Machine Learning

Lecture 12: Introduction to Reinforcement Learning

Nevin L. Zhang
lzhang@cse.ust.hk

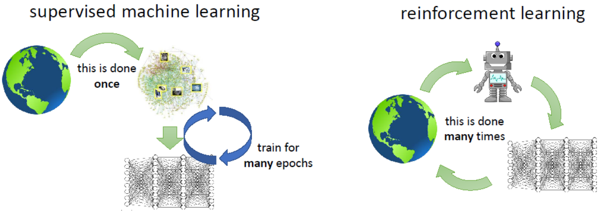
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

This set of notes is based on internet resources and
Richard Sutton and Andrew Barto (1998). *Reinforcement Learning*. MIT Press.

Outline

Introduction to RL

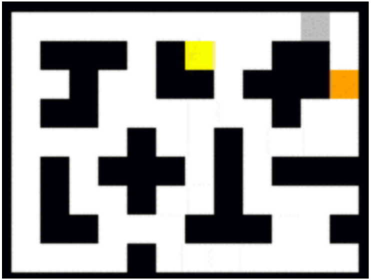
- **Supervised learning**: Learn how to predict from labelled data $\{\mathbf{x}_i, y_i\}_{i=1}^N$.
- **Unsupervised learning**: Understand unlabelled data $\{\mathbf{x}_i\}_{i=1}^N$, learn how to generate and manipulate data.
  In both cases, the data are collected beforehand.



- **Reinforcement Learning**: Learn how to act from experiences $\{(s, a, r, s')\}$ with the environment, which are collected by the learning agent itself.
  So, RL is a kind of **active learning**.

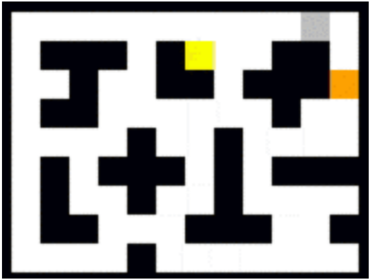
The cat-mouse-cheese example

github.com/vmayoral/basic_reinforcement_learning/blob/master/tutorial1/README.md



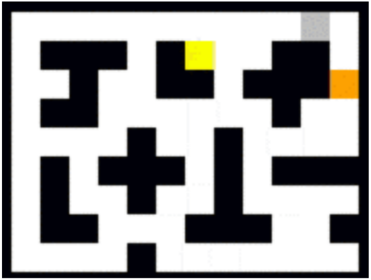
- A cat (orange), a mouse (gray), a piece of cheese (yellow) in a maze environment.
- Actions for the mouse and the cat: Move up, down, left and right.
- The mouse gets a **reward** at each time step:
  - -100: if eaten by the cat
  - 50: if eats the cheese
  - -1: otherwise

The cat-mouse-cheese example



- The cat is preprogrammed to catch the the mouse.
- Focus of this lecture: Make the mouse smart through learning,
    - Learn to do the right thing at each time step.

Learning a policy



- At each step, the mouse knows the current situation s.
  - The maze, locations of cat and cheese.
  - s is called a **state**.
  - **State space** S: set of all possible states.
  - The mouse needs to pick an **action** a from a **set** A **of possible actions**.
- Needs to learn **policy** π: $S \rightarrow A$
  - $\pi(s)$ is the action to take in situation s.

Learning a policy

- The mouse needs to learn a **policy** π: $S \rightarrow A$.
- From?
  - Experiences with the games.
    - If wins, do the same again.
    - If loses, avoid making the same mistake.
- This is a new type of problem:
  - Training data not in the form $\{\mathbf{x}_i, y_i\}_{i=1}^N$.
  - It is not an unsupervised learning problem.
- It is a **reinforcement learning** problem: The mouse needs to learn from interactions with the environment to improve its behavior over time.

Reinforcement Learning and Markov Decision Process

- The cat is preprogrammed.
- If the mouse knows how that program works, then it could figure out the best policy. No need to learning from experiences.
- In general, if an agent has a **MDP (Markov Decision Process)** model of its environment, it can figure out the optimal policy.
- Next:
  - What are MDPs? How to derive optimal policies from MDPs?
- Later:
  - What to do if we don't have an MDP model of environment?
  - Answer: reinforcement learning.
  - Discussions on MDPs will give us the framework for discussing reinforcement learning.

Outline

Markov Decision Process

- **Markov Decision Process (MDP)** is a model about how an agent interacts with its environment.
- At each step,
  - Environment is in some **state** s.
  - Depending on what s is, the agent takes an **action** a:
    - Environments moves to another state s'.
    - Agents gets an **immediate reward/penality** r.
  - Repeat

Markov Decision Process

A **MDP** consists of:

- A finite **state space**: S
- A **space of actions**: A
- A **transition probability**: $P(s'|s, a)$
  - Suppose environment is current in s and agent takes action a.
  - The probability of the state at the next time being s' is $P(s'|s, a)$.
- Immediate **reward function**: $r(s, a, s')$

Given an MDP, we want to find a **policy** that specifies an action for each possible state

$$\begin{aligned} \pi : \quad & S \mapsto A \\ & s \rightarrow \pi(s) \end{aligned}$$

The Reward Function

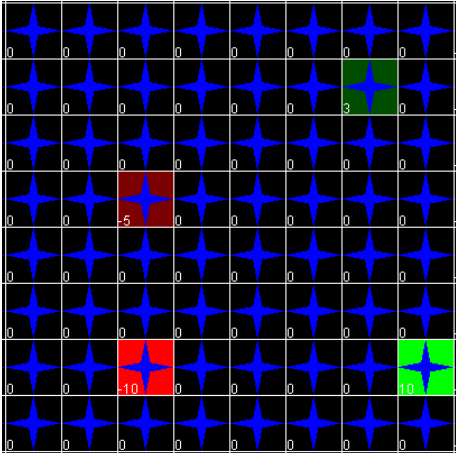
A **MDP** consists of:

- The immediate reward function: $r(s, a, s')$ is influenced by only the current action, not by future actions.
- After taking action a, we can calculate the expected reward:

$$r(s, a) = \sum_{s'} r(s, a, s') P(s'|s, a)$$

- In the following, we will sometimes regard $r(s, a)$ as the reward we obtain right away after taking action a in state s.

MDP example



- S: 8×8 grid
- A: 4 actions: up, down, left and right.
- Transition probability:
  - 0.7 chance move in intended direction, 0.1 chance in each of the other 3 directions.
  - Does not move when bumping against the wall.
- Reward: 4 reward states (reward obtained when leaving those states); -1 for bumping into walls.

MDP: The process

$$\pi : S \longrightarrow A$$

* process

agent: $\pi(s_0) = a_0$ world: $p(s' \mid s_0, a_0)$ agent: $\pi(s_1) = a_1$ world: $p(s' \mid s_1, a_1)$

$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_2]{a_1} s_2 \longrightarrow \cdots$$

Discounted total rewards

Discount factor $0 < \gamma < 1$

$$R^{\pi}(s_0) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots$$

0.95

value function of π

$$V^{\pi}(s) = E[R^{\pi}(s)]$$

MDP: The Process

- Initial state of environment: s_0.
- For $t = 0$ to ∞
  - Environment in state s_t
  - Agent takes action $a_t = \pi(s_t)$
  - Receives reward r_t.
  - Environment change to another state s_{t+1} according to transition probability $P(s_{t+1}|s_t, a_t)$.
- **Trajectory (rollout)**: $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$
- Discounted total reward **depends on** π:

$$R^\pi(s_0) = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$$

Value Function of Policy π

- $R^{\pi}(s)$ might be different in different runs. There are randomness in the system.
- Define

$$V^{\pi}(s) = E[R^{\pi}(s)]$$

  - total reward expected to get if follow policy π starting from state s
  - Called value function of policy π.

Optimal Policy

- Different policies have different value functions.
- There exist a policy, π^*, such that, for any other policy π:

$$V^{\pi^*}(s) \geq V^{\pi}(s) \quad \forall s$$

- It is called the optimal policy.
- Its value function is called the optimal state value function, denoted by $V^*(s)$:

$$V^*(s) = V^{\pi^*}(s)$$

Planning vs Reinforcement Learning vs Unsupervised Learning

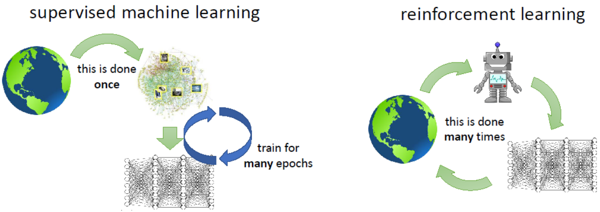
- **Planning**:

$$P(s'|s,a), r(s,a) \Rightarrow \pi^*$$

- **Reinforcement Learning**:

$$\{(s,a,r,s')\} \Rightarrow \pi^*$$

The **experience tuples** (s,a,r,s') are collected by the learning agent itself. So, RL is a kind of **active learning**.



Value Iteration

- **Planning**:

$$P(s'|s,a), r(s,a) \Rightarrow \pi^*$$

- Do it in two steps:
  - **Value iteration**: $P(s'|s,a), r(s,a) \Rightarrow v^*$
  - $V^* \Rightarrow \pi^*$

Value Iteration

- **Value Iteration (VI)**:
  - Pick $V_0(s)$, $k = 0$
  - Repeat:
    - $V_{k+1}(s) = \max_a \{r(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s')\}$
    - $k = k + 1$
  - until $\max_s |V_{k+1}(s) - V_k(s)| \leq \epsilon$
- AKA: Dynamic programming for MDPs.
- The mapping from V_k to V_{k+1} is called the **Bellman Operator**.

Value Iteration

- The sequence $\{V_0, V_1, V_2, \ldots, \}$ converges to V^*, regardless of the choice of V_0,
  - Because of **contraction property** of value iteration (or Bellman Operator):

$$\max_s |V_{k+1}(s) - V_k(s)| \leq \gamma \max_s |V_k(s) - V_{k-1}(s)|$$

Bellman's Optimality Equations

- In particular, it VI starts with V^*, it converges in one step:

$$V^*(s) = \max_a \{r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')\}$$

  This is called Bellman's optimality equation.

- Optimal **state-action value function**:

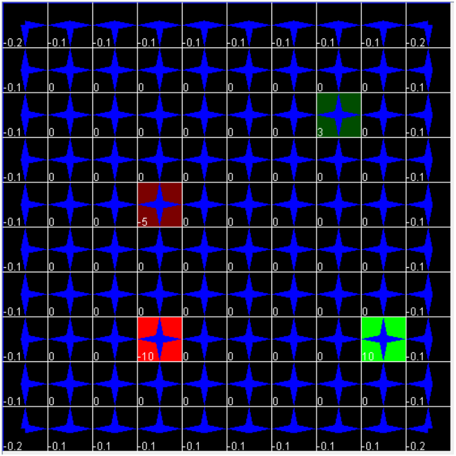
$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')$$

  Total reward for, starting from s, taking action a and acting optimally after that.

- The optimal policy can be obtained from Q^*:

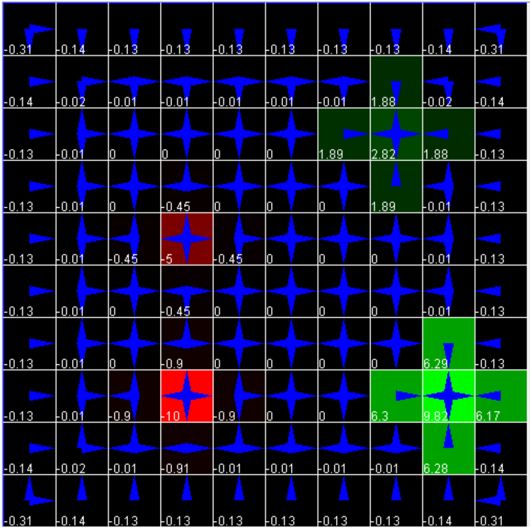
$$\pi^*(s) = \arg\max_a Q^*(s,a).$$

Value Iteration Example



- S: 10×10 grid
- A: 4 actions: up, down, left and right.
- Transition probability:
  - 0.7 chance move in intended direction, 0.1 chance in each of the other 3 directions.
  - Does not move when bumping against the wall.
- Reward: 4 reward states (reward obtained when leaving); -1 for bumping into walls.
- **Figure shows V_1 ($V_0 = 0$)**

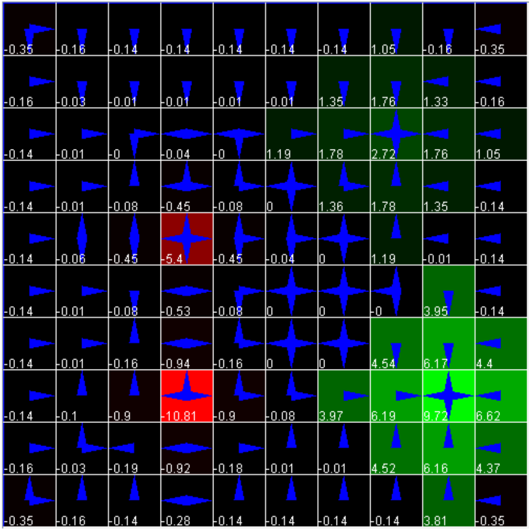
Value Iteration Example



V_2:

- States near positive-reward states
  - Values changed drastically.
  - Actions decided: Go there!
- States near negative-reward states
  - Values changed slight.
  - Actions decided: avoid going there!
- Avoid walls.

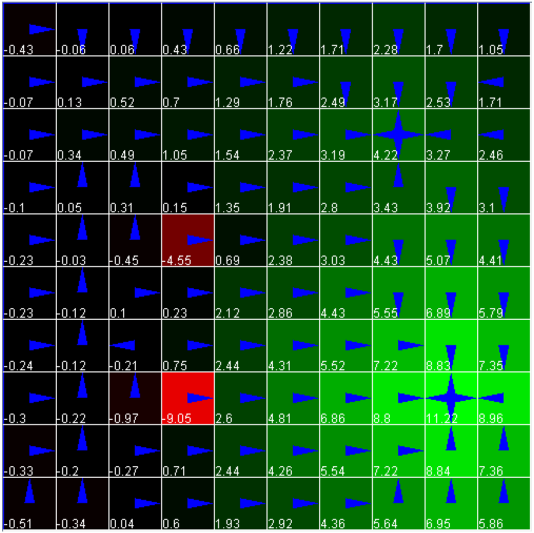
Value Iteration Example



V_3:

- For more states near positive-reward states
  - Values changed drastically.
  - Actions decided: Go there!
- For more states near negative-reward states
  - Values changed slightly.
  - Actions decided: avoid going there!

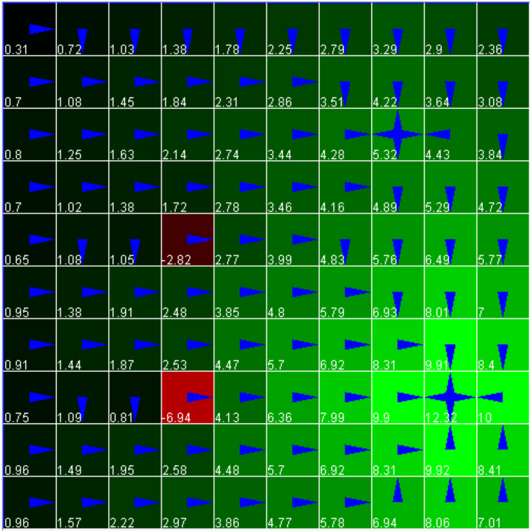
Value Iteration Example



V_{10}:

- A clear policy emerged
  - Move toward the state with reward 10 if it is not too far away compared with the state with reward 3.

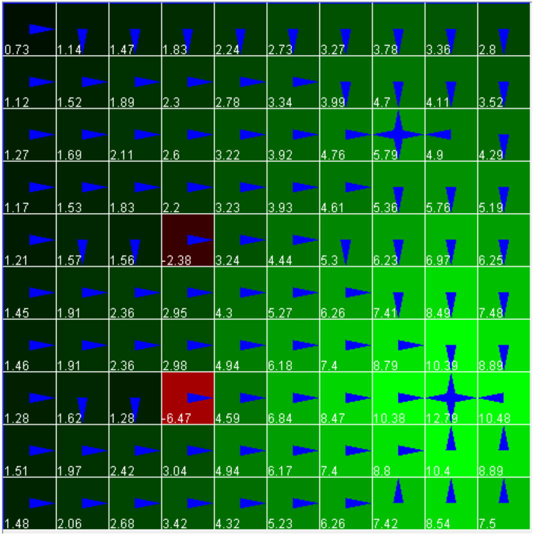
Value Iteration Example



V_{20}:

- Values changed quite a lot from V_{10}
- Policy did not change much:
    - Only the action for (5, 7) is changed.

Value Iteration Example



- V_{30}:
- Values changed some more from V_{20}
- Policy did not change at all.
    - Optimal policy found.
- Note that how the policy avoids the negative-reward states.

Value Iteration

- Can also carry out **Value Iteration (VI)** in terms of Q function directly:
  - Pick $Q_0(s,a)$, $k=0$
  - Repeat:
    - $Q_{k+1}(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q_k(s',a')$
    - $k = k+1$
  - until $\max_s |max_a Q_{k+1}(s,a) - \max_a Q_k(s,a)| \leq \epsilon$
- The mapping from Q_k to Q_{k+1} is called the **Bellman Operator**.

Value Iteration

- The sequence $\{Q_0, Q_1, Q_2, \ldots, \}$ converges to Q^*, regardless of the choice of Q_0.
- The optimal **action-value function** Q^* also satisfies Bellman's optimality equation.

$$Q^*(s,a) = \ r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')\}$$

- The **greedy policy** π_k based on Q_k is given by:

$$\pi_k(s) = \arg\max_a Q_k(s,a)$$

- π_k will approach and stabilize at π^* in a finite number of steps.

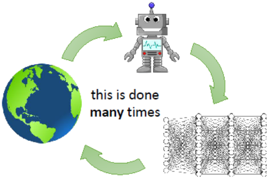
Outline

Reinforcement Learning (RL)

- RL comes into play when we
  - Don't have a model about the environment, i.e., no $P(s'|s, a)$, $r(s, a)$
  - But can interact with the environment
- RL is about learning what to do through interactions with environment.
  - Experience with environment: Trajectories/rollouts

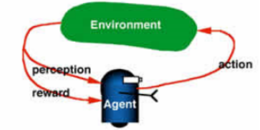
$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots$$

  - Learn for the experience so that later actions become better and better



Q-Learning

- **Q-Learning** problem statement: $\{(s, a, r, s')\} \Rightarrow Q^*(s, a)$
  Recall: $\pi^*(s) = \arg\max_s Q^*(s, a)$.



- **Algorithm**:
  - Represent $Q(s, a)$ as a table (later as neural network)
  - **Initialize** $Q(s, a)$
  - **Repeat**
    - Collect experience tuple (s, a, r, s')
    - Update Q for the observed pair (s, a) using the tuple

HOW?

Q-Learning

- Recall value iteration:
  $Q_{k+1}(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q_k(s',a')$
- If we have samples $s'_1, \ldots, s'_m \sim P(s'|s,a)$, then

$$Q_{k+1}(s,a) \approx r(s,a) + \gamma \frac{1}{m} \sum_{j=1}^{m} \max_{a'} Q_k(s'_j, a')$$

- If we have only one sample $s' \sim P(s'|s,a)$, then

$$Q_{k+1}(s,a) \approx r(s,a) + \gamma \max_{a'} Q_k(s',a')$$

  RHS is called the **temporal difference (TD) target**. It is an unbiased estimation of Bellman update, which is known to be an improvement of the current estimation $Q_k(s,a)$. However, the variance is high.
- So, we use it to update Q slightly:

$$\begin{aligned} Q_{k+1}(s,a) &\leftarrow (1-\alpha)Q_k(s,a) + \alpha(r(s,a) + \gamma \max_{a'} Q_k(s',a')) \\ &= Q_k(s,a) + \alpha[(r(s,a) + \gamma \max_{a'} Q_k(s',a')) - Q_k(s,a)] \end{aligned}$$

Q-Learning

- **Initialize** $Q(s, a)$
- **Repeat**
  - Collect experience tuple (s, a, r, s')

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

Q-learning converges to $Q^*(s, a)$ **if each** (s, a) **pair is updated infinitely often.**

The exploration vs exploitation tradeoff

- The agent needs to collect data for learning by exploring the environment.
- Exploration
  - Explore new parts of state space so as to gain more experiences.
  - Reward might not maximized.
- Exploitation
  - Make use of experience gained so far to maximize reward.
  - Might not gain new experiences
- ϵ-greedy policy:
  - With small probability ϵ, chose an action at random.
  - With probability $1 - \epsilon$, chose the action with the highest reward according to current estimates.

Terminal States

- Terminal/absorbing states: Cannot leave once entered.
- Example: ‘Game Over’.
- To continue training, need to restart the game.
- Episode: The process from initial state to terminal state.

The Q-Learning Algorithm

- Initialize $Q(s, a)$ arbitrarily.
- Repeat (for each episode)
  - Pick initial state s.
  - **Repeat**
    - Choose a for the state s (ϵ-greedy with $\arg\max_a Q(s, a)$)
    - Take action a, observe r and s'
    - Update:

$$\begin{aligned} Q(s,a) &\leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s,a)] \\ s &\leftarrow s' \end{aligned}$$

  - **until** s is terminal

Try it out:

`github.com/vmayoral/basic_reinforcement_learning/blob/master/tutorial1/README.md`

On-policy v.s. Off-policy

- An **on-policy** agent learns the value based on its current action a derived from the current policy, whereas its **off-policy** counter-part learns it based on the action a^* obtained from another policy.
- Q-learning is off-policy. It updates its Q-values using the Q-value of the next state s' and the greedy action a'.
- In other words, it estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed despite the fact that it's not following a greedy policy.

Sarsa

Another algorithm for temporal difference learning:

- Initialize $Q(s, a)$ arbitrarily.
- Repeat (for each episode)
    - Pick initial state s.
    - Choose a for the state s (ϵ-greedy with $\arg\max_a Q(s, a)$)
    - **Repeat**
        - Take action a, observe r and s'
        - Choose a' for s' (ϵ-greedy with $\arg\max_a Q(s', a)$)
        - Update:

$$\begin{aligned} Q(s, a) &\leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \\ s &\leftarrow s', a \leftarrow a' \end{aligned}$$

    - **until** s is terminal

Sarsa is On-Policy

- SARSA is on-policy.
- It updates its Q-values using the Q-value of the next state s' and the current policy's action a'.
- It estimates the return for state-action pairs assuming the current policy continues to be followed.