

ITCS 6156/8156 Spring 2024 Machine Learning

Reinforcement Learning

Instructor: Hongfei Xue

Email: hongfei.xue@charlotte.edu

Class Meeting: Mon & Wed, 4:00 PM – 5:15 PM, Denny 109

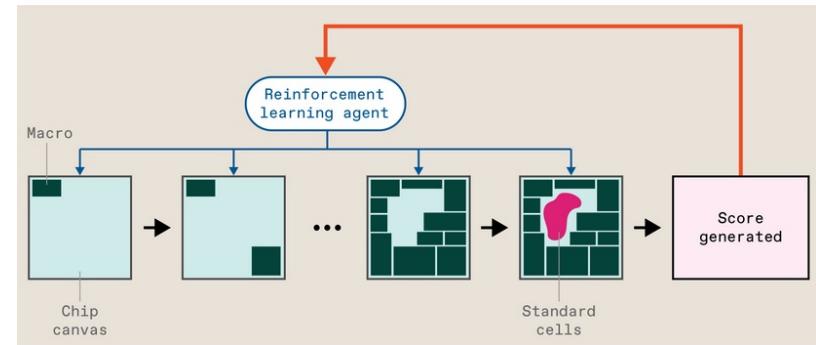


Some content in the slides is based on Dr. Raquel Urtasun's lecture

Fancy Applications of RL

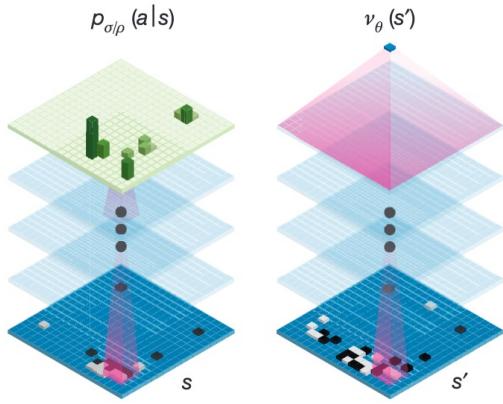


Play Dota

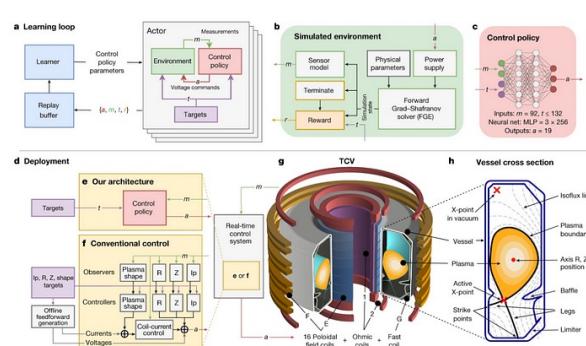


Chip Design

Policy network



AlphaGo



Nuclear Fusion



AWS Deep Racer

Playing Games: Atari

From YouTube: <https://www.youtube.com/watch?v=V1eYniJ0Rnk>

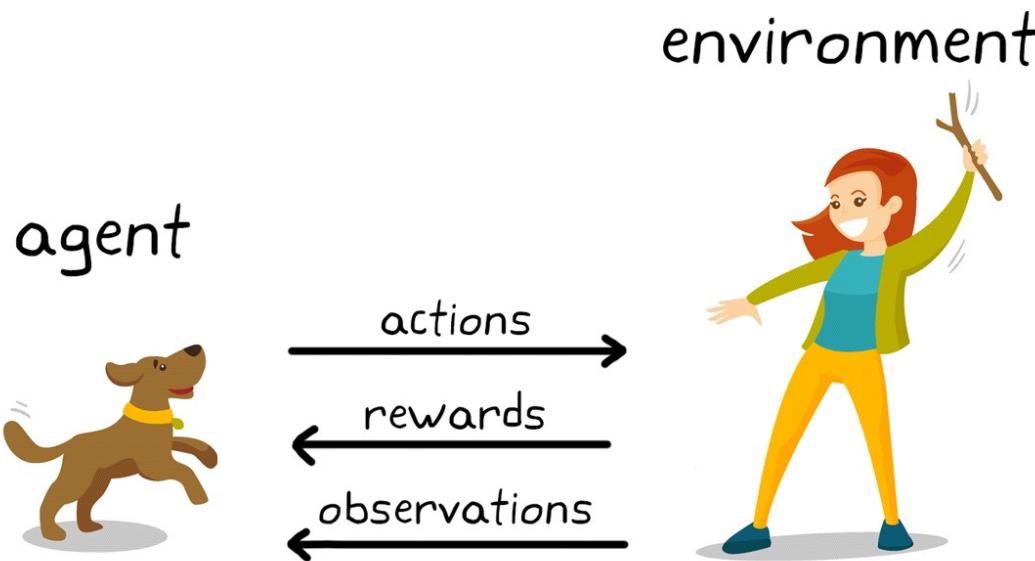
Playing Games: Hide and Seek

From YouTube: <https://www.youtube.com/watch?v=Lu56xVlZ40M>

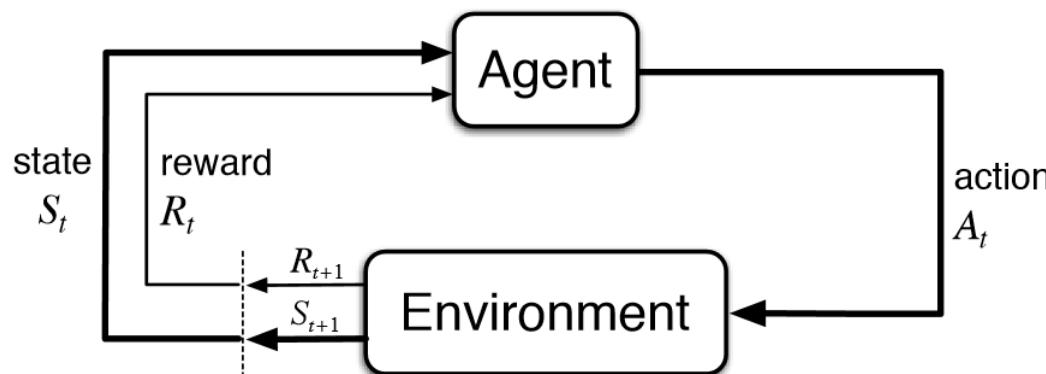
Reinforcement Learning

- Learning algorithms differ in the information available to learner
 - ▶ Supervised: correct outputs
 - ▶ Unsupervised: no feedback, must construct measure of good output
 - ▶ Reinforcement learning
- More realistic learning scenario:
 - ▶ Continuous stream of input information, and actions
 - ▶ Effects of action depend on state of the world
 - ▶ Obtain reward that depends on world state and actions
 - ▶ not correct response, just some feedback

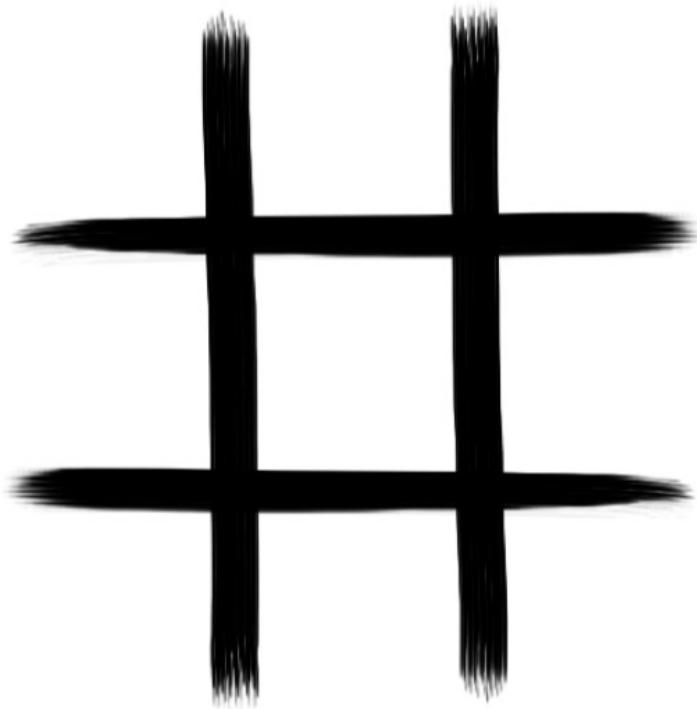
Reinforcement Learning



Reinforcement Learning in Dog Training

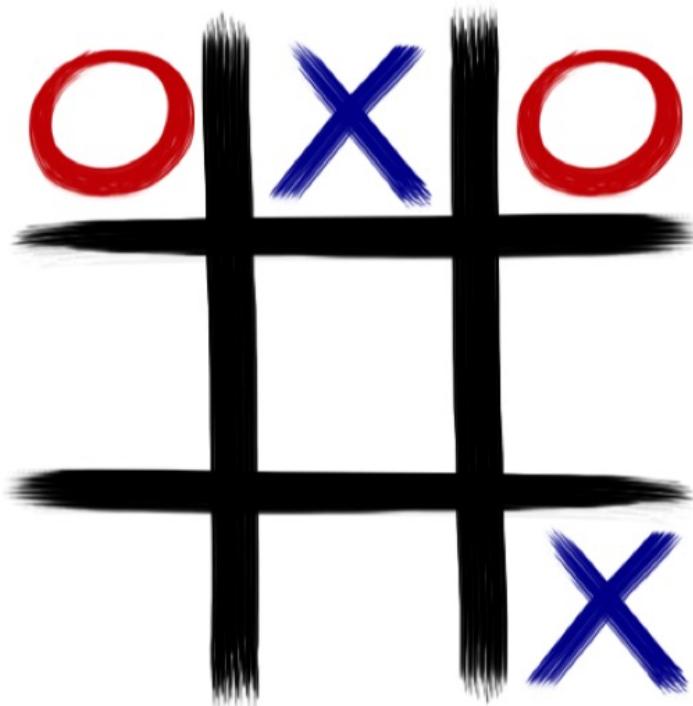


Example: Tic Tac Toe, Notation



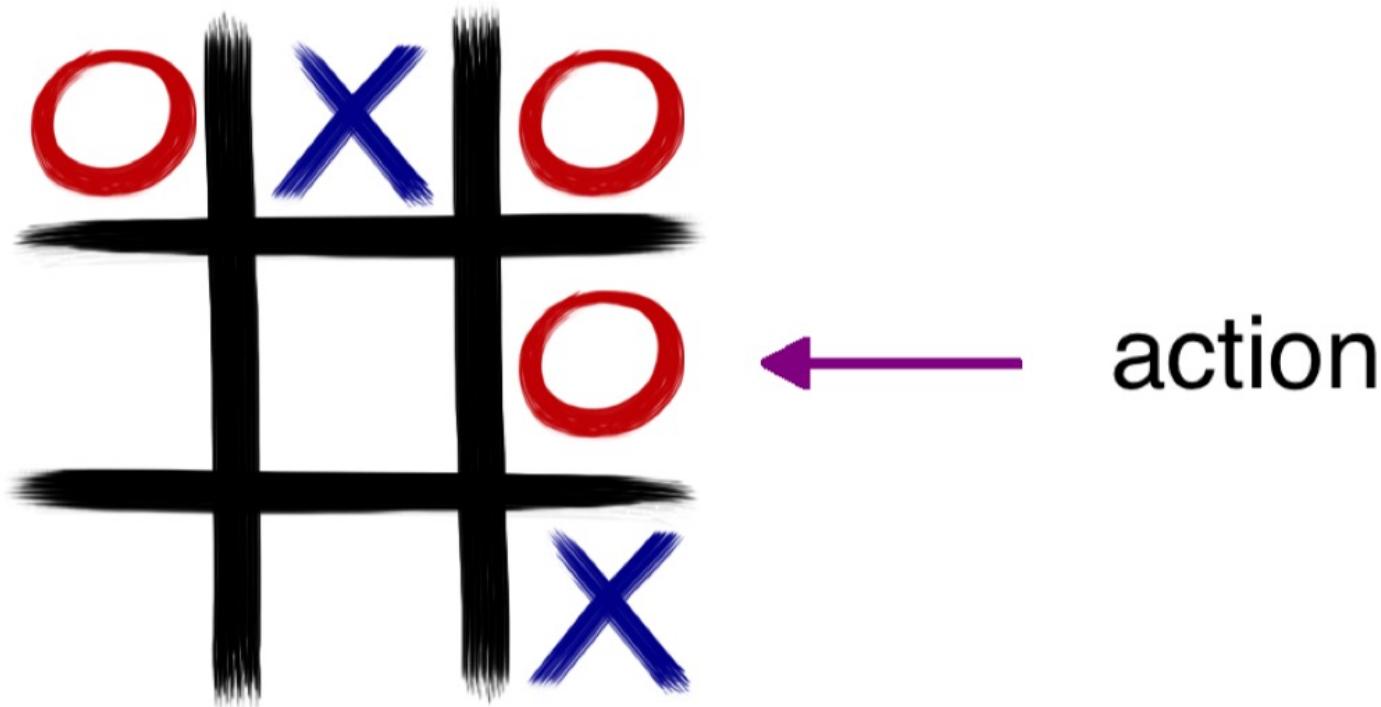
environment

Example: Tic Tac Toe, Notation

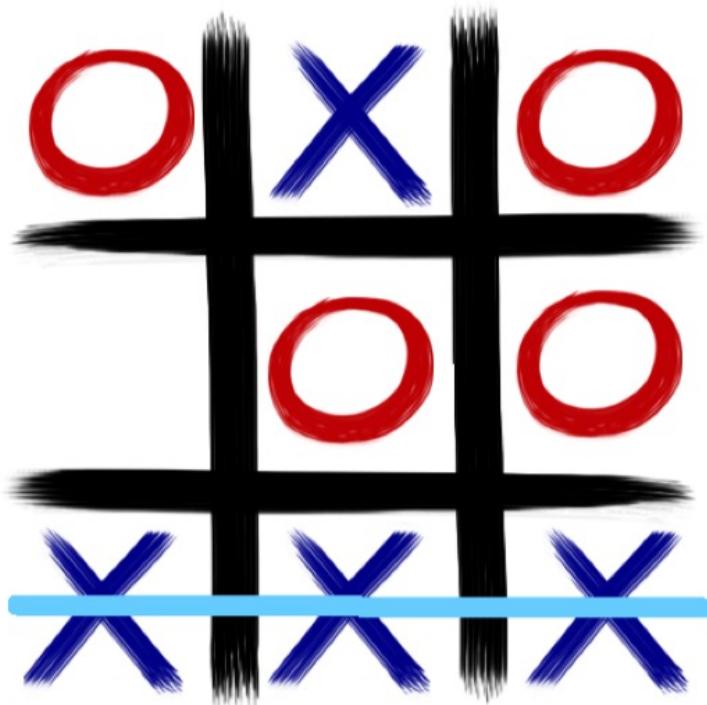


(current)
state

Example: Tic Tac Toe, Notation

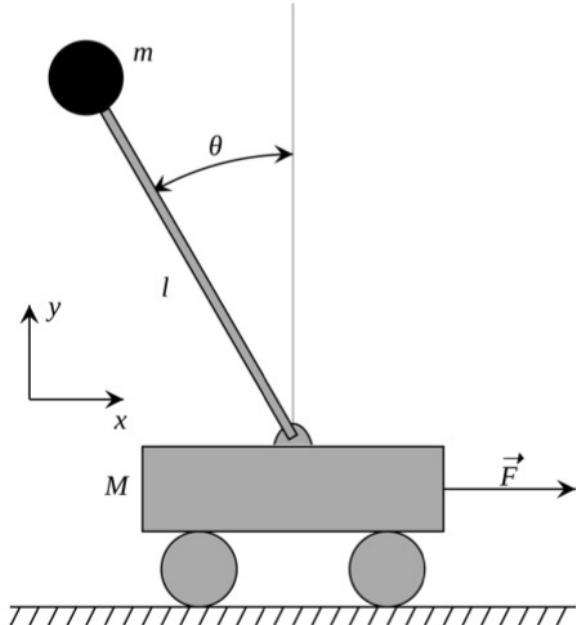


Example: Tic Tac Toe, Notation



reward
(here: -1)

Example: Cart-Pole Problem



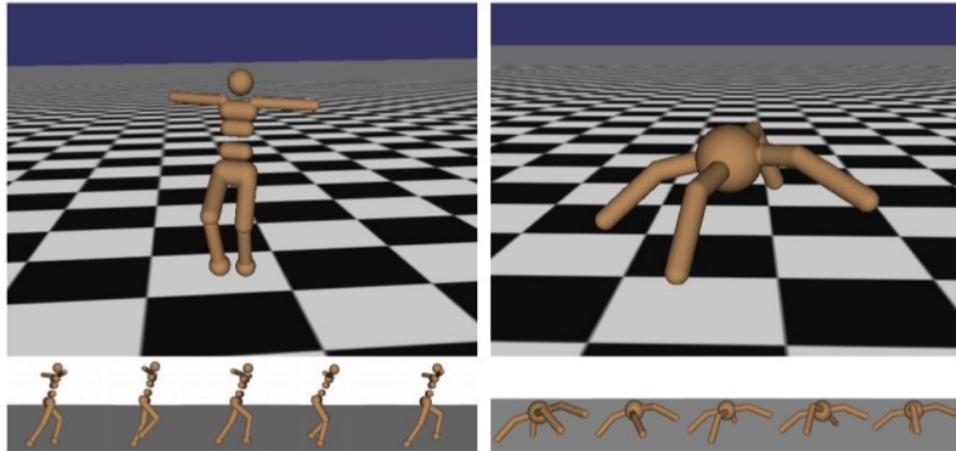
Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity

Action: horizontal force applied on the cart

Reward: 1 at each time step if the pole is upright

Example: Robot Locomotion



Objective: Make the robot move forward

State: Angle and position of the joints

Action: Torques applied on joints

Reward: 1 at each time step upright + forward movement

Example: Atari Games



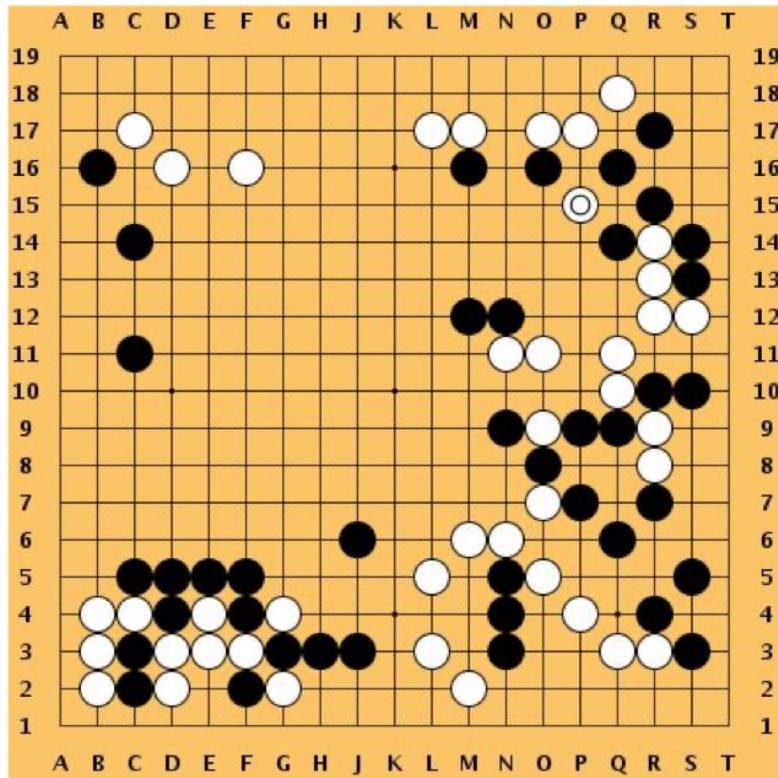
Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state

Action: Game controls e.g. Left, Right, Up, Down

Reward: Score increase/decrease at each time step

Example: Go



Objective: Win the game!

State: Position of all pieces

Action: Where to put the next piece down

Reward: 1 if win at the end of the game, 0 otherwise

Rewards

- ▶ A **reward** R_t is a scalar feedback signal
- ▶ Indicates how well agent is doing at step t — defines the goal
- ▶ The agent's job is to maximize cumulative reward

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots$$

- ▶ We call this the **return**

Reinforcement learning is based on the **reward hypothesis**:

Any goal can be formalized as the outcome of maximizing a cumulative reward

Values

- ▶ We call the expected cumulative reward, from a state s , the **value**

$$\begin{aligned}v(s) &= \mathbb{E}[G_t \mid S_t = s] \\&= \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \dots \mid S_t = s]\end{aligned}$$

- ▶ The value depends on the actions the agent takes
- ▶ Goal is to **maximize value**, by picking suitable actions
- ▶ Rewards and values define **utility** of states and action (no supervised feedback)
- ▶ Returns and values can be defined recursively

$$\begin{aligned}G_t &= R_{t+1} + G_{t+1} \\v(s) &= \mathbb{E}[R_{t+1} + v(S_{t+1}) \mid S_t = s]\end{aligned}$$

Maximizing Value by Taking Actions

- ▶ Goal: **select actions to maximise value**
- ▶ Actions may have long term consequences
- ▶ Reward may be delayed
- ▶ It may be better to sacrifice immediate reward to gain more long-term reward
- ▶ Examples:
 - ▶ Refueling a helicopter (might prevent a crash in several hours)
 - ▶ Defensive moves in a game (may help chances of winning later)
 - ▶ Learning a new skill (can be costly & time-consuming at first)
- ▶ A mapping from states to actions is called a **policy**

Action Values

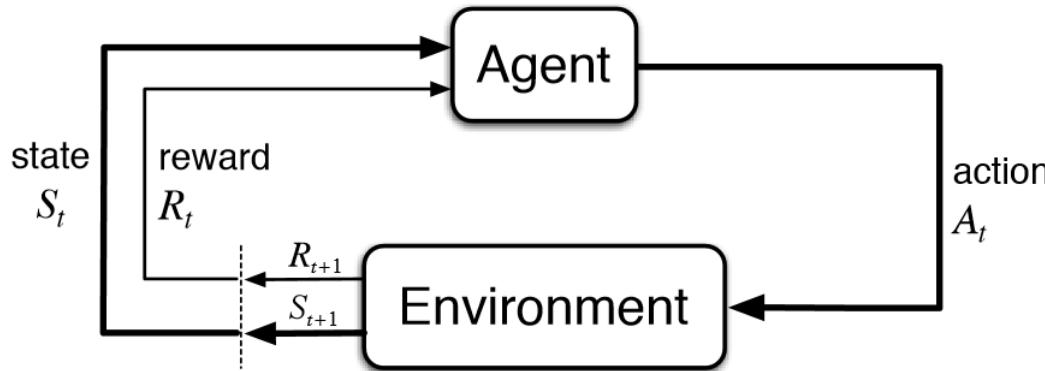
- ▶ It is also possible to condition the value on **actions**:

$$\begin{aligned} q(s, a) &= \mathbb{E}[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \dots \mid S_t = s, A_t = a] \end{aligned}$$

- ▶ We will talk in depth about state and action values later

Markov Decision Process

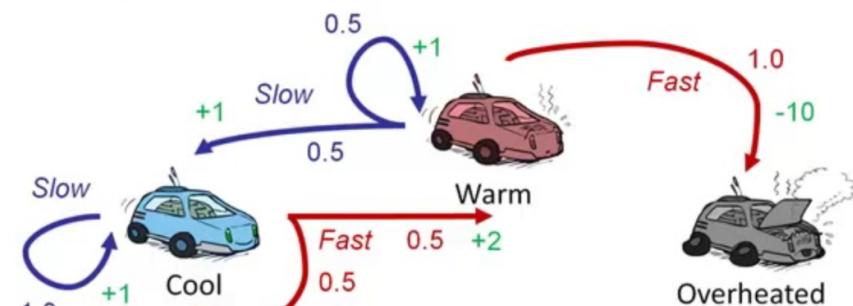
How can we mathematically formalize the RL problem?



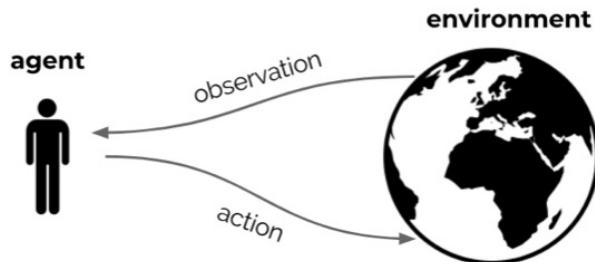
Markov Decision Process:

- Mathematical formulation of the RL problem
- **Markov property:** Current state completely characterises the state of the world : $p(r, s | S_t, A_t) = p(r, s | \mathcal{H}_t, A_t)$

- \mathcal{H}_t is the history.
- This means that the state contains all we need to know from the history.
- Doesn't mean it contains everything, just that adding more history doesn't help. \Rightarrow Once the state is known, the history may be thrown away.
- Typically, the agent state S_t is some compression of \mathcal{H}_t
- Note: we use S_t to denote the agent state, not the environment state



Fully Observable Environments



Full observability

Suppose the agent sees the full environment state

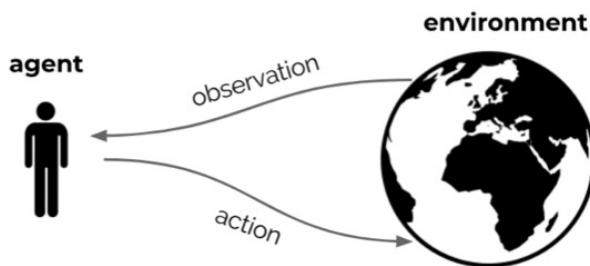
- ▶ observation = environment state
- ▶ The agent state could just be this observation:

$$S_t = O_t = \text{environment state}$$

Partially Observable Environments

- ▶ **Partial observability:** The observations are not Markovian
 - ▶ A robot with camera vision isn't told its absolute location
 - ▶ A poker playing agent only observes public cards
- ▶ Now using the observation as state would not be Markovian
- ▶ This is called a **partially observable Markov decision process** (POMDP)
- ▶ The **environment state** can still be Markov, but the agent does not know it
- ▶ We might still be able to construct a Markov agent state

Agent State



- ▶ The agent's actions depend on its state
- ▶ The **agent state** is a function of the history
- ▶ For instance, $S_t = O_t$
- ▶ More generally:

$$S_{t+1} = u(S_t, A_t, R_{t+1}, O_{t+1})$$

where u is a ‘state update function’

- ▶ The agent state is often **much** smaller than the environment state

Agent State

The full environment state of a maze



Agent State

A potential observation



Agent State

An observation in a different location



Agent State

The two observations are indistinguishable



Partially Observable Environments

- ▶ To deal with partial observability, agent can construct suitable state representations
- ▶ Examples of agent states:
 - ▶ Last observation: $S_t = O_t$ (might not be enough)
 - ▶ Complete history: $S_t = \mathcal{H}_t$ (might be too large)
 - ▶ A generic update: $S_t = u(S_{t-1}, A_{t-1}, R_t, O_t)$ (but how to pick/learn u ?)
- ▶ Constructing a fully Markovian agent state is often not feasible
- ▶ More importantly, the state should allow good policies and value predictions

Markov Decision Process

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

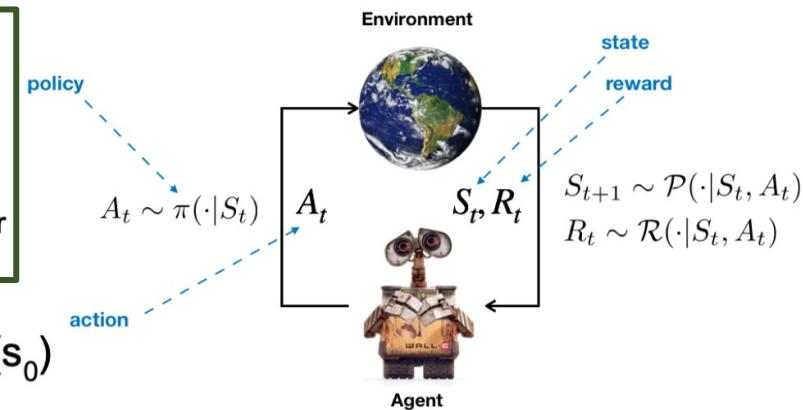
\mathcal{S} : set of possible states

\mathcal{A} : set of possible actions

\mathcal{R} : distribution of reward given (state, action) pair

\mathbb{P} : transition probability i.e. distribution over next state given (state, action) pair

γ : discount factor



- At time step $t=0$, environment samples initial state $s_0 \sim p(s_0)$
- Then, for $t=0$ until done:
 - Agent selects action a_t
 - Environment samples reward $r_t \sim R(\cdot | s_t, a_t)$
 - Environment samples next state $s_{t+1} \sim P(\cdot | s_t, a_t)$
 - Agent receives reward r_t and next state s_{t+1}
 - A policy π is a function (the agent's behavior) from S to A that specifies what action to take in each state.
 - Deterministic policy: $a = \pi(s)$
 - Stochastic policy: $\pi(a | s) = P[a_t=a | s_t=s]$
 - **Objective:** find policy π^* that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

γ is called a **discount rate**, and it is always $0 \leq \gamma \leq 1$

If γ close to 1, rewards further in the future count more, and we say that the agent is “farsighted”

γ is less than 1 because there is usually a time limit to the sequence of actions needed to solve a task (we prefer rewards sooner rather than later)

The optimal policy π^*

We want to find optimal policy π^* that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi \right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$

A simple MDP: Grid World

actions = {

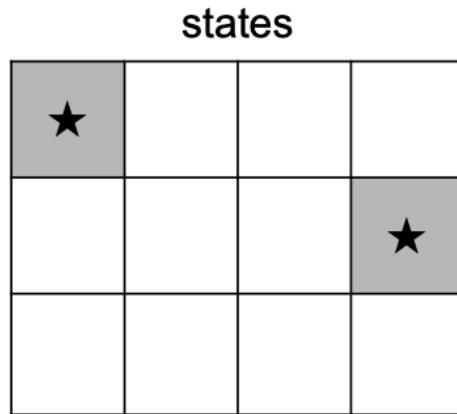
1. right →

2. left ←

3. up ↑

4. down ↓

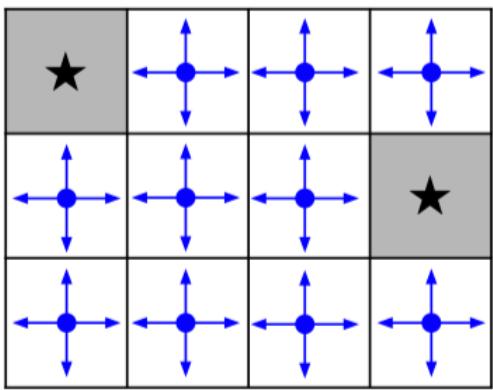
}



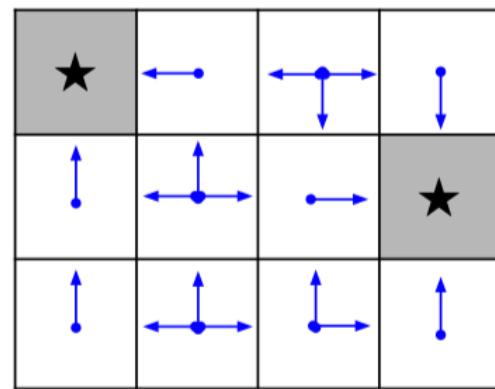
Set a negative “reward”
for each transition
(e.g. $r = -1$)

Objective: reach one of terminal states (greyed out) in
least number of actions

A simple MDP: Grid World



Random Policy



Optimal Policy

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

How good is a state?

The **value function** at state s , is the expected cumulative reward from following the policy from state s :

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right]$$

How good is a state-action pair?

The **Q-value function** at state s and action a , is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Bellman Equation

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Q^* satisfies the following **Bellman equation**:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Intuition: if the optimal state-action values for the next time-step $Q^*(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

The optimal policy π^* corresponds to taking the best action in any state as specified by Q^*

Solving for the optimal policy: Q-learning

Q-learning: Use a function approximator to estimate the action-value function

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

- Simplest function approximator: Table!

Example: Table for Q-learning

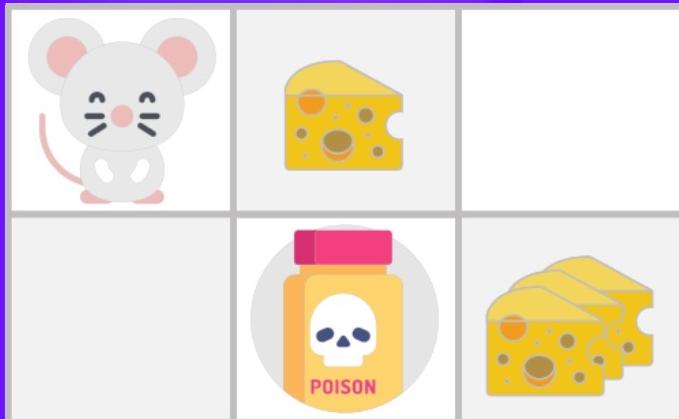


- You're a mouse in this tiny maze. You always **start at the same starting point**.
- The goal is **to eat the big pile of cheese at the bottom right-hand corner** and avoid the poison. After all, who doesn't like cheese?
- The episode ends if we eat the poison, **eat the big pile of cheese**, or if we take more than five steps.
- The learning rate is 0.1
- The discount rate (gamma) is 0.99

Example: Table for Q-learning

Example

- The reward function:
 - 0: Going to a state **with no cheese in it.**
 - **+1:** Going to a state with a **small cheese in it.**
 - **+10:** Going to the state with **the big pile of cheese.**
 - **-10:** Going to the state **with the poison and thus die.**



Example: Table for Q-learning

Example, Step 1

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\text{terminal-state}, \cdot) = 0$)

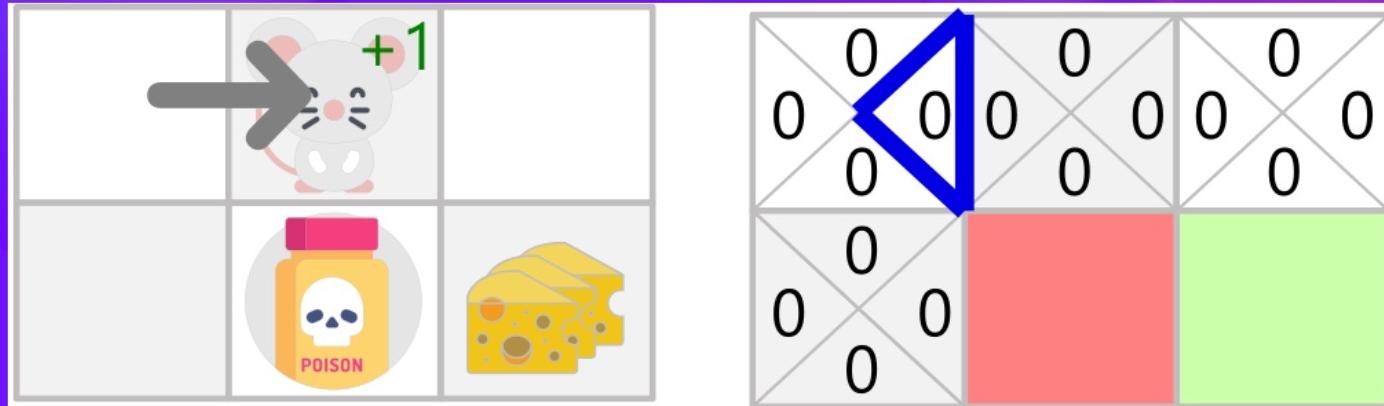
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

We initialize the Q-Table

Example: Table for Q-learning

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Example: Table for Q-learning

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Example: Table for Q-learning

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

New
Q-value
estimation

Former
Q-value
estimation

Learning
Rate

Immediate
Reward

Discounted Estimate
optimal Q-value
of next state

Former
Q-value
estimation

TD Target

TD Error

Update our Q-value estimation

Example: Table for Q-learning

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

$$Q(\text{Initial state, Right}) = 0 + 0.1 * [1 + 0.99 * 0 - 0]$$

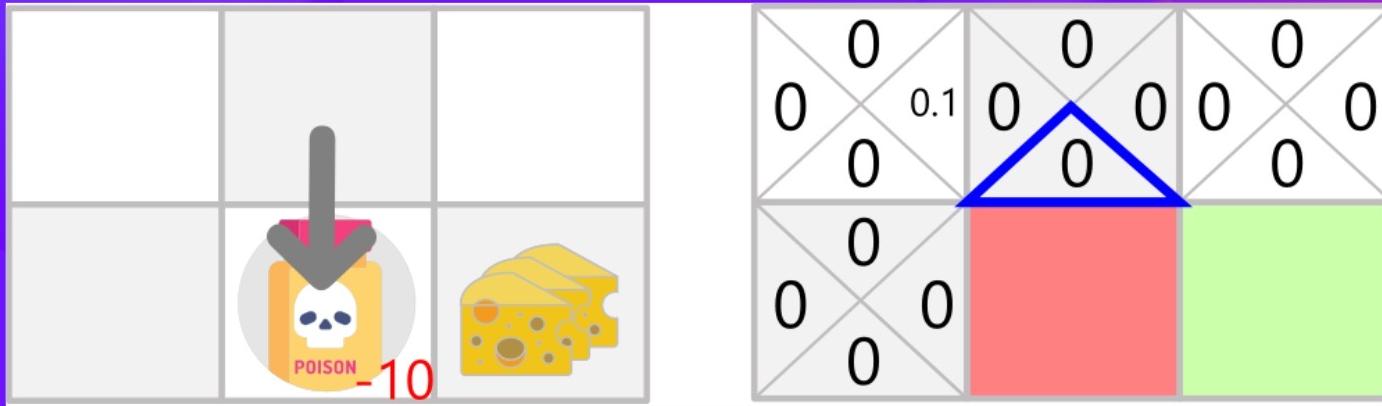
$$Q(\text{Initial state, Right}) = 0.1$$

	0	0.1	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Example: Table for Q-learning

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Example: Table for Q-learning

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Example: Table for Q-learning

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

$$Q(\text{State 2, Down}) = 0 + 0.1 * [-10 + 0.99 * 0 - 0]$$

$$Q(\text{State 2, Down}) = -1$$

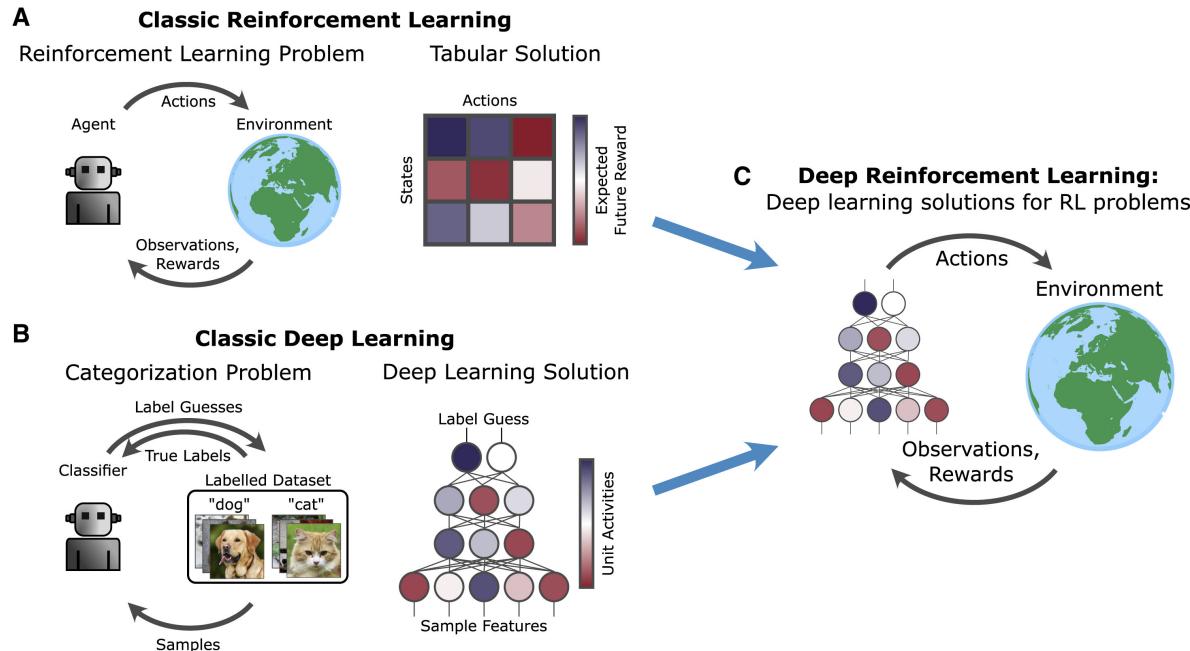
	0	0.1	0	0
	0	0	0	-1
	0	0	0	0
	0	0	0	0
	0	0	0	0

Deep Q-Learning

What's the problem with this?

Not scalable. Must compute $Q(s,a)$ for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate $Q(s,a)$. E.g. a neural network!



Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

If the function approximator is a deep neural network => deep q-learning!

Deep Q-Learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

Deep Q-Learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)

Backward Pass

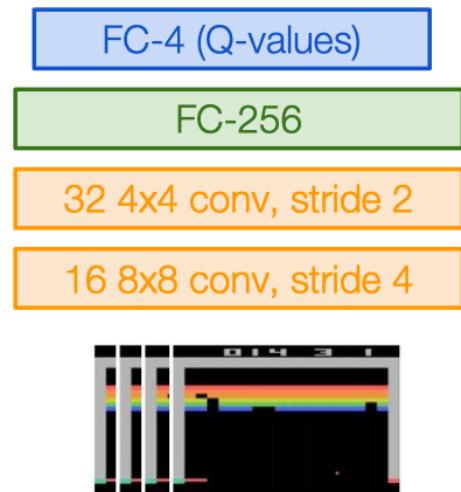
Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Deep Q-Learning

$Q(s, a; \theta)$:
neural network
with weights θ

A single feedforward pass
to compute Q-values for all
actions from the current
state => efficient!



Last FC layer has 4-d output (if 4 actions),
corresponding to $Q(s_t, a_1), Q(s_t, a_2), Q(s_t, a_3), Q(s_t, a_4)$

Number of actions between 4-18
depending on Atari game

Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Deep Q-Learning: Experience Replay

Learning from batches of consecutive samples is problematic:

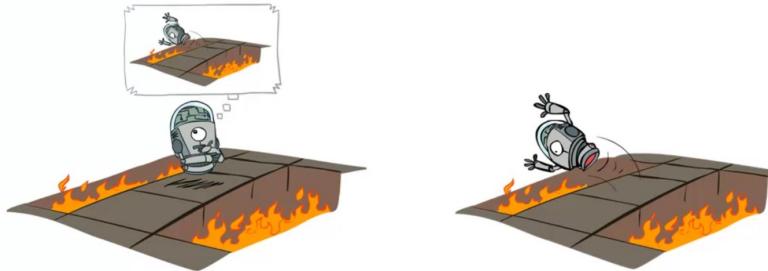
- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Exploration vs. Exploitation

Learning: We don't know which states are good or what the actions do. We must try out the actions and states to learn what to do



- If we knew how the world works (embodied in P), then the policy should be deterministic
 - ▶ just select optimal action in each state
- Reinforcement learning is like trial-and-error learning
- The agent should discover a good policy from its experiences of the environment
- Without losing too much reward along the way
- Since we do not have complete knowledge of the world, taking what appears to be the optimal action may prevent us from finding better states/actions
- Interesting trade-off:
 - ▶ immediate reward (**exploitation**) vs. gaining knowledge that might enable higher future reward (**exploration**)

Examples

- Restaurant Selection
 - ▶ **Exploitation:** Go to your favourite restaurant
 - ▶ **Exploration:** Try a new restaurant
- Online Banner Advertisements
 - ▶ **Exploitation:** Show the most successful advert
 - ▶ **Exploration:** Show a different advert
- Oil Drilling
 - ▶ **Exploitation:** Drill at the best known location
 - ▶ **Exploration:** Drill at a new location
- Game Playing
 - ▶ **Exploitation:** Play the move you believe is best
 - ▶ **Exploration:** Play an experimental move

Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)
- Can choose actions to maximize $\hat{Q}(s, a)$
- Good idea?
- Can instead employ stochastic action selection (policy):

$$P(a_i|s) = \frac{\exp(k\hat{Q}(s, a_i))}{\sum_j \exp(k\hat{Q}(s, a_j))}$$

- Can vary k during learning
 - ▶ more exploration early on, shift towards exploitation

Or a simpler way (**ϵ -greedy**): Exploits the best-known action with probability $(1-\epsilon)$ and explores a random action with probability ϵ , balancing between exploration and exploitation.

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N  **Initialize replay memory, Q-network**
Initialize action-value function Q with random weights
for episode = 1, M **do**
 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
 for $t = 1, T$ **do**
 With probability ϵ select a random action a_t
 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
 end for
end for

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do** ← Play M episodes (full games)
 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
 for $t = 1, T$ **do**
 With probability ϵ select a random action a_t
 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
 end for
end for

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t
 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



Initialize state
(starting game
screen pixels) at the
beginning of each
episode

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do**
 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
 for $t = 1, T$ **do**
 With probability ϵ select a random action a_t
 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
 end for
end for



For each timestep t of the game

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do**
 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
 for $t = 1, T$ **do**
 With probability ϵ select a random action a_t
 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
 end for
end for

← With small probability,
select a random
action (explore),
otherwise select
greedy action from
current policy

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



Take the action (a_t) ,
and observe the
reward r_t and next
state s_{t+1}

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



Store transition in
replay memory

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

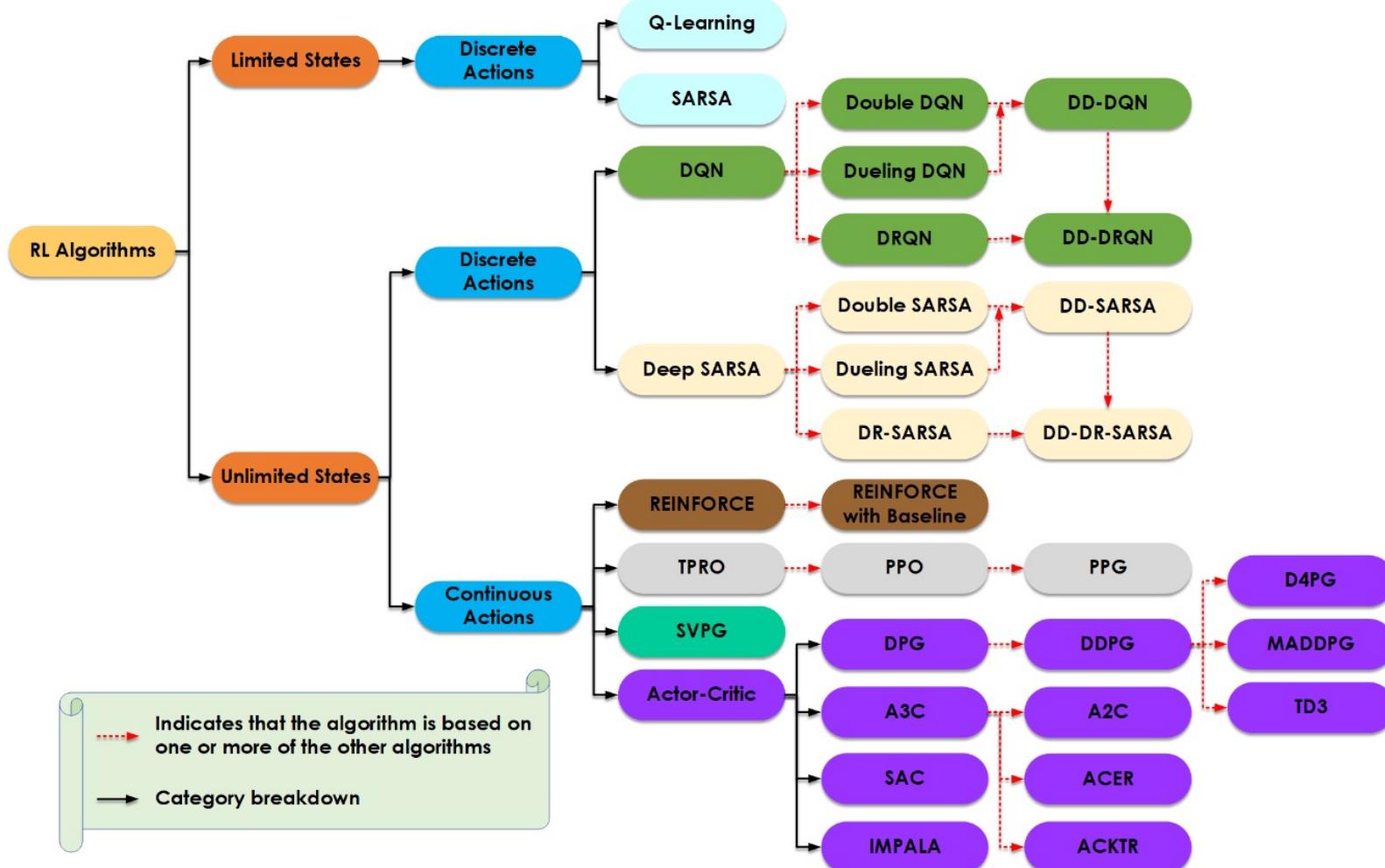
end for

end for

Experience Replay:
Sample a random
minibatch of transitions
from replay memory
and perform a gradient
descent step



Reinforcement Learning Algorithms



Questions?