning

- [Demos]

- **Basic idea:**
  - Receive feedback in the form of rewards
  - Must learn to act so as to maximize expected rewards
  - Agent’s utility is defined by the reward function
  - Change the rewards, change the behavior!

- **Examples:**
  - Playing a game, reward at the end for winning / losing
  - Vacuuming a house, reward for each piece of dirt picked up
  - Automated taxi, reward for each passenger delivered
Markov Decision Processes

- Markov decision processes (MDPs)
  - A set of states \( s \in S \)
  - A model \( T(s,a,s') = P(s' | s,a) \)
    - Probability that action \( a \) in state \( s \) leads to \( s' \)
  - A reward function \( R(s) \) (or \( R(s,a,s') \))

- MDPs are the simplest case of reinforcement learning
  - In general reinforcement learning, we don’t know the model or the reward function

MDP Solutions

- In state-space search, want an optimal sequence of actions from start to a goal
- In an MDP, want an optimal policy \( \pi(s) \)
  - A policy gives an action for each state
  - Optimal policy is the one which maximizes expected utility (i.e. expected rewards) if followed
  - Gives a reflex agent!

Optimal policy when \( R(s) = -0.04 \):
Example Optimal Policies

Stationarity

- In order to formalize optimality of a policy, need to understand utilities of reward sequences
- Typically consider stationary preferences:

\[
[r; r_0, r_1, r_2, \ldots] > [r; r_0', r_1', r_2', \ldots]
\]

\[
[r_0, r_1, r_2, \ldots] > [r_0', r_1', r_2', \ldots]
\]

- Theorem: only two ways to define stationary utilities
  - Additive utility:
    \[
    U([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \cdots
    \]
  - Discounted utility:
    \[
    U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots
    \]
How (Not) to Solve an MDP

- The inefficient way:
  - Enumerate policies
  - Calculate the expected utility (discounted rewards) starting from the start state
    - E.g. by simulating a bunch of runs
    - Choose the best policy
  - We’ll return to a (better) idea like this later

Utilities of States

- Idea: calculate the utility (value) of each state
  \[ U(s) = \text{expected (discounted) sum of rewards assuming optimal actions} \]
  - Given the utilities of states, MEU tells us the optimal policy
  \[
  \pi^U(s) = \arg \max_a E_{P(s'|s,a)} U(s')
  \]
  \[
  = \arg \max_a U(s') T(s, a, s')
  \]
Infinite Utilities?!

- Problem: infinite state sequences with infinite rewards

- Solutions:
  - Finite horizon:
    - Terminate after a fixed T steps
    - Gives nonstationary policy (\( \pi \) depends on time left)
  - Absorbing state(s): guarantee that for every policy, agent will eventually “die”
  - Discounting: for \( 0 < \gamma < 1 \)
    \[
    U([s_0, \ldots, s_\infty]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \frac{R_{\text{max}}}{1 - \gamma}
    \]
  - Smaller \( \gamma \) means smaller horizon

The Bellman Equation

- Definition of state utility leads to a simple relationship amongst utility values:

  Expected rewards = current reward + 
  \( \gamma \) x expected sum of rewards after taking best action

- Formally:

  \[
  U(s) = R(s) + \gamma \max_a E_{P(s'|a,s)} U(s')
  \]
  \[
  = R(s) + \gamma \max_a \sum_{s'} U(s') T(s, a, s')
  \]
  \[
  = R(s) + \gamma \sum_{s'} U(s') T(s, \pi^U(a), s')
  \]
**Example: Bellman Equations**

![Bellman Equation Example]

\[ U(1,1) = -0.04 + \gamma \text{ max}\{0.8U(1,2)+0.1U(2,1)+0.1U(1,1),\]
\[ 0.9U(1,1)+0.1U(1,2),\]
\[ 0.9U(1,1)+0.1U(2,1),\]
\[ 0.8U(2,1)+0.1U(1,2)+0.1U(1,1)\}\]

**Value Iteration**

- **Idea:**
  - Start with bad guesses at utility values (e.g. \( U_0(s) = 0 \))
  - Update using the Bellman equation (called a value update or Bellman update):
    \[ U_{i+1}(s) = R(s) + \gamma \max_a E_P(s'|a,s)U_i(s') \]
    \[ = R(s) + \gamma \max_a \sum_{s'} U_i(s')T(s,a,s') \]
  - Repeat until convergence

- **Theorem:** will converge to unique optimal values
  - Basic idea: bad guesses get refined towards optimal values
  - Policy may converge before values do
Example: Bellman Updates

\[ U_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} U_i(s')T(s, a, s') \]

\[ = 0 + 0.9 \sum_{s'} U_i(s')T(3, 3, \text{right}, s') \]

\[ = 0 + 0.9 [0.8 \cdot 1 + 0.1 \cdot 0 + 0.1 \cdot 0] \]

Example: Value Iteration

- Information propagates outward from terminal states and eventually all states have correct value estimates
- [DEMO]
Convergence*

- Define the max-norm: \( ||U|| = \max_s |U(s)| \)

- Theorem: For any two approximations U and V

\[
||U^{t+1} - V^{t+1}|| \leq \gamma ||U^t - V^t||
\]

  \[\text{i.e. any distinct approximations must get closer to each other, so, in particular, any approximation must get closer to the true U and value iteration converges to a unique, stable, optimal solution}\]

- Theorem:

\[
||U^{t+1} - U^t|| < \varepsilon, \Rightarrow ||U^{t+1} - U|| < 2\varepsilon/(1 - \gamma)
\]

  \[\text{i.e. one the change in our approximation is small, it must also be close to correct}\]

---

Policy Iteration

- Alternate approach:
  - Policy evaluation: calculate utilities for a fixed policy
  - Policy improvement: update policy based on resulting utilities

- Repeat until convergence

- This is policy iteration
  - Can converge faster under some conditions
Policy Evaluation

- If we have a fixed policy \( \pi \), use simplified Bellman equation to calculate utilities:

\[
U_{i+1}^\pi(s) = R(s) + \gamma \sum_{s'} U_i(s') T(s, \pi(s), s')
\]

Policy Improvement

- For fixed utilities, easy to find the best action according to one-step lookahead

\[
\pi_{i+1}^U(s) = \arg \max_a \sum_{s'} U(s') T(s, a, s')
\]
Comparison

- In value iteration:
  - Every pass (or “backup”) updates both policy (based on current utilities) and utilities (based on current policy)

- In policy iteration:
  - Several passes to update utilities
  - Occasional passes to update policies

- Hybrid approaches (asynchronous policy iteration):
  - Any sequences of partial updates to either policy entries or utilities will converge if every state is visited infinitely often

Next Class

- In real reinforcement learning:
  - Don’t know the reward function $R(s)$
  - Don’t know the model $T(s,a,s')$
  - So can’t do Bellman updates!

- Need new techniques:
  - Q-learning
  - Model learning
  - Agents actually have to interact with the environment rather than simulate it!