Markov Decisions Processes
Partially observable Markov decision Processes

Value Iteration for Motion Planning

Value Function and Policy Iteration

- Often the optimal policy has been reached long before the value function has converged.
- Policy iteration calculates a new policy based on the current value function and then calculates a new value function based on this policy.
- This process often converges faster to the optimal policy.

Previously

- Variations of MDP’s
- Continuous state MDP’s – discrete set of actions – Value function approximation
- Today – reinforcement learning
- finite horizon MDP’s
- LQR – continuous linear systems MDP’s
- Stochastic Policy search
- POMDP’s
Value Function Approximation

- Avoid visiting every state - use machine learning methods to approximate value functions
- Pick some function which is easy to compute approximate the value function with fewer parameters - (find the parameters such that the error will be minimized) e.g. linear regression
  \[ V^T(s) = \theta_0 + \theta_1 x + \theta_2 y \]
- After each trial get the values - solve for parameters

Reinforcement Learning

- Passive learning – policy is fixed (learn the utilities of states)
- Active learning – learn what to do (exploration/exploitation)
- MDP’s know transition function and reward function
- Now: do not know either

Types of environments
- deterministic
- stochastic

Reinforcement Learning

- All we can is
  1. act 2. perceive state 3. get reward
- Example trials, using some fixed policy

  (1,1) -0.04 \rightarrow (1,2) -0.04 \rightarrow (1,3) -0.04 \rightarrow ... \rightarrow (4,3) +1
  (1,1) -0.04 \rightarrow (1,2) -0.04 \rightarrow (1,3) -0.04 \rightarrow (2,3) -0.04 \rightarrow (3,3) ... \rightarrow (4,3) +1
  (1,1) -0.04 \rightarrow (2,1) -0.04 \rightarrow (3,1) -0.04 \rightarrow (3,2) -0.04 \rightarrow (4,2) -1

Use the information about rewards to learn the expected utility of each state

Passive Reinforcement Learning

- Learn the utility – expected sum of rewards
  \[ U^t(s) = E \left[ \sum_{s_0}^T R(s_0) | e, s_0 = s \right] \]
- Direct utility estimation (Widrow, Hoff 1960)
- At the end of each sequence calculate the observed reward-to-go for each state and update the utility – one sample
- Keep track of the average utility – over all visits to a particular state
- Does not exploit the information that the utilities of neighboring states are related
  \[ U^t(s) = R(s) + \gamma \sum_{s'} T(s, s', \alpha) U^t(s') \]
Adaptive dynamic programming

- How to update the utilities – such that they will be satisfy the Bellman equation
- We need to know how the states are related
- Need to learn at the same time the transition model
- keep track of frequencies of reaching \( x' \) from \( x \) by executing the action \( u \)
- Use the Bellman equation to determine the next utility value for each state – solve linear system of linear eq.
- Learns the utility function faster, exploits correlations between the states

Temporal Difference Learning

- Another way how to use the Bellman equation

\[ U^\pi(s) = R(s) + \gamma \sum_{s' \pi(s)} P(s, a, s') U^\pi(s') \]
\[ U^\pi(1, 3) = -0.04 + U^\pi(2, 3) \]

After few iterations the above constraint is not satisfied

- Adjust the value function based on difference between the utilities of successive states

\[ U^\pi(s) \leftarrow U^\pi(s) + \alpha (R(s) + \gamma U^\pi(s') - U^\pi(s)) \]

- Simpler – instead of doing value determination – just update the value

Active reinforcement learning

- Decide what actions to take – no fixed policy
- At each step – follow optimal policy given the current estimate of the utility function
- Greedy agents it mail fail to learn the correct utilities unless it explores also other states
- Choosing always optimal actions can lead overall to suboptimal results
- Fundamental trade-off exploitation (maximize its reward) and exploitation (maximize overall well being)
- Choose random action \( 1/t \) times – otherwise follow optimal policy – alternatively design some function which will tradeoff greed vs curiosity (taking an action which yields lower utility – but has not been tried often)

Q-learning

- Instead of learning utilities - learn action value function \( Q(s, a) \)

\[ Q(s, a) \leftarrow R(s) + \gamma \max_{a'} Q(s', a') \]

- Active TD Q-learning agent

\[ Q(s, a) \leftarrow Q(s, a) + \alpha (R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a)) \]

- TD learning – too expensive to store the value functions
- In large models it is very hard to learn visit every state
Policy search

- Policy maps states to actions
- We would like to learn directly the policy
- Parameterize the policy by a collection of functions

\[ \pi(s) = \max_a Q^\pi(s, a) \]

\[ v(s) = \text{softmax} Q^\pi(s, a) \]

Value of the policy

POMDPs

- In POMDPs we apply the very same idea as in MDPs.
- Since the state is not observable, the agent has to make its decisions based on the belief state which is a posterior distribution over states.
- Let \( b \) be the belief of the agent about the state under consideration.
- POMDPs compute a value function over belief space:

\[
V^\pi(b) = \gamma \max_u \left[ r(b, u) + \int V^\pi_{t-1}(b') p(b' | u, b) \, db' \right]
\]

Problems

- Each belief is a probability distribution, thus, each value in a POMDP is a function of an entire probability distribution.
- This is problematic, since probability distributions are continuous.
- Additionally, we have to deal with the huge complexity of belief spaces.
- For finite worlds with finite state, action, and measurement spaces and finite horizons, however, we can effectively represent the value functions by piecewise linear functions.

An Illustrative Example
The Parameters of the Example

- The actions $u_1$ and $u_2$ are terminal actions.
- The action $u_3$ is a sensing action that potentially leads to a state transition.
- The horizon is finite and $\gamma = 1$.

\[
\begin{align*}
  r(x_1, u_1) &= -100 \\
  r(x_1, u_2) &= +100 \\
  r(x_1, u_3) &= 0 \\
  r(x_2, u_1) &= -100 \\
  r(x_2, u_2) &= -50 \\
  r(x_2, u_3) &= -1 \\
  p(x'_1|x_1, u_3) &= 0.2 \\
  p(x'_1|x_1, u_3) &= 0.8 \\
  p(x'_2|x_2, u_3) &= 0.7 \\
  p(x'_2|x_2, u_3) &= 0.3 \\
  p(x_1|x_1) &= 0.7 \\
  p(x_2|x_2) &= 0.3
\end{align*}
\]

Payoff in POMDPs

- In MDPs, the payoff (or return) depended on the state of the system.
- In POMDPs, however, the true state is not exactly known.
- Therefore, we compute the expected payoff by integrating over all states:

\[
  r(b, u) = E_x[r(x, u)] = \int r(x, u)p(x) \, dx = p_1 r(x_1, u) + p_2 r(x_2, u)
\]

Payoffs in Our Example (1)

- If we are totally certain that we are in state $x_1$ and execute action $u_1$, we receive a reward of -100.
- If, on the other hand, we definitely know that we are in $x_2$ and execute $u_1$, the reward is +100.
- In between it is the linear combination of the extreme values weighted by the probabilities

\[
\begin{align*}
  r(b, u_1) &= -100 \cdot p_1 + 100 \cdot p_2 \\
  &= -100p_1 + 100(1 - p_1) \\
  r(b, u_2) &= 100p_1 - 50(1 - p_1) \\
  r(b, u_3) &= -1
\end{align*}
\]

Payoffs in Our Example (2)

\[
V_1(b) = \max_u r(b, u)
\]
The Resulting Policy for $T=1$

- Given we have a finite POMDP with $T=1$, we would use $V_1(b)$ to determine the optimal policy.
- In our example, the optimal policy for $T=1$ is
  \[
  \pi_1(b) = \begin{cases} 
  u_1 & \text{if } p_1 \leq \frac{3}{7} \\
  u_2 & \text{if } p_1 > \frac{3}{7} 
  \end{cases}
  \]
- This is the upper thick graph in the diagram.

Piecewise Linearity, Convexity

- The resulting value function $V_1(b)$ is the maximum of the three functions at each point
  \[
  V_1(b) = \max_{b} r(b, u) = \max \left\{ 
  \begin{array}{c}
  -100 \, p_1 + 100 \, (1 - p_1) \\
  100 \, p_1 - 50 \, (1 - p_1) \\
  -1
  \end{array}
  \right\}
  \]
- It is piecewise linear and convex.

Pruning

- If we carefully consider $V_1(b)$, we see that only the first two components contribute.
- The third component can therefore safely be pruned away from $V_1(b)$.
  \[
  V_1(b) = \max \left\{ 
  \begin{array}{c}
  -100 \, p_1 + 100 \, (1 - p_1) \\
  100 \, p_1 - 50 \, (1 - p_1)
  \end{array}
  \right\}
  \]

Increasing the Time Horizon

- Assume the robot can make an observation before deciding on an action.
- To be continued in Special Topics Class …
Additional topics

- LQR solving MDP’s exactly
- Finite horizon problems
- Policy search, Reinforce and Pegasus alg.

- Robotics Self Assembly
  http://www.youtube.com/ssrlab0/

- Weasle balls – sensorless control
  http://www.youtube.com/watch?v=P7vTZbpX5k&lr=1

- Petman Boston Dynamics
  http://www.youtube.com/watch?v=Dl4OuEjcP3o&feature=fvst

- Sand Flea jumping robots
  http://www.youtube.com/watch?v=6b4ZZQkcNEo

- Sand swimming robot

- Medical robotics – needle steering
  http://www.youtube.com/watch?v=yFbUvmsXX4

- Laundry folding
  http://www.youtube.com/watch?v=gy5g3350Gzo
Wumpus World PEAS description

- **Performance measure**
  - gold +1000, death -1000
  - -1 per step, -10 for using the arrow
- **Environment**
  - Squares adjacent to wumpus are smelly
  - Squares adjacent to pit are breezy
  - Glitter iff gold is in the same square
  - Shooting kills wumpus if you are facing it
  - Shooting uses up the only arrow
  - Grabbing picks up gold if in same square
  - Releasing drops the gold in same square
- **Sensors:** Stench, Breeze, Glitter, Bump, Scream
- **Actuators:** Left turn, Right turn, Forward, Grab, Release, Shoot

Wumpus world characterization

- **Fully Observable** No – only local perception
- **Deterministic** Yes – outcomes exactly specified
- **Episodic** No – sequential at the level of actions
- **Static** Yes – Wumpus and Pits do not move
- **Discrete** Yes
- **Single-agent?** Yes

Exploring a wumpus world

No stench or breeze in [1 1], nearby states are ok