Reinforcement Learning II

So far..

- MDP
- "Value of a state"
- Bellman's optimality equation
- Dynamic Programming
- Value Iteration/Policy Iteration
- Key Assumption: Models are known
  - What are the models? P & R
  - When are they known? Before learning
  - Unreasonable in many domains where we learn to act for the first time. Can we reuse models from other agents? What if they are not complete?
- Solution: Reinforcement "**Learning**"

What is RL?

Imagine playing a new game whose rules you don’t know; after a hundred or so moves, your opponent announces, “You lose”.

-Russell and Norvig

Introduction to Artificial Intelligence

What is different?

- Agent placed in an environment and must learn to behave optimally in it
- Assume that the world behaves like an MDP, except:
  - Agent can act but does ***not know*** the transition model P
  - Agent observes its current state and its reward but does ***not know*** the reward function (R is unknown)
- Goal: learn an optimal policy
- Challenge: How do you get to know the models (reward function and transition probabilities)
- Solution: Learn them as you explore the environment

$R \rightarrow$ observe $\qquad$ P – estimate

Why is RL difficult?

if $p(s'|s,a) = 1 \ \forall a,s \rightarrow$ deterministic

- Actions have non-deterministic effects – which are initially unknown and must be learned
- Rewards / punishments can be infrequent
  - Often at the end of long sequences of actions
  - How do we determine what action(s) were really responsible for reward or punishment? (credit assignment problem)
- World is large and complex

$\hookrightarrow$ curse of dimensionality.

Passive vs. Active learning

→ flow to act.

- Passive learning
  - The agent acts based on a fixed policy π and tries to learn how good the policy is by observing the world go by
  - Analogous to policy evaluation in policy iteration
- Active learning
  - The agent attempts to find an optimal (or at least good) policy by exploring different actions in the world
  - Analogous to solving the underlying MDP

Model-Based vs. Model-Free RL

- *Model based approach to RL:* → P
  - learn the MDP model (T and R), or an approximation of it
  - use it to find the optimal policy
- *Model free approach to RL:*
  - derive the optimal policy without explicitly learning the model

Qlearning

We will consider both types of approaches

Passive Learning: Monte Carlo Methods

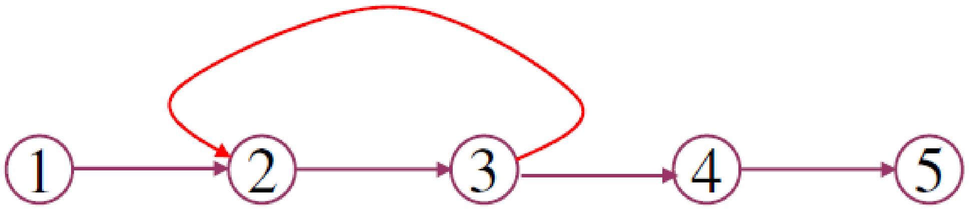
- Monte Carlo methods learn from complete sample returns
  - **Here**: only for episodic tasks
- Monte Carlo methods learn directly from experience
  - **On-line**: No model necessary and still attains optimality
  - **Simulated**: No need for a full model

Monte Carlo Policy Evaluation

Goal: learn $V^{\pi}(s)$

Given: some number of episodes under π which contain s

Idea: Average returns observed after visits to s



Every-Visit MC: average returns for *every* time s is visited in an episode

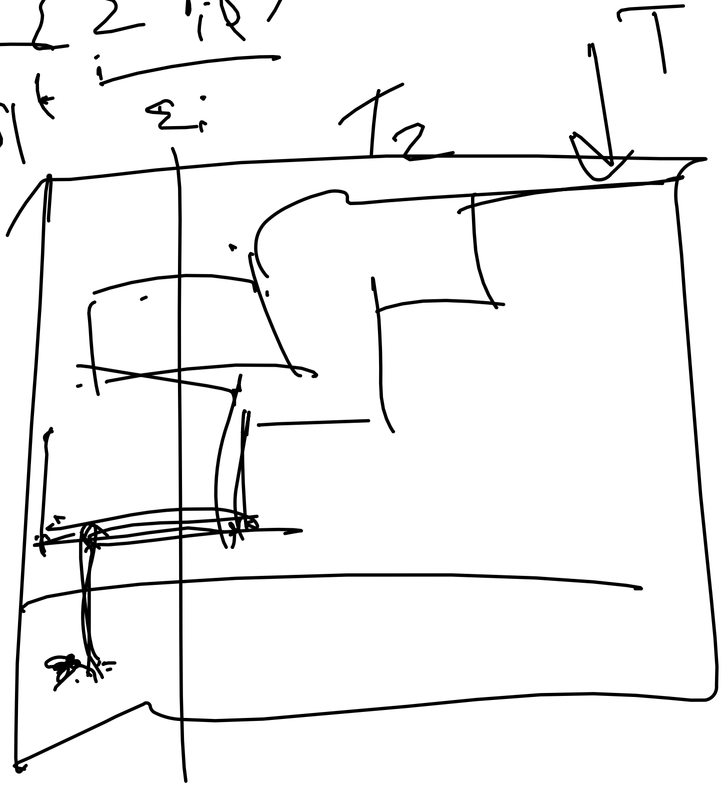
First-visit MC: average returns only for *first* time s is visited in an episode

Both converge asymptotically

$$R(r) = \frac{1}{n_{|S|}^{t}} \left\{ \sum_i r_i(s) \right\}$$

$\downarrow$

Every visit



T_1

s_0

0

G_{s1}

$:$

2

s_{10}

s_0

s_2

s_3

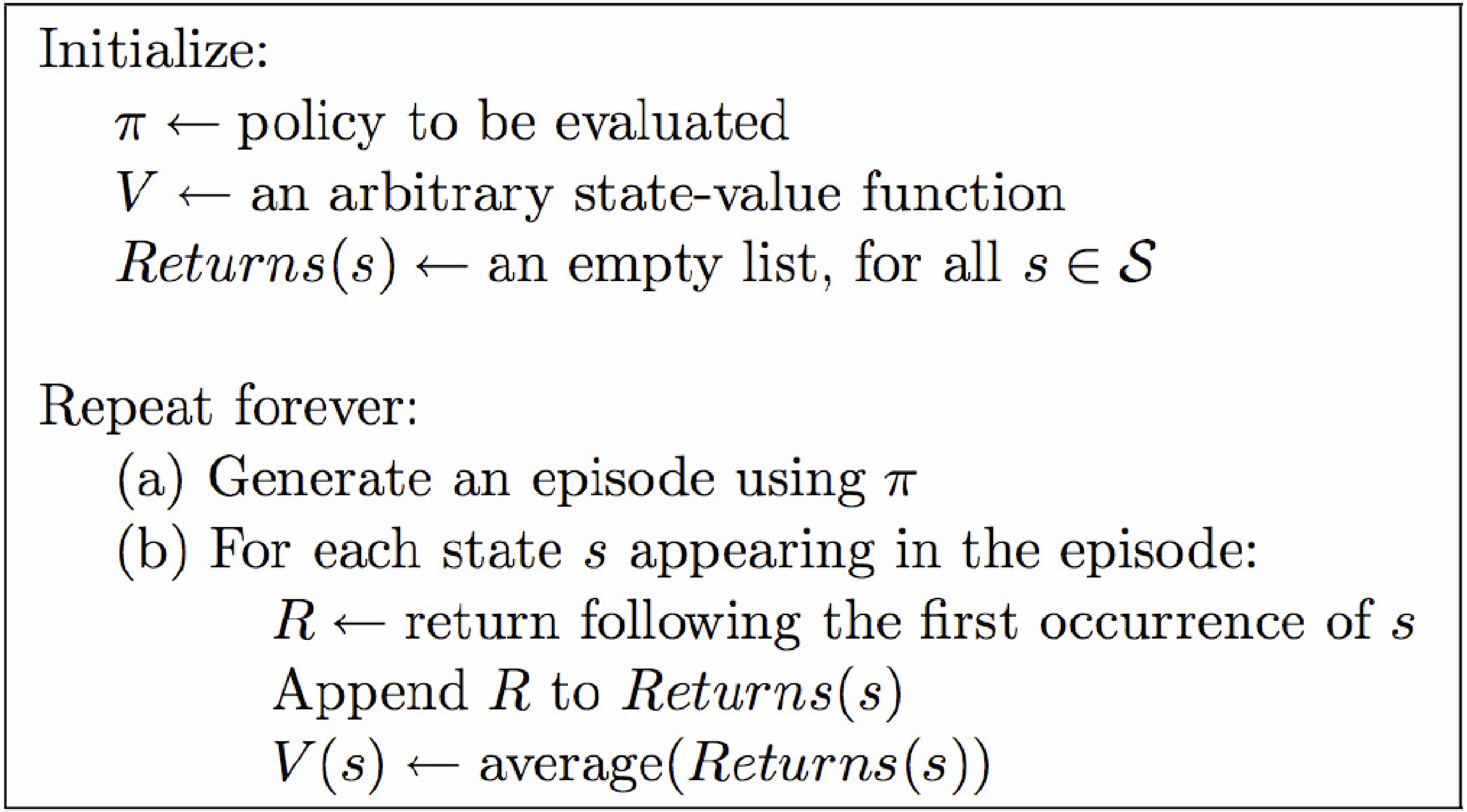
s_1

$:$

s_{10}

$$R(s) = \frac{\sum \text{firstvisit } t(s)}{n_{|S|}}$$

First Visit Monte Carlo policy evaluation



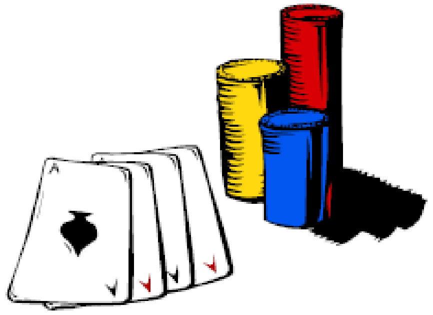
```
Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each state $s$ appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s$
            Append $R$ to $Returns(s)$
            $V(s) \leftarrow$ average($Returns(s)$)
```

Blackjack example

Object: Have your card sum be greater than the dealers without exceeding 21.

States (200 of them):

- current sum (12-21)
- dealer's showing card (ace-10)
- do I have a useable ace?

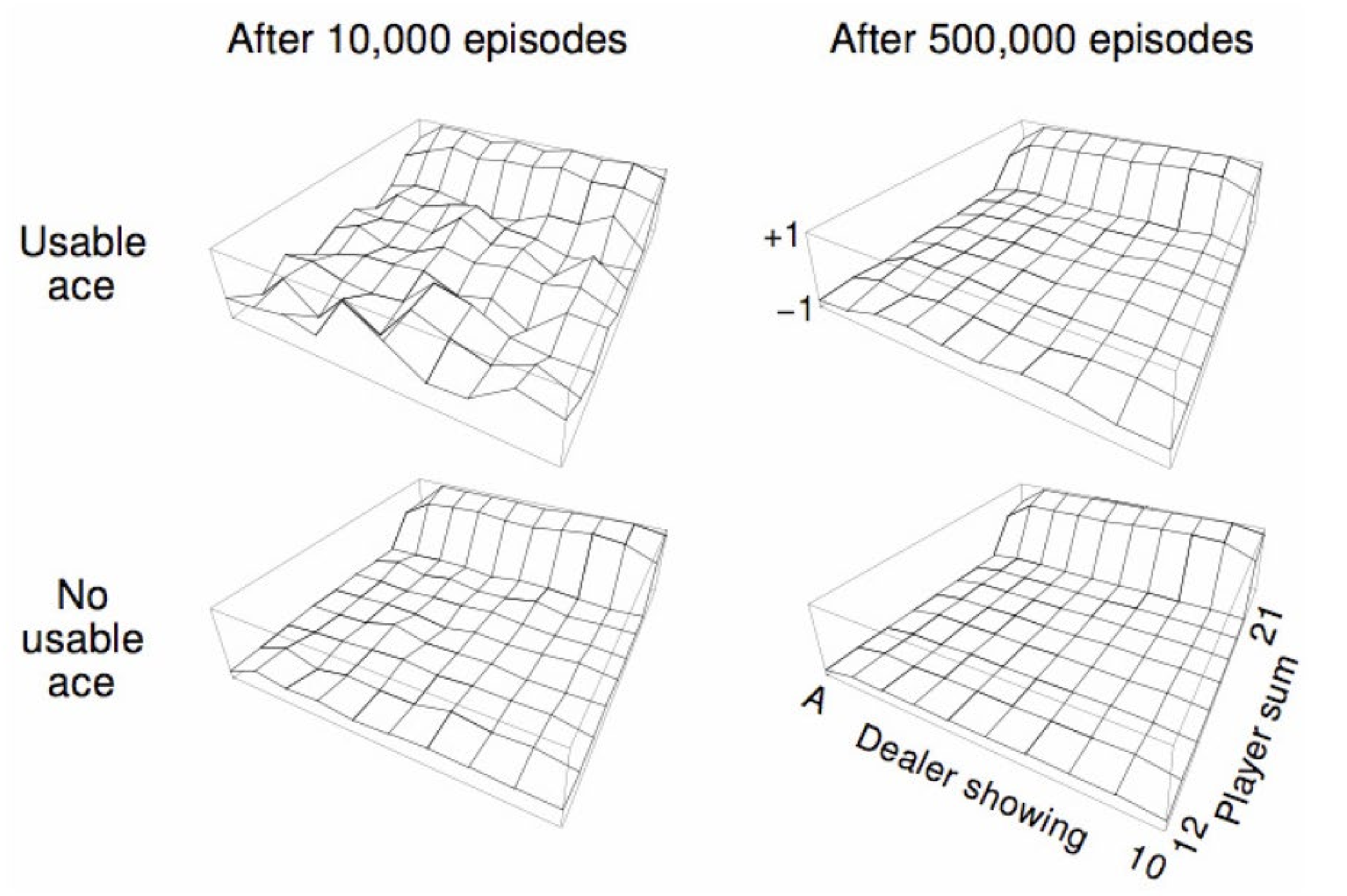


Reward: +1 for winning, 0 for a draw, -1 for losing

Actions: stick (stop receiving cards), hit (receive another card)

Policy: Stick if my sum is 20 or 21, else hit

Learned value functions



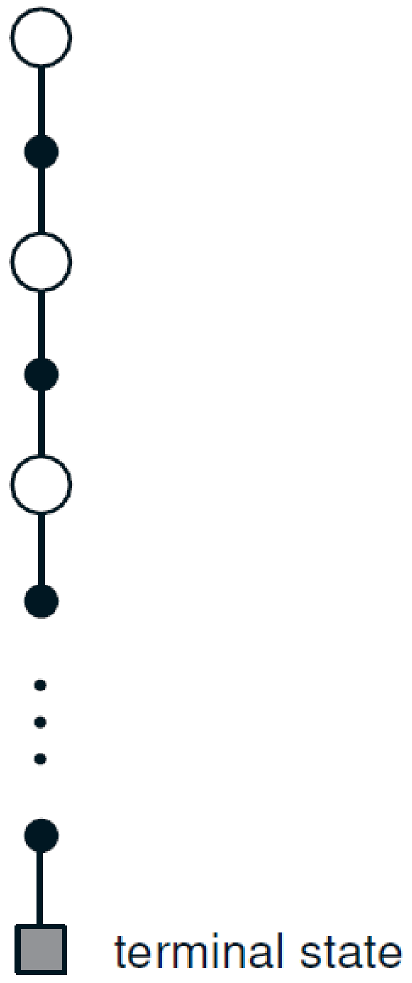
Monte Carlo Backup diagram

Entire episode included

Only one choice at each state (unlike DP)

MC does not bootstrap

Time required to estimate one state does not depend on the total number of states



Monte Carlo Estimation of Action Values (Q values)

Monte Carlo is most useful when a model is not available

- We want to learn Q^*

$Q^\pi(s,a)$ - average return starting from state s and action a following π

Also converges asymptotically *if* every state-action pair is visited

Exploring starts: Every state-action pair has a non-zero probability of being the starting pair

Instead of $V(s)\ \forall s$, estimate $Q(s,a)$ $\forall s,a$

Monte Carlo Control

$$\begin{tikzcd}
\pi \arrow[r, bend left, "\text{evaluation}", "Q \to Q^{\pi}"'] & Q \arrow[l, bend left, "\text{improvement}", "\pi \to \text{greedy}(Q)"']
\end{tikzcd}$$

MC policy iteration: Policy evaluation using MC methods followed by policy improvement

Policy improvement step: greedify with respect to value (or action-value) function

Approach 2: Adaptive Dynamic Programming

- Basically it learns the transition model **T** and the reward function **R** from the training sequences
- Based on the learned MDP (**T** and **R**) we can perform policy evaluation (which is part of policy iteration previously taught)

get reward r(s)

~~r(s,a)~~

ADP

Estimate

$P(s'|s,a) = \frac{n_{s',s,a}}{n_{s,a}}$

- ADP is a model based approach
  - Follow the policy for awhile
  - Estimate transition model based on observations
  - Learn reward function
  - Use estimated model to compute utility of policy

$$V^{\pi}(s) = R(s) + \beta \sum_{s'} T(s,a,s') V^{\pi}(s')$$

learned

'Passive'

- How can we estimate transition model T(s,a,s')?
  - Simply the fraction of times we see s' after taking a in state s.

Approach 3: Temporal Difference Learning

- Instead of calculating the exact utility for a state can we approximate it and possibly make it less computationally expensive?
- Yes we can! Using Temporal Difference (TD) learning

$$V_\pi(s) = R(s) + \gamma \underbrace{\sum_{s'} T(s, \pi(s), s') V_{\pi}(s')}$$

- Instead of doing this sum over all successors, only adjust the utility of the state based on the successor observed in the trial.
- It does not estimate the transition model – model free

Why is this correct? – Online Mean Estimation

- Suppose that we want to incrementally compute the mean of a sequence of numbers
  - E.g. to estimate the mean of a r.v. from a sequence of samples.

$$\hat{X}_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n+1}\left(x_{n+1} - \frac{1}{n}\sum_{i=1}^{n} x_i\right)$$

$$= \hat{X}_n + \frac{1}{n+1}\left(x_{n+1} - \hat{X}_n\right)$$

average of n+1 samples   learning rate   sample n+1

- Given a new sample x(n+1), the new mean is the old estimate (for n samples) plus the weighted difference between the new sample and old estimate

$$\bar{x}_n + \alpha(x_{n+1} - \bar{x}_n)$$

Temporal Difference Learning (TD)

- TD update for transition from s to s':

$$V^\pi(s) = \underline{V^\pi(s)} + \alpha(\underbrace{R(s) + \gamma V^\pi(s')}_{\text{New (noisy) sample of utility based on next state}} - \underline{V^\pi(s)})$$

learning rate

- So the update is maintaining a "mean" of the (noisy) utility samples
- If the learning rate decreases with the number of samples (e.g. 1/n) then the utility estimates will eventually converge to true values!

$$V^\pi(s) = R(s) + \gamma \sum_{s'} T(s,a,s') V^\pi(s')$$

or $V^{\pi}(s) = (1-\alpha)V^{\pi}(s) + \alpha\left[R(s) + \gamma V^{\pi}(s')\right]$

Temporal Difference Update

When we move from state s to s', we apply the following update rule:

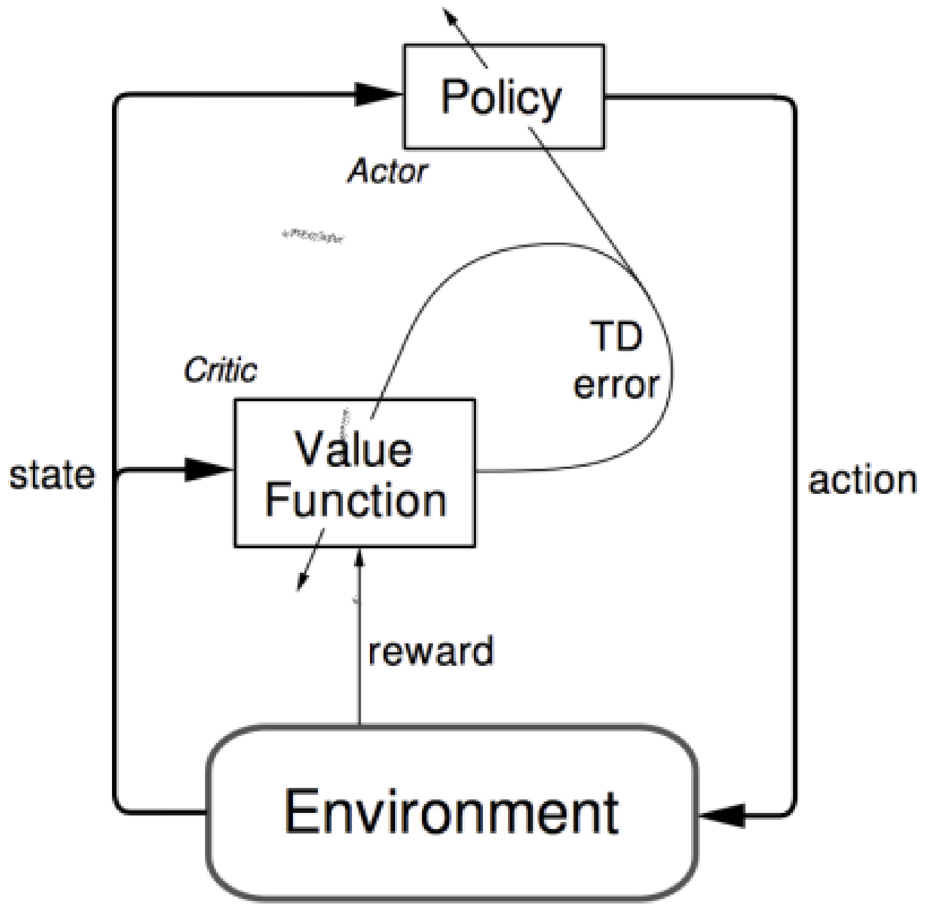
$$V^{\pi}(s) = V^{\pi}(s) + \alpha(R(s) + \gamma V^{\pi}(s') - V^{\pi}(s))$$

This is similar to one step of value iteration

We call this equation a "backup"

TD uses the observed states instead of the sum over all states that ADP uses.

Temporal Difference Learning



Comparison of Passive Learning methods

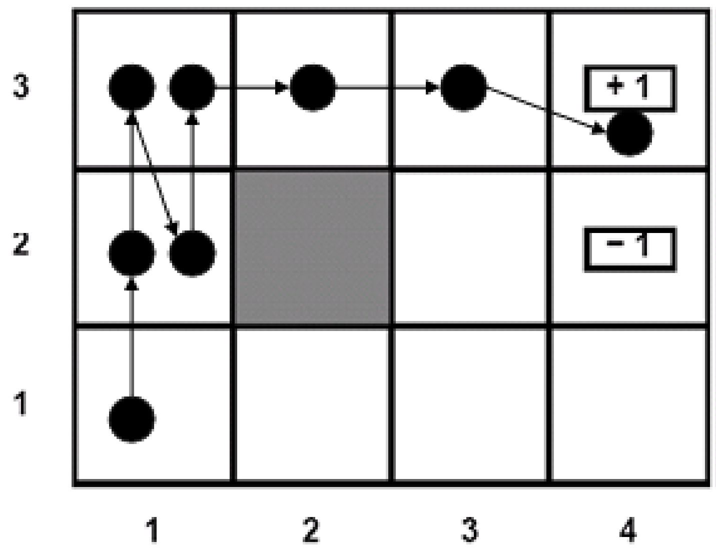
- Monte-Carlo Direct Estimation (model free)
  - Simple to implement
  - Each update is fast
  - Does not exploit Bellman constraints
  - Converges slowly
- Adaptive Dynamic Programming (model based)
  - Harder to implement
  - Each update is a full policy evaluation (expensive)
  - Fully exploits Bellman constraints
  - Fast convergence (in terms of updates)
- Temporal Difference Learning (model free)
  - Update speed and implementation similiar to direct estimation
  - Partially exploits Bellman constraints---adjusts state to 'agree' with observed successor
    - Not ***all*** possible successors
  - Convergence in between direct estimation and ADP

TD vs MC

- TD methods do not require a model of the environment, only experience
- TD, but not MC, methods can be fully incremental
  - You can learn before knowing the final outcome
- Less memory
- Less peak computation
  - You can learn without the final outcome
- From incomplete sequences

Passive learning

- Here we assume that the agent executes a fixed policy π
- The goal is to evaluate how good π is, based on some sequence of trials performed by the agent – i.e., compute $V^{\pi}(s)$



Learning $V^{\pi}(s)$ does not lead to a optimal policy, why?

• the models are incomplete/inaccurate

• the agent has only tried limited actions, we cannot gain a good overall understanding of T

• This is why we need active learning

Goal of Active Learning

- Let’s first assume that we still have access to some sequence of trials performed by the agent
  - The agent is not following any specific policy
  - We can assume for now that the sequences should include a thorough exploration of the space
  - We will talk about how to get such sequences later
- The goal is to learn an optimal policy from such sequences

Naïve Approach

1. Act Randomly for a (long) time
   - Or systematically explore all possible actions
2. Learn
   - Transition function
   - Reward function
3. Use value iteration, policy iteration, …
4. Follow resulting policy thereafter.

Will this work? Yes (if we do step 1 long enough and there are no "dead-ends")

Any problems? We will act randomly for a long time before exploiting what we know.

Revision of Naïve Approach

1. Start with initial (uninformed) model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Execute action suggested by policy in current state
4. Update estimated model based on observed transition
5. Goto 2

This is just ADP but we follow the greedy policy suggested by current value estimate

Will this work? No. Can get stuck in local minima.

What can be done?

Exploration versus Exploitation

- Two reasons to take an action in RL
  - **Exploitation**: To try to get reward. We exploit our current knowledge to get a payoff.
  - **Exploration**: Get more information about the world. How do we know if there is not a pot of gold around the corner.
- To explore we typically need to take actions that do not seem best according to our current model.
- Managing the trade-off between exploration and exploitation is a critical issue in RL
- Basic intuition behind most approaches:
  - Explore more when knowledge is weak
  - Exploit more as we gain knowledge

Active RL Exploration

- Several strategies exist for exploration vs exploitation trade-off
  - Greedy in the limit of infinite exploration – Choose the best action with probability *p* and explore with *1-p*. Increase *p* with time.
  - Boltzmann exploration: Choose an action with probability proportional to the q value of that action

$$\Pr(a \mid s) = \frac{\exp\left(Q(s,a)/T\right)}{\sum_{a' \in A} \exp\left(Q(s,a')/T\right)}$$

  - Optimistic exploration: Agent always acts greedily according to a model that assumes all “unexplored” states are maximally rewarding

Q-Learning

- Instead of learning the optimal value function V, directly learn the optimal Q function.
  - Recall Q(s,a) is the expected value of taking action a in state s and then following the optimal policy thereafter
- The optimal Q-function satisfies $V(s) = \max_{a'} Q(s,a')$ which gives:

$$Q(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') V(s')$$

$$= R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s,a')$$

- Given the Q function we can act optimally by selecting action greedily according to Q(s,a) without a model

How can we learn the Q-function directly?

Q-learning – Model-free Learning

Bellman constraints on optimal Q-function:

$$Q(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s,a')$$

- We can perform updates after each action just like in TD.
  - After taking action a in state s and reaching state s' do: (note that we directly observe reward R(s))

$$Q(s,a) \leftarrow Q(s,a) + \alpha(\underbrace{R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)}_{\text{(noisy) sample of Q-value based on next state}})$$

Q-learning: Model-free learning

$$\underbrace{Q(a,s)}_{\text{New estimate of Q(a,s)}} \leftarrow \underbrace{Q(a,s)}_{\text{Old estimate of Q(a,s)}} + \underbrace{\alpha}_{\text{Learning rate } 0 < \alpha < 1}\Big(\underbrace{R(s) + \gamma \max_{a'} Q(a',s') - Q(a,s)}_{\text{Difference between old estimate Q(a,s) and the new noisy sample after taking action a}}\Big)$$

Q-learning: Estimating the Policy

Q-Update: After moving from state s to state s' using action a:

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

Note that T(s,a,s') does not appear anywhere!

Further, once we converge, the optimal policy can be computed without T.

This is a completely model-free learning algorithm.

Q-learning

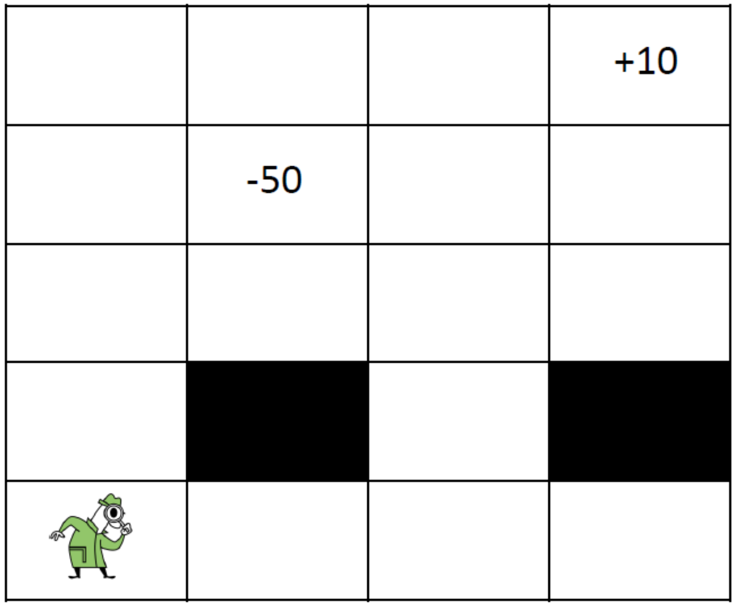
1. Start with initial Q-function (e.g. all zeros)
2. Take action from explore/exploit policy giving new state s' (should converge to greedy policy, i.e. GLIE)
3. Perform TD update

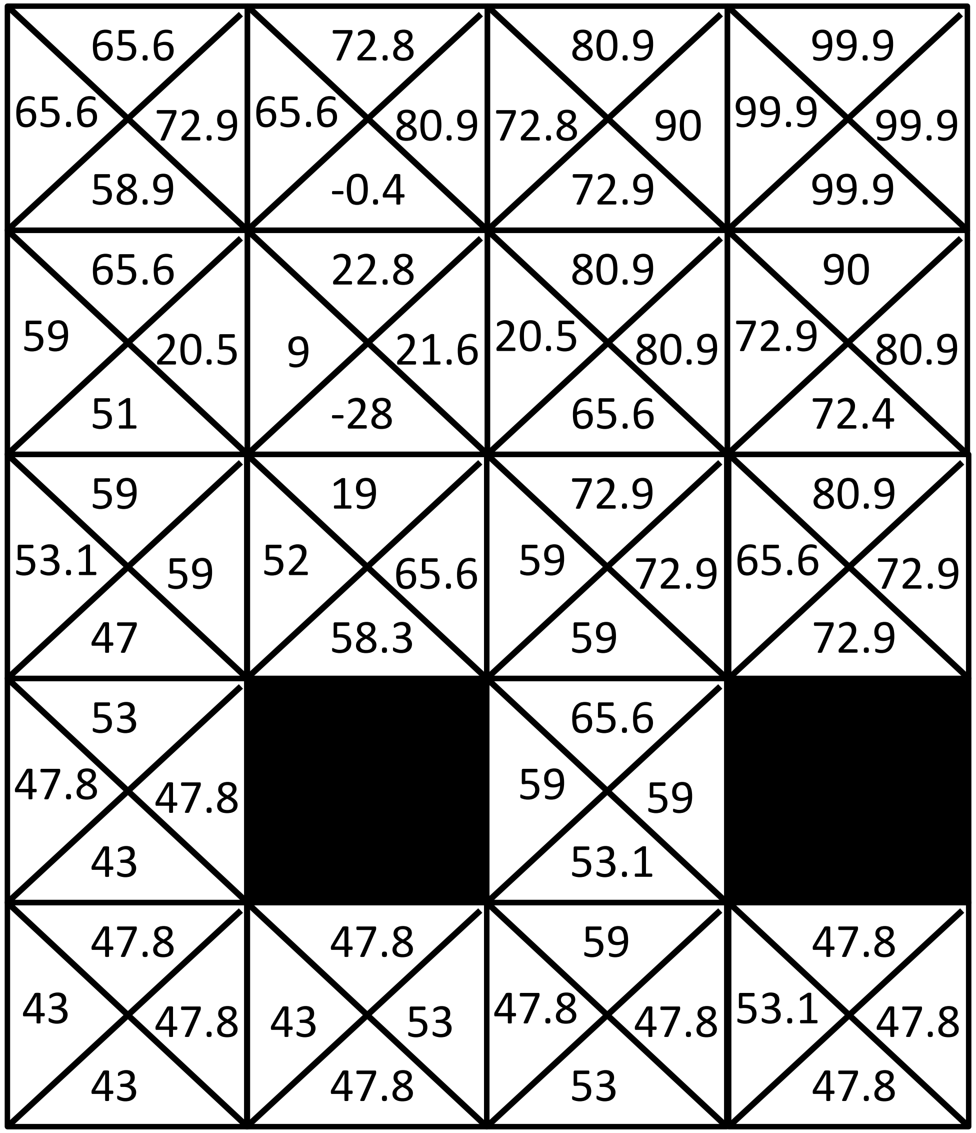
   $$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

   Q(s,a) is current estimate of optimal Q-function.
4. Goto 2

- Does not require model since we learn Q directly!
- Uses explicit |S|x|A| table to represent Q
- Explore/exploit policy directly uses Q-values
  - E.g. use Boltzmann exploration.

Q-values after 5000 iterations





$\varepsilon = 0.5$

$\alpha = 0.1$

$\gamma = 0.9$

Reduced $\varepsilon = \dfrac{\varepsilon}{1+\varepsilon}$ after every

10 iterations

(Some) Important Issues in Real World RL

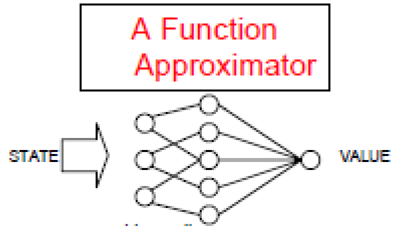
- Large State spaces
- Large number of objects
- Continuous states and continuous action spaces
- Partial Observability

Large State Spaces

- When a problem has a large state space we can not longer represent the V or Q functions as explicit tables
- Even if we had enough memory
  - Never enough training data!
  - Learning takes too long

- What to do??

Solution 1: Function Approximation

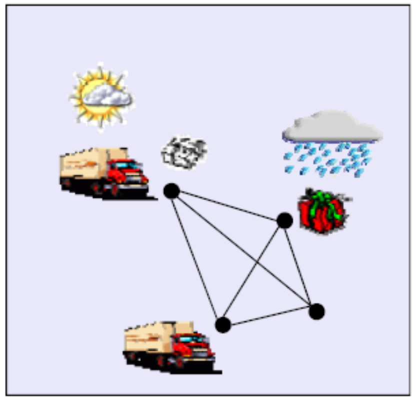


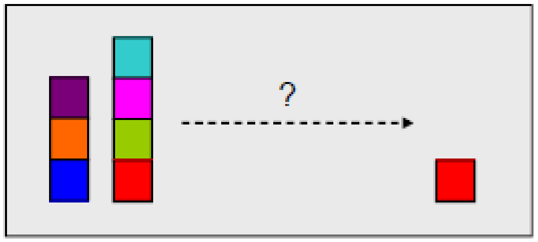
- Never enough training data!
  - Must generalize what is learned from one situation to other "similar" new situations
- Idea:
  - Instead of using large table to represent V or Q, use a parameterized function
    - The number of parameters should be small compared to number of states (generally exponentially fewer parameters)
  - Learn parameters from experience
  - When we update the parameters based on observations in one state, then our V or Q estimate will also change for other similar states
    - I.e. the parameterization facilitates generalization of experience
  - Examples are Linear functions, Neural networks etc.

Solution 2: Factored state spaces

- Solution: describe a state using a vector of features
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - 1 / (dist to dot)^2
    - Is Pacman in a tunnel? (0/1)
    - …… etc.
  - Can also describe a q-state (s, a) with features (e.g. action moves closer to food)

Large number of objects





- Many objects of various types in complex interactions
- Good agents can generalize across situations involving distinct object configurations
- Move many objects around with many other objects
- Identities and number of objects always changing
- Reasoning about relationship between objects key to solving the problem
- Would like a policy that is independent of the number of objects/blocks

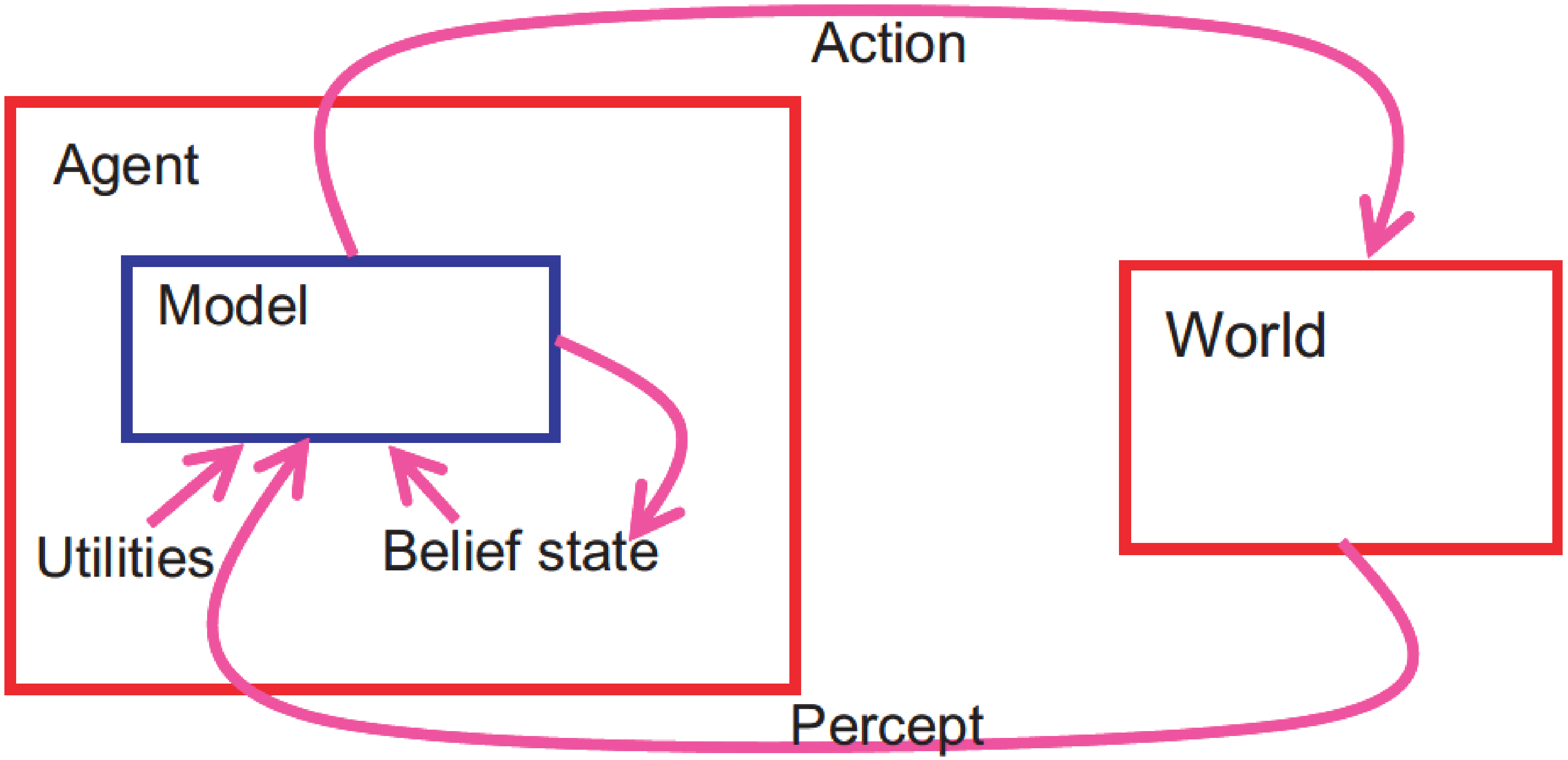
Relational MDPs (RMDPs)

- RMDPs are MDPs that make the notion of object explicit in the state and action spaces
- **States:** each state is described by set of objects, object properties, and relations among objects. E.g.
  - Objects: agent1, agent2, weapon1, location1, location2, ….
  - Properties: strength(agent1,100), empty(location1)
  - Relations: holding(agent1,weapon1), at(agent2,location2)
- **Actions:** parameterized by objects
  - Abstract actions: Attack(X,Y), Pickup(X,Y)
  - Ground actions: Attack(agent1,agent2), Pickup(agent1,weapon1)
  - Effects of actions can change properties of and relationships among objects, or sometimes even create new objects
- **Reward**: depends on object properties and relations
  - # of packages delivered, # of enemies killed

Relational RL

- RMDPs with fixed set of objects can be “propositionalized”
  - Describe states via traditional feature vectors that list values of all properties and relations.
    - [on(a,b)=true, on(b,a)=false, ontable(a)=false, ontable(b)=true]
- Can then directly apply feature-based RL to this representation
  - Loses the relational structure provided by objects
  - Policies can’t be applied directly to new object domains
  - Can be difficult to learn from such large feature vectors
- Relational RL attempts to learn policies that directly exploit relational structure:
  - Faster learning w.r.t. propositionalization even with fixed # of objects
  - Learn policies that generalize across object domains

Partial Observability



POMDPs

- A special case of the Markov Decision Process (MDP). In an MDP, the environment is fully observable, and with the Markov assumption for the transition model, the optimal policy depends only on the current state.
- For POMDPs, the environment is only partially observable

POMDP Implications

- Since current state is not necessarily known, agent cannot execute the optimal policy for the state.
- A POMDP is defined by the following:
  - Set of states *S*, set of actions *A*, set of observations *O*
  - Transition model *T(s, a, s')*
  - Reward model *R(s)*
  - Observation model *O(s, o)* – probability of observing observation *s* in state *o*.

POMDP Implications (cont.)

- Optimal action depends not on current state but on agent’s current *belief state*.
  - Belief state is a probability distribution over all possible states
- Given a belief state, if agent does an action *a* and perceives observation *o*, new belief state is
  - $b'(s') = \alpha\, O(s', o)\, \Sigma\, T(s, a, s')\, b(s)$
- Optimal policy $\pi^*(s)$ maps from belief states to actions

POMDP Solutions

- Solving POMDP on a physical state space is equivalent to solving an MDP on the belief state space
- However, state space is continuous and very high-dimensional, so solutions are difficult to compute.
- Even finding approximately optimal solutions is PSPACE-hard (i.e. **really** hard)

Why Study POMDPs?

- In spite of the difficulties, POMDPs are still very important.
  - Many real-world problems and situations are not fully observable, but the Markov assumption is often valid.
- Active area of research
  - Google search on “POMDP” returns ~5000 results
  - A number of current papers on the topic