INF5390 - Kunstig intelligens

Reinforcement Learning

Roar Fjellheim

Outline

- Reinforcement learning
- Sequential decision processes
- Passive learning
- Active learning
- RL applications
- Summary

AIMA Chapter 17: Making Complex Decisions
AIMA Chapter 21: Reinforcement Learning

Reinforcement learning

- Reinforcement learning (RL) is *unsupervised* learning: The agent receives no examples, and starts with no model or utility information
- The agent must use trial-and-error, and receives *rewards*, or *reinforcement*, to guide learning
- Examples:
  - √ Learning a game by making moves until lose or win: Reward only at the end
  - √ Learning to ride a bicycle without any assistance: Rewards received more frequently
- RL can be seen to encompass all of AI: An agent must learn to behave in an unknown environment

Variations of RL

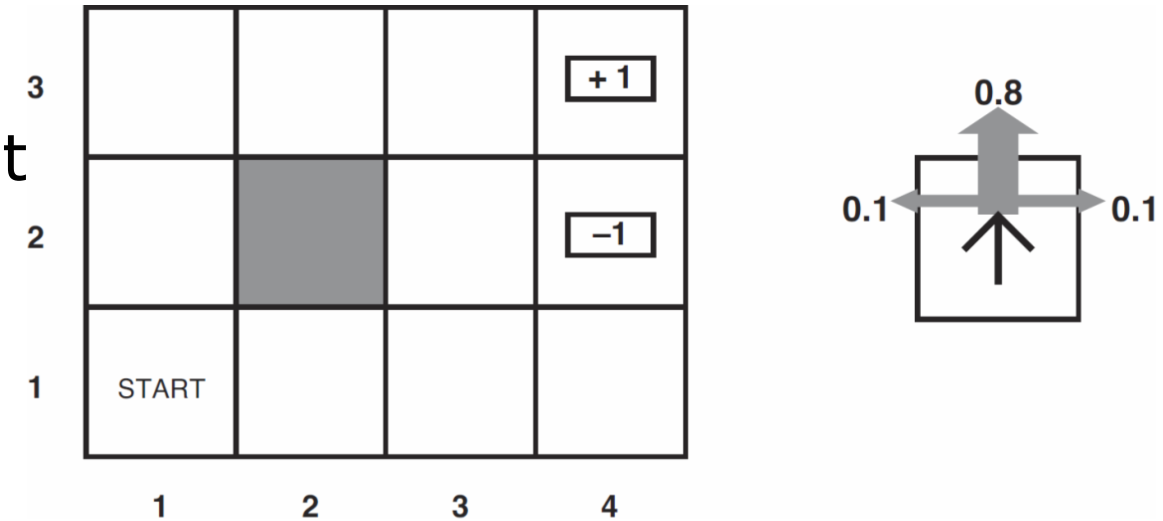
- *Accessible* environment (agent can use percepts) vs. *inaccessible* (must have some model)
- Agent may have some *initial* knowledge, or not have any domain model
- Rewards can be received only in *terminal* states, or in *any state*
- Rewards can be *part* of the actual utility, or just *hint* at the actual utility
- The agent can be a *passive* (watching) or an *active* (exploring) learner
- RL uses results from *sequential decision processes*

Sequential decision processes

- In a sequential decision process, the agent's utility depends on a sequence of decisions
- Such problems involve *utility* (of states) and *uncertainty* (of action outcomes)
- The agent needs a *policy* that tells it what to do in any state it might reach: $a = \pi(s)$
- An optimal policy $\pi^*(s)$ is a policy that gives the highest expected utility

Example sequential decision process

- 4x3 environment
- Agent starts in (1,1)
- Probabilistic transition model
  - √ Sequence [*Up, Up ,Right, Right, Right*] only has a probability of $0.8^5 = 0.33$ of reaching +1 (4,3)
- Terminal rewards are +1 (4,3) and -1 (4,2)
- Other state rewards are -0.04



Markov Decision Processes (MDP)

- An MDP is a sequential decision process with the following characteristics
- Fully observable environment (agent knows where it is at any time)
- State transitions are *Markovian*, i.e. $P(s'|s,a)$ - probability of reaching *s'* from *s* by action *a* depends only on *s* and *a*, not earlier state history
- Agent receives a reward $R(s)$ in each state *s*
- The total utility $U(s)$ of *s* is the sum of the rewards received, from *s* until a terminal state is reached

Optimal policy and utility of states

- The utility of a state *s* depends on rewards received but also on the policy π followed

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} R(S_t)\right]$$

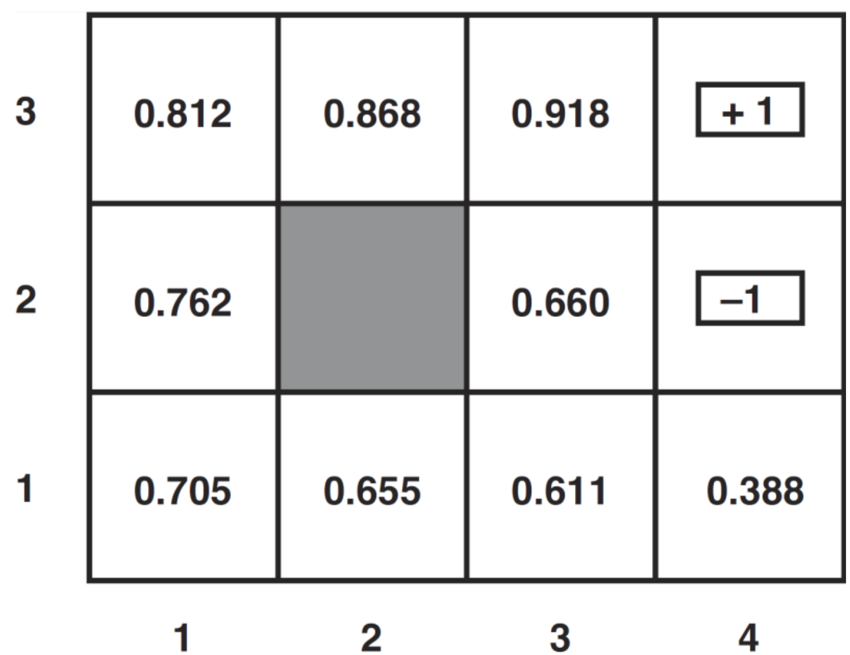
- Of all the possible policies the agent could follow, one gives the highest expected utility

$$\pi^*{}_s = \underset{\pi}{\operatorname{argmax}}\, U^{\pi}(s)$$

- This is the *optimal policy.* Under certain assumptions, it is independent of starting state

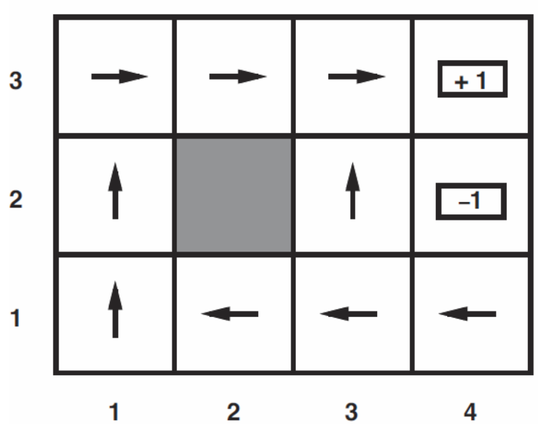
Utility of states (cont.)

- For an MDP with known transition model, reward function and assuming the optimal policy, we can calculate the utility *U(s)* of each state *s*
- For the 4x3 example, the utilities are:

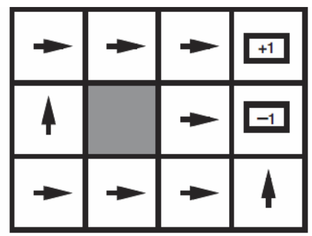


Optimal policy

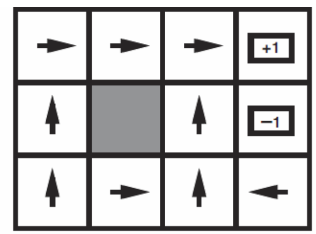
- Knowing *U(s)* allows the agent to select the optimal action:
$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P\left(s' \middle| s,a\right) U\left(s'\right)$$
- Optimal policy depends on non-final reward *R(s)*



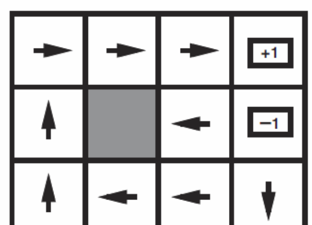
R(s) = -0.04



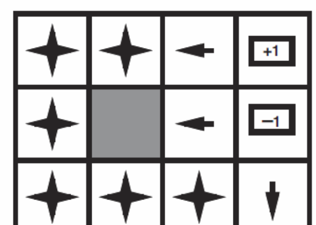
$R(s) < -1.6284$



$-0.4278 < R(s) < -0.0850$



$-0.0221 < R(s) < 0$



$R(s) > 0$

Bellman equations

- We need to be able to calculate utilities *U(s)* in order to define optimal policy
- Can exploit dependence between states: The utility of a state *s* is the reward *R(s)* plus the maximum expected utility of the next state

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

- This is the *Bellman equation.* There are *n* equations for *n* states, containing utilities *U(s)* as *n* unknowns
- Solving the equations yields the utilities *U(s)*

Value iteration

- The Bellman equations are *nonlinear* (due to the *max* opr.) and cannot be solved by linear algebra. Can use an *iterative* approach instead
  - √ Start with arbitrary initial values
  - √ Calculate right hand side
  - √ Plug the value into left-hand sides *U(s)*
  - √ Iterate until the values stabilize (within a margin)
- This VALUE-ITERATION algorithm is guaranteed to converge and to produce unique solutions

Policy iteration

- Instead of iterating to find *U(s)* and derive an optimal policy, we can iterate directly in policies
- We can iterate the policy to get an optimal one:
  - √ Policy evaluation: Given a policy π_i, calculate U_i, the utility of each state if π_i is followed
  - √ Policy improvement: Calculate new policy π_{i+1} that selects the action that maximizes successor state value (MEU)
  - √ Repeat until values no longer change
- This POLICY-ITERATION algorithm is guaranteed to converge and to produce an optimal policy

Reinforcement learning of MDP

- We could find an optimal policy for an MDP if we know the transition model $P(s'|s,a)$
- However, an agent in an *unknown environment* does not know the transition model nor in advance what rewards it will get in new states
- We want the agent to learn to behave rationally in an unsupervised process

- **The purpose of RL is to learn the optimal policy based only on received rewards**

Different RL agent designs

- **Utility-based** agents learn a utility function on states and uses it to select actions that maximize expected utility
  - √ Requires also a model of where actions lead
- **Q-learning** agents learn an action-utility function (*Q-function*), giving expected utility of taking a given action in a given state
  - √ Can select actions without knowing where they lead, at the expense not being able to look ahead
- **Reflex** agents learn a policy that maps directly from states to actions, i.e. π^*

Direct utility estimation

- In passive learning, the agent's policy π is fixed, it only needs to how good it is
- Agent runs a number of *trials,* starting in (1,1) and continuing until it reaches a terminal state
- The utility of a state is the expected total remaining reward (*reward-to-go*)
- Each trial provides a *sample* of the reward-to-go for each visited state
- The agent keeps a running average for each state, which will converge to the true value
- This is a *direct utility estimation* method

Example: Direct utility estimation

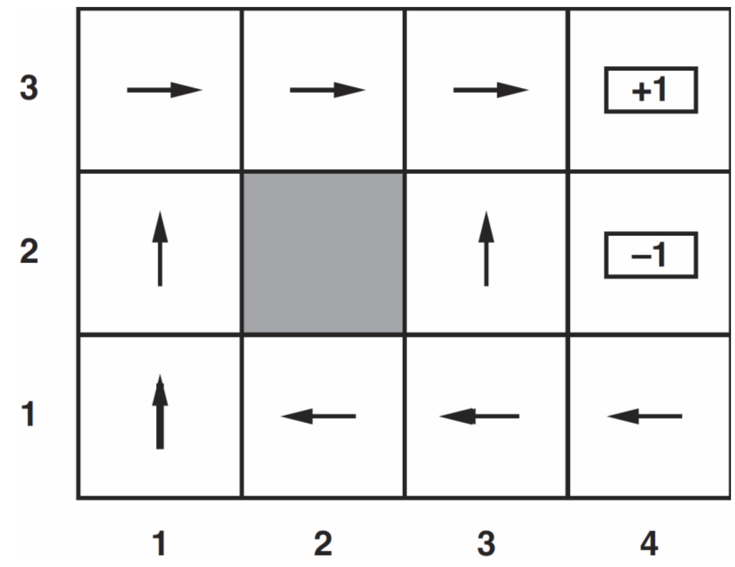
- Training trials for (4,3) matrix

$$(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (3,2)_{-0.04} \rightarrow (4,3)_{+1}$$

$$(1,1)_{-0.04} \rightarrow (2,1)_{-0.04} \rightarrow (3,1)_{-0.04} \rightarrow (3,2)_{-0.04} \rightarrow (4,2)_{-1}$$

Etc.

- Sample *U(s)* in first trial
  - (1,1) 0.72
  - (1,2) 0.76 and 0.84
  - (1,3) 0.80 and 0.88
  - Etc.
- Direct utility estimation converges slowly



Exploiting state dependencies

- Direct utility fails to exploit the fact that states are dependent as shown by Bellman equations

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

- Learning can be speeded up by using these dependencies
- Direct utility estimation can be seen to search a too large hypothesis space that contains many hypotheses violating Bellman equations

Adaptive Dynamic Programming (ADP)

- An ADP agent uses dependencies between states to speed up value estimation
- It follows a policy π and can use observed transitions to incrementally build the transition model $P(s'|s,\pi(s))$
- It can then plug the learned transition model and observed rewards $R(s)$ into the Bellman equations to get $U(s)$
  - The equations are linear because there is no *max* operator, and therefore easier to solve
- The result is $U(s)$ for the given policy π

Temporal Difference (TD) learning

- TD is another passive utility value learning algorithm using Bellman equations
- Instead of solving the equations, TD uses the observed transitions to adjust the utilities of the observed states to agree with Bellman
- TD uses a *learning rate* parameter α to select the rate of change of utility adjustment
- TD does not need a transition model to perform its updates, only the observed transitions

Active reinforcement learning

- While a passive RL agent executes a fixed policy π, an *active RL agent* has to decide which actions to take
- An active RL agent is an extension of a passive one, e.g. the passive ADP agent, and adds
  - √ Needs to learn a complete transition model for *all* actions (not just π), using passive ADP learning
  - √ Utilities need to reflect the optimal policy π^*, as expressed by the Bellman equations
  - √ Equations can be solved by the VALUE-ITERATION or POLICY-ITERATION methods described before
  - √ Action to be selected as the optimal/maximizing one

Exploration behavior

- The active RL agent may select maximizing actions based on a faulty learned model, and fail to incorporate observations that might lead to a more correct model
- To avoid this, the agent design could include selecting actions that lead to more correct models at the cost of reduced immediate rewards
- This called *exploitation* vs. *exploration* tradeoff
- The issue of *optimal* exploration policy is studied in a subfield of statistical decision theory dealing with so-called *bandit problems*

Q-learning

- An *action-utility* function *Q* assigns an expected utility to taking a given action in a given state: *Q(a,s)* is the value of doing action *a* in state *s*
- Q-values are related to utility values:

$$U(s) = \max_a Q(a,s)$$

- Q-values are sufficient for decision making *without* needing a transition model *P(s'|s,a)*
- Can be learned directly from rewards using a TD-method based on an update equation (s -> s'):

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

Generalization in RL

- In simple domains, U and Q can be represented by *tables*, indexed by state *s*
- However, for large state spaces the tables will be too large to be feasible, e.g. chess 10^{40} states
- Instead *functional approximation* can sometimes be used, e.g. $\breve{U}(s) = \sum parameter_i \times feature_i(s)$
- Instead of e.g. 10^{40} table entries, *U* can be estimated by e.g. 20 *parameterized* features
- Parameters can be found by supervised learning
- Problem: Such a function may not exist, and learning process may therefore fail to converge

Policy search

- A policy π maps states to actions: $a = \pi(s)$, and *policy search* tries to derive π directly
- Normally interested in *parameterized* policy in order to get a compact representation
- E.g., π can be represented by a collection of functional approximations $\hat{Q}(s,a)$, one per action *a*, and policy will be to maximize over *a*

$$\pi(s) = \max_a \hat{Q}(s,a)$$

- Policy search has been investigated in continuous/ discrete and deterministic/stochastic domains

Some examples of reinforcement learning

- Game playing
  - Famous program by Arthur Samuel (1959) to play *checkers* used linear evaluation function for board positions, updated by reinforcement learning
  - *Backgammon* system (Tesauro, 1992) used TD-learning and self-play (200.000 games) to reach level comparable to top three human world masters
- Robot control
  - *Inverted pendulum* problem used as test case for several successful reinforcement learning programs, e.g. BOXES (Michie, 1968) learned to balance pole after 30 trials

Summary

- *Reinforcement learning* (RL) examines how the agent can learn to act in an unknown environment just based on percepts and rewards
- Three RL designs are *model-based*, using a model *P* and utility function *U*, *model-free*, using action-utility function *Q,* and *reflex,* using a *policy*
- The *utility of a state* is the expected sum of rewards received up to the terminal state. Three methods are direct estimation, Adaptive dynamic programming (ADP), and Temporal-Difference (TD)

Summary (cont.)

- *Action-value function* (Q-functions) can be learned by ADP or TD approaches
- In *passive* learning the agent just observes the environment, while an *active* learner must select actions to trade off immediate reward vs. *exploration* for improved model precision
- In domains with very large state spaces, utility tables *U* are replaced by approximate *functions*
- Policy search work directly on a representation of the policy, improving it in an iterative cycle
- Reinforcement learning is a very active research area, especially in robotics