Deep Reinforcement Learning

CS440/ECE448 Lecture 35

Mark Hasegawa-Johnson, 4/2022
CC-BY 4.0: you may remix or redistribute if you cite the source

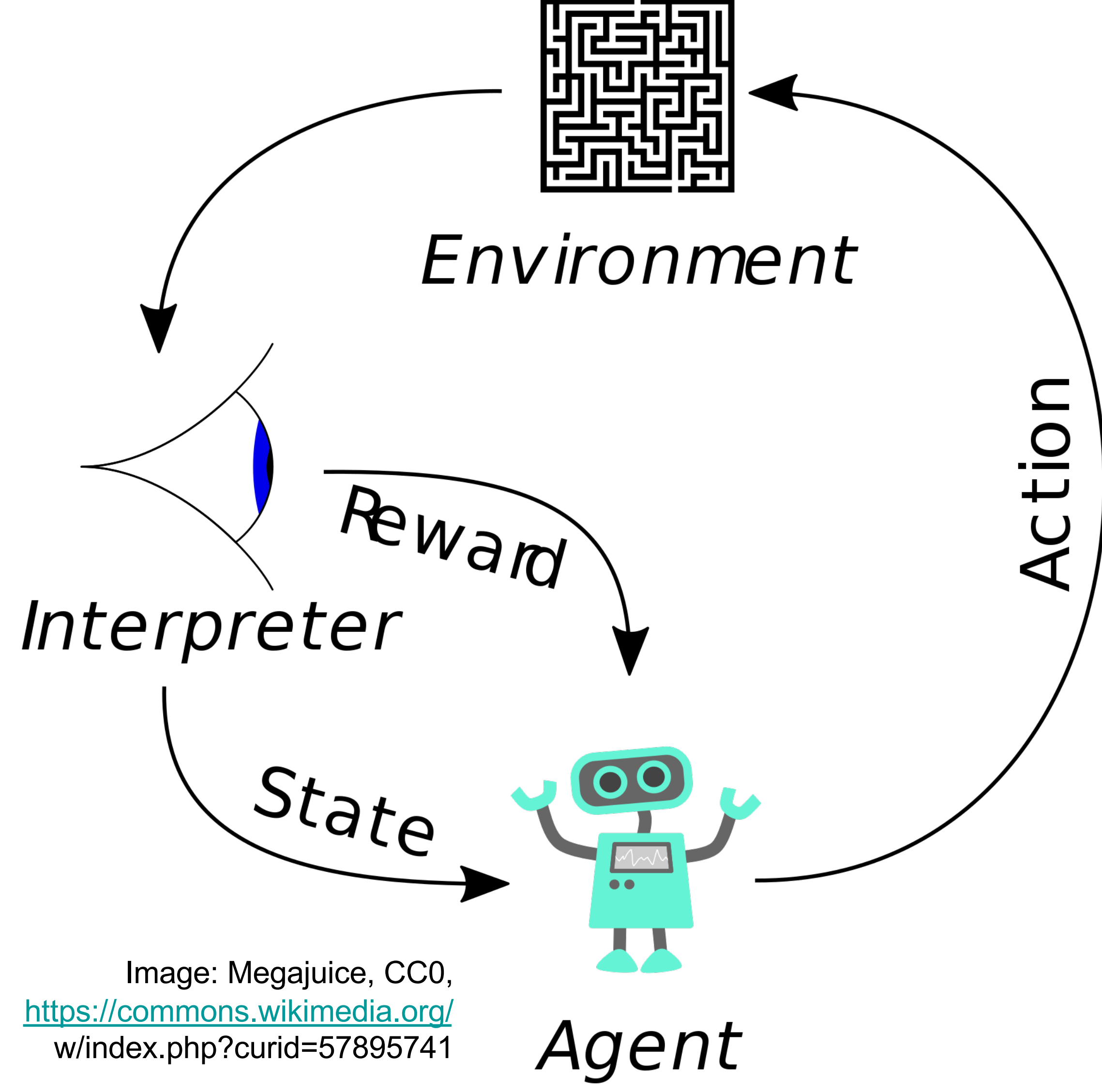
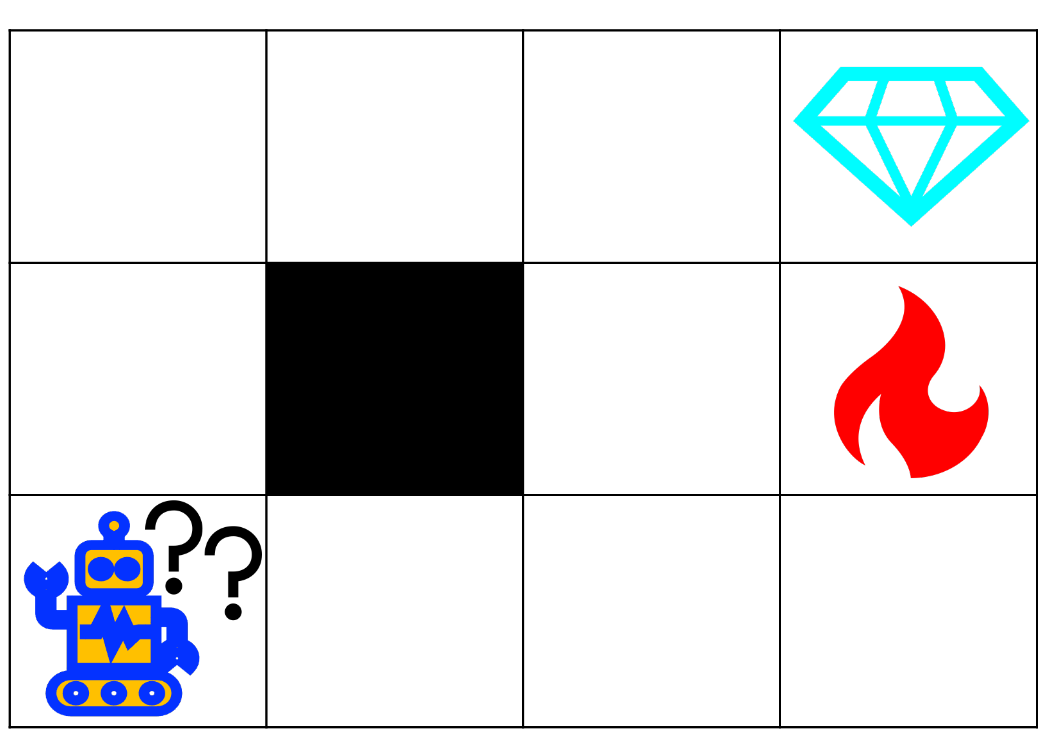


Image: Megajuice, CC0, https://commons.wikimedia.org/w/index.php?curid=57895741

Outline

- Review: MDP and Q-Learning
- Deep Q-Learning
- Imitation Learning
- Actor-Critic Learning

Review: Markov Decision Process



A Markov decision process (MDP) is defined by the rewards earned in each state:

$$R(s) = \begin{cases} +1 & s = (4,3) \\ -1 & s = (4,2) \\ -0.04 & \text{otherwise} \end{cases}$$

…and the transition probability function, which tells the probability of reaching state s' if you take action a in state s:

$$P(s'|s,a) = \begin{cases} 0.8 & \text{intended} \\ 0.1 & \text{fall left} \\ 0.1 & \text{fall right} \end{cases}$$

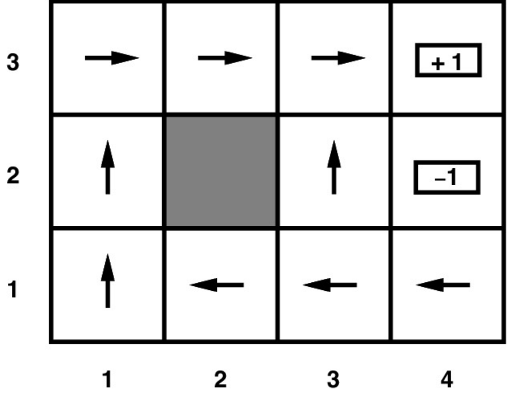
…and the discount factor, $0 \le \gamma \le 1$, which tells you what fraction of a reward today is worth one reward tomorrow.

Review: Markov Decision Process

An MDP is "solved" by finding:

- $U(s)$, the "utility," which is defined to be the expected sum of all future rewards obtained by starting in state s and proceeding according to the best possible policy, and
- $\pi(s)$, the "policy", defined as the best action to take in each state.

0.81	0.87	0.92	
0.76		0.66	
0.71	0.66	0.61	0.39



Review: Markov Decision Process

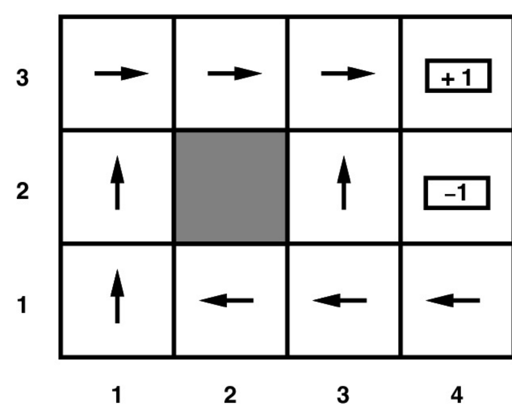
The solution to an MDP is found by solving Bellman's equation:

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

Bellman's equation gives the exact solution. The only reasons we might choose not to use it are:

- It is N nonlinear equations in N unknowns, so exact solution may be NP-complete (exhaustive search).
- In many real-world scenarios, we don't know $P(s'|s,a)$, so we can't use Bellman's equation anyway.

0.81	0.87	0.92	
0.76		0.66	
0.71	0.66	0.61	0.39



Review: Q-Learning

Suppose we don't know $P(s'|s,a)$, and we don't care to learn it. We can avoid learning it by instead learning $Q(s,a)$, defined as the maximum expected sum of all future rewards if we start with action a in state s. In other words, divide Bellman's equation:

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

…into two parts:

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) U(s')$$

$$U(s) = \max_{a \in A(s)} Q(s,a)$$

Review: Q-Learning

Rewrite Bellman's equation in terms of only Q:

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

Here are the problems we need to solve.

- $R(s)$ is unknown $\Longrightarrow$ use $R_t(s)$, the reward at the tth time step.
- $P(s'|s,a)$ is unknown $\Longrightarrow$ perform action a_t in state s_t, observe the resulting state s_{t+1}, and pretend that $P(s_{t+1}|s_t, a_t) = 1$.
- $Q(s',a')$ is unknown $\Longrightarrow$ use $Q_t(s',a')$, our current estimate.

Result:

$$Q_{local}(s_t, a_t) = R_t(s_t) + \gamma \max_{a\prime \in A(s_{t+1})} Q_t(s_{t+1}, a')$$

Review: Q-Learning

$$Q_{local}(s_t, a_t) = R_t(s_t) + \gamma \max_{a' \in A(s_{t+1})} Q_t(s_{t+1}, a')$$

Q_{local} has a high variance because of all the approximations involved. The variance can be reduced by averaging it over time using some averaging constant $0 < \alpha < 1$:

$$Q_{t+1}(s_t, a_t) = (1 - \alpha) Q_t(s_t, a_t) + \alpha Q_{local}(s_t, a_t)$$

$$= Q_t(s_t, a_t) + \alpha \big(Q_{local}(s_t, a_t) - Q_t(s_t, a_t) \big)$$

Summary: Q-Learning

- Q(s,a) – the “quality” of an action

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

- Q-learning
- Off-policy learning: TD

$$Q_{local}(s_t, a_t) = R_t(s_t) + \gamma \max_{a' \in A(s_{t+1})} Q_t(s_{t+1}, a')$$
$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha\left(Q_{local}(s_t, a_t) - Q_t(s_t, a_t)\right)$$

- On-policy learning: SARSA

$$a_{t+1} = \pi_t(s_{t+1})$$
$$Q_{local}(s_t, a_t) = R_t(s_t) + \gamma Q_t(s_{t+1}, a_{t+1})$$

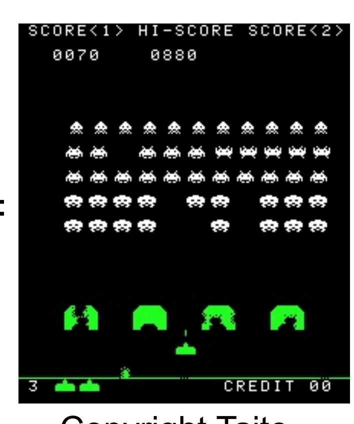
Outline

- Review: MDP and Q-Learning
- Deep Q-Learning
- Imitation Learning
- Actor-Critic Learning

Deep Q learning

Instead of discrete s, suppose $\vec{s}$ is a vector of real numbers, e.g., the image from the robot's eye camera:

$$\vec{s} = [s_1, \ldots, s_D] =$$

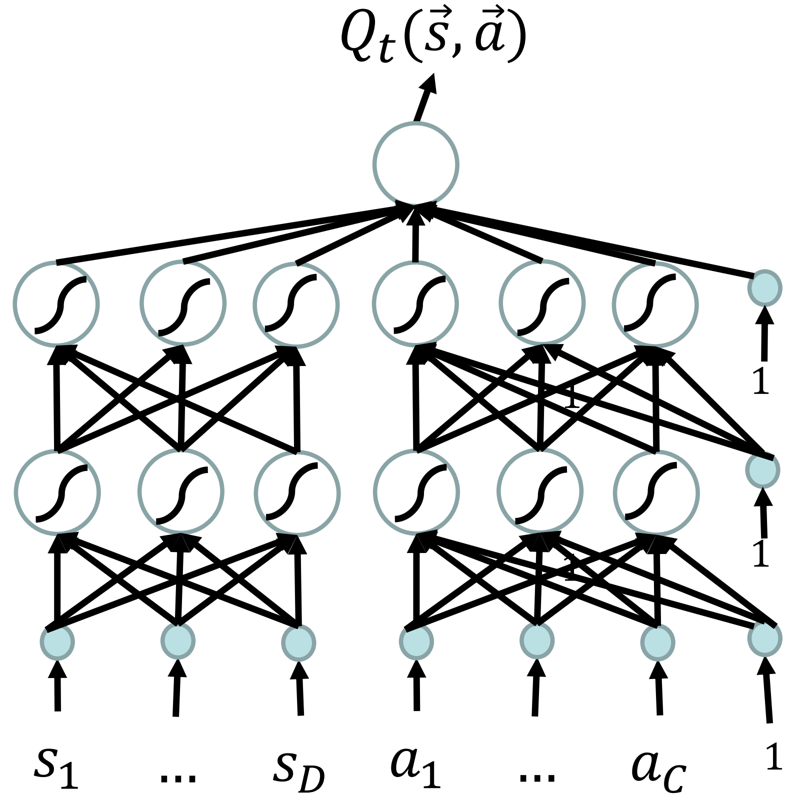


Copyright Taito.

Instead of discrete a, suppose $\vec{a}$ is a vector, e.g., cannon angle and velocity,

$$\vec{a} = [a_1, \ldots, a_C]$$

Deep Q-learning uses a neural network to compute an estimate $Q_t(\vec{s}, \vec{a})$ which is as close as possible to $Q(\vec{s}, \vec{a})$.



MMSE Deep Q learning

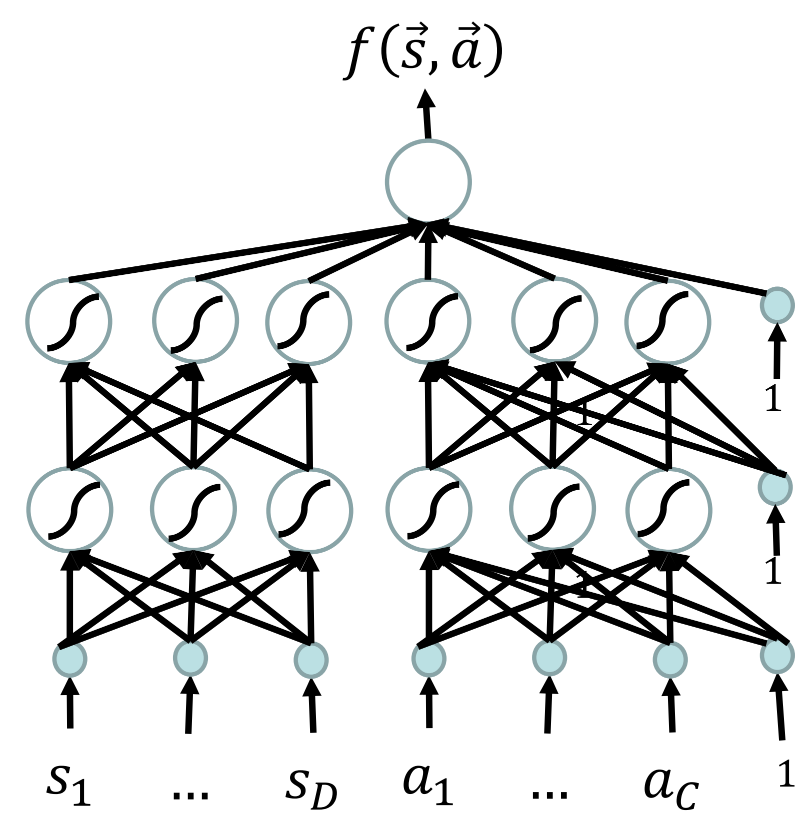
Suppose we train the neural network weights in order to minimize the mean-squared error (MMSE):

$$\mathcal{L} = \frac{1}{2} E[(Q_t(\vec{s}, \vec{a}) - Q(\vec{s}, \vec{a})\ )^2]$$

(where I'm using $E[\cdot]$ as a lazy way to write "average over all training runs of the game").

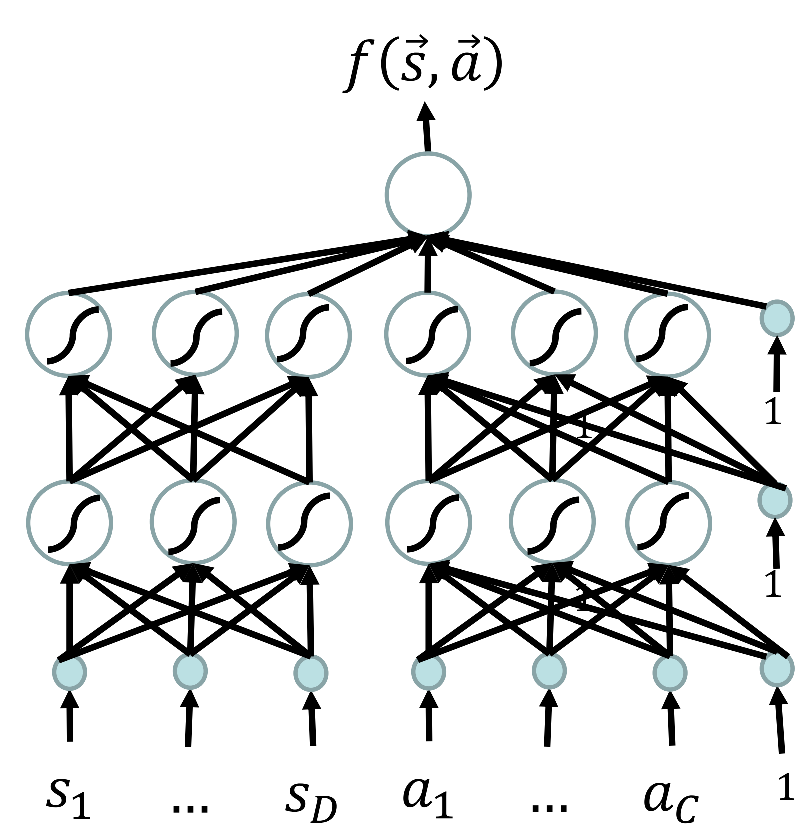
Then, for each weight w, we update as

$$w \leftarrow w - \eta \frac{d\mathcal{L}}{dw}$$



What makes deep Q learning harder than normal neural network training

- We don't know the true value of $Q(\vec{s}, \vec{a})$ for ***<u>any</u>*** of the training runs!
- $Q(\vec{s}, \vec{a})$ is defined to be the expected value of performing action $\vec{a}$. We never know its true expected value: all we know is whether we won or lost that particular game.
- So we can't compute $\mathcal{L}$, and we can't compute $\frac{d\mathcal{L}}{dw}$, and we can't update w!



The solution: Q_{local}

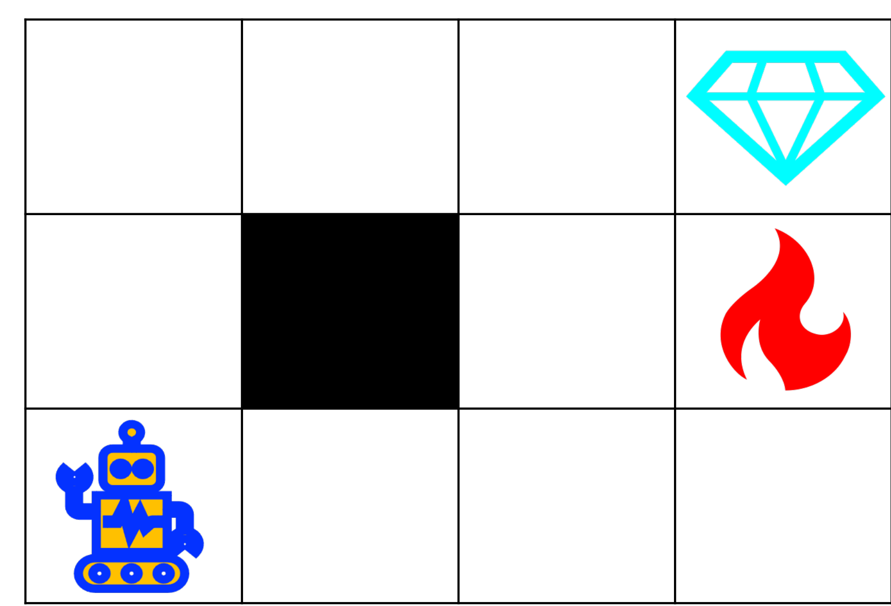
Remember that Q learning was defined as

$$Q_{t+1}(s_t, a_t) \\ = Q_t(s_t, a_t) + \alpha\big(Q_{local}(s_t, a_t) - Q_t(s_t, a_t)\big)$$

where $Q_{local}(s_t, a_t)$ is defined, e.g., in TD as

$$Q_{local}(s_t, a_t) = R_t(s_t) + \gamma \max_{a'} Q_t(s_{t+1}, a')$$

…for s_{t+1} equal to the next state we reach after action a_t on **this particular game**.



The solution: Q_{local}

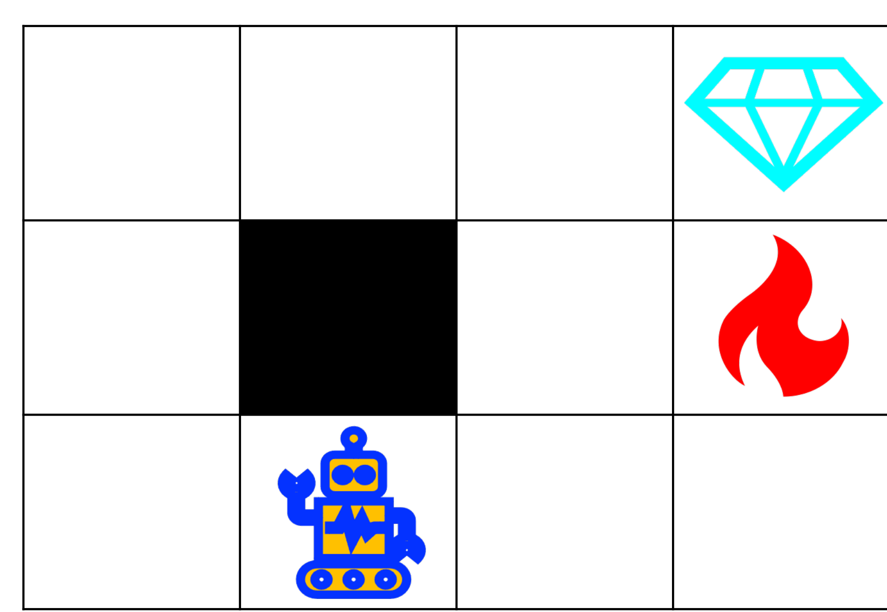
Let's define deep Q learning using the same Q_{local}:

$$\mathcal{L} = \frac{1}{2} E[(Q_t(\vec{s}_t, \vec{a}_t) - Q_{local}(\vec{s}_t, \vec{a}_t)\ )^2]$$

where $Q_{local}(\vec{s}_t, \vec{a}_t)$ is:

$$Q_{local}(\vec{s}_t, \vec{a}_t) = R_t(\vec{s}_t) + \gamma \max_{\vec{a}'} Q_t(\vec{s}_{t+1}, \vec{a}')$$

Now we have an L that depends only on things we know ($Q_t(\vec{s}_t, \vec{a}_t)$, $R_t(\vec{s}_t)$, and $Q_t(\vec{s}_{t+1}, \vec{a}')$), so it can be calculated, differentiated, and used to update the neural network.



Dealing with training instability

- Challenges
  - Target values are not fixed
  - Successive experiences are correlated and dependent on the policy
  - Policy may change rapidly with slight changes to parameters, leading to drastic change in data distribution
- Solutions
  - Freeze target Q network
  - Use *experience replay*

Experience replay

- At each time step:
  - Take action $\vec{a}_t$ according to epsilon-greedy policy
  - Store experience $(\vec{s}_t, \vec{a}_t, R_t, \vec{s}_{t+1})$ in *replay memory buffer*

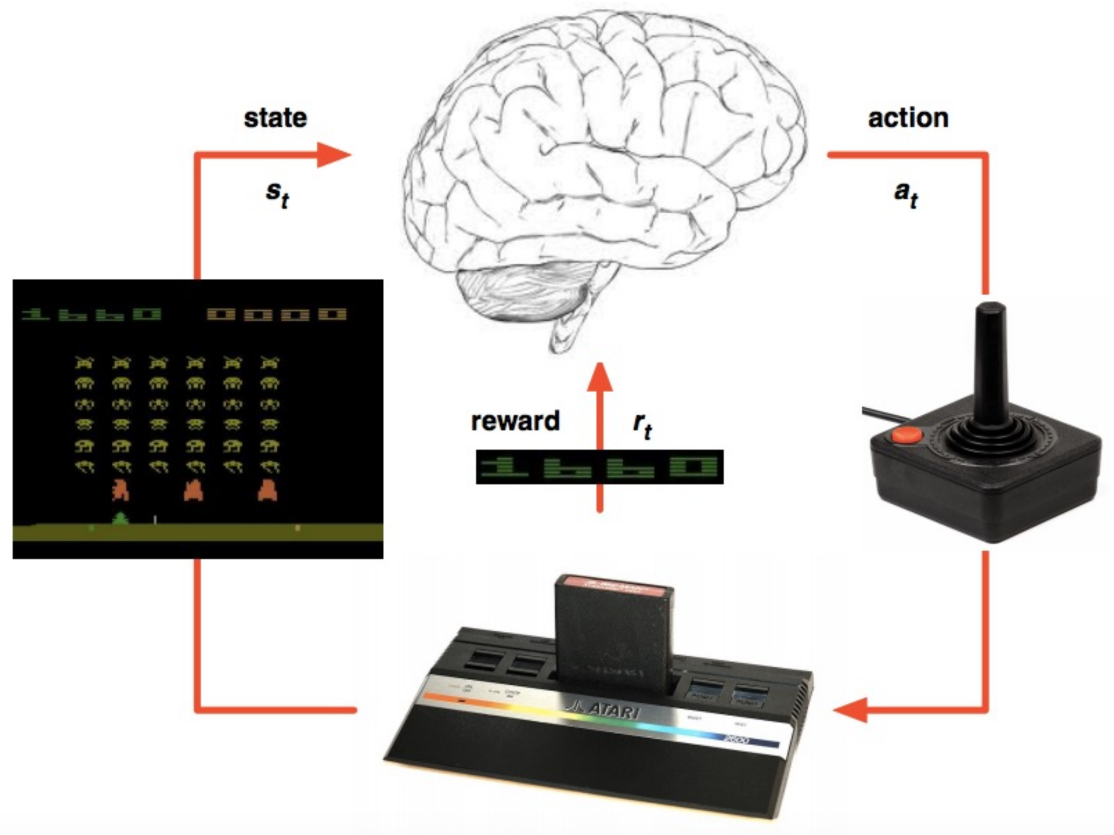
$(\vec{s}_1, \vec{a}_1, R_1, \vec{s}_2)$
$(\vec{s}_2, \vec{a}_2, R_2, \vec{s}_3)$
...

$(\vec{s}_t, \vec{a}_t, R_t, \vec{s}_{t+1})$

- Learning:
  - Randomly sample a minibatch, $\mathcal{D}$, from the replay buffer.

$\mathcal{D}$ =randomly sampled set of tuples

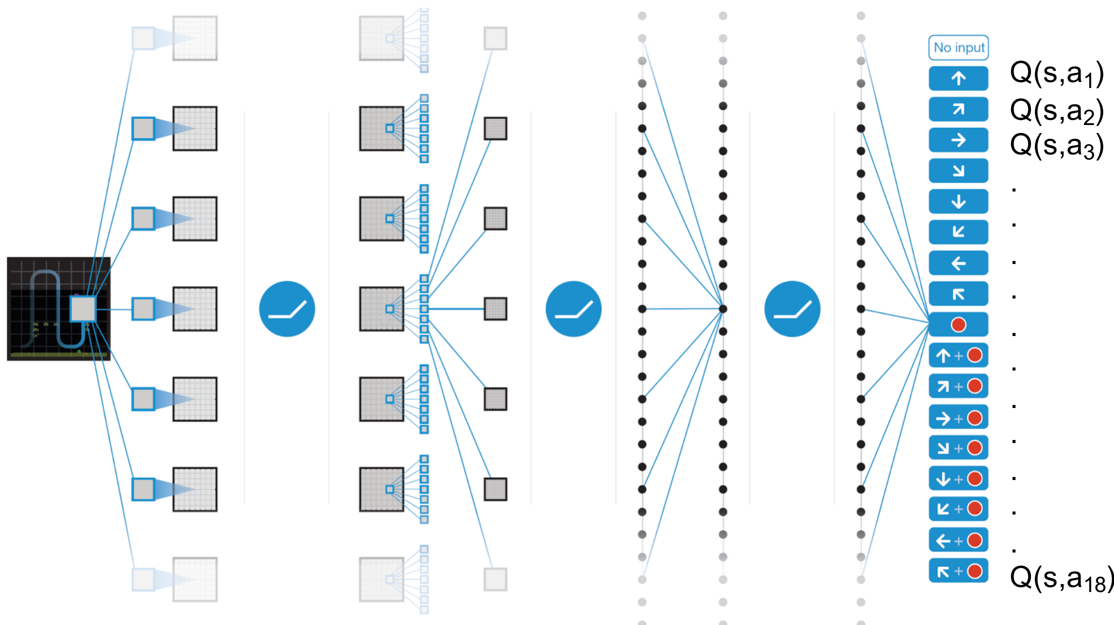
Deep Q learning in Atari



Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Deep Q learning in Atari

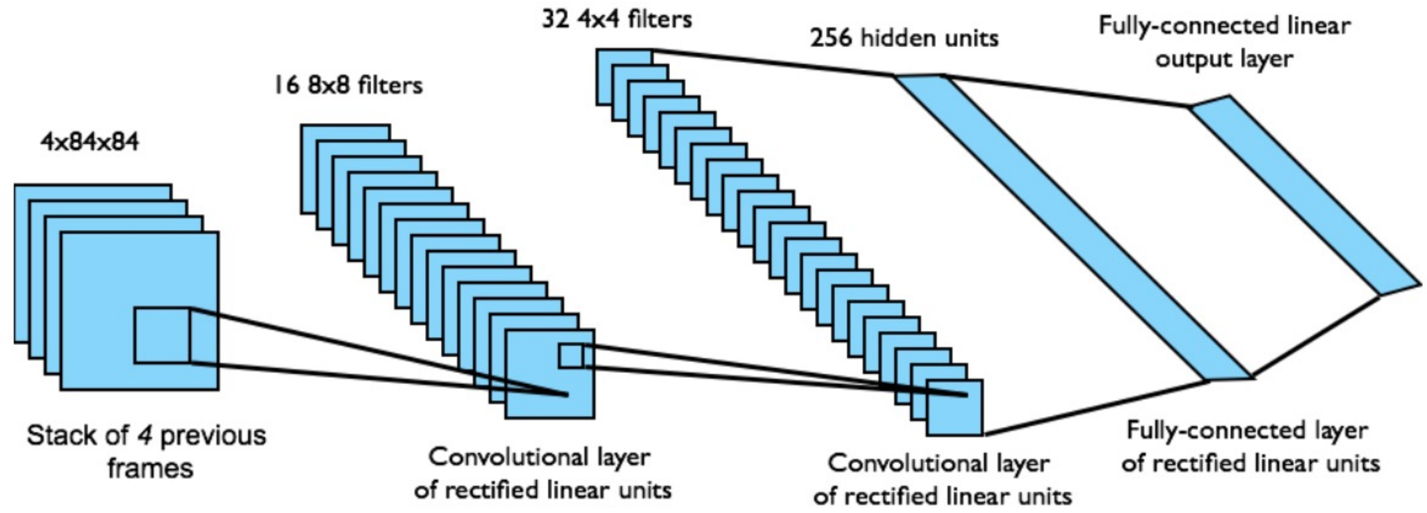
- End-to-end learning of $Q_t(\vec{s}, a)$ from pixels $\vec{s}$
- Output is $Q_t(\vec{s}, a)$ for $a \in 18$ joystick/button configurations
- Reward is change in score for that step



Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Deep Q learning in Atari

- Input state $\vec{s}$ is stack of raw pixels from last 4 frames
- Network architecture and hyperparameters fixed for all games



Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Deep Q learning in Atari

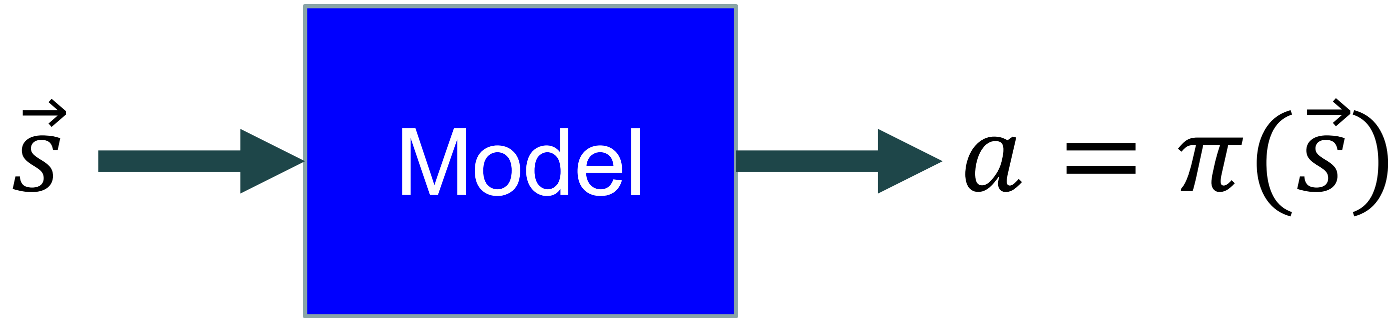
Deep Q-Learning Playing Atari Breakout

Outline

- Review: MDP and Q-Learning
- Deep Q-Learning
- Imitation Learning
- Actor-Critic Learning

Policy Learning

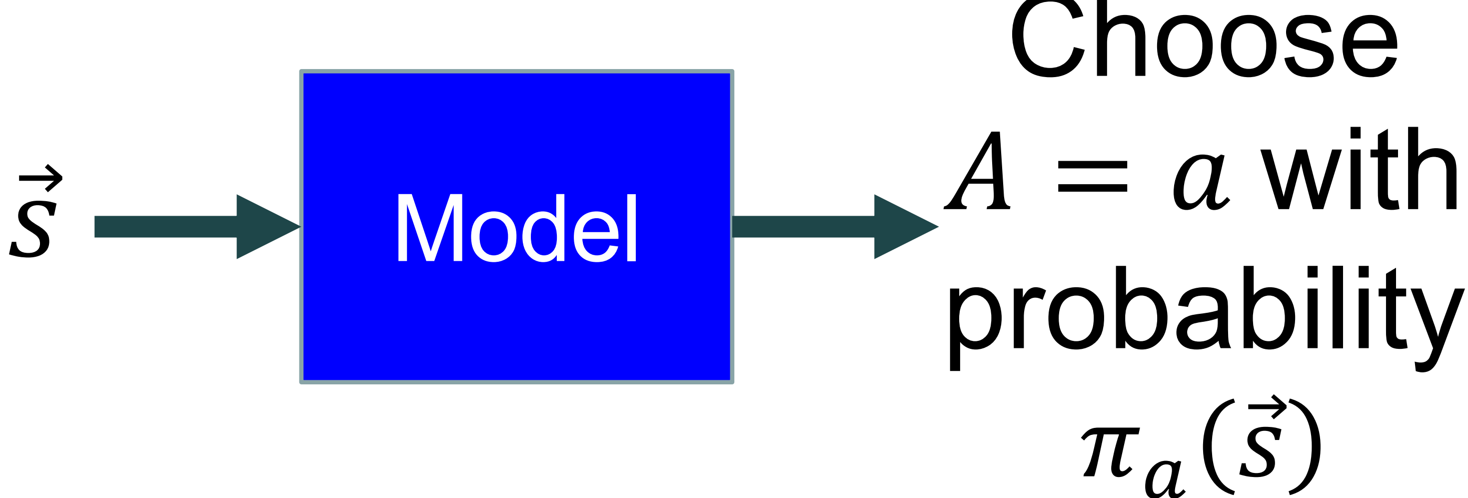
Why can't we just learn a model (neural net, or even a table lookup) that does this:



Policy Learning

Actually, let's redefine $\vec{\pi}(s)$ to be a probability vector:

$$\pi_a(\vec{s}) = \frac{\exp\left(\vec{w}_a^T \vec{h}(\vec{s})\right)}{\sum_{k=1}^{|A|} \exp\left(\vec{w}_k^T \vec{h}(\vec{s})\right)} = P(A = a | S = \vec{s})$$



Probabilistic Policy

If we have $|A|$ possible, actions, $1 \leq a \leq |A|$, we could train the network to learn a hidden layer $\vec{h}(\vec{s})$ so that:

$$\pi_a(\vec{s}) = \frac{\exp\left(\vec{w}_a^T \vec{h}(\vec{s})\right)}{\sum_{k=1}^{|A|} \exp\left(\vec{w}_k^T \vec{h}(\vec{s})\right)} = P(A = a | S = \vec{s})$$

Meaning “the probability that the best action is a.”

How do we train it?

- Training data only give us (s_i, a_i, R_i, s_{i+1}).
- BAD IDEA: train the network to choose $\mathrm{A} = a_i$ that maximizes the immediate reward, R_i, and just ignore future rewards.
- GOOD IDEA: Train the network to maximize $R_i + \gamma U(s_{i+1})$ = sum of all future rewards.
- PROBLEM: we don't know $U(s_{i+1})$.

$$(s_1, a_1, R_1, s_2)$$
$$(s_2, a_2, R_2, s_3)$$
$$(s_3, a_3, R_3, s_4)$$
$$(s_4, a_4, R_4, s_5)$$
$$(s_5, a_5, R_5, s_6)$$
$$\vdots$$

How to make Policy Learning trainable

1. Watch a human perform the task,
2. assume that each of the human's actions is the best action that could possibly have been taken at that time,
3. train the neural net to imitate the human's actions with high probability.

This is called "apprenticeship learning," which is a type of "imitation learning."

Imitation learning

- In some applications, you cannot bootstrap yourself from random policies
  - High-dimensional state and action spaces where most random trajectories fail miserably
  - Expensive to evaluate policies in the physical world, especially in cases of failure
- **Solution:** learn to imitate sample trajectories or demonstrations
  - This is also helpful when there is no natural reward formulation

Imitation learning

- $\vec{s}_t$ = a representation of the state of the environment at time t (can be a real-valued vector)
- a_t= the action that a human actor performed in response to this state (must be discrete)
- $\pi_k(\vec{s}_t)$ = k^{th} element in the softmax output of a neural network, given $\vec{s}_t$ as the input
- Training criterion: train the neural network in order to minimize the cross-entropy:

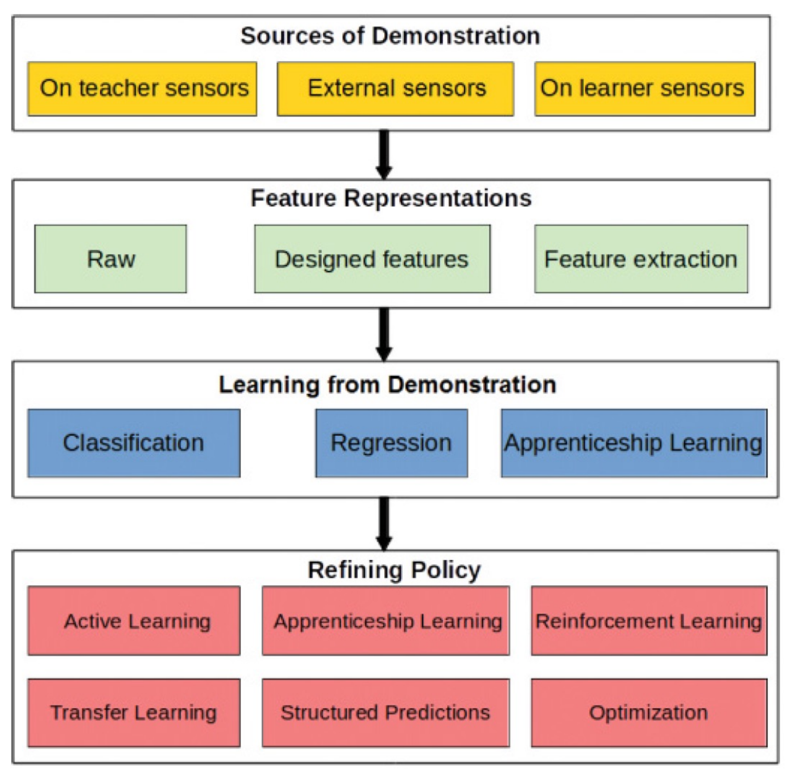
$$\mathcal{L} = -\log \pi_{a_t}(\vec{s}_t)$$

How do we train it?

- Now our training data don't even require R_i or s_{i+1}!
- All we need is a set of state vectors, s_i, and the action that the human performed in the same situation, a_i.

$$(s_1, a_1)$$
$$(s_2, a_2)$$
$$(s_3, a_3)$$
$$(s_4, a_4)$$
$$(s_5, a_5)$$
$$\vdots$$

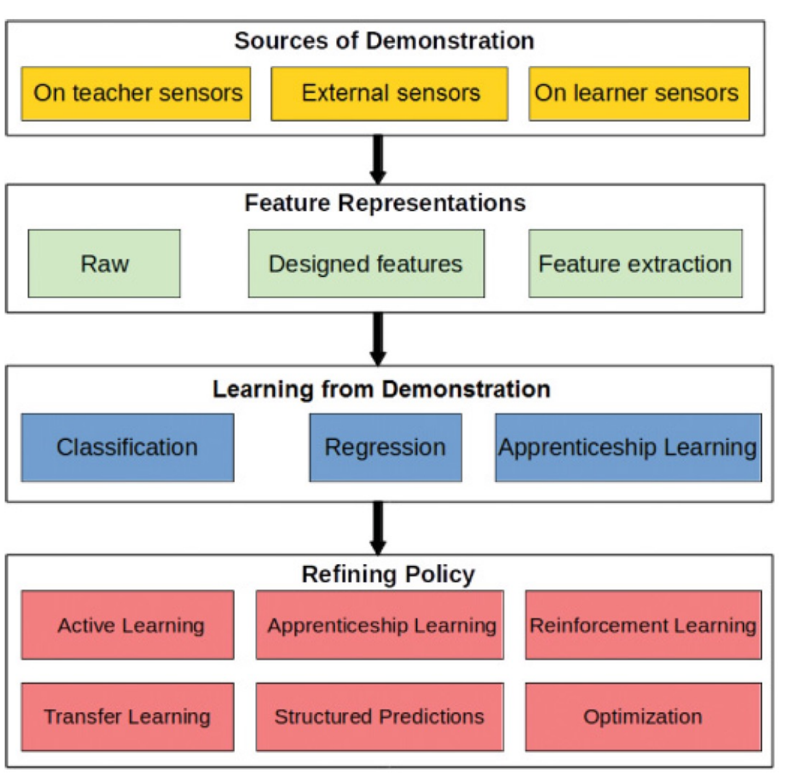
Overview of imitation learning methods



Apprenticeship learning (copy the human's action) needs a lot of human supervision. Sometimes we can benefit by combining apprenticeship learning with other methods (active learning, transfer learning, structured prediction, reinforcement learning, optimization).

Hussein et al. Imitation Learning: A Survey of Learning Methods, 2018.

Overview of imitation learning methods

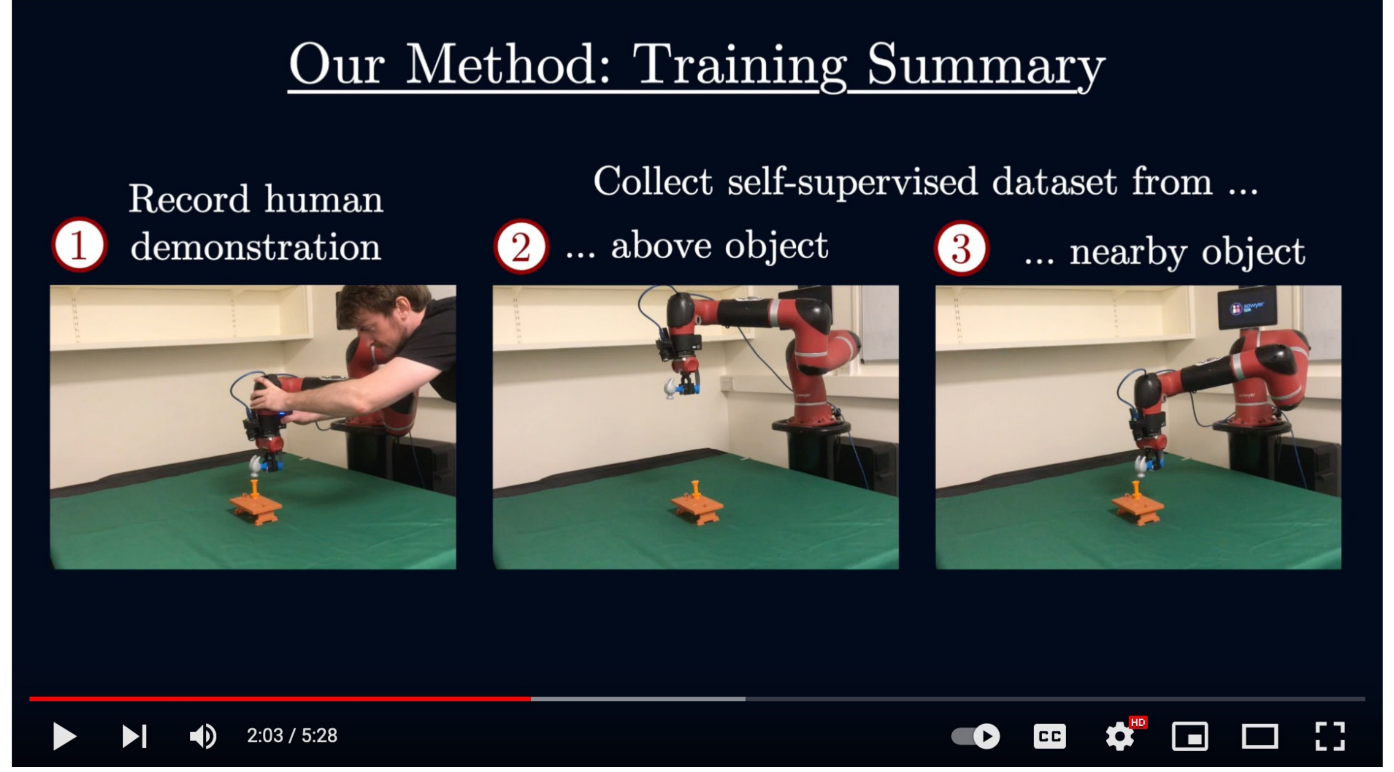


Other ways to make imitation learning efficient:

- Use many sensors
- Use designed features, or features extracted w/a pre-trained neural net
- Balance between classification (discrete actions) vs regression (continuous actions)

Hussein et al. Imitation Learning: A Survey of Learning Methods, 2018.

Example: Coarse-to-Fine Imitation Learning



Edward Johns, [Coarse-to-Fine Imitation Learning: Robot Manipulation from a Single Demonstration](), 2021.

Outline

- Review: MDP and Q-Learning
- Deep Q-Learning
- Imitation Learning
- Actor-Critic Learning

The Deep RL Methods We've Learned So Far

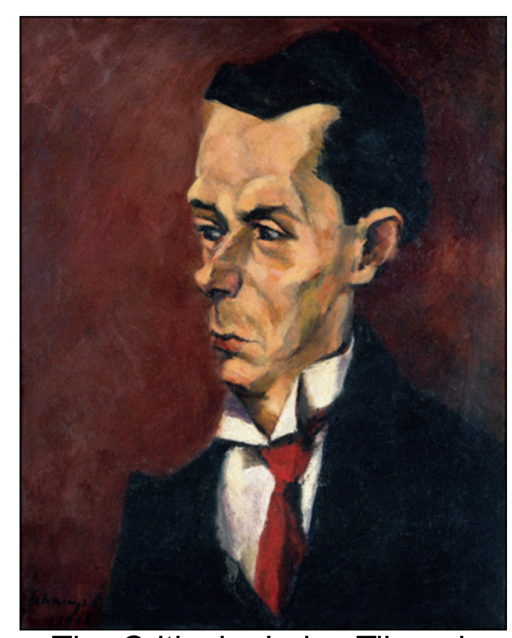
- Deep Q-learning gives us a network Q(s,a) which is very noisy, so we don't really want to trust it
- A policy network can directly estimate $\pi(s)$. The only problem is that we have no way to train it, unless we imitate human behavior.

Actors from the Comédie Française, by Antoine Watteau, 1720. Public Domain, https://commons.wikimedia.org/w/index.php?curid=15418670

Actor-critic algorithm

So let's train two neural nets!

- $Q_t(s,a)$ is the **critic**, and is trained according to the deep Q-learning algorithm (MMSE).
- $\pi_a(s)$ is the **actor**, and is trained to satisfy the critic



The Critic, by Lajos Tihanyi. Oil on canvas, 1916. Public Domain, https://commons.wikimedia.org/w/index.php?curid=17837438

The Actor-Critic Algorithm

Main idea:

- The **actor** is a policy network that decides what action to perform:

  $\pi_a(s) =$ Probability that a is the best action in state s

- The **critic** is a deep Q-learning network that estimates the quality of that action ($Q(s,a)$).

  $Q(s,a) =$ Expected sum of future rewards if (s,a)

- The critic is noisy, so they don't get to decide the action. Instead, we only use the critic to help us to train the actor.

The Actor-Critic Algorithm

$\pi_a(s) =$ Probability that a is the best action in state s

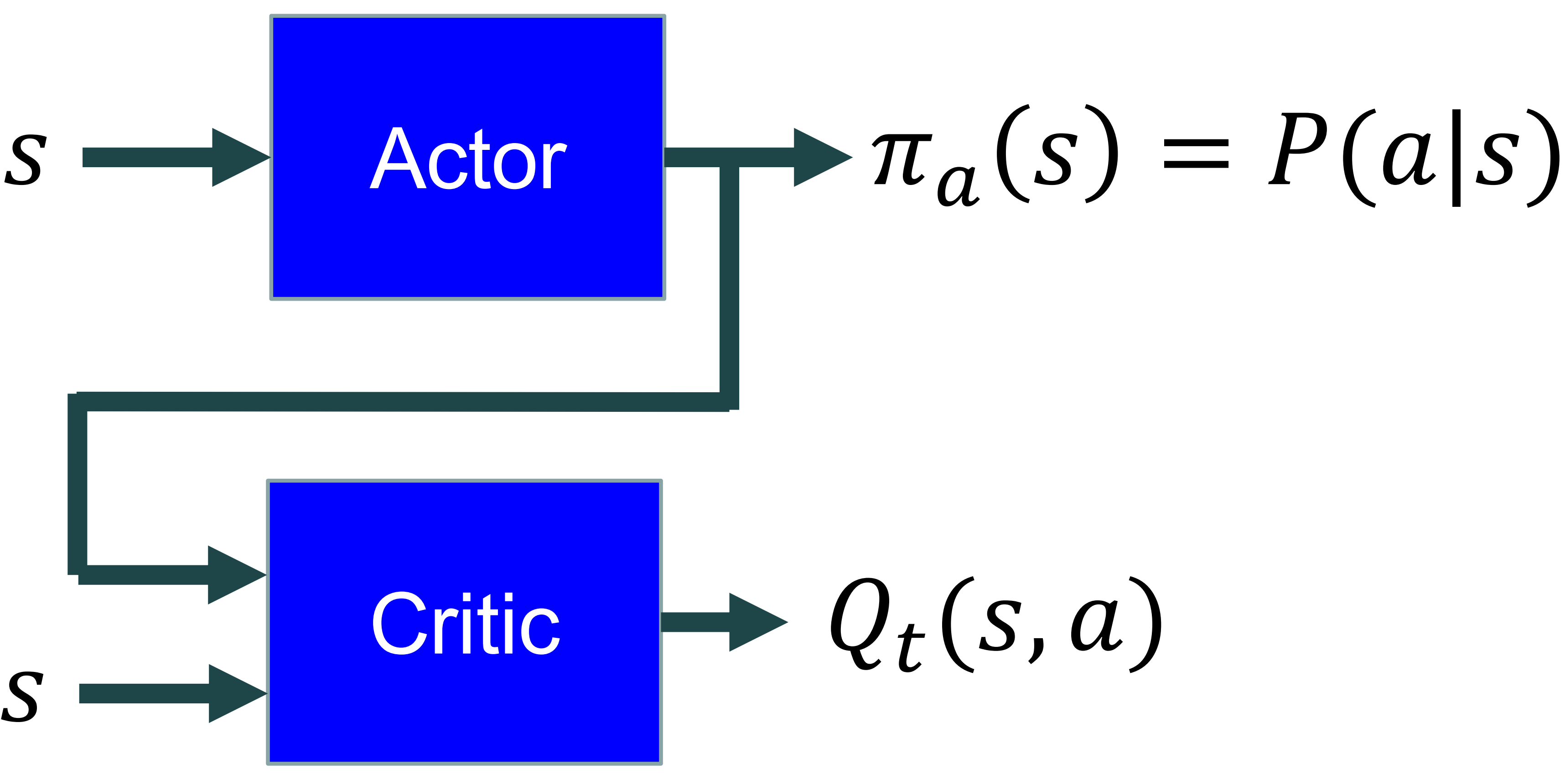
$Q(s,a) =$ Expected sum of future rewards if (s,a)

- The critic is noisy, so they don't get to decide the action. Instead, we only use the critic to help us to train the actor.

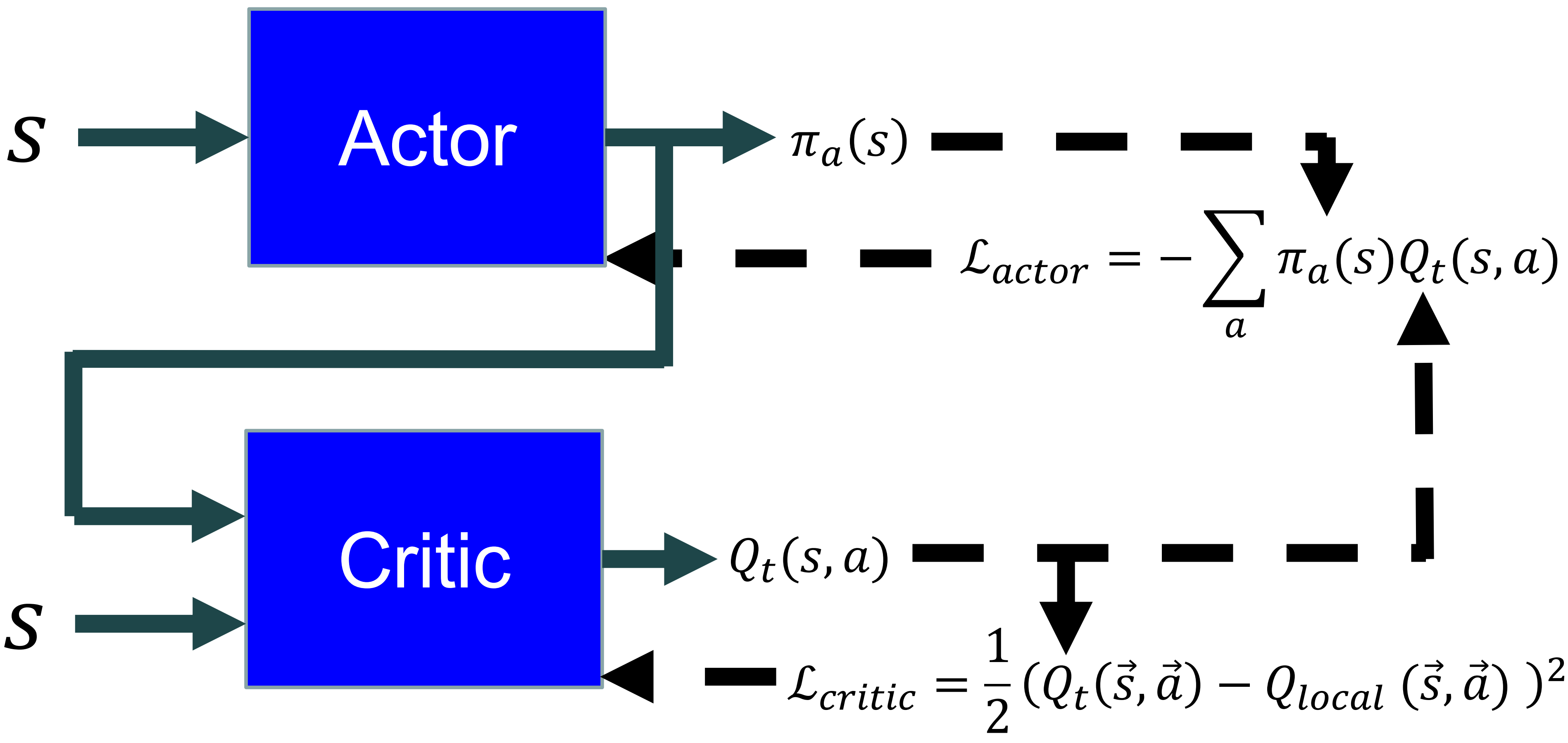
$$\mathcal{L} = -\sum_a \pi_a(s) Q(s,a)$$

- The training loss = negative expected sum of future rewards given action a, averaged over the probability with which the actor chooses action a.

The Actor-Critic Algorithm: Forward-Prop



The Actor-Critic Algorithm: Back-Prop



Asynchronous advantage actor-critic (A3C)

TORCS car racing simulation video

Mnih et al. Asynchronous Methods for Deep Reinforcement Learning. ICML 2016

Summary: Deep Reinforcement Learning

- Deep Q-learning:

$$\mathcal{L} = \frac{1}{2} E[(Q_t(\vec{s}_t, \vec{a}_t) - Q_{local}(\vec{s}_t, \vec{a}_t)\ )^2]$$

- Imitation Learning:

$$\mathcal{L} = -\log \pi_{a_t}(\vec{s}_t)$$

- Actor-Critic:

$$\mathcal{L}_{actor} = -\sum_a \pi_a(s) Q_t(s, a)$$