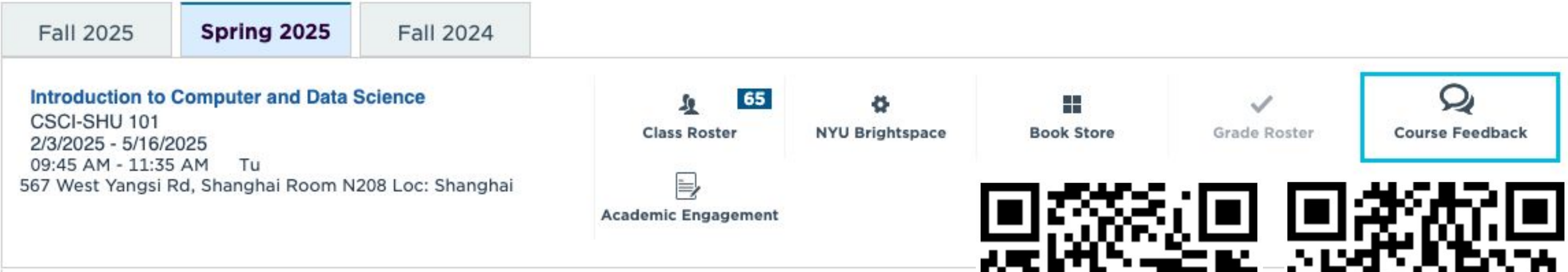
ICDS Spring 2025

Advanced Topic in DS

An Introduction to Reinforcement Learning

Course Evaluation Open!

- **May 5 to 16 on Albert:**



- Gradescope: Upload proof/screenshot to get extra **0.5% bonus** in final grade!
  - Lecture for me
  - Recitation for your TA!

Intro to CS and DS - Spring 2025, Lecture (SP25:CSCI-SHU:101:SH:001)

Intro to CS and DS - Spring 2025, Lecture (SP25:CSCI-SHU:101:SH:002)

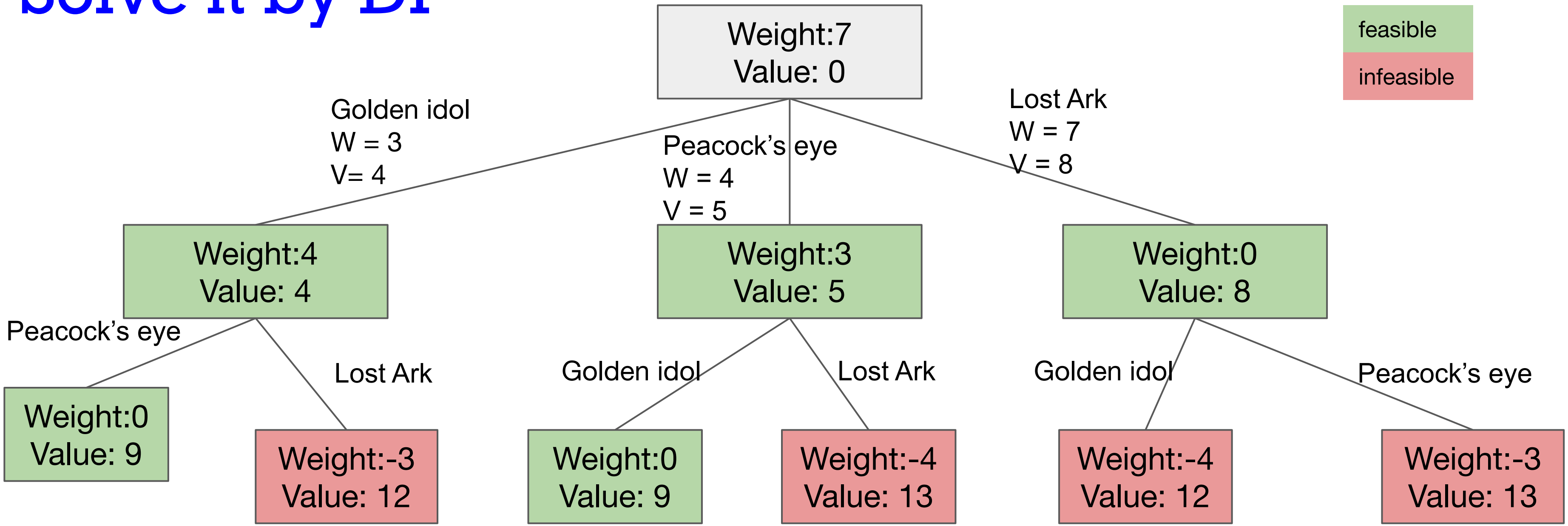
Helping Indiana Jones

Once upon a time, Indiana Jones came to a lost temple. Fortunately, he found 6 different treasures whose weights and values are listed in the following table.

Treasure	Golden Idol	Peacock's eye	Lost Ark
Weight(kg)	3	4	7
Value	4	5	8

Unfortunately, he only had one knapsack that could take 7 kg weights maximum (we don't consider the size of the bag). Now, please pick out some items so that maximize the total values of items that Indiana Jones takes.

Solve it by DP



- Dynamic Programming = Visiting nodes + memoization
- In DP, we obtain the memoization by visiting every not-yet-visited feasible node (It is like a lazy learner, not summarizing the experience, but reciting everything.)

Reinforcement Learning: a smart learner

RL: an algorithm that can learn a more informative memoization than DP

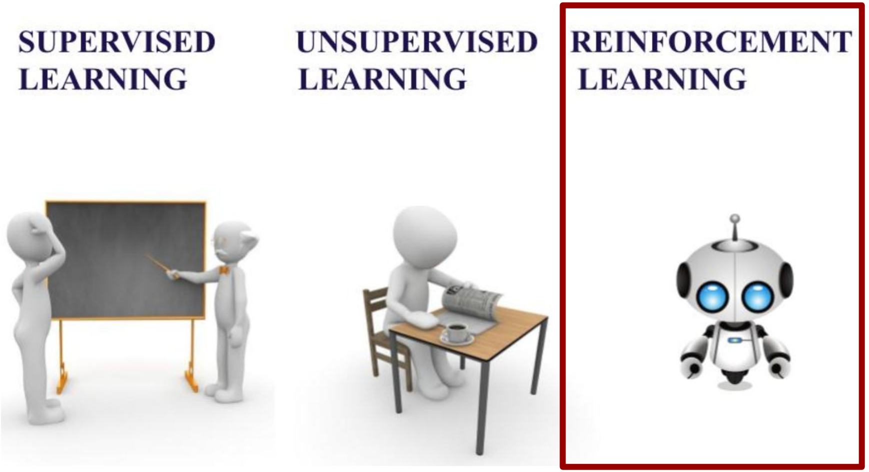
- It gains experience by running trails.
- It summaries the experience, not simply stores what have happened.

It simulates the learning process of intelligent animals; the algorithm contains the following components,

- Agent model
- Epsilon-Greedy (a model for decision making process)
- Smart memoization (using the reward)

Types of machine learning methods

Depending on how much guidance is needed when they "learn":



- Supervised learning (need clear guidance): works with labelled data sets; it needs the labels to tell it whether its prediction is correct or not.
- Unsupervised learning (no guidance needed): no labels are needed; it is self-organized, using a predefined way to process the data.
- **Reinforcement learning (vague guidance** is ok; like the intelligent creatures) : it "labels" the data items during learning; "learn by doing".

Agenda

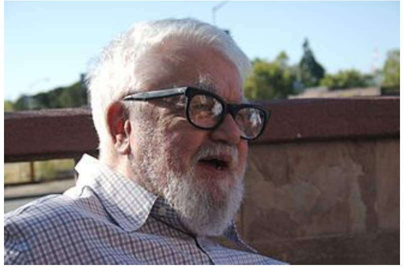
- The agent model
- Exploration v.s. Exploitation
  - Epsilon greedy
- Reinforcement learning
  - Modelling the rewards: Q-learning
  - Reinforcement learning vs dynamic programming
- Appendix: Deep reinforcement learning

What is an agent in computer science?

In computer science, the AI is about the study of "Intelligent agent".

- Intelligent agent: Any device that perceives its environment and takes actions that **maximize** its chance of successfully achieving its goals. (Wikipedia)

"Artificial Intelligence" → coined by John McCarthy at a workshop at Dartmouth College in 1956.

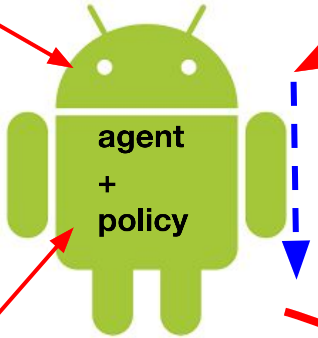


John McCarthy (1927-2011), Turing Award winner (1971)

The agent-environment framework

Agent: It is an autonomous entity that acts upon an environment using sensors and actuators for achieving goals.

Sensors:
perceive the states of the environment



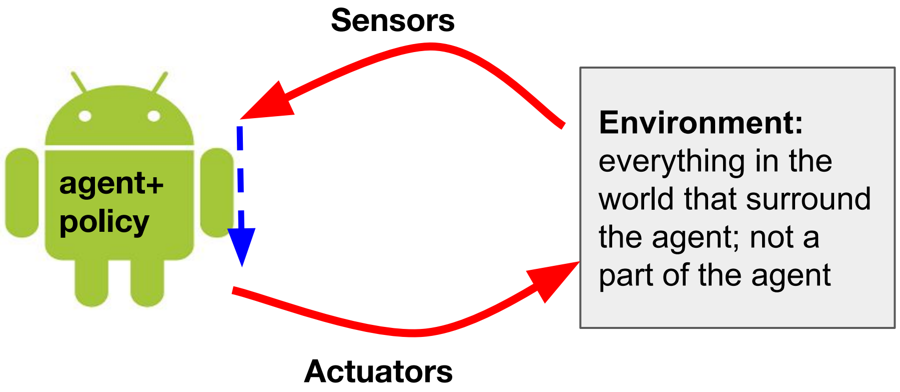
Environment:
everything in the world that surround the agent; not a part of the agent

- State: the situations of the environment; it is changing.

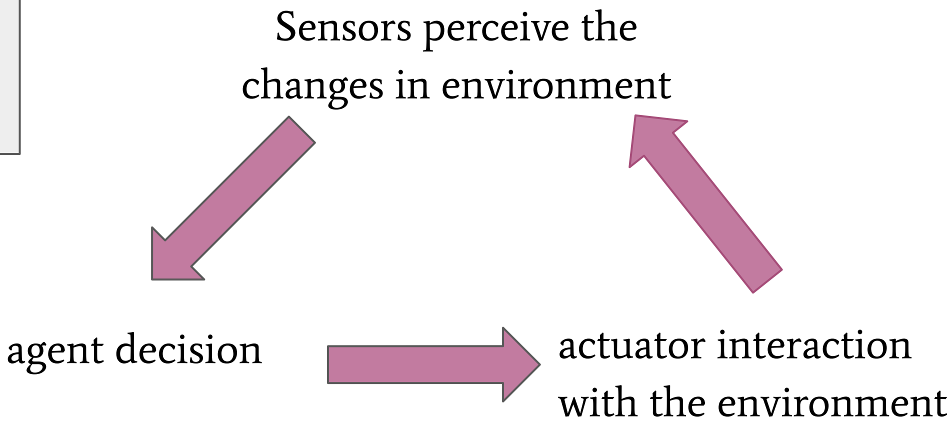
Control policy: the function that maps sensors to actuators; it tells how the agent make decisions;

Actuators:
affect the states

Perception action cycle

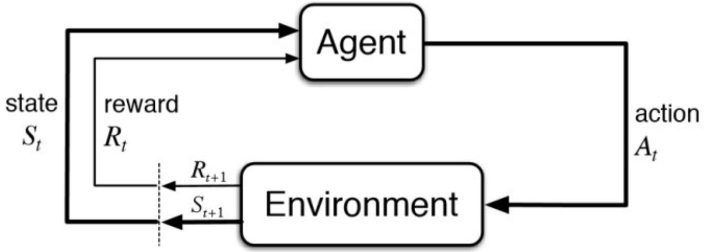


- The agent carries out actuations based on the sensor data it gets.
- It is a loop, called **perception action cycle**: (this is how we train the agent in RL)

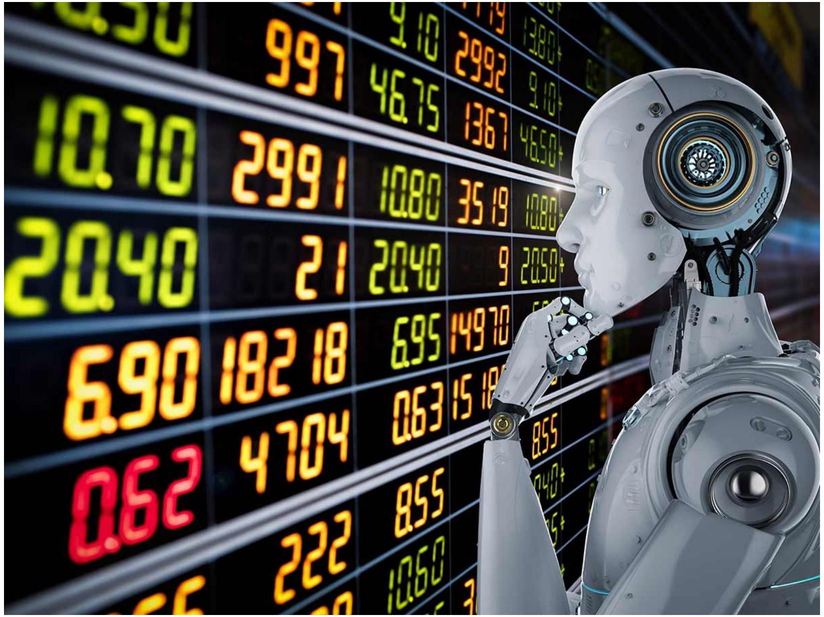


Agent model

- **Objective:** Maximizing the chance of success ⇒ maximizing the total reward
- **Components:**
  - **Agent**: an autonomous entity
  - **Environment**: the world through which the agent moves
  - **State**: a situation at a specific moment in the environment
  - **Action**: often being represented as a list of discrete possible moves
  - **Reward**: the feedback by which we measure the success or failure of an action in a given state
  - **Policy**: the strategy that the agent employs to determine the next action based on the current state



Example: Stock trading agents



The agent observes the price changes and decides the time to buy, sell, or keep a stock so that the money it earns can be maximized

- Environment: the stock market; the changes of prices
- State: a specific time during the trading period, e.g., 10 am;
- Actions: [buy, sell, keep]
- Reward: the money the agent earned

Example: Game agents in car racing game

The game agents play against human players to maximize the fun of the game.

- Environment: the human player
- States: specific frames in the race
- Actions: [left, right, stop, accelerate]
- Reward: scores the agent get

agent

Your move
(state)

Its move
(action)

You are the environment

you

Question:

- How do we adjust the game level (e.g., easy, normal, hard)?

OOP Implementation of agent model

Two types of objects in this framework: Agent and Environment

Agent class

- **Policy:** finding the best action based on the state and reward
- **Action:** methods for taking action

Environment class

- **State:** methods that provide the state
- **Reward:** methods that provide rewards

- **Policy** = the **total reward** of every available action + a decision making algorithm

Reward Table			
action/state	State 1	State 2	State 3
Take action 1	10	20	0
Take action 2	20	5	10

Modeling the decision processing

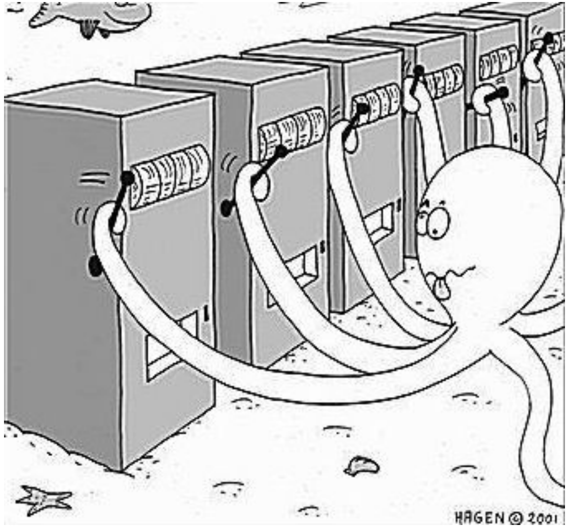
For intelligent creatures, decision is a <u>tradeoff</u> between exploration and exploitation.

- e.g., choose a restaurant for today's dinner
  - Exploitation: Go to your favorite restaurant
  - Exploration: Try a new/nameless restaurant

- The tradeoff can be described by **probability**

K-armed bandit problem

There is a slot machine with 4 arms; each arm has its own probability of success.



Arm 1	Arm 2	Arm 3	Arm 4
??%	??%	??%	??%
current success rate	current success rate	current success rate	current success rate

- The exact success rate of each arm is unknown. But they can be estimated by trials.

K-armed bandit problem

Now, you are given 100 coins; your goal is to maximize the total rewards collected.

- What can you do?

Arm 1	Arm 2	Arm 3	Arm 4
??% current success rate	??% current success rate	??% current success rate	??% current success rate

At the very beginning, you randomly select one arm, insert a coin and see if it returns some coins.

- The more you try it, the more you know about it.

K-armed bandit problem

After 30 trials, you got some knowledge about the arms

Now, what will you do in the 31st trial?

Arm 1	Arm 2	Arm 3	Arm 4
40%	50%	60%	30%
current success rate	current success rate	current success rate	current success rate

You have two options:

- Pull Arm 3 so that win at probability of 60% → (exploitation)
- Pull anyone of the arms (e.g., Arm 4) to try your luck → (exploration)

Exploration-exploitation dilemma

Optimal performance requires some balance between exploratory and exploitative behaviors.

- Exploration gives new knowledge
- Exploitation uses the existing knowledge

The trade-off between the need to obtain new knowledge and the need to use that knowledge to improve performance is one of the most basic trade-offs in nature.

Exploration v.s. Exploitation

- Investment Portfolio:
  - Exploitation: Investing in stocks that have historically performed well
  - Exploration: Allocating a portion of the portfolio to new, high-risk/high-reward stocks

- Course registration:
  - Exploitation: Register one that you are confident to get an A
  - Exploration: Try a course of different area you are interested in

- Clinical trial:
  - Exploitation: Choose the best treatment so far
  - Exploration: Try a new treatment

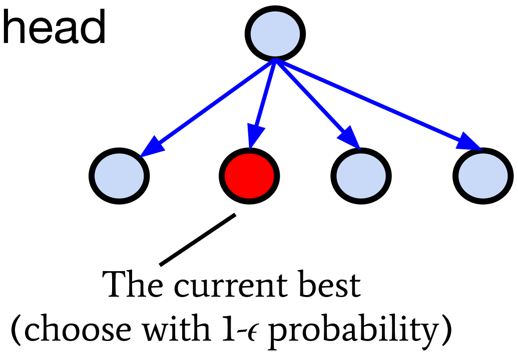
Modeling the trade-off: ϵ-greedy

The ϵ-greedy algorithm:

- With chance ϵ: explore randomly (to explore)
- Otherwise: choose the current best (to exploit)

It is like you throw a biased coin with probability ϵ to get the head up when you have to make a decision.

- If the head is up, you do exploration;
- Otherwise, you do exploitation;



Implement epsilon-greedy

By using `random.random()` we get a random number in [0, 1]; if it < epsilon, we do exploration; otherwise, we do exploitation.

```python
def epsilon_greedy(epsilon:float, arms:list):

    idx = [i for i in range(len(arms))]
    if random.random() < epsilon:    # explore
        choice = random.choice(idx)
        print("explore")
    else: # exploit
        choice = 0
        for i in range(1, len(arms)):
            if arms[choice].get_win_p() < arms[i].get_win_p():
                choice = i
    return arms[choice]
```

```python
class Arm:

    def __init__(self, idx, win_prob):
        self.idx = idx
        self.win_prob = win_prob

    def get_win_p(self):
        return self.win_prob
```

- arms is a list of Arm objects.

Reinforcement learning

--- To find out the reward

Origin: The Law of Effect

TL;DR: Animals associate their actions to the situations by the rewards.

"Of several responses made to the same situation, those which are accompanied or closely followed by **satisfaction** to the animal will, other things being equal, be more firmly **connected** with the **situation**, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by **discomfort** to the animal will, other things being equal, have their **connections** with that **situation** weakened, so that, when it recurs, they will be less likely to occur. **The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond**"

--- The law of effect [Thorndike, 1911]

Edward Lee Thorndike (1874-1949), Psychologist, Father of Educational Psychology

Reinforcement learning (RL)

Goal:

- To find the best action-state relations ⇒ a **policy** by which the agent will **maximize** its **total reward** if it follows the policy.
  - If we know the *action-state-reward table*, we can achieve the goal by choosing the action that brings the highest reward at each state.
  - The reward at each state represents the long-term reward (we may call it as the value).

Learn to play an Atari game (breakout)

https://elgoog.im/breakout/

In each episode, you move the pad to hold the ball

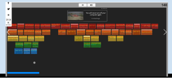
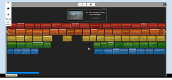
- Success: + a few points
- Failed: game over, start a new episode

Goal: get as many points as you can!

What is the actions?
What is the environment and state?
What can be the reward?

If we know the long term rewards,

Q: the action-value function; it can be represented by a table, if the number of actions and states are finite.

action/state				……(many states)
Move left	10	12	0	……
Move right	0	15	10	……
Don’t move	5	0	20	……

- Each cell represents the highest rewards of an (action, state) in a long run;
- Once the Q is known, the agent can maximize the total rewards by taking the action that gives the highest long term reward at each state.

The reward of a (state, action) (i.e., long term reward)

Mathematically, reinforcement learning is to maximize the sum of rewards in the long term:

$$\sum_{t=0}^{t=\infty} \gamma^t r(s(t), a(t)),$$

Find a r(s, t) that can maximize the long term reward.

where $\gamma \in [0, 1]$ is a discount factor that quantifies the difference in importance between immediate rewards and future rewards; $r(s, a)$ is a reward function. For state s and action a, $r(s, a)$ gives you the reward associated with taking action a at state s.

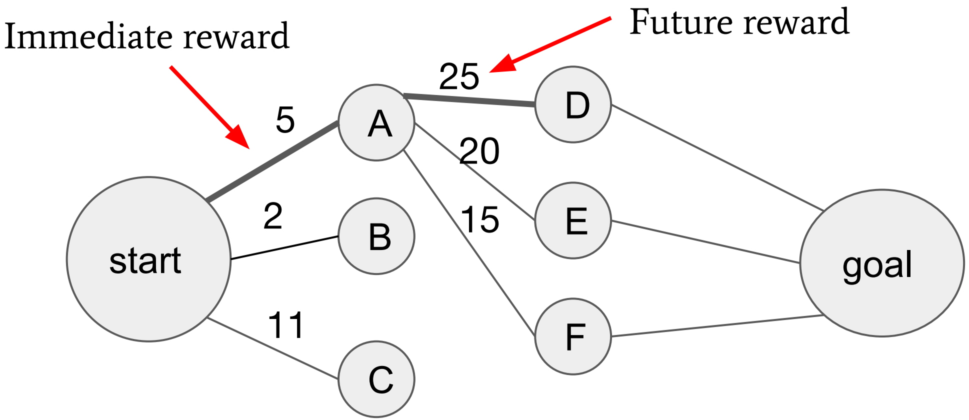
With the equation, we formalize the task of reinforcement learning as maximizing the sum of long-term rewards by taking the best action in each state.

The two components of the long term reward

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma * \max\{Q(s_{t+1}, \text{all valid } a)\}$$

The **long-term reward** = the immediate (transition) reward + the future reward

- action a_t will change state s_t to s_{t+1} ⇒ we calculate the future reward based on it
- **Immediate reward**: the value of the action from **the transition** from the current state to the next;
- **Future reward**: the value of **the best action in the next state** (being discounted)



Reward discounting

- Reward is influenced by time: the value of the reward is often higher if it is given to you right now than in future.
  - $100 today has higher value than $100 in 10 years later

- Therefore, we need to discount the rewards by time.

Q-learning

https://youtu.be/z48JCQZwwzA

- **Q-learning:** finding the reward of each action at each state ⇒ using the “learn by doing” strategy

- Once we know what the Q table is, for each state, we can choose the action that gives the largest reward.

Q-learning: estimating the reward iteratively

Let $Q(s, a)$ be an action-value function that maps a $(state, action)$ pair to an expected reward (i.e., a real number); R be a matrix represent the transition rewards (i.e., the reward of moving from one state to another), and γ be the discount factor. We update the Q by the following

$$\underbrace{Q(s_t, a_t)}_{\text{Updated Q}} = \underbrace{R(s_t, a_t)}_{\text{Transition reward}} + \gamma \times \boxed{\max\{Q(s_{t+1}, \text{all valid } a)\}}$$

Expected reward, i.e., the max reward of the possible actions at next state (i.e., t+1)

Q before updated

R is a given function; Q is updated by adding the current transition reward and the max achievable future reward based on the current Q. (We will update Q for many times until the Q doesn't change.)

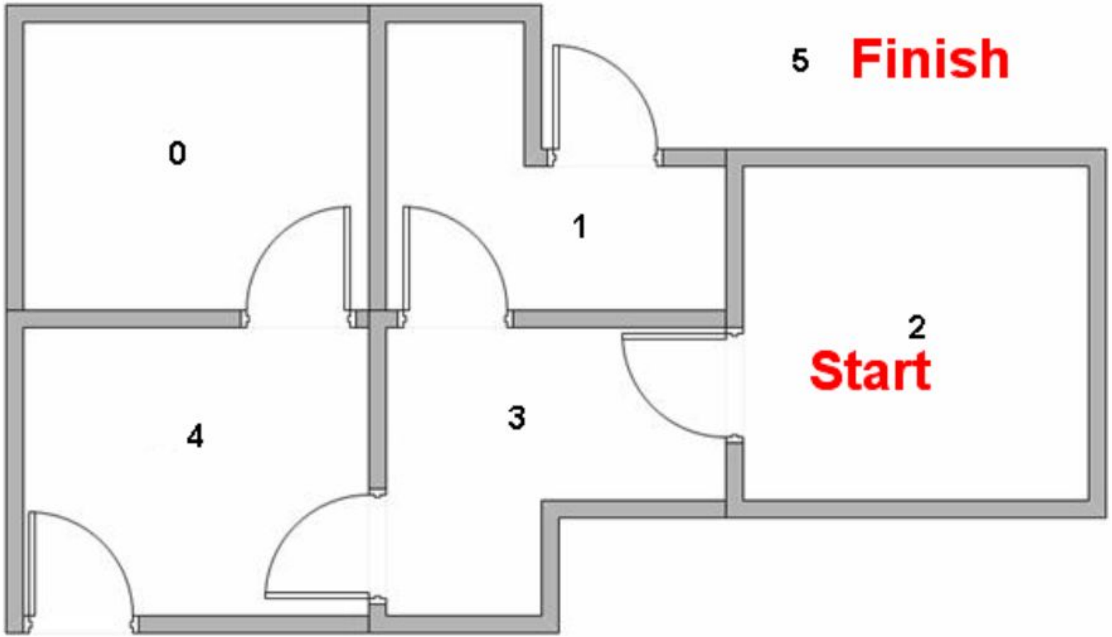
- Q is reinforced in each iteration (the perception-action cycle)
- γ = 0: ignore the future reward, total greedy; γ = 1: emphasize future reward

Let's have an example: getting out of a house

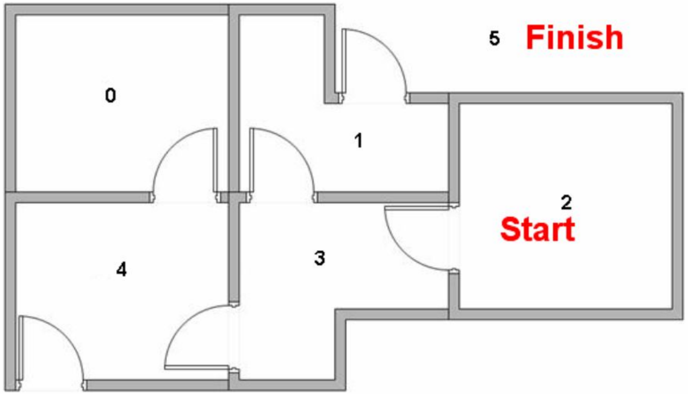
We have a 5 bedroom house

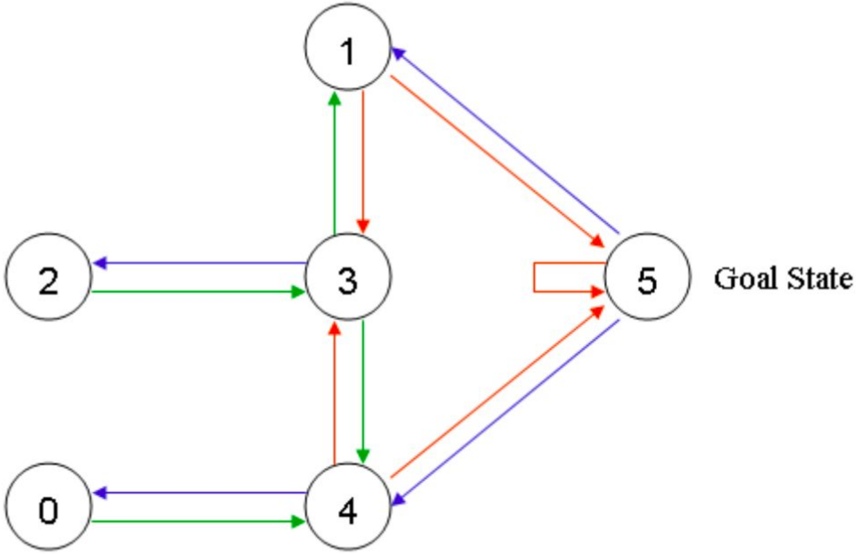
- Each room has a number 0 through 4
- The outside can be thought of as one big room 5
- Doors in room 1 and 4 lead into room 5 (outside)

Q: How can a robot learn to get out?



Defining the immediate awards

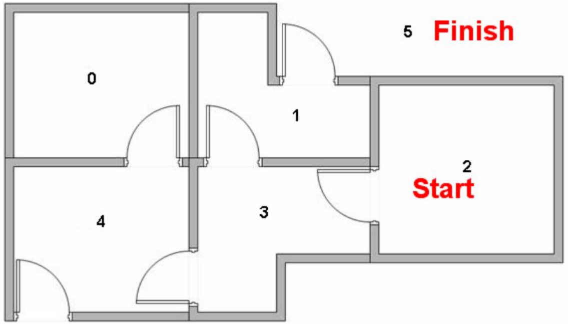




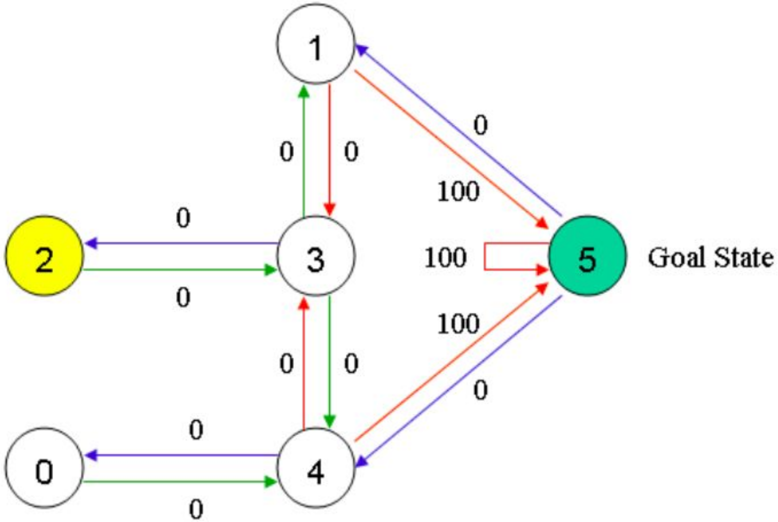
The goal room is 5. Let's set

- The doors that lead immediately to the goal have an instant reward of 100.
- Other doors have 0 reward. (since we know nothing at the beginning.)

R: Reward[state, action]



$$
\begin{array}{rcc}
 & & \text{Action} \\
 & \text{State} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
R= & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix}
-1 & -1 & -1 & -1 & 0 & -1 \\
-1 & -1 & -1 & 0 & -1 & 100 \\
-1 & -1 & -1 & 0 & -1 & -1 \\
-1 & 0 & 0 & -1 & 0 & -1 \\
0 & -1 & -1 & 0 & -1 & 100 \\
-1 & 0 & -1 & -1 & 0 & 100
\end{bmatrix}
\end{array}
$$



- **"-1s"** in the table represent null values (i.e., where there isn't a link between nodes)
- **R** is obtained from the current environment

Q: action-value at a given state

$$Q(\text{state}, \text{action}) = R(\text{state}, \text{action}) + \gamma * (\text{Max}\{Q(\text{next state}, \text{all valid actions})\})$$

Algorithm:

0. initialize Q to be all zero

Loop until converge (or bored; each loop is an episode in which the agent gains experience):

1. Pick a random starting state
2. Pick the best action according to Q, or randomly explore - controlled with the parameter epsilon (i.e., epsilon-greedy)
3. Update Q(state, action)
4. State = next state, go to 2; if terminate go to 1

Question:

- Do we have to record the goal state (5) in the R matrix?

Q-learning in action: 1st episode

Q(state, action) = R(state, action) + γ*(Max{Q(next state, all valid actions)})
We start by setting:
γ = 0.8
action: 1 → 5 (i.e., initial state as room 1, we randomly select to go room 5)

R(1, 5) = 100
max{ Q(5, 5), Q(5, 1), Q(5, 4)} = 0

$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Because 5 is the goal state,
we've finished one episode.

Q-learning in action: 2nd episode

Q(state, action) = R(state, action) + γ*(max{Q(next state, all valid actions)})
We start by setting:
$\gamma = 0.8$
action: 3 → 1 (i.e., initial state as room 3, we randomly select to go room 1)

Action

$$R = \begin{array}{c|c} \text{State} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c|c} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

3→1:
R(3, 1) = 0
max{Q(1, 5), Q(1, 3)} = 100
Q(3, 1) = R(3,1) +0.8*100 = 80

$$Q = \begin{array}{c|c} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Q-learning in action: 2nd episode

Q(state, action) = R(state, action) + γ*(max{Q(next state, all valid actions)})
We start by setting:
$\gamma = 0.8$
action: 3 → 1 → 5 (i.e., initial state as room 3, we randomly select to go room 1, then randomly go to 5)

1→5:
R(1, 5) = 100
max{Q(5, 1), Q(5, 4), Q(5, 5)} = 0
Q(1, 5) = 100 + 0.8*0 = 100

Action

$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

The 2nd episode ends as the agent reaches room 5.

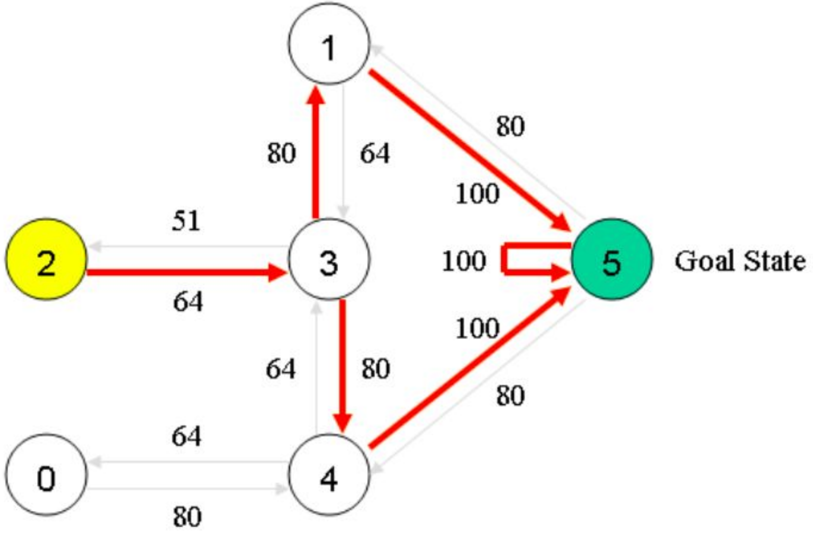
Eventually…

After many episodes…

$$Q = \begin{array}{cc} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix} \end{array}$$

Normalize to [0, 100]

$$Q = \begin{array}{cc} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$



The matrix Q is a guide to let the agent go to the goal state.

The "world"

```python
''' a simple case: 0 <-> 1 <-> 2; room #2 is the exit'''
R_simple = [
            [-1, 0, -1],
            [0, -1, 100],
            [-1, 0, 100]
            ]

''' a more complex world, see lecture example '''
R_room = [
    [-1, -1, -1, -1, 0, -1],
    [-1, -1, -1, 0, -1, 100],
    [-1, -1, -1, 0, -1, -1],
    [-1, 0, 0, -1, 0, -1],
    [0, -1, -1, -1, -1, 100],
    [-1, 0, -1, -1, 0, 100]
    ]
```

The the rows are the states and the columns are the actions.

```python
class World():
    def __init__(self, rewards, final_state):
        self.rewards = rewards
        self.num_states = len(rewards)
        self.num_acts = len(rewards[0])
        self.final_state = final_state

    def is_final_state(self, s):
        return s == self.final_state

    def get_num_states(self):
        return self.num_states

    def get_num_acts(self):
        return self.num_acts

    def get_valid_acts(self, s):
        acts = rl_helper.get_valid_acts(self, s)
        return acts

    def get_reward_in_state_act(self, s, a):
        return self.rewards[s][a]
```

The "world"

You need to implement this. The function returns all possible actions of a state s.

The agent

```python
class Agent():
    def __init__(self, n_state, n_act, gamma = 0.0, epsilon = 0.0):
        self.Q = [ [0 for i in range(n_act)] for j in range(n_state)]
        self.gamma = gamma
        self.epsilon = epsilon

    def get_maxQ(self, s):
        return max(self.Q[s])

    def get_maxQ_act(self, s, acts):
        max_q = -1
        for a in acts:
            if self.Q[s][a] > max_q:
                max_q = self.Q[s][a]
                max_a = a
            elif self.Q[s][a] == max_q: # make some randomness when tie
                if random.random() < 0.5:
                    max_a = a
        return max_a

    def make_move(self, s, valid_moves):
        act = rl_helper.make_move(self, s, valid_moves)
        return act

    def update_Q(self, s, a, next_s, r):
        rl_helper.update_Q(self, s, a, next_s, r)
        return

    def set_Q(self, s, a, val):
        self.Q[s][a] = val
```

You need to implement these two functions.

- In make_move(), you need to apply epsilon-greedy to determine the action. And, remember to call random.choice() for randomly select a action.
- Remember to use self.set_Q() in update_Q().

Set up the learning

```
if __name__ == "__main__":
    ''' set up the world'''
    w = World(R, 5)
    num_states = w.get_num_states()
    num_actions = w.get_num_acts()

    ''' set up the agent '''
    gamma = 0.8
    epsilon = 0.2
    agent = Agent(num_states, num_actions, gamma, epsilon)

    ''' set up the runs '''
    total_iter = 500
    num_iter = 0
    interactive = False
    verbose = False
    report_steps = 100
```

The learning loop

```python
    ''' start learning '''
    while num_iter < total_iter:
        ''' init one random state '''
        s = random.choice(list(range(num_states)))
        moves = [s]
        while True:
            ''' make one move '''
            valid_moves = w.get_valid_acts(s)
            act = agent.make_move(s, valid_moves)

#            print(s, act, valid_moves)
            this_reward = w.get_reward_in_state_act(s, act)
            ''' for this example, next state is equal to act '''
            next_s = act
            agent.update_Q(s, act, next_s, this_reward)
            s = next_s

            moves.append(s)
            if w.is_final_state(s):
                break

        if verbose:
            print("moves: ", moves)

        num_iter += 1
        if num_iter % report_steps == 0:
            print(agent)
```

Homework

[RL] Implement the Q-learning algorithm

- **Environment**: get_valid_acts on page 42
- **Agent**: make_move, update_Q on page 43
- Play with the code: change parameter setting

Useful function:

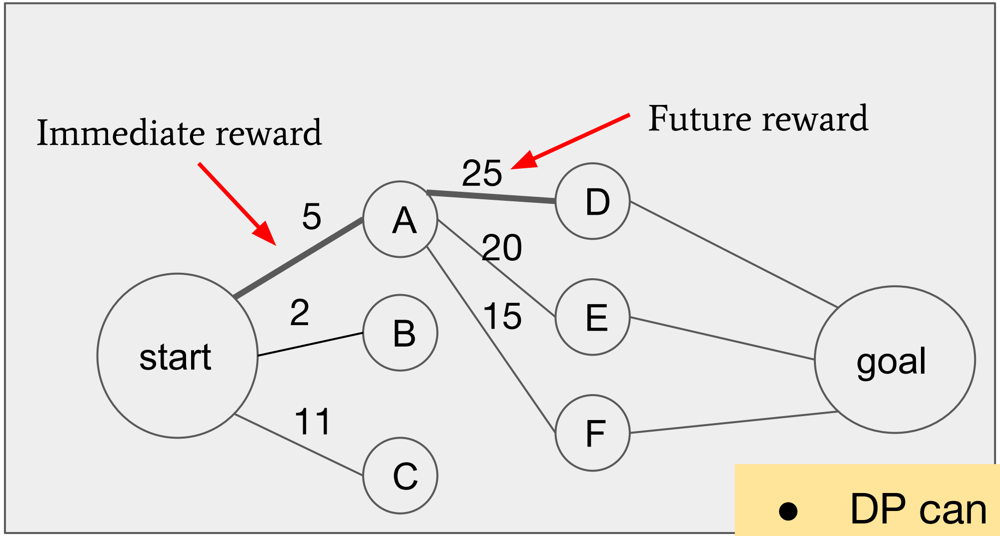
random.random(), random.choice(your_list)

Again:

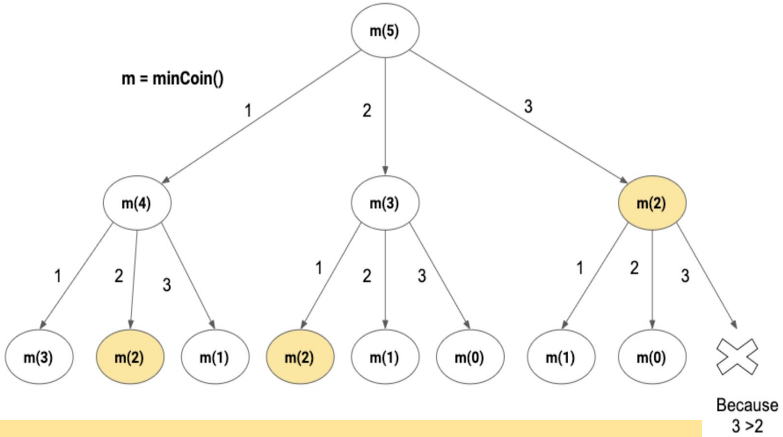
Q(state, action) = R(state, action) + γ*(max{Q(next state, all valid actions)})

Reinforcement Learning vs. Dynamic Programming

RL: evaluating the values by many trials



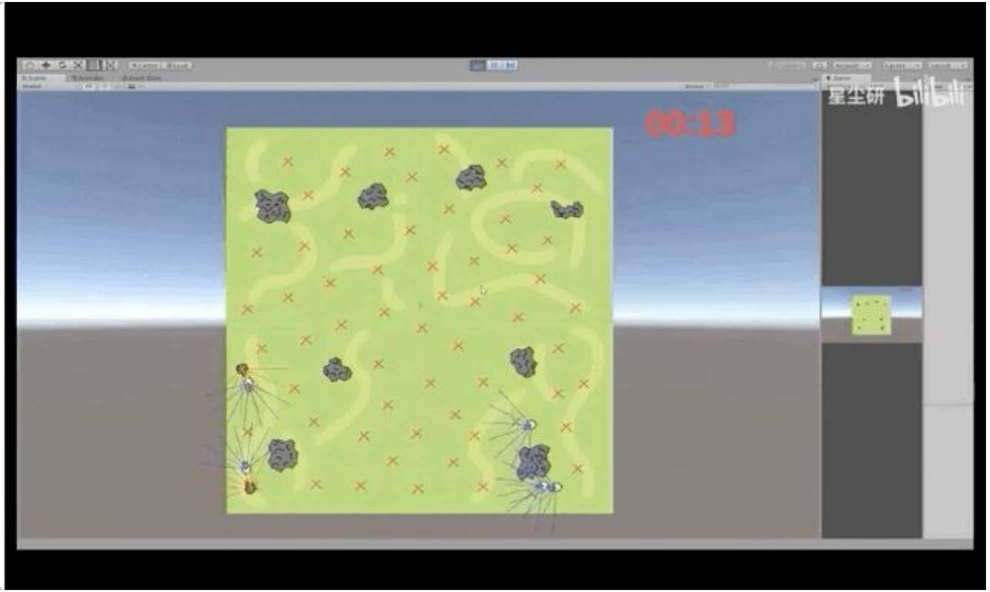
DP: calculating the values by visiting all paths



- DP can be a special case of RL: in each trial, it chooses a new path. (A “dummy” agent, who only recite the visited nodes.)
- We can solve DP problems using RL with a **proper defined reward**.

Reward is critical. (You may refer to Reward is enough.)

What will happen the wolf feels sad?



Involution happens to the wolf AI https://www.youtube.com/watch?v=TBn1aDrPAWw (in Chinese)

Train the agents by competition: a wolf get more sheep is the winner
The rewards for the wolves:

- Successfully catch a sheep: 10
- Punishment: hit the rock (-1) and (-0.1) every second until catching a sheep

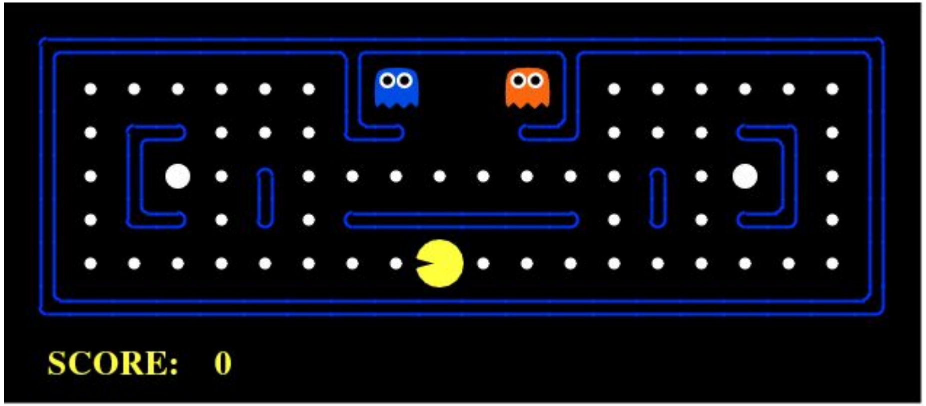
Wolves sucide themselves because of the unbalance between positive reward and punishment: die earlier (devolution 躺平) ⇒ less punishment ⇒ win others in the game

DeepMind

- General-purpose learning machine
  - Learn automatically from raw inputs
  - Operate across a wide range of tasks

Full video: https://www.youtube.com/watch?v=rbsqaJwpu6A&feature=youtu.be&t=301

Appendix: Deep Reinforcement Learning



A problem

Q is a table of size $m \times n$.

- If we have 4 different actions and 10 different states, then Q is in the size of 10×4.

But what if we have a much bigger state space?

A problem

But what if we have a bigger state space?

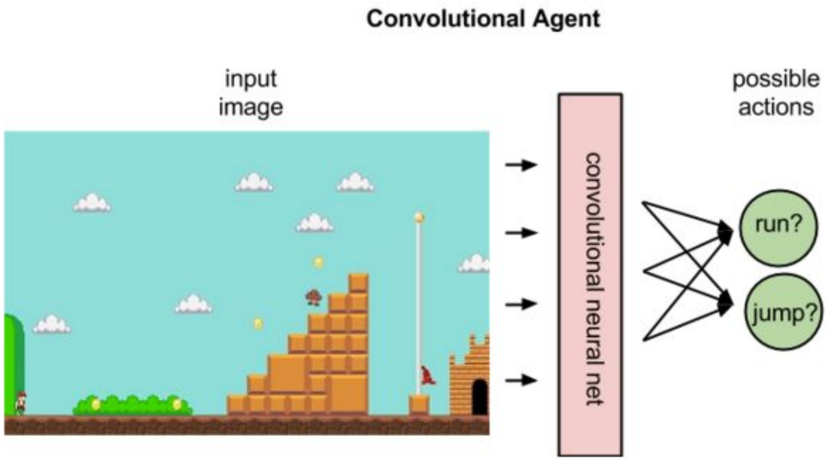
- For example, in an Atari game where each frame in the game is treated as a single state, then we may have millions of states.

So, we have to store a table that have millions of rows in the memory, which is not a good idea.

One solution

We approximate the Q by a neural network.

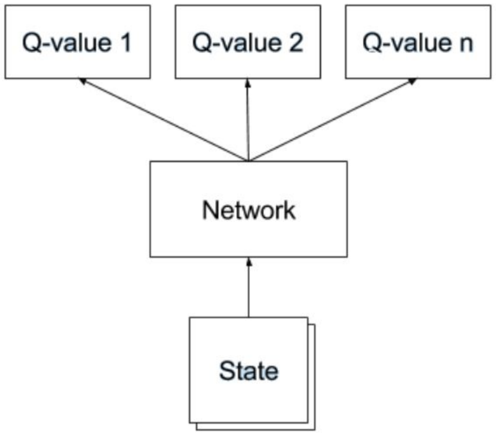
- Instead of finding the Q-value in the table, we calculate it by a neural network each time.



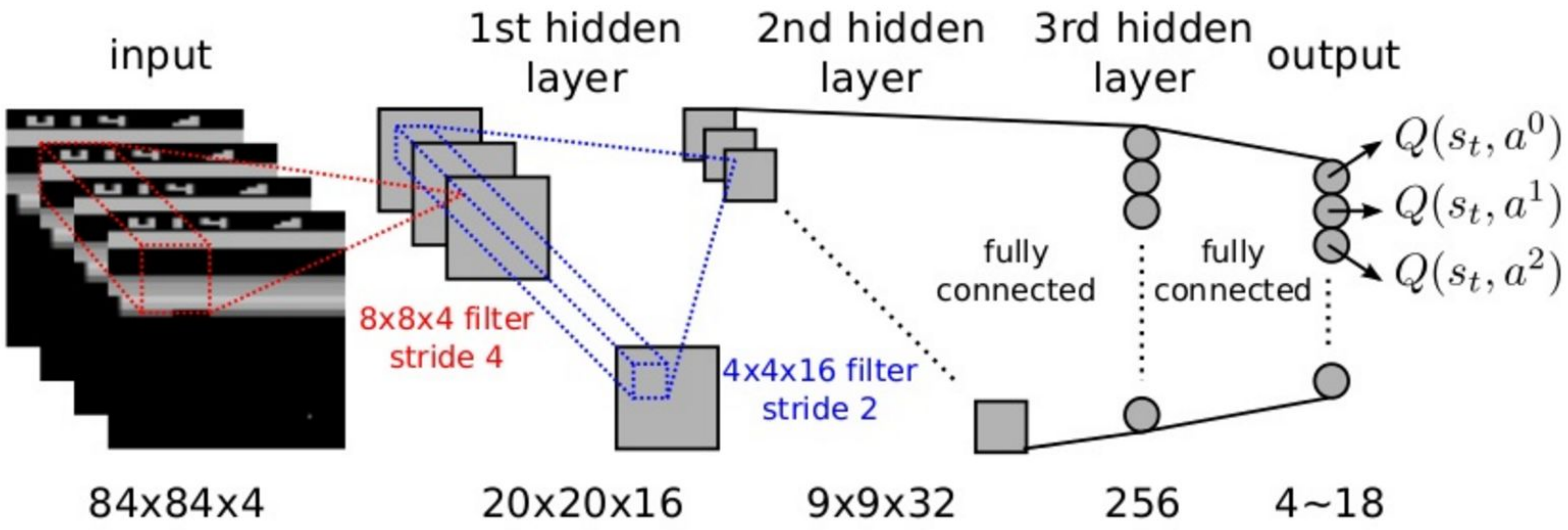
How?

The network takes a state as the input and produce the Q-values for every action in the action space.

- The NN should learn the parameters for such predictions.
- Once the NN is trained, we use it to predict the next Q-values and take the action corresponding to the highest Q-value



Deep reinforcement learning



When connected with a deep NN, it is called **deep reinforcement learning**.

Examples

- PacMAN: https://github.com/tychovdo/PacmanDQN
- Atari games:
  https://github.com/Madhu009/Deep-math-machine-learning.ai/tree/master/Reinforcement%20learning

Summary

- Deep reinforcement learning =

  Deep learning + Reinforcement learning

- “Deep learning without labels and reinforcement learning without tables.”

Appendix: DeepMind Protein folding

Superhuman performance achieved:

Protein folding explained

AlphaFold: The making of a scientific breakthrough

AlphaGo

If you are a gamer,

Starcraft II, Dota 2