22년 삼성 AI 전문가과정
6월 9일 목요일 2교시
장병탁

인공지능

17차시 : Reinforcement Learning

서울대학교 컴퓨터공학부
담당 교수: 장병탁

Seoul National University
Byoung-Tak Zhang

Lecture Overview

인공지능

17차시 : Reinforcement Learning

서울대학교 컴퓨터공학부
담당 교수: 장병탁

Seoul National University
Byoung-Tak Zhang

Introduction: Reinforcement Learning

- **Supervised Learning** (Previous lectures)
  - With **supervised learning**, an agent learns by passively observing example input/output pairs provided by a "teacher."
  - **Deep learning** models also learn this way, including feedforward neural networks, convolutional neural networks, and recurrent neural networks.
- **Reinforcement Learning** (This lecture)
  - In reinforcement learning, the agents can actively learn from their own experience, without a teacher, by considering their own ultimate success or failure.
  - We see how experiencing rewards and punishments can teach an agent how to maximize rewards in the future
  - Passive/Active RL, Generalization in RL, Policy Search, Inverse RL, etc.

Problems with Supervised Learning

What's wrong with supervised learning?

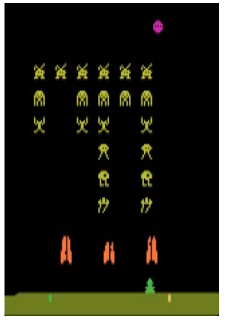
- A supervised learning agent learns by passively observing example input/output pairs provided by a "teacher"
- Can we train the chess agent with supervised learning?
  - Data: examples of chess positions (each labeled with the correct move)
  - Available training data set for winners: 10^8
  - The space of all possible chess positions: 10^{40}

→ Chess cannot be solved by supervised learning

Reinforcement Learning

Reinforcement learning (RL)

- An agent interacts with the world and periodically receives rewards (reinforcements) that reflect how well it is doing
  - For example, in case of chess, the reward is 1 for winning, 0 for losing, and $\frac{1}{2}$ for a draw
  - As long as we can provide the correct reward signal to the agent, RL provides a very general way to build AI systems (i.e. build right reward function!)
- Deep Reinforcement Learning (Deep Learning + RL) Applications



Deep Q-learning (Mnih et al., 2013)

Multi-task Robotic RL (Chebotar et al., 2021)

Poker AI (Brown et al., 2017)

Reinforcement Learning

- Model-based vs. Model-free
- Passive learning agent vs. Active learning agent
- Exploration vs. Safe exploration
  - Exploration, Exploitation, Bayesian RL
- Generalization in Reinforcement Learning
  - Function approximation, Deep RL, Reward shaping, Hierarchical RL
- Apprenticeship Learning
  - Imitation learning, Inverse reinforcement learning

Outline (Lecture 17)

Stuart Russell & Peter Norvig (2021), Artificial Intelligence: A Modern Approach (4th Edition)

17.1 Learning from Rewards

17.1 Learning from Rewards (1/4)

What's wrong with Supervised Learning (SL)?

- With supervised learning, an agent learns by passively observing example input/output pairs provided by a "teacher"

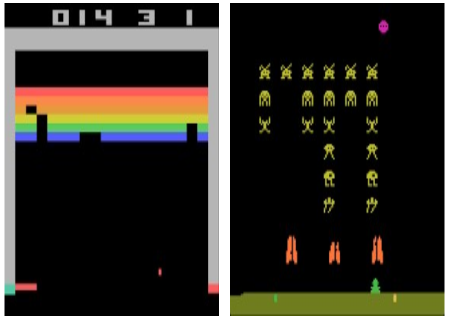
- Can we train the chess agent with supervised learning?
  - Data: examples of chess positions (each labeled with the correct move)
  - Few examples of winner: 10^8
  - The space of all possible chess positions: 10^{40}

→ Chess cannot be solved by supervised learning

17.1 Learning from Rewards (2/4)

Reinforcement Learning (RL)

- An agent interacts with the world and periodically receives rewards (reinforcements) that reflect how well it is doing
  - For example, in case of chess, the reward is 1 for winning, 0 for losing, and $\frac{1}{2}$ for a draw
  - As long as we can provide the correct reward signal to the agent, RL provides a very general way to build AI systems (i.e. build right reward function!)
- Deep Reinforcement Learning (Deep Learning + RL) Applications


Deep Q-learning (Mnih et al., 2013)


Multi-task Robotic RL (Chebotar et al., 2021)

Poker AI (Brown et al., 2017)

17.1 Learning from Rewards (3/4)

Categorization of Reinforcement Learning

- Model-based Reinforcement Learning
  - An agent uses a transition model of the environment.
  - The model may be initially unknown or already be known
    e.g. A chess program may know the rules of chess even if it does not know how to choose good move.
  - In partially observable environments, the transition model is also useful for state estimation.
  - Model-based RL systems often learn a utility function $U(s)$, defined in terms of the sum of rewards from state s onward.

17.1 Learning from Rewards (4/4)

Categorization of Reinforcement Learning

- Model-free Reinforcement Learning
  - An agent neither knows nor learns a transition model for the environment.
  - Instead, the agent learns a more direct representation of how to behave.
  - Action-utility learning
    - Most common form is Q-learning, where the agent learns a Q-function $Q(s, a)$
    - Given Q-function (sum of rewards from s and a), the agent can choose what to do in state s by finding the action with the highest Q-value
  - Policy search
    - The agent learns a policy $\pi(s)$ that maps directly from states to actions.
    - This is a reflex agent.


17.2 Passive Reinforcement Learning
VERI TAS LUX MEA

17.2 Passive Reinforcement Learning (1/6)

Passive learning agent

- An agent already has a fixed policy $\pi(s)$ that determines its actions.
- The agent is trying to learn the utility function $U^{\pi}(s)$—the expected total discounted reward if policy π is executed beginning in state s.
- Passive learning task is similar to the policy evaluation task (part of policy iteration algorithm).

- What are the differences between passive learning and policy evaluation?
  - Passive learning agent does not know the transition model $P(s'|s,a)$.
  - Passive learning agent does not know the reward function $R(s,a,s')$.

17.2 Passive Reinforcement Learning (2/6)

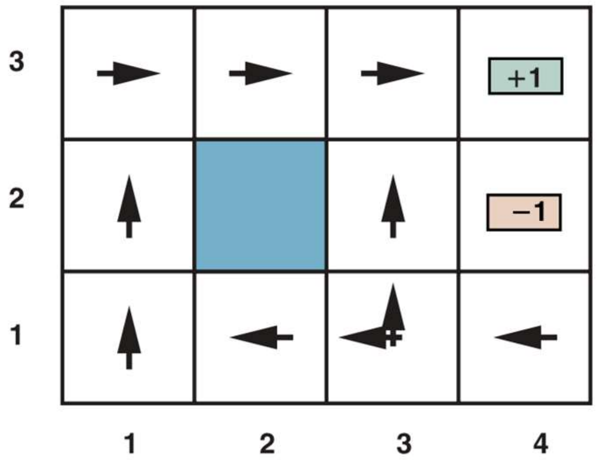
Passive learning agent

- The optimal policies for the 4×3 world
  - $R(s, a, s') = -0.04$ (nonterminal state)

$$(1,1) \xrightarrow[Up]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (2,3) \xrightarrow[Right]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{+1}} (4,3)$$

$$(1,1) \xrightarrow[Up]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (2,3) \xrightarrow[Right]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{-.04}} (3,2) \xrightarrow[Up]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{+1}} (4,3)$$

$$(1,1) \xrightarrow[Up]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (2,3) \xrightarrow[Right]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{-.04}} (3,2) \xrightarrow[Up]{\text{-1}} (4,2)$$



	1	2	3	4
3	0.8516	0.9078	0.9578	+1
2	0.8016		0.7003	−1
1	0.7453	0.6953	0.6514	0.4279

$$U^{\pi}(s) = E\left[\sum_{t=1}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right]$$

17.2 Passive Reinforcement Learning (3/6)

Direct utility estimation

- The utility of a state is defined as the expected total reward from that state onward (called the expected **reward-to-go**)
  - In the limit of infinitely many trials, the sample average will converge to the true expectation in equation below

$$U^{\pi}(s) = E\left[\sum_{t=1}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right]$$

  - We have reduced reinforcement learning to a standard supervised learning problem in which each example is a (*state*, *reward-to-go*)
  - But the utility of a state is determined by the reward and the expected utility of the successor states. Specifically, the utility values obey the Bellman equations for a fixed policy

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

17.2 Passive Reinforcement Learning (4/6)

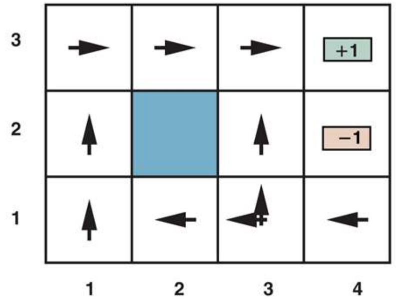
Adaptive dynamic programming (ADP)

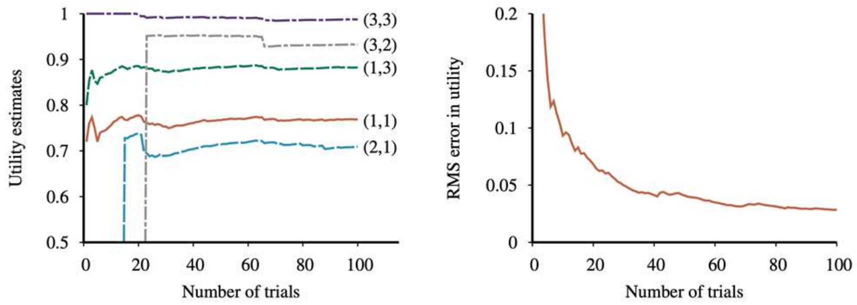
- ADP agent takes advantage of the constraints among the utilities of the states by learning the transition model that connects them and solving the corresponding Markov decision process (MDP) using dynamic programming.

```
function PASSIVE-ADP-LEARNER(percept) returns an action
  inputs: percept, a percept indicating the current state s′ and reward signal r
  persistent: π, a fixed policy
              mdp, an MDP with model P, rewards R, actions A, discount γ
              U, a table of utilities for states, initially empty
              N_{s′|s,a}, a table of outcome count vectors indexed by state and action, initially zero
              s, a, the previous state and action, initially null

  if s′ is new then U[s′] ← 0
  if s is not null then
    increment N_{s′|s,a}[s, a][s′]
    R[s, a, s′] ← r
    add a to A[s]
    P(· | s, a) ← NORMALIZE(N_{s′|s,a}[s, a])
    U ← POLICYEVALUATION(π, U, mdp)
    s, a ← s′, π[s′]
    return a
```

Figure 23.2 A passive reinforcement learning agent based on adaptive dynamic programming. The agent chooses a value for γ and then incrementally computes the P and R values of the MDP. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 567.





17.2 Passive Reinforcement Learning (5/6)

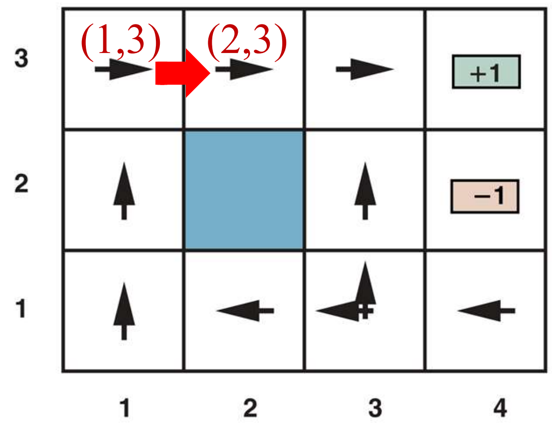
Temporal-difference learning (TD learning)

- Another way for solving MDP is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations
  - For example, if the transition from (1,3) to (2,3): $U^{\pi}(1,3) = -0.04 + U^{\pi}(2,3)$
  - More generally: $U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha[R(s,\pi(s),s') + \gamma U^{\pi}(s') - U^{\pi}(s)]$

function PASSIVE-TD-LEARNER(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: π, a fixed policy
    s, the previous state, initially null
    U, a table of utilities for states, initially empty
    N_s, a table of frequencies for states, initially zero

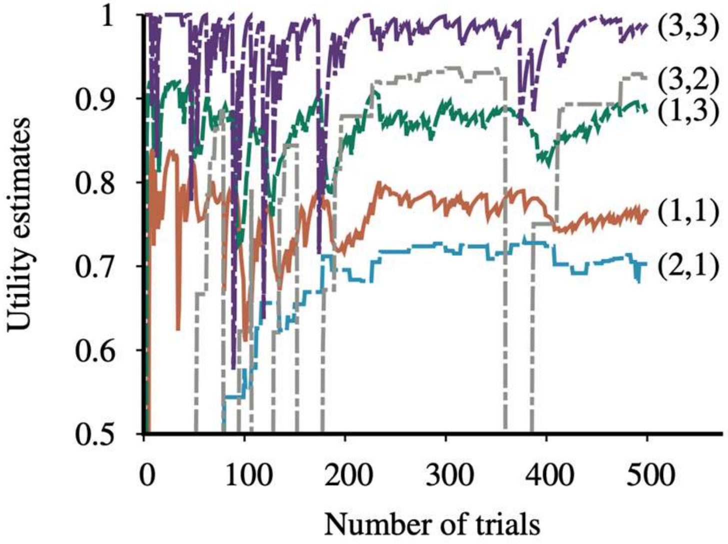
  if s' is new **then** $U[s'] \leftarrow 0$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s]) \times (r + \gamma\, U[s'] - U[s])$
  $s \leftarrow s'$
  return $\pi[s']$

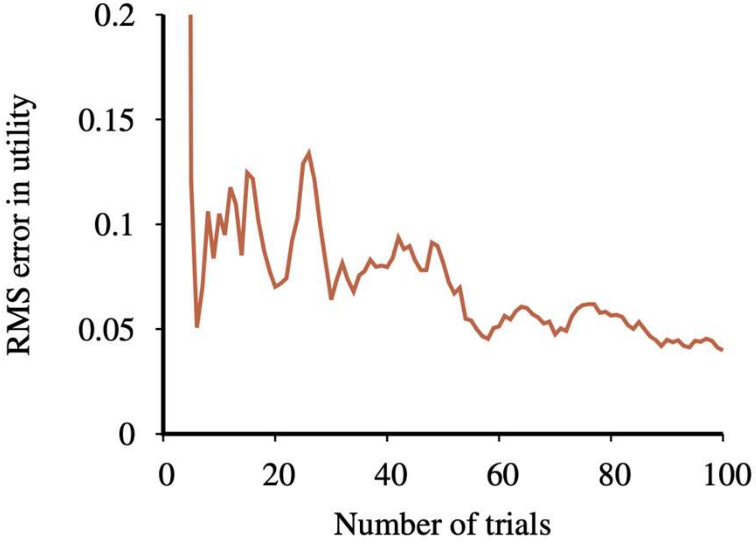
Figure 23.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence.



17.2 Passive Reinforcement Learning (6/6)

Temporal-difference learning (TD learning)





17.3 Active Reinforcement Learning

17.3 Active Reinforcement Learning (1/8)

Active learning agent

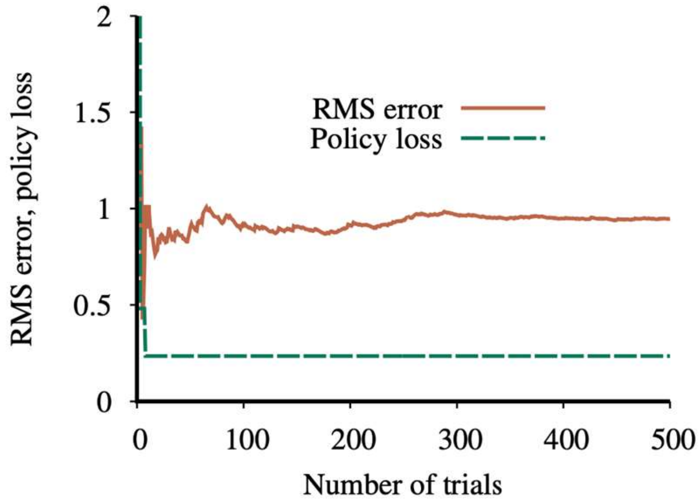
- A passive learning agent has a fixed policy that determines its behavior.
- An active learning agent gets to decide what actions to take.
- Consider ADP agent that how it can be modified to take advantage
  - The agent will need to learn a complete transition model with outcome probabilities for all actions.
  - We need to take into account the fact that the agent has a choice of actions.
  - The utilities it needs to learn are those defined by the optimal policy; they obey the Bellman equations (value iteration or policy iteration)

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

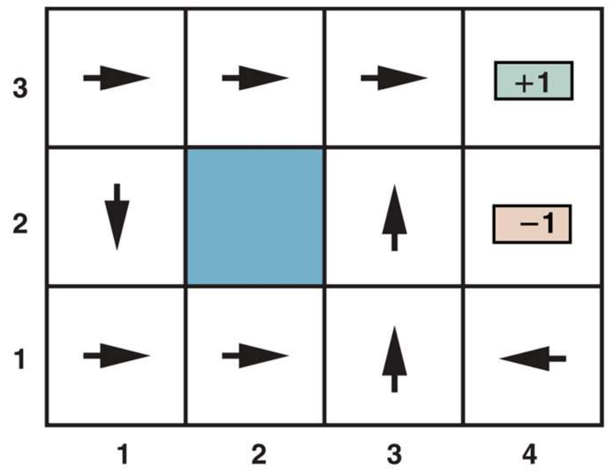
17.3 Active Reinforcement Learning (2/8)

Active learning agent

- The greedy agent does not learn the true utilities or the true optimal policy! Then, what should it do?



(a)



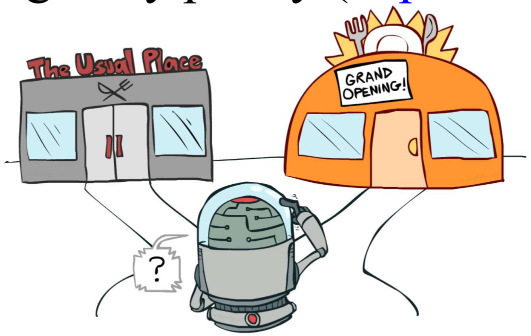
(b)

<출처> Stuart J. Russell and Peter Norvig (2021). Artificial Intelligence: A Modern Approach (4th Edition). Pearson

17.3 Active Reinforcement Learning (3/8)

Exploration

- The greedy agent overlooks the fact that actions do more than provide *reward.*
- An agent must make a tradeoff between exploitation and exploration.
- GLIE (Greedy in the Limit of Infinite Exploration)
  - Choose random action at time step t with probability $\frac{1}{t}$ (exploration)
  - Otherwise, follow the greedy policy (exploitation)



<출처> Avery Parkinson' blog: The epsilon-greedy algorithms for reinforcement learning

17.3 Active Reinforcement Learning (4/8)

Exploration

- Can we do better exploration?
- **A better approach:** give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility
  - $U^+(s)$: optimistic estimate of the utility (i.e. the expected reward-to-go)
  - $N(s,a)$: the number of times action a in state s
  - $f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise,} \end{cases}$

    R^+ is best possible reward (optimistic estimate)
    N_e is a fixed parameter
    Determine how greed is traded off against curiosity
  - The we can rewrite the update equation with exploration function f as below

$$U^+(s) \leftarrow \max_a f\left(\sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U^+(s')], N(s,a)\right)$$

17.3 Active Reinforcement Learning (5/8)

Exploration

- An agent with exploration shows a rapid convergence toward zero policy loss (unlikely with the greedy approach)

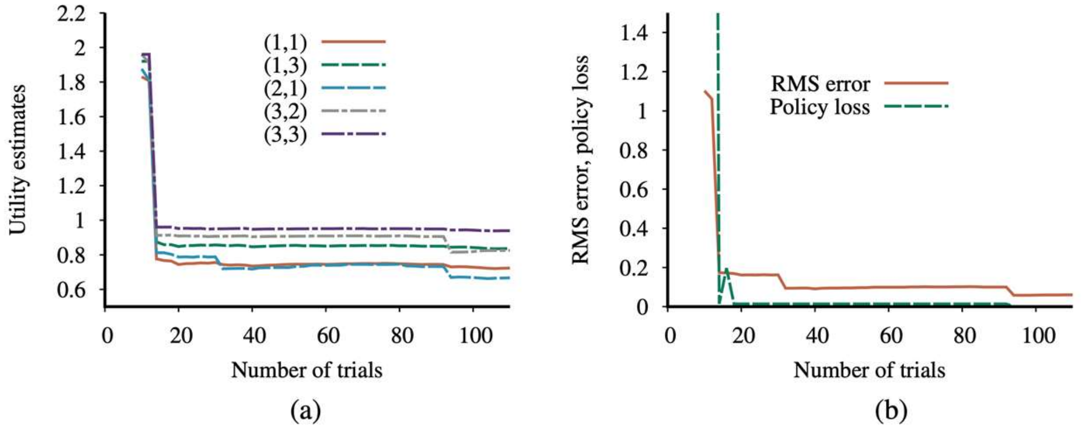


Figure 23.7 Performance of the exploratory ADP agent using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

17.3 Active Reinforcement Learning (6/8)

Safe exploration

- In a simulation environment, any accidents give us more information, and we can just hit the reset button.
- How about a real world setting? (e.g. robot learning)
  - In the worst case, the agent enters an absorbing state where no actions have any effect and no rewards are received.
- One of safe exploration approaches: **Bayesian reinforcement learning**
  - Assume a prior probability $P(h)$ over hypotheses h about the true model
  - The posterior $P(h|\mathbf{e})$ is obtained by Bayes' rule (i.e. $P(h|\mathbf{e}) = \frac{P(h)P(\mathbf{e}|h)}{P(\mathbf{e})}$)
  - The utility obtained by executing policy π in model h. Then we have,

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_{h} P(h|\mathbf{e}) U_h^{\pi}$$

17.3 Active Reinforcement Learning (7/8)

Temporal-difference Q-learning

- Two algorithms of TD learning for active ADP agent: SARSA, Q-learning
  - TD learning does not need a transition model $P(s'|s,a)$
- SARSA (for State, Action, Reward, State, Action)
  - SARSA updates with the Q-value of the action a' that is actually taken:
    $$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma Q(s',a') - Q(s,a)]$$
  - SARSA is an on-policy algorithm (behavior policy = target policy)
- Q-learning
  - Q-learning TD update is calculated whenever a is executed in s learning to s'
    $$Q(s,a) \leftarrow Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$
  - Q-learning is an off-policy algorithm (not always behavior policy = target policy)

17.3 Active Reinforcement Learning (8/8)

Temporal-difference Q-learning

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$ The exploration function $f(u,n)$ we have learned
  return a

Figure 23.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s,a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model.

<출처> Stuart J. Russell and Peter Norvig (2021). Artificial Intelligence: A Modern Approach (4th Edition). Pearson

17.4 Generalization in Reinforcement Learning

17.4 Generalization in Reinforcement Learning (1/6)

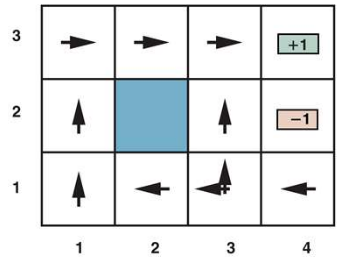
Function approximation

- Two dimensional grid environment: about 10^6 states space
- What about real-world environment?
  - Backgammon is simpler than most real-world applications, yet it has 10^{20} states.
  - We cannot easily visit them all in order to learn how to play the game.
- We need to use function approximation for approximating utility function
  - e.g. weighted linear combination of features

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

→ Instead of learning 10^6 states values (in table), an agent can learn 20 values for parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_{20})$

17.4 Generalization in Reinforcement Learning (2/6)



Approximating direct utility estimation

- Consider 4×3 world environment (with x and y coordinates)

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

- As with neural network learning, we write an error function and compute its gradient w.r.t. paramters ($u_j(s)$ is the observed total reward in jth trial)

$$E_j(s) = \left(\hat{U}_\theta(s) - u_j(s)\right)^2 / 2$$

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha\left[u_j(s) - \hat{U}_\theta(s)\right] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

  - This is called the Widrow–Hoff rule (or delta rule). We get three simple update rules

$$\theta_0 \leftarrow \theta_0 + \alpha\left[u_j(s) - \hat{U}_\theta(s)\right],$$
$$\theta_1 \leftarrow \theta_1 + \alpha\left[u_j(s) - \hat{U}_\theta(s)\right]x,$$
$$\theta_2 \leftarrow \theta_2 + \alpha\left[u_j(s) - \hat{U}_\theta(s)\right]y.$$

17.4 Generalization in Reinforcement Learning (3/6)

Approximating temporal-difference learning

- We can apply the idea in previous slide equally well to TD learners
  - For utility

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

  - For Q-value

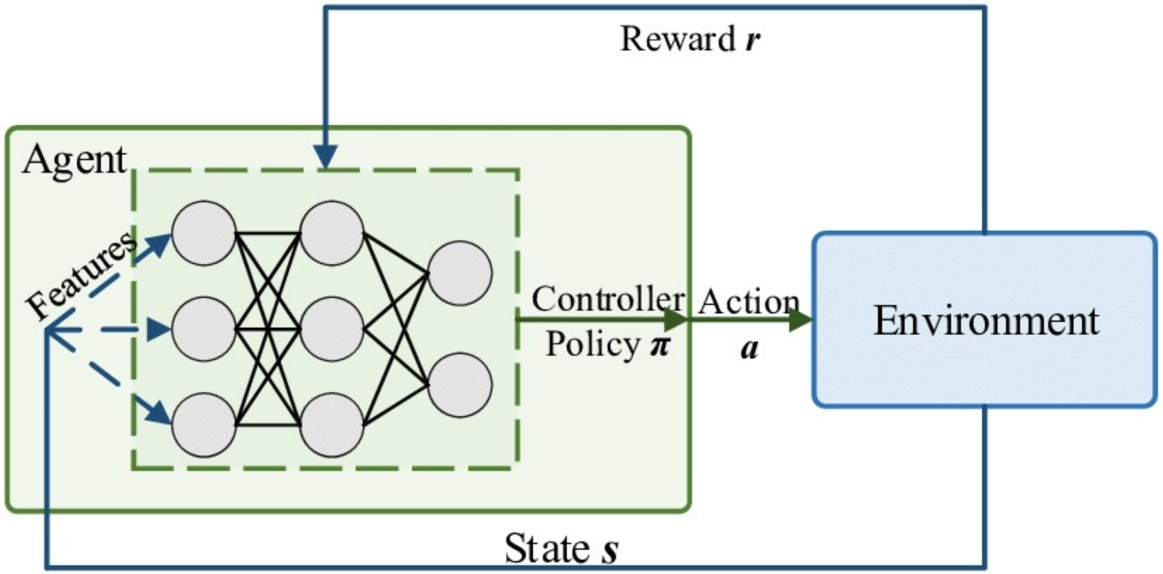
$$\theta_i \leftarrow \theta_i + \alpha\left[R(s,a,s') + \gamma \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)\right]\frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

- The problem of catastrophic forgetting
  - One of solutions is using experience replay.
  - The learning algorithm can retain trajectories from the entire learning process and replay those trajectories to ensure that its value function is still accurate for parts of the state space it no longer visits.

17.4 Generalization in Reinforcement Learning (4/6)

Deep reinforcement learning

- The linear approximator may be insufficient → Use deep neural networks!
  - e.g. Video games, AlphaGo, training robots etc.

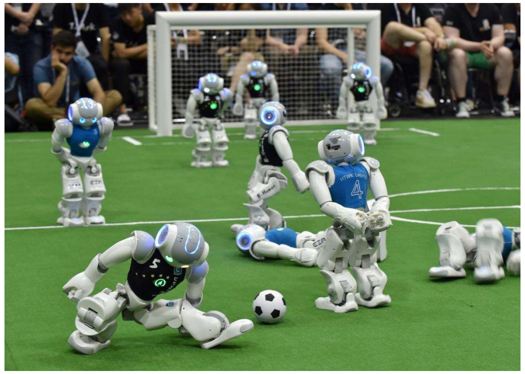


<출처> Gu et al., Machine Learning for Intelligent Optical Networks: A Comprehensive Survey

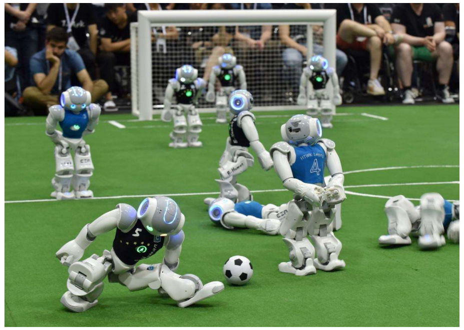
17.4 Generalization in Reinforcement Learning (5/6)

Reward shaping

- The credit assignment problem
  - Real-world environments may have very sparse rewards → many primitive actions are required to achieve any nonzero reward
  - For example, a soccer-playing robot might send a hundred thousand motor control commands to its various joints before conceding a goal. → Now it has to work out what it did wrong.
- One of good solutions is reward shaping
  - For any potential function Φ,
    $$R'(s,a,s') = R(s,a,s') + \gamma\Phi(s') - \Phi(s)$$
  - e.g. potential function Φ for soccer playing robot
    → A bonus for reducing distance of the ball from the opponents' goal



17.4 Generalization in Reinforcement Learning (6/6)



Hierarchical reinforcement learning

- Break long action sequences up into a few smaller pieces

  while not IS-TERMINAL(s) **do**
  if BALL-IN-MY-POSSESSION(s) **then choose**({PASS, HOLD, DRIBBLE})
  else choose({STAY, MOVE, INTERCEPT-BALL}).

- HRL agent is solving a Markovian decision problem with following elements:
  - The states are the choice states σ of the joint state space
  - The actions at σ are the choices c available in σ according to the partial program
  - The reward function $\rho(\sigma, c, \sigma')$ is the expected sum of rewards
  - The transition model $\tau(\sigma, c, \sigma')$ is defined in the obvious way:
    - if c invokes a physical action a, then τ borrows from the physical model $P(s'|s, a)$;
    - if c invokes a computational transition, such as calling a subroutine, then the transition deterministically modifies the computational state m according to the rules of the programming language

17.5 Policy Search

17.5 Policy Search (1/2)

Policy representation

- Policy representation in terms of Q-functions

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, \hat{Q}_\theta(s,a)$$

  - Problem: policy change discontinuously, which makes gradient-based search difficult

- (Stochastic) Policy representation with softmax

$$\pi_\theta(s,a) = \frac{e^{\beta \hat{Q}_\theta(s,a)}}{\sum_{a'} e^{\beta \hat{Q}_\theta(s,a)}}$$

  - Use a stochastic policy representation, which specifies the *probability* of actions
  - Parameter $\beta > 0$ modulates softness of the softmax (high: hard max, low: uniform)

17.5 Policy Search (2/2)

Policy gradient methods

- $\rho(\theta)$ is policy value and $\nabla_\theta \rho(\theta)$ is policy gradient

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a R(s_0, a, s_0)\pi_\theta(s_0, a) = \sum_a R(s_0, a, s_0)\nabla_\theta \pi_\theta(s_0, a)$$

- Monte Carlo approximation: approximate by samples generated from $\pi_\theta(s_0, a)$

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{R(s_0, a, s_0)\nabla_\theta \pi_\theta(s_0, a)}{\pi_\theta(s_0, a)}$$

$$\approx \frac{1}{N} \sum_{j=1}^{N} \frac{R(s_0, aj, s_0)\nabla_\theta \pi_\theta(s_0, aj)}{\pi_\theta(s_0, a_j)}$$

- For the sequential case

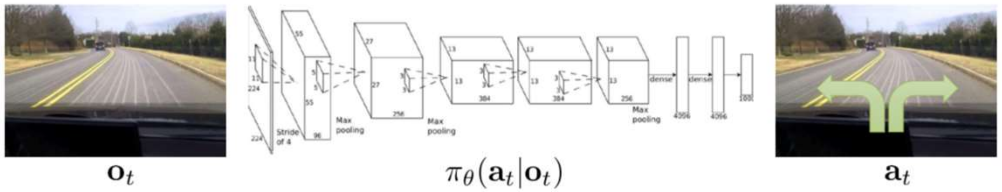
$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{u_j(s)\nabla_\theta \pi_\theta(s_0, a_j)}{\pi_\theta(s_0, a_j)}$$


17.6 Apprenticeship and Inverse
Reinforcement Learning
VERI LUX
TAS MEA

17.6 Apprenticeship and Inverse RL (1/2)

Apprenticeship learning

- How to behave well given observations of expert behavior
- We can apply supervised learning using state-action pairs to learn policy
  → Imitation learning



<출처> Bojarski et al. '16, NVIDIA

17.6 Apprenticeship and Inverse RL (2/2)

Inverse reinforcement learning

- Learn rewards by observing a policy, rather than learning a policy by observing rewards
- How to find the expert's reward function, given the expert's actions
- Feature matching method
  - Assume the reward function as a weighted linear combination of features

$$R_\theta(s,a,s') = \sum_{i=1}^{n} \theta_i f_i(s,a,s') = \theta \cdot \mathbf{f}$$

  - Recall the utility function of executing a policy π. We can derive feature expectation $\mu_i(\pi)$ (expected discounted value of the feature f_i)

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right] = E\left[\sum_{t=0}^{\infty} \gamma^t \sum_{i=1}^{n} \theta_i f_i(S_t, \pi(S_t), S_{t+1})\right]$$

$$= \sum_{i=1}^{n} \theta_i\, E\left[\sum_{t=1}^{\infty} \gamma^t f_i(S_t, \pi(S_t), S_{t+1})\right] = \sum_{i=1}^{n} \theta_i \mu_i(\pi) = \theta \cdot \mu(\pi)$$

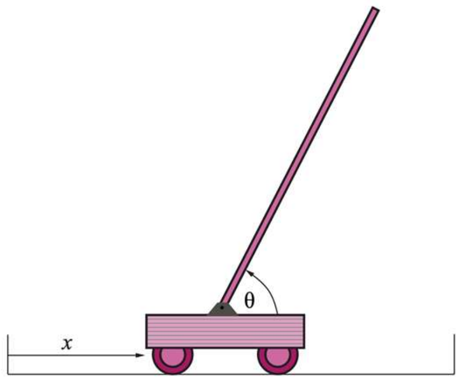
17.7 Applications of Reinforcement Learning

17.7 Applications of Reinforcement Learning (1/1)

Application of Reinforcement Learning

- Game playing
  - Checker program written by Arthur Samuel (1959, 1967)
  - TD-GAMMON (Backgammon) program by Gerry Tesauro (1992)
  - AlphaGo program by Google DeepMind (2016)

- Robot control
  - Cart-pole balancing problem by Michie and Chambers (1968)
  - Helicopter flight using policy search by Bagnell et al. (2001)
  - Deep RL for robotics and self-driving cars (2016~present)



Cart-pole balancing

Helicopter flight

<출처> Stuart J. Russell and Peter Norvig (2021). Artificial Intelligence: A Modern Approach (4th Edition). Pearson

Summary

1. Direct utility estimation uses the total observed reward-to-go for a given state as direct evidence for learning its utility.
2. A model-based reinforcement learning agent acquires a transition model for the environment and learns a utility function.
3. A model-free reinforcement learning agent may learn an action-utility function or a policy
4. Adaptive dynamic programming (ADP) learns a model and a reward function from observations and then uses value or policy iteration to obtain the utilities or an optimal policy.
5. Temporal difference (TD) methods update utility estimates to match those of successor states.
6. Action-utility functions (Q-functions) can be learned by an ADP approach or a TD approach.
7. Policy-search methods operate directly on a representation of the policy, attempting to improve it based on observed performance.
8. Apprenticeship learning through observation of expert behavior can be an effective solution when a correct reward function is hard to specify.
9. Imitation learning formulates the problem as supervised learning of a policy from the expert's state-action pairs.
10. Inverse reinforcement learning infers reward information from the expert's behavior.