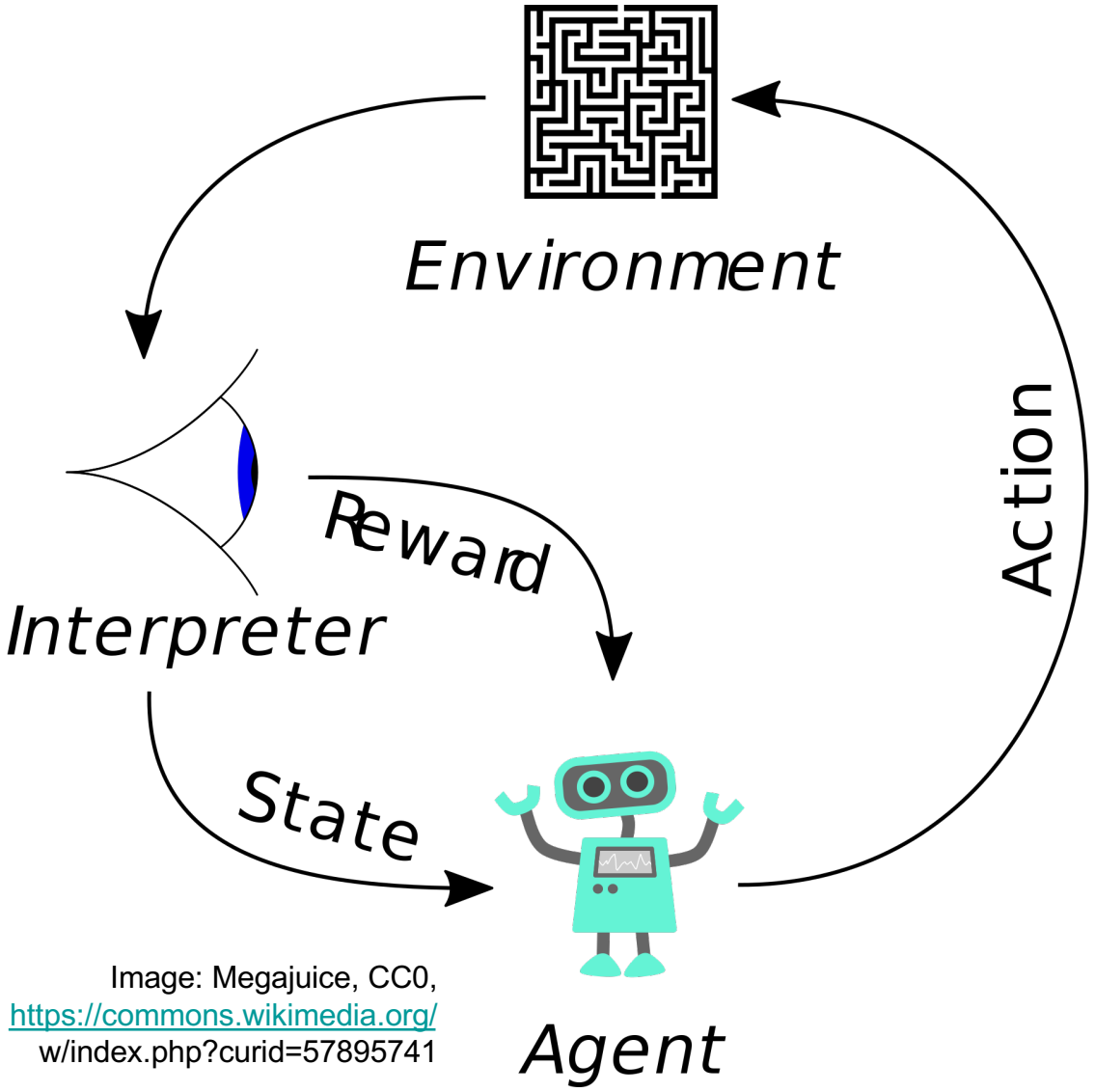
Model-Free Reinforcement Learning

CS440/ECE448 Lecture 24



Mark Hasegawa-Johnson, 4/2021,
including slides by Svetlana Lazebnik, 11/2017
CC-BY 4.0: you may remix or redistribute if you cite the source

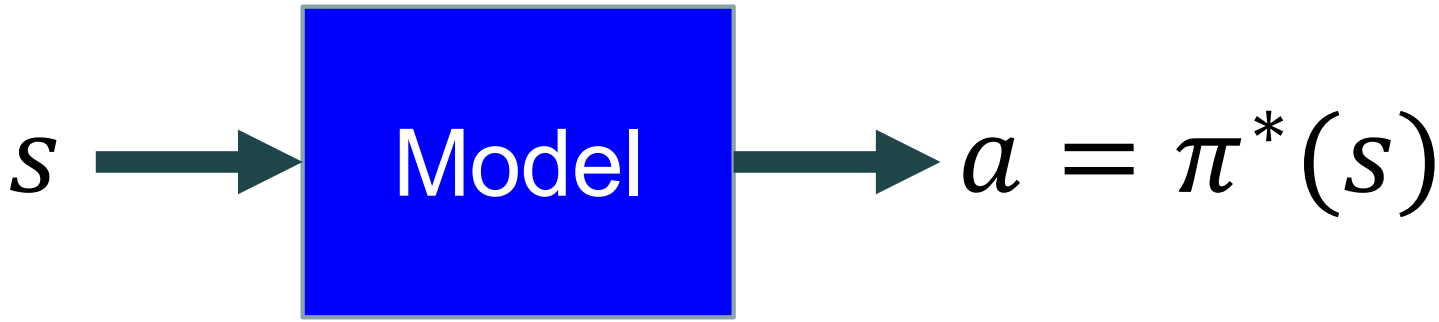
Image: Megajuice, CC0, https://commons.wikimedia.org/w/index.php?curid=57895741

What we've learned so far

- Markov Decision Process (MDP): Given P(s'|s,a) and R(s), you can solve for $\pi^*(s)$, the optimal policy, by finding U(s), the value of each state, using either value iteration or policy iteration.
- Model-Based Reinforcement Learning: If P(s'|s,a) and R(s) are unknown, you can find for $\pi(s)$ by using the observation-model-policy loop:
  - Observation: Create a training dataset by trying n consecutive actions, using an exploration-exploitation tradeoff like epsilon-first or epsilon-greedy
  - Model: Estimate P(s'|s,a) and R(s) using maximum likelihood estimation or Laplace smoothing
  - Policy: Find the optimum policy using value iteration or policy iteration.

Today: Model-Free Learning

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

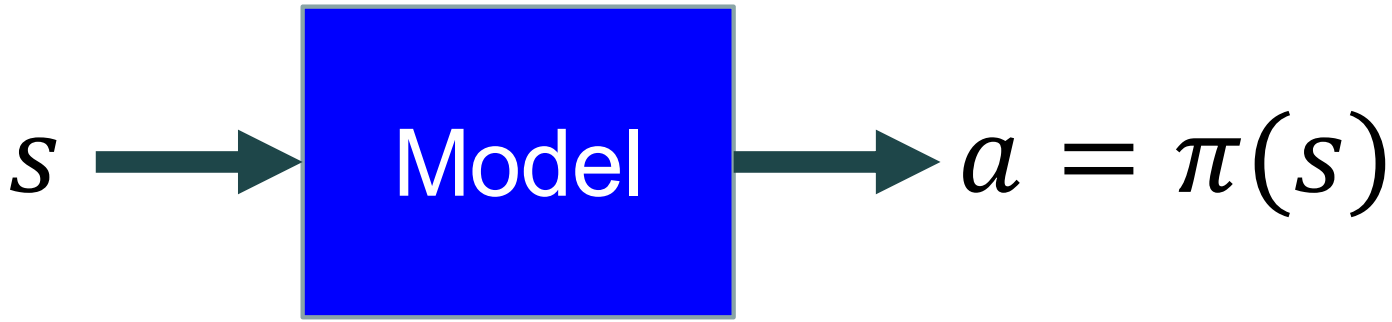


Outline

- Policy learning
  - Imitation learning
- Q-learning
  - Table-based: TD, SARSA
  - Deep Q-learning
- Actor-Critic RL

Policy Learning

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



Probabilistic Policy

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

$$\pi_a(s) = \frac{\exp(w_a^T h(s))}{\sum_{k=1}^{|A|} \exp(w_k^T h(s))} = P(A = a|S = s)$$

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

How do we train it?

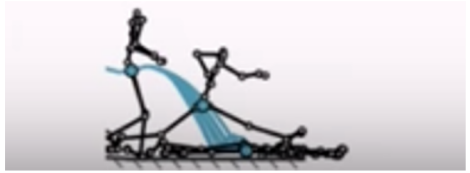
- Training data only give us (s_i, a_i, s_i', R_i).
- BAD IDEA: train the network to choose $\mathrm{A} = a_i$ that maximizes the immediate reward, R_i, and just ignore future rewards.
- GOOD IDEA: Train the network to maximize $U(s_i')$ = sum of all future rewards.
- PROBLEM: we don't know $U(s_i')$.

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

How to make Policy Learning trainable

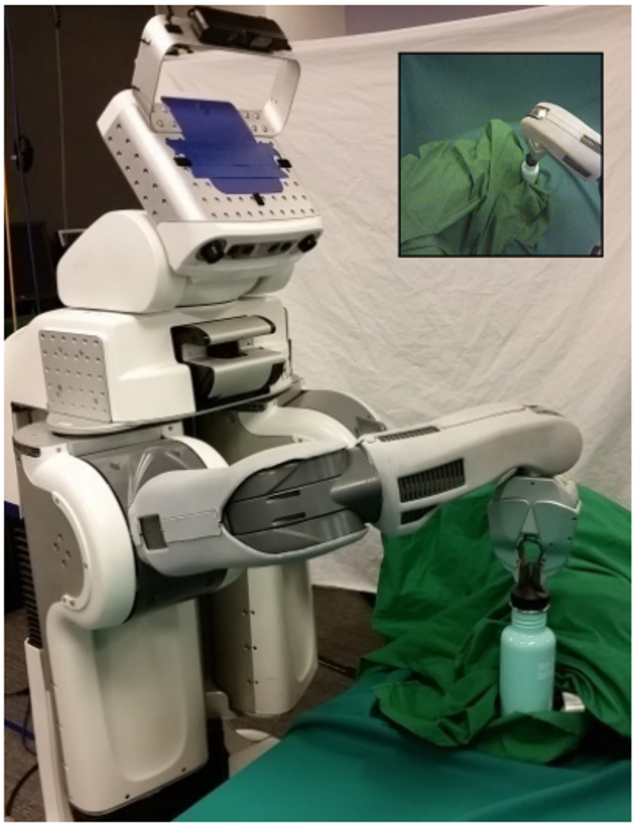
1. Actor-Critic RL. We'll come back to this.
2. Imitation learning.

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

Learning visuomotor policies

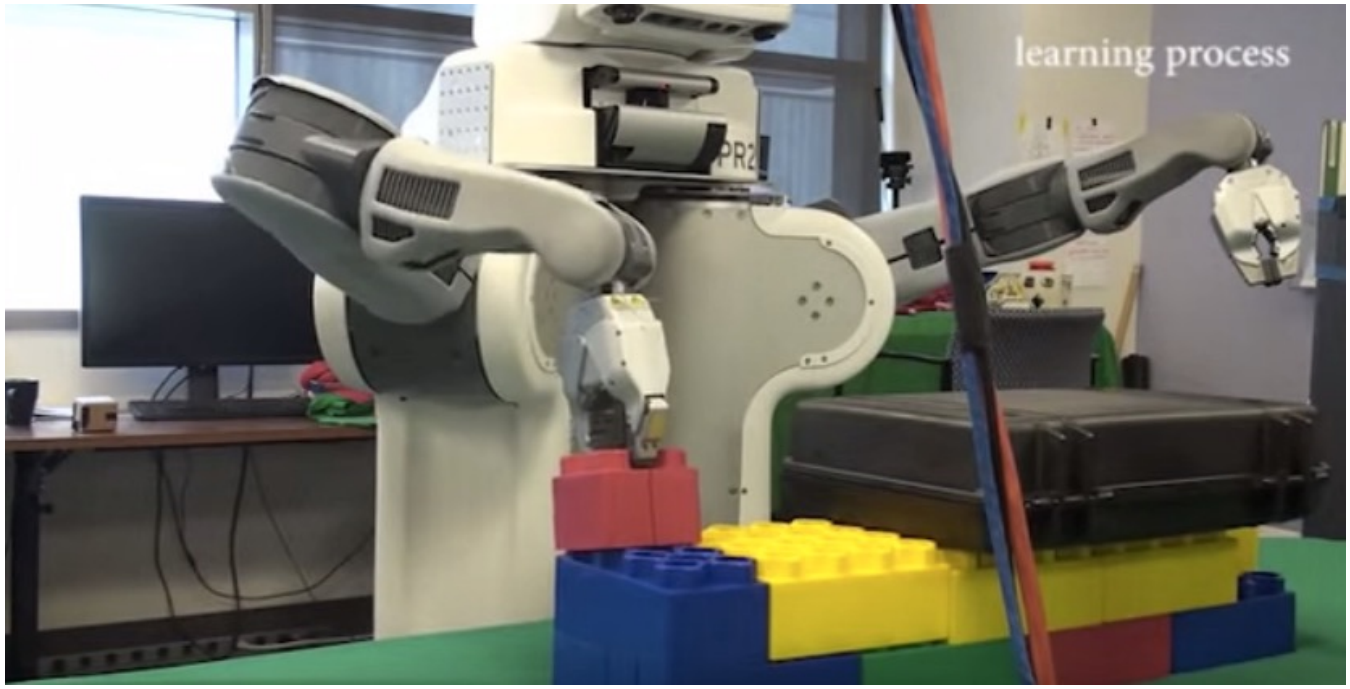


- *Underlying state* x: true object position, robot configuration
- *Observations* o: image pixels

- Two-part approach:
  - Learn *guiding policy* $\pi(a|x)$ using trajectory-centric RL and control techniques
  - Learn *visuomotor policy* $\pi(a|o)$ by imitating $\pi(a|x)$

S. Levine et al. End-to-end training of deep visuomotor policies. JMLR 2016

Learning visuomotor policies



Overview video, training video

S. Levine et al. End-to-end training of deep visuomotor policies. JMLR 2016

Outline

- Policy learning
  - Imitation learning
- Q-learning
  - Table-based: TD, SARSA
  - Deep Q-learning
- Actor-Critic RL

Bellman's Equation

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

When we talked about solving Bellman's equation before, we said that the optimum policy is given by the "max" operation: the action that gives you that maximum is the action you should take.

The Quality of an Action

The goal of Q-learning is to learn a function, Q(s,a), such that the best action to take is the action that maximizes Q:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} Q(s, a)$$

How about if we define Q(s,a) to be “The expected future reward I will achieve if I take action a in state s?”

The Quality of an Action

Suppose we know everything: we know P(s'|s,a), R(s), γ, and U(s). Then we collect our total expected future reward by doing these things:

- Collect our current reward, R(s)
- Discount all future rewards by γ
- Make a transition to a future state, s', according to P(s'|s,a)
- Then collect all future rewards, U(s')

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

The Quality of an Action

Whoa! So Bellman's equation is actually just a simplified version of the Q-function:

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

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

The Q-function: recursive definition

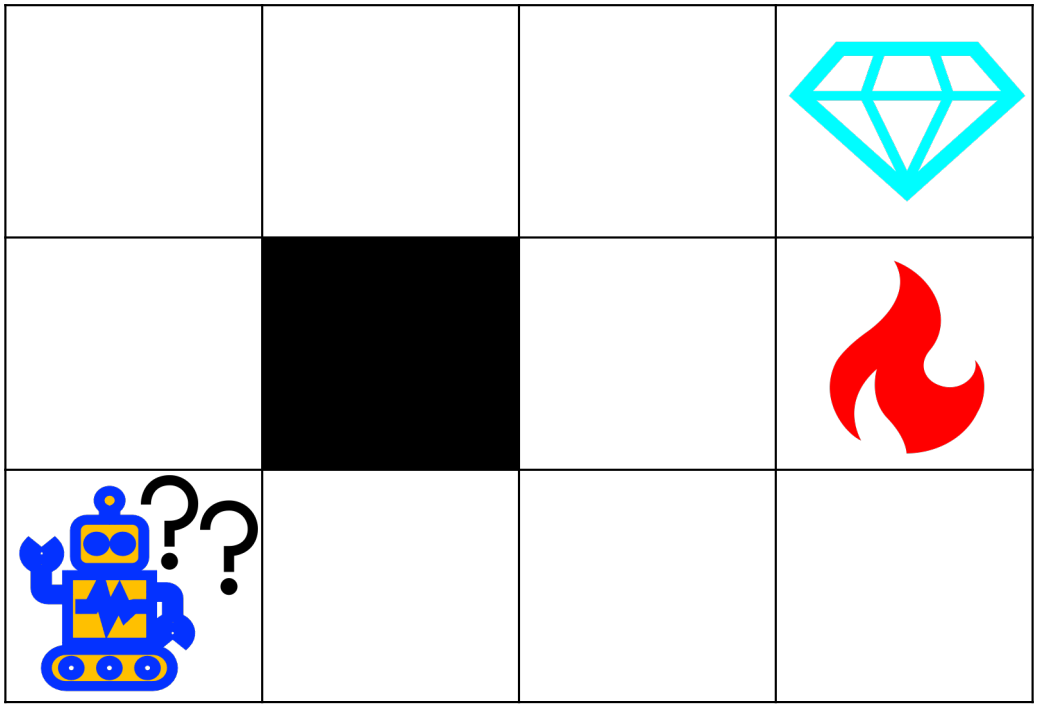
Or, to look at it another way, we could plug $U(s') = \max_{a' \in A(s')} Q(s', a')$ into the definition of the Q-function in order to get

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

Remember, it has these steps::

- Collect our current reward, R(s)
- Discount all future rewards by γ
- Make a transition to a future state, s', according to P(s'|s,a)
- Choose the optimum action, a', from state s', and collect all future rewards.

Example: Gridworld



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

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

$$\gamma = 1$$

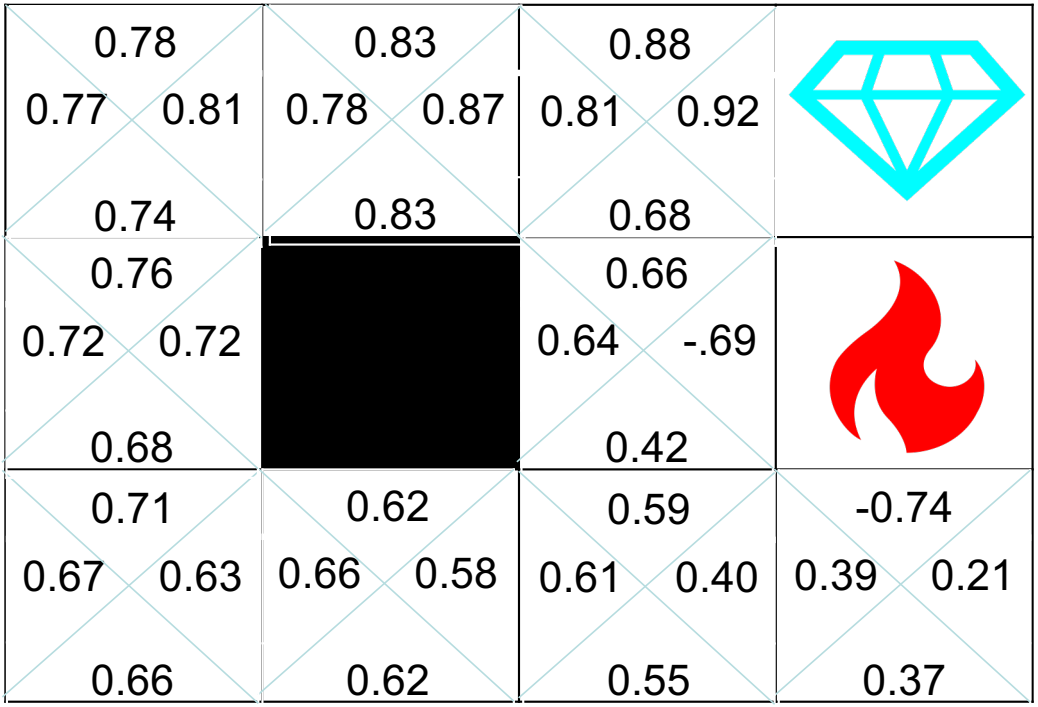
Gridworld: Utility of each state

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

0.81	0.87	0.92	
0.76		0.66	
0.71	0.66	0.61	0.39

(Calculated using value iteration.)

Gridworld: The Q-function



Calculated using a two-step value iteration:

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

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

Gridworld: Relationship between Q and U

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

0.78 0.77 0.81 0.74	0.83 0.78 0.87 0.83	0.88 0.81 0.92 0.68		0.81	0.87	0.92	
0.76 0.72 0.72 0.68		0.66 0.64 -.69 0.42		0.76		0.66	
0.71 0.67 0.63 0.66	0.62 0.66 0.58 0.62	0.59 0.61 0.40 0.55	-0.74 0.39 0.21 0.37	0.71	0.66	0.61	0.39

Reinforcement learning: Key concepts

Key concept: What if you don't know P(s'|s,a) and R(s)? Can you still estimate Q(s,a)?

1. Method #1: Model-based learning. Estimate P(s'|s,a) and R(s), then use them to compute Q(s,a).
2. Method #2 (today): Model-free learning. Try some stuff, observe the results, use the results to estimate Q(s,a).

Q-learning

Q(s,a) is the total of all current & future rewards that you expect to get if you perform action a in state s.

…so how about this strategy…

1. Play the game an infinite number of times.
2. Each time you try action a in state s, measure the reward that you receive from that point onward for the rest of the game.
3. Average.

Q-learning: a slightly more practical version

Q(s,a) is the total of all current & future rewards that you expect to get if you perform action a in state s.

…so how about this strategy…

1. Play the game ~~an infinite~~ **finite** number of times. **Keep track of $Q_t(s,a)$, the estimate of Q after the tth iteration.**
2. Each time you try action a in state s, measure the reward that you receive ~~from that point onward for the rest of the game.~~ **in the current state, plus γ times $Q_t(s',a')$.**
3. Average **Q_t with #2 in order to get Q_{t+1}**.

TD learning

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

Let's solve these problems as follows:

- Instead of $R(s)$, use $R_t(s)$, the reward we got this time.
- Instead of summing over $P(s'|s,a)$, just set s' equal to whatever state followed s this time.
- Instead of the true value of $Q(s,a)$, use our current estimate, $Q_t(s,a)$.

TD learning

$$Q_{local}(s_t, a_t) = R_t(s_t) + \gamma \max_{a\prime \in A(s_t\prime)} Q_t(s_t', a')$$

Let's solve these problems as follows:

- Instead of $R(s)$, use $R_t(s)$, the reward we got this time.
- Instead of summing over $P(s'|s, a)$, just set s' equal to whatever state followed s this time.
- Instead of the true value of $Q(s, a)$, use our current estimate, $Q_t(s, a)$.

TD learning

$$Q_{local}(s_t, a_t) = R_t(s_t) + \gamma \max_{a\prime \in A(s_t\prime)} Q_t(s_t', a')$$

Problem: NOISY!

- s_t' is random,
- $Q_t(s_t', a')$ is not the real value of Q, only our current estimate,
- $Q_{local}(s, a)$ might be very far away from $Q(s, a)$. It might even be worse than $Q_t(s, a)$.

TD learning

Solution: interpolate, using a small interpolation constant α that's $0 < \alpha < 1$:

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

$$= Q_t(s,a) + \alpha\big(Q_{local}(s,a) - Q_t(s,a)\big)$$

TD learning

$Q_{local}(s,a) - Q_t(s,a)$ is called the "time difference" or TD.

1. If the TD is positive, it means action a was **better** than we expected, so $Q_{t+1}(s,a) = Q_t(s,a) + \alpha TD$ is an increase.
2. If the TD is negative, it means action a was **worse** than we expected, so $Q_{t+1}(s,a) = Q_t(s,a) + \alpha TD$ is a decrease.

Exploration versus exploitation

- TD-learning has one gap, still: when you reach state s, how do you choose an action?
- You might think that you just choose $a^* = \max_{a \in A(s)} Q_t(s, a)$, but that has the following problem: what if $Q_t(s, a)$ is wrong?
- The solution is to use an exploration strategy. For example,
  - Epsilon-first strategy: if there's an action we've chosen less than ϵN times, then choose that. Otherwise, choose a^*.
  - Epsilon-greedy strategy: with probability $1 - \epsilon$, choose a^*. With probability ϵ, choose an action uniformly at random.

TD learning

Putting it all together, here's the whole TD learning algorithm:

1. When you reach state s, use your current exploration versus exploitation policy, $\pi_t(s)$, to choose some action $a = \pi_t(s)$.
2. Observe the state s' that you end up in, and the reward you receive, and then calculate Qlocal:

$$Q_{local}(s, a) = R_t(s) + \gamma \max_{a' \in A(s')} Q_t(s', a')$$

3. Calculate the time difference, and update:

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

Repeat.

TD learning

Putting it all together, here's the whole TD learning algorithm:

1. When you reach state s, use your current exploration versus exploitation policy, $\pi_t(s)$, to choose some action $a = \pi_t(s)$.
2. Observe the state s' that you end up in, and the reward you receive, and then calculate Qlocal:

$$Q_{local}(s,a) = R_t(s) + \gamma \max_{a' \in A(s')} Q_t(s',a')$$

3. Calculate the time difference, and update:

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

Repeat.

The action you actually perform

The action TD-learning assumes you will perform

TD learning is an off-policy learning algorithm

TD learning is called an off-policy learning algorithm because it assumes an action

$$\underset{a\prime \in A(s\prime)}{\operatorname{argmax}}\, Q_t(s', a')$$

…which is different from the action dictated by your current exploration versus exploitation policy

$$a' = \pi_t(s')$$

Sometimes off-policy learning converges slowly, for example, because the TD-learning update is not taking advantage of your exploration.

On-policy learning: SARSA

We can create an "on-policy learning" algorithm by deciding in advance which action (a') we'll perform in state s', and then using that action in the update equation:

1. Use your current exploration versus exploitation policy, $\pi_t(s)$, to choose some action $a = \pi_t(s)$.
2. Observe the state s' that you end up in, and then use your current policy to choose $a' = \pi_t(s')$.
3. Calculate Qlocal and the update equation as:

$$Q_{local}(s,a) = R_t(s) + \gamma Q_t(s',a')$$

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

4. Go to step 2.

On-policy learning: SARSA

This algorithm is called SARSA (state-action-reward-state-action) because:

- In order to compute the TD-learning version of Q_{local}, you only need to know the tuple (s, a, R, s'):

$$Q_{local}(s, a) = R_t(s) + \gamma \max_{a' \in A(s')} Q_t(s', a')$$

- In order to compute the SARSA version of Q_{local}, you need to have already picked out (s, a, R, s', a'):

$$Q_{local}(s, a) = R_t(s) + \gamma Q_t(s', a')$$

Outline

- Policy learning
  - Imitation learning
- Q-learning
  - Table-based: TD, SARSA
  - Deep Q-learning
- Actor-Critic RL

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:

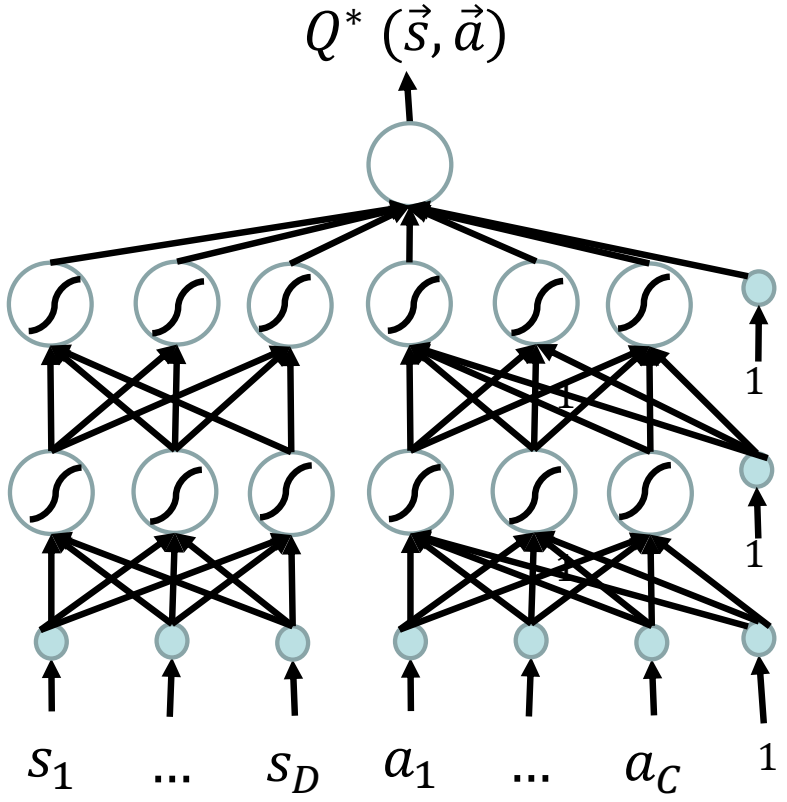
$$\vec{s} = [s_1, \ldots, s_D] =$$

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 $Q^*(\vec{s}, \vec{a})$ which is as close as possible to $Q(\vec{s}, \vec{a})$.



MMSE Deep Q learning

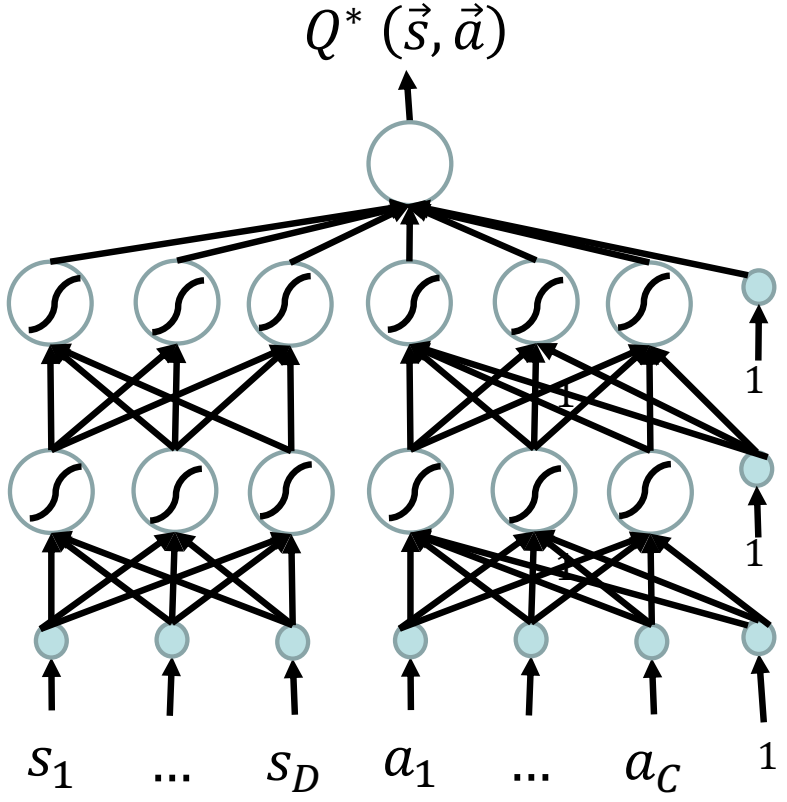
Suppose we train the neural network weights in order to minimize the mean-squared error (MMSE):

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

(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

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

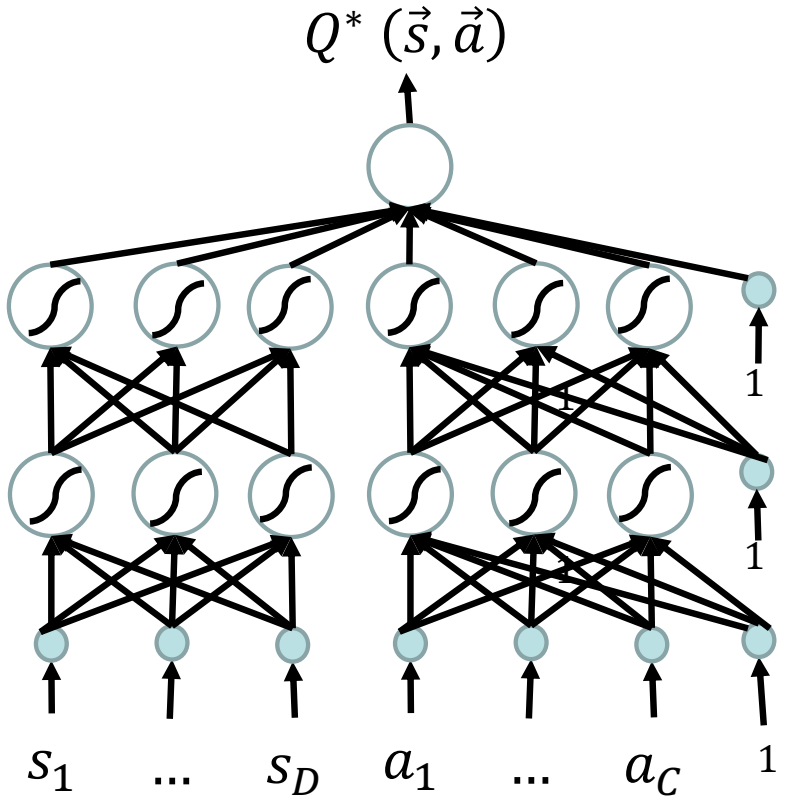


What makes deep Q learning harder than normal neural network training

- 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}

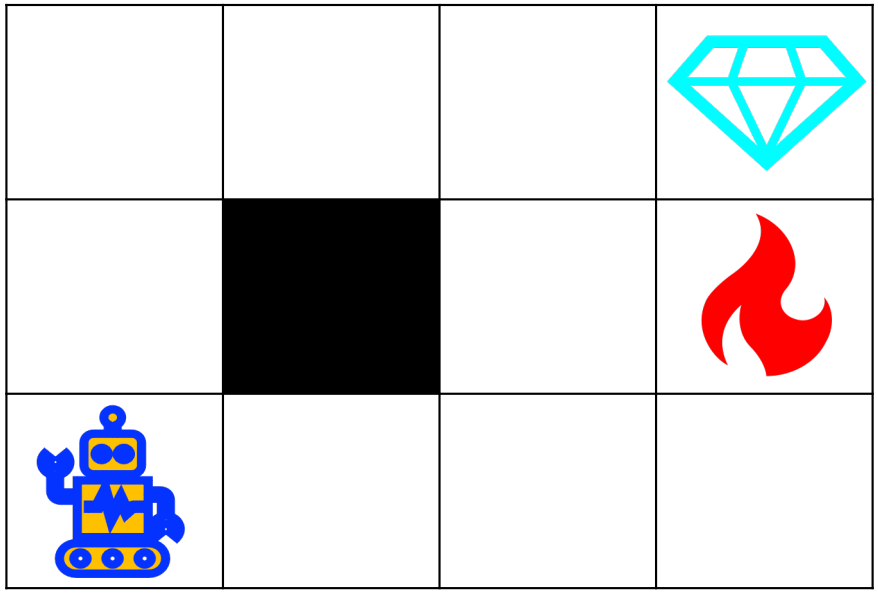
Remember that Q learning was defined as

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

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

$$Q_{local}(s,a) = R_t(s) + \gamma \max_{a'} Q_t(s',a')$$

…for s' equal to the next state we reach after action a on **this particular game**.



The solution: Q_{local}

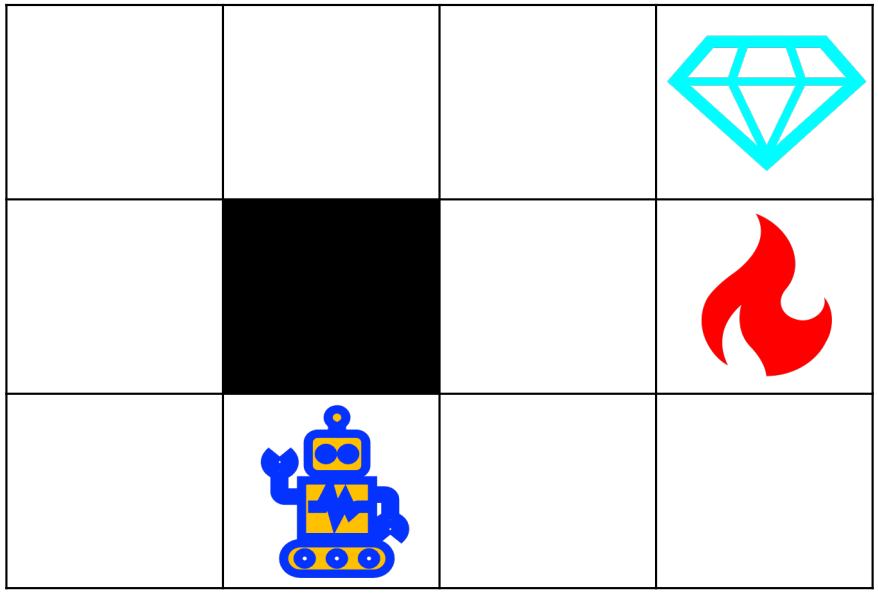
Let's define deep Q learning using the same Q_{local}:

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

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

$$Q_{local}(\vec{s}, \vec{a}) = R_t(\vec{s}) + \gamma \max_{\vec{a}'} Q^*(\vec{s}', \vec{a}')$$

Now we have an L that depends only on things we know ($Q^*(\vec{s}, \vec{a})$, $R_t(\vec{s})$, and $Q^*(\vec{s}', \vec{a}')$), so it can be calculated, differentiated, and used to update the neural network.

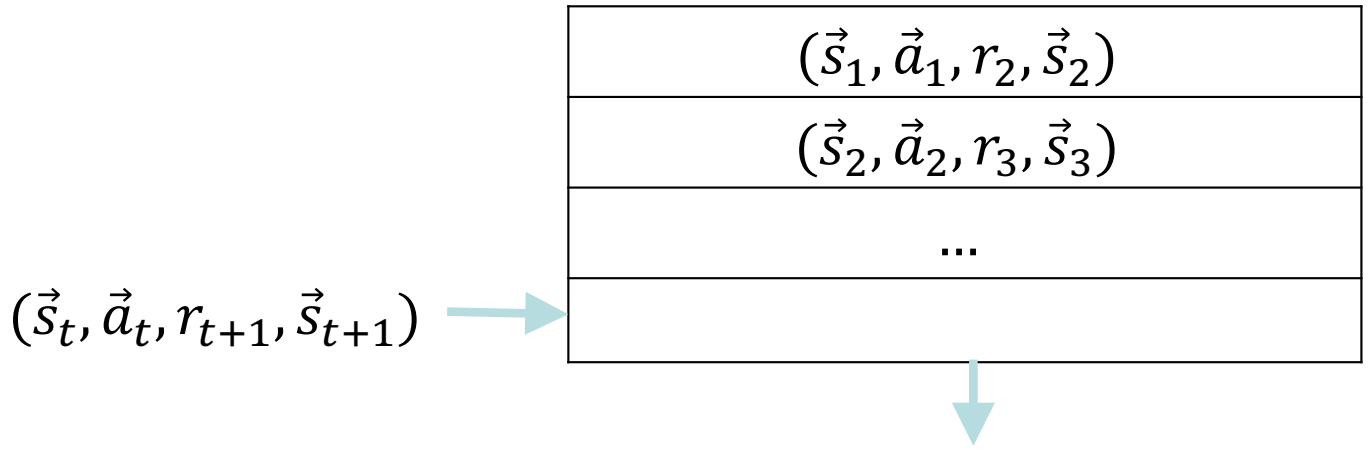


Dealing with training instability

- 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*

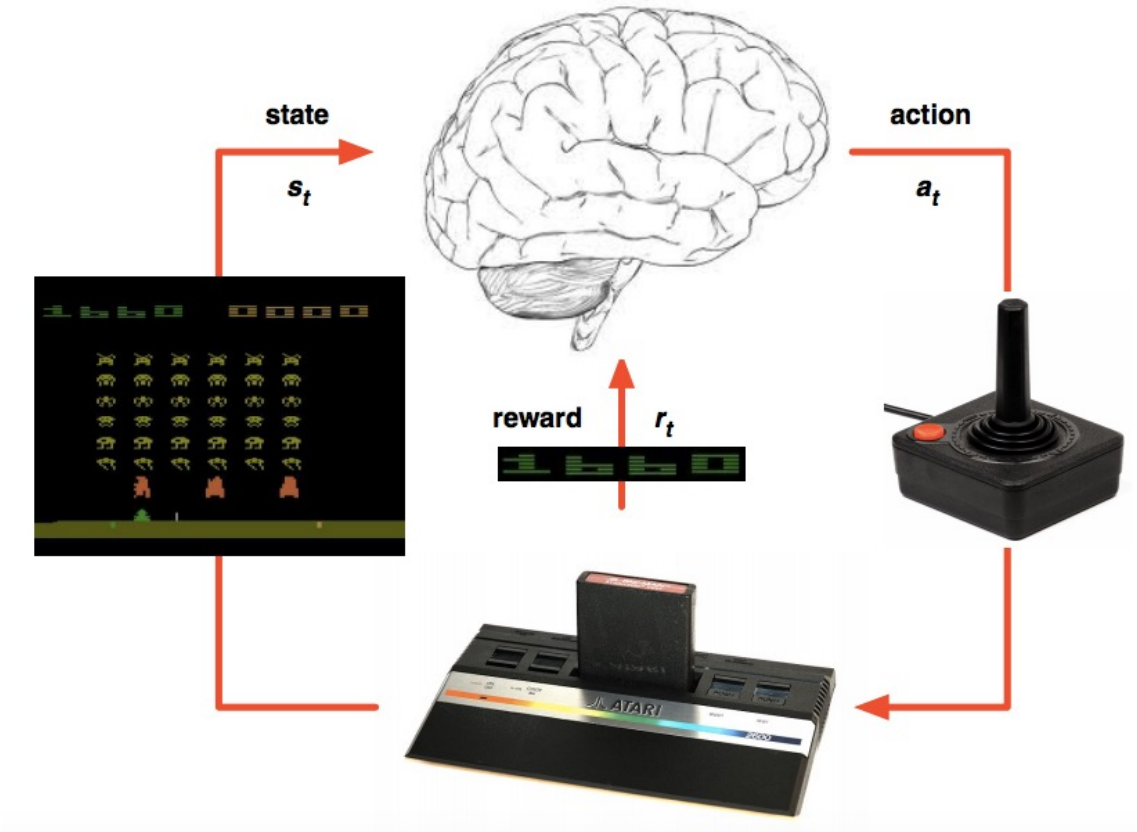
Experience replay

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



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

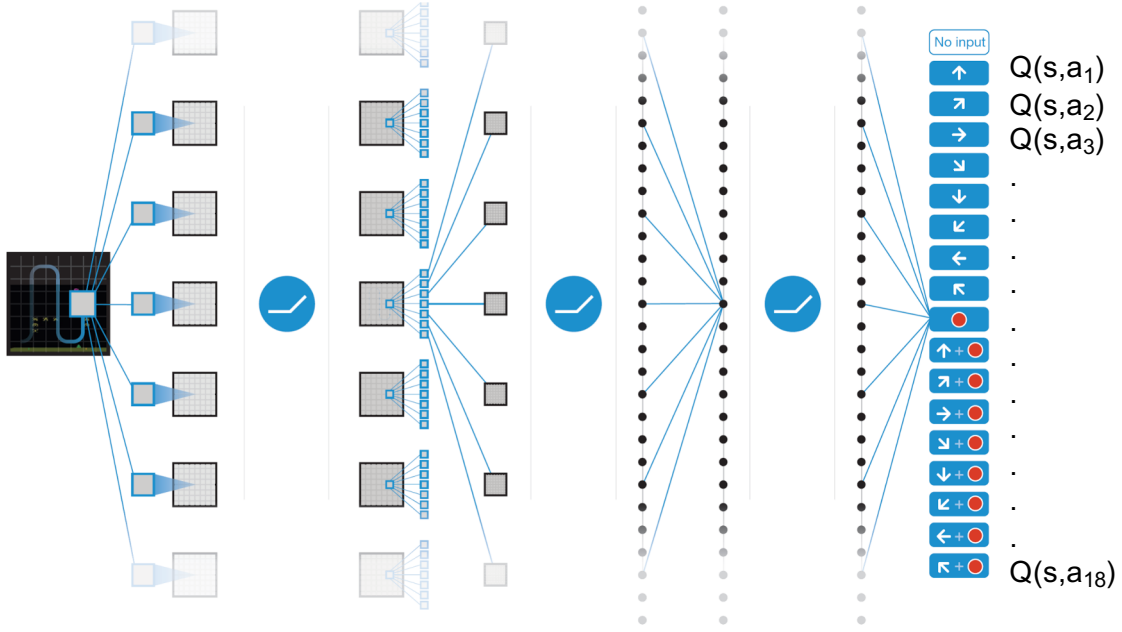
Deep Q learning in Atari



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

Deep Q learning in Atari

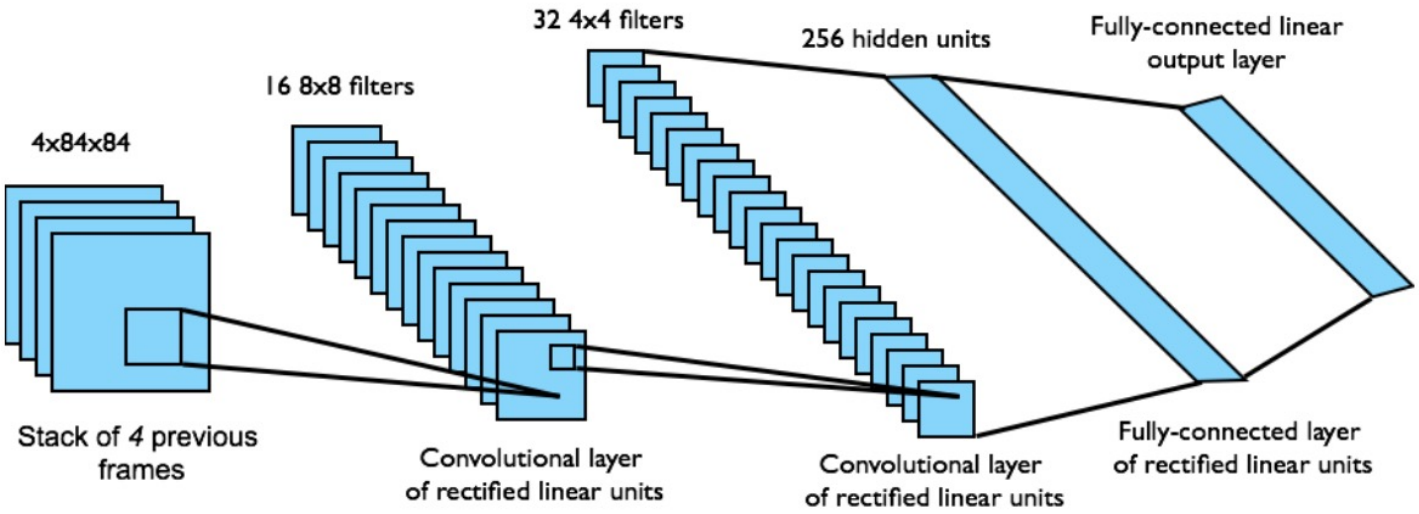
- End-to-end learning of Q(s,a) from pixels s
- Output is Q(s,a) for 18 joystick/button configurations
- Reward is change in score for that step



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

Deep Q learning in Atari

- Input state 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

Outline

- Policy learning
  - Imitation learning
- Q-learning
  - Table-based: TD, SARSA
  - Deep Q-learning
- Actor-Critic RL

Policy learning methods

- Suppose that s is continuous, but a is discrete (e.g., a one-hot vector).
- Then learning the policy directly can be much faster than learning Q values.
- We can train a neural network for a stochastic policy---a policy that chooses an action at random, using the probability distribution:

$$\pi(s,a) = \frac{e^{f(s,a)}}{\sum_{a'} e^{f(s,a')}}$$

How do we train $\pi(s, a)$?

- Expected total reward = Bellman's utility, $U(s)$.
- If we always choose the best action, then

$$U(s) = \max_a Q(s, a)$$

- With a stochastic policy, the utility of state s is suboptimal, given by:

$$U^{\pi}(s) = \sum_a \pi(s, a) Q(s, a)$$

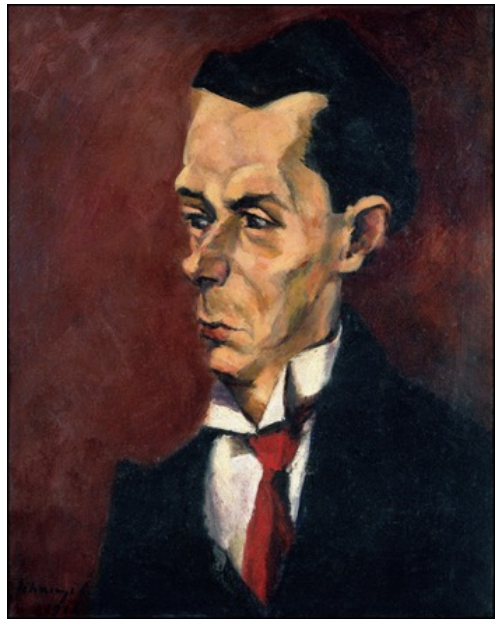
- If we knew $Q(s, a)$, then we'd learn $\pi(s, a)$ to maximize $U^{\pi}(s)$…

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).
- $\pi(s,a)$ 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

$$\pi(s,a) = \text{argmax} \sum_a \pi(s,a) Q_t(s,a)$$

Asynchronous advantage actor-critic (A3C)

TORCS car racing simulation video

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

Outline

- Policy learning: learn $\pi(s)$ directly
  - Imitation learning
- Q-learning: learn $Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a)U(s')$
  - Table-based: TD, SARSA
  - Deep Q-learning
- Actor-Critic: learn both