Reinforcement Learning (Single Agent Q-learning)

ECE 277

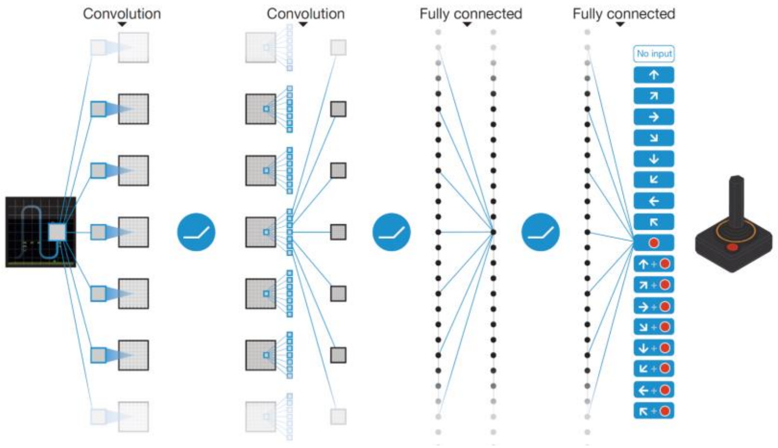
Cheolhong An

Deep Reinforcement learning: RL + Deep Learning



Learned to play 49 games for the Atari 2600 game console, without labels or human input, from self-play and the score alone

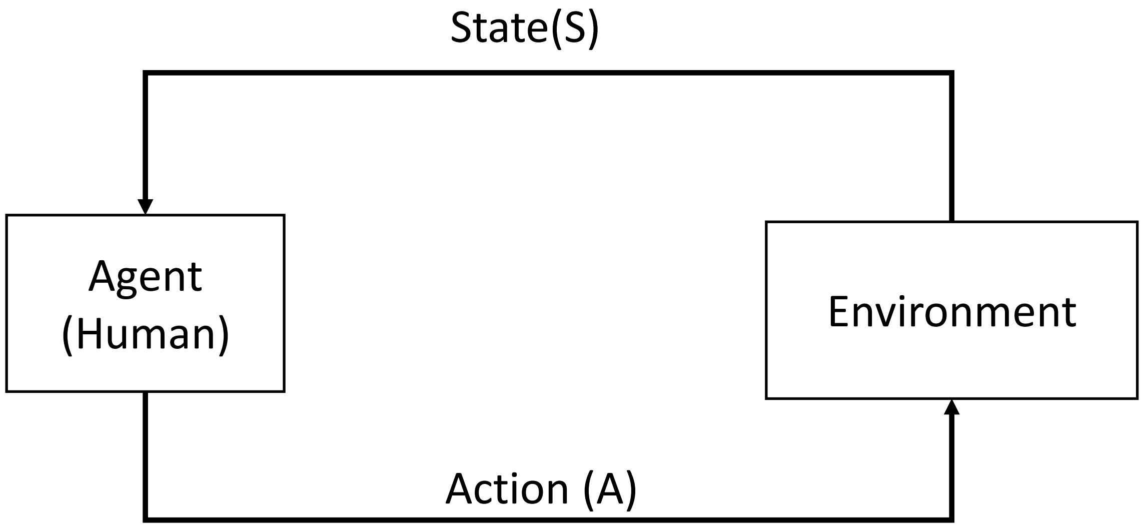
mapping raw screen pixels

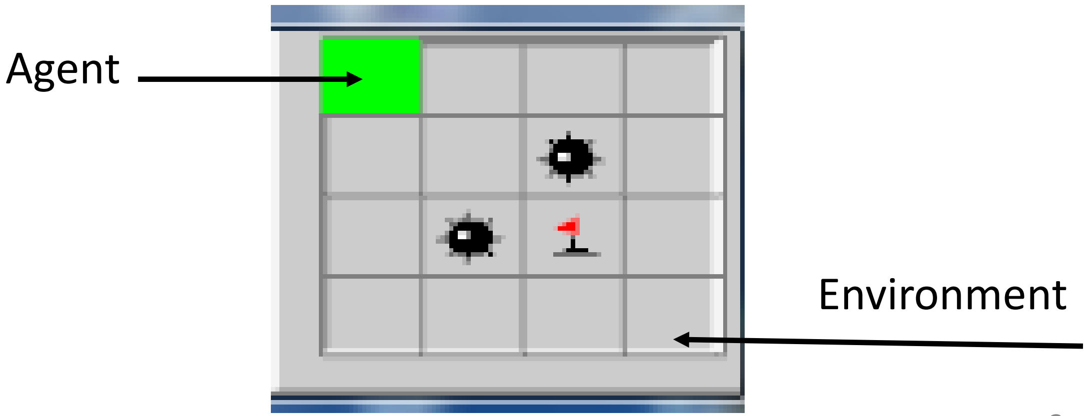


to predictions of final score for each of 18 joystick actions

Reinforcement learning environment: No learning (Human agent)

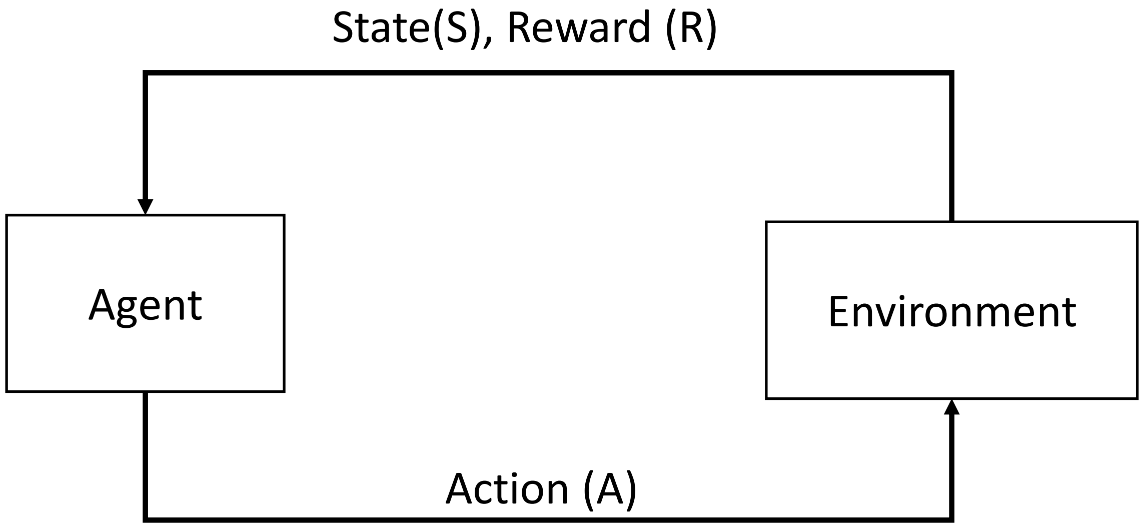
- Goal
  catch flag
- Environment
  4x4 grid world
  two mines and one flag
- State
  (x,y) position of an agent
- Reward
  catch flag: +1
  step mine: -1
  otherwise: 0
- Action
  0: right, 1: down , 2: left, 3: up
- Episode end
  Agent reaches one of mines or a flag
  Every episode restarts from (0,0)

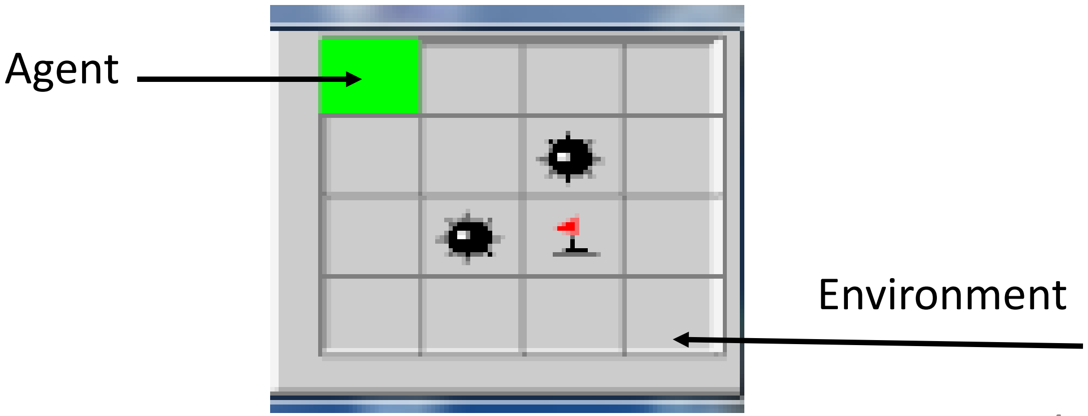




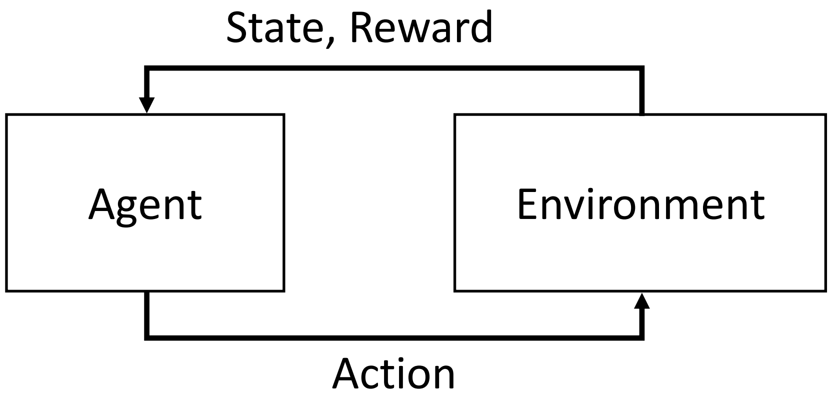
Reinforcement learning: Q-learning (single agent)

- Goal
  catch flag
- Environment
  4x4 grid world
  two mines and one flag
- State
  (x,y) position of an agent
- Reward
  catch flag: +1
  step mine: -1
  otherwise: 0
- Action
  0: right, 1: down , 2: left, 3: up
- Episode end
  Agent reaches one of mines or a flag
  Every episode restarts from (0,0)





Q-learning



Initialize $Q(s,a) = 0, \forall s \in S, a \in A(s)$ ← agent_init()

Repeat (for each episode):

Initialize S

Repeat (for each step of episode:)

Choose A from current state S using policy derived from Q (e.g. ϵ-greedy)

Take action A

agent_action()

Observe next state S' and R

$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

agent_update()

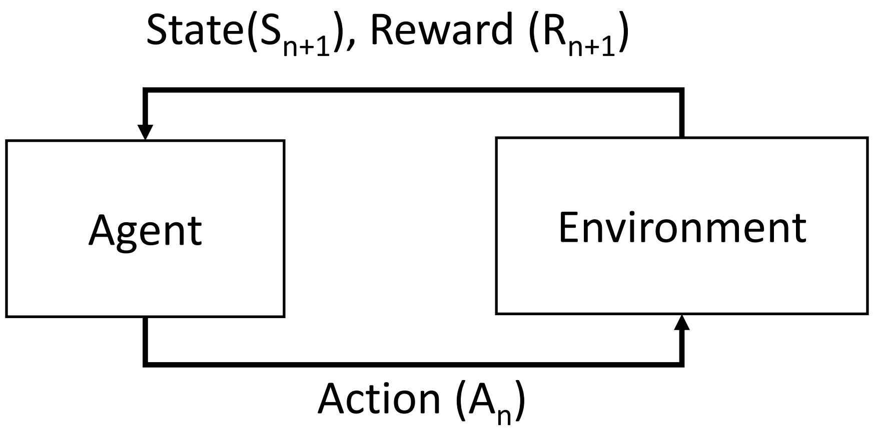
$S \leftarrow S'$

Until S is terminal

R.S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction

Agent: Action

- Policy: ε-Greedy
- Action
  0: right, 1: down , 2: left, 3: up
- S_n: current state
- You need to decrease ε every episode
  ex) initial value: 1.0 -> 0.1 (minimum value)
  Exploration is more important from beginning
- Exploitation: Make the best decision given current information
- Exploration: Gather more information



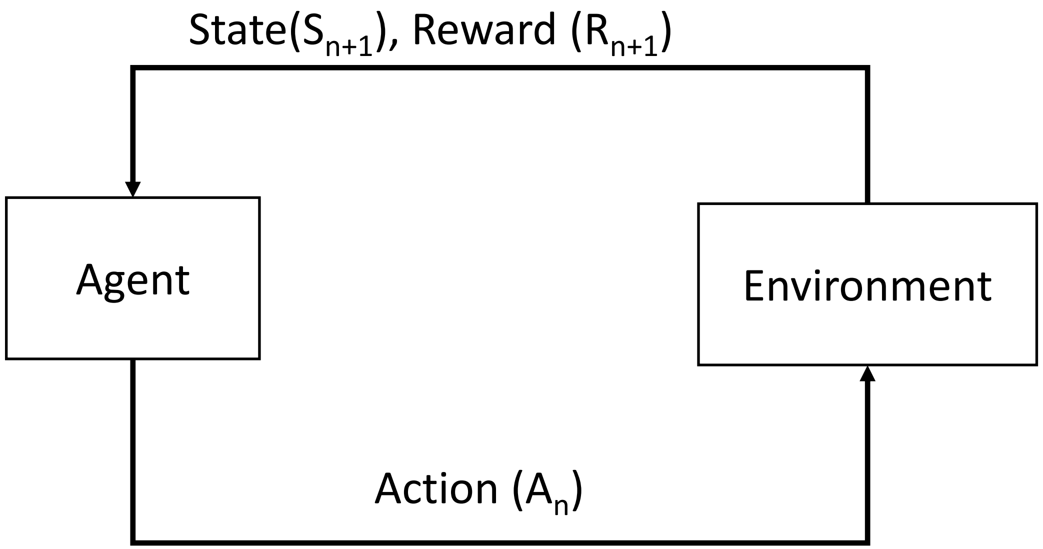
if (uniform(0,1) $<$ ϵ)
  a = uniform(0, #actions) ← Exploration
else
  a $=$ $\arg\max_{a'} Q(S_n, a')$ ← Exploitation (Greedy policy)

Agent: update Q-table: Q-learning

- Update Q-table
- R_{n+1}: reward
- S_{n+1}: next state
- S_n: current state

α : learning rate

γ : discount factor



Gradient ascent

error

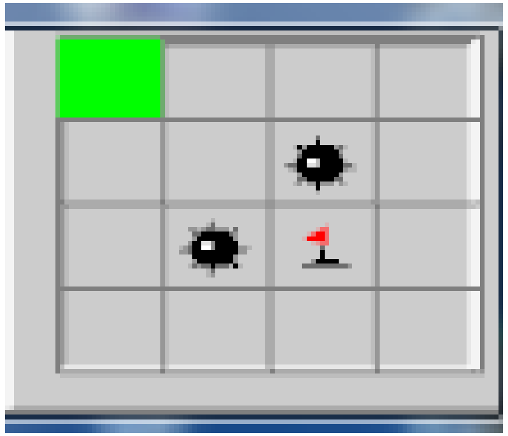
$$Q(S_n, A_n) \mathrel{+}= \alpha\left[R_{n+1} + \gamma \max_{a'} Q(S_{n+1}, a') - Q(S_n, A_n)\right]$$

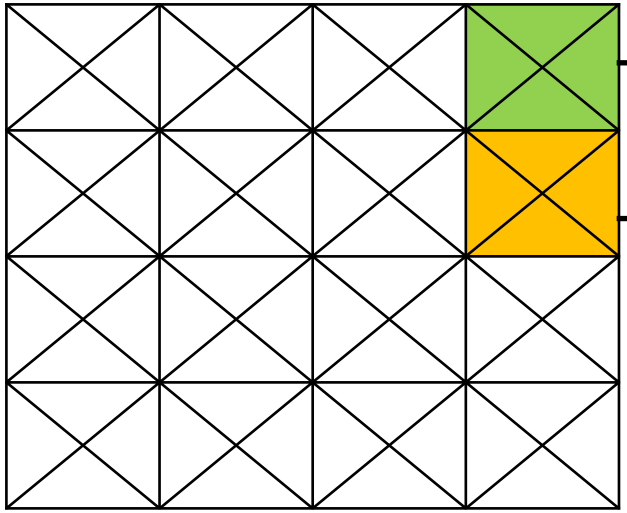
prediction to Q

Q-table

- This is an example of Q-table
- Action A

  0: right, 1: down , 2: left, 3: up





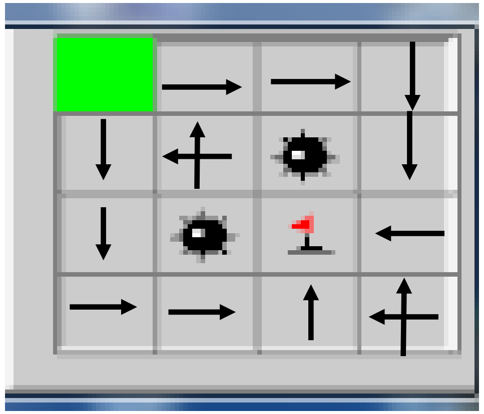
→	↓	←	↑	
0.1	0.5	0.3	0.1	$Q(S_n, A), \text{where } S_n = (0, 3)$
0.01	0.69	0.1	0.2	$Q(S_n, A), \text{where } S_n = (1, 3)$

Q(S,A) table

Optimal Policy: Greedy after Q-learning

- For given state, S_n

$$\mathrm{a} \;=\; \arg\max_{a'} Q(S_n, a')$$



Source code view

```
if (m_episode == 0 && m_steps==0) {// only for first episode
    env.reset(m_sid);
    agent_init(); // initQ table + self initialization
}else {
    active_agent = checkstatus(board, env.m_state, flag_agent);

    if (m_newepisode) {
        env.reset(m_sid); // clear buffer
        float epsilon = agent_adjustepsilon(); // adjust epsilon
        m_steps = 0;
        printf("EP=%4d, eps=%4.3f\n", m_episode, epsilon);
        m_episode++;
    }else {
        short* action = agent_action(env.d_state[m_sid]);
        env.step(m_sid, action);
        agent_update(env.d_state[m_sid], env.d_state[m_sid ^ 1], env.d_reward);

        m_sid ^= 1;
        episode = m_episode;
        steps = m_steps;
    }}
m_steps++; env.render(board, m_sid); return m_newepisode;
```

Algorithm view

Initialize $Q(s,a) = 0, \forall s \in S, a \in A(s)$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode:)
    Choose A from current state S using policy derived from Q (e.g. ϵ-greedy)
    Take action A
    Observe next state S' and R
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$
  Until S is terminal

Exchange data types

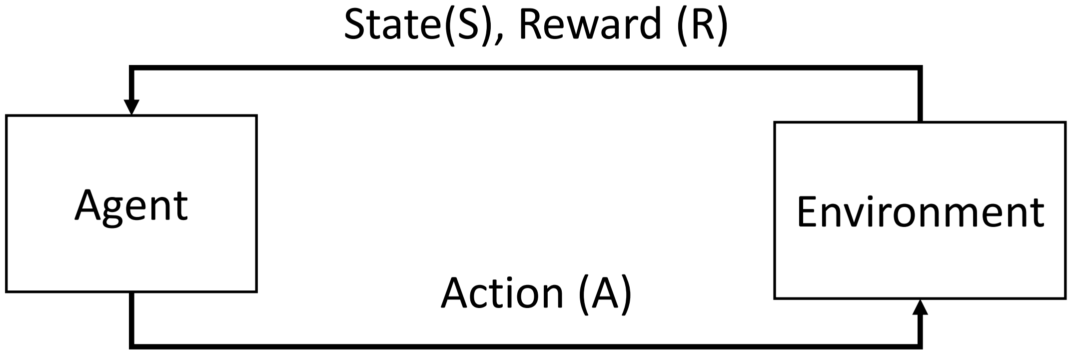
- S_n: current state
  env.d_state[m_sid]: *int2 for each agent
- S_{n+1}: next state
  env.d_state[m_sid^1]: *int2 for each agent
- Each state indicates of position
  d_state[agent_id].x: x position
  d_state[agent_id].y: y position

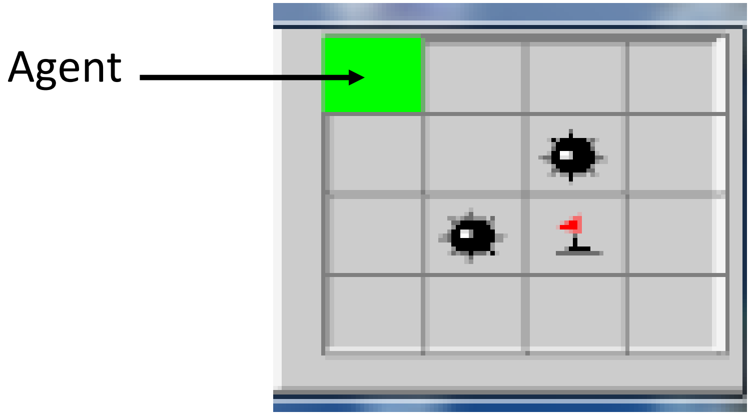
- Reward:
  env.d_reward: *float for each agent
  d_reward[agent_id]: catch flag: +1, step mine: -1, otherwise: 0

Reinforcement learning: Q-learning

- Current status
  (x, y): position of an agent
  env.d_state[m_sid]

- Update agent
  current status: env.d_state[m_sid]
  next status: env.d_state[m_sid ^ 1]
  reward: env.d_reward
- Action
  0: right, 1: bottom , 2: left, 3: top
- Agent status
  inactive (nonzero reward) or active (zero reward)





- curand_uniform usage

  - Onetime allocation for every thread
    - curandState *states
    - cudaMalloc((void **)&states, sizeof(curandState) * threads_per_block * blocks_per_grid)

    - You should pass a pointer of **curandState**

    __global__ void device_api_kernel(**curandState *states**, float *out, int N)

  - Onetime Initialization of a kernel: All the threads need to maintain their own states

```
__global__ void init_randstate(int size, curandState *state)
{
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    curand_init(clock() + tid, tid, 0, &state[tid]);
}
```

- __global__ void kernel_fun (curandState *d_randstate)

  int tid = blockIdx.x*blockDim.x + threadIdx.x;

  - Just call curand_uniform(&d_randstate[tid])
  - **Not necessary to reinitialize curandState**