Model-Free Value-Based RL

CMPT 419/983

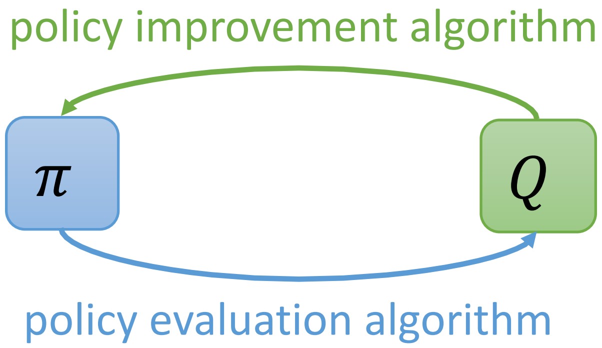
Mo Chen

SFU Computing Science

30/10/2019

Monte-Carlo Value Function Estimate

- Start with initial policy π and value function V or Q
- Use policy π to update Q: $a = \pi(s)$
  - Repeat for many episodes:
    - $N(s,a) \leftarrow N(s,a) + 1$
    - $Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)}\big(R(s,a) - Q(s,a)\big)$
- Use Q to update policy π
  - ϵ-greedy policy
    - With probability ϵ, choose random control
    - With probability $1-\epsilon$, choose $a = \arg\max_{a'}\{Q(s,a')\}$
  - Pick $\epsilon = \frac{1}{k}$, where k is the # of algorithm iterations
    - Explore less as value function becomes more accurate

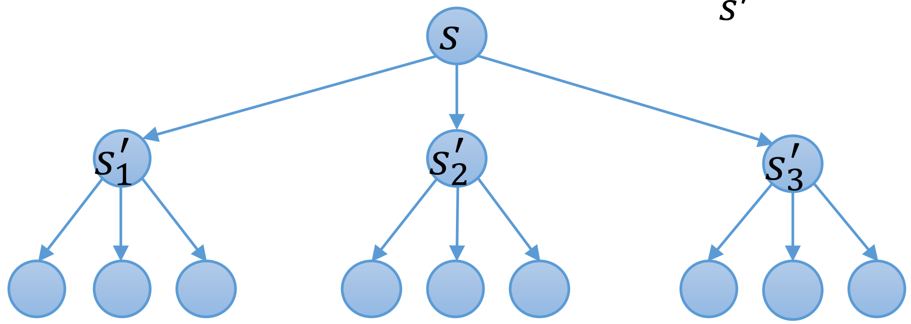


DP vs. MC Policy Evaluation

- Suppose the policy π is given
  - Dynamic Programming

    $$V(s) \leftarrow \max_a Q(s,a)$$

    $$Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} [p(s'|s,a)V(s')]$$

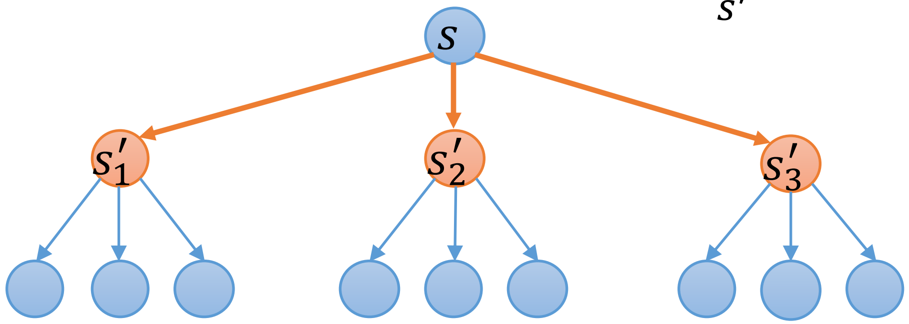
  - Monte-Carlo

DP vs. MC Policy Evaluation

- Suppose the policy π is given
  - Dynamic Programming

    $$V(s) \leftarrow \max_a Q(s,a)$$

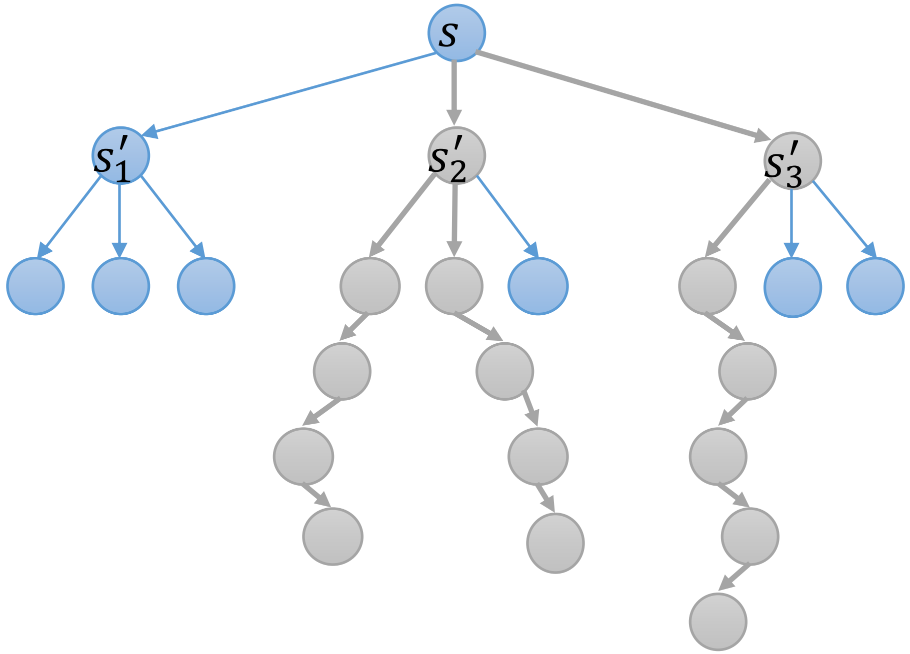
    $$Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} [p(s'|s,a)V(s')]$$



- Monte-Carlo
  - Repeat for many episodes:

    $$N(s,a) \leftarrow N(s,a) + 1$$

    $$Q(s,a) \leftarrow Q(s,a) + \alpha\big(R - Q(s,a)\big)$$

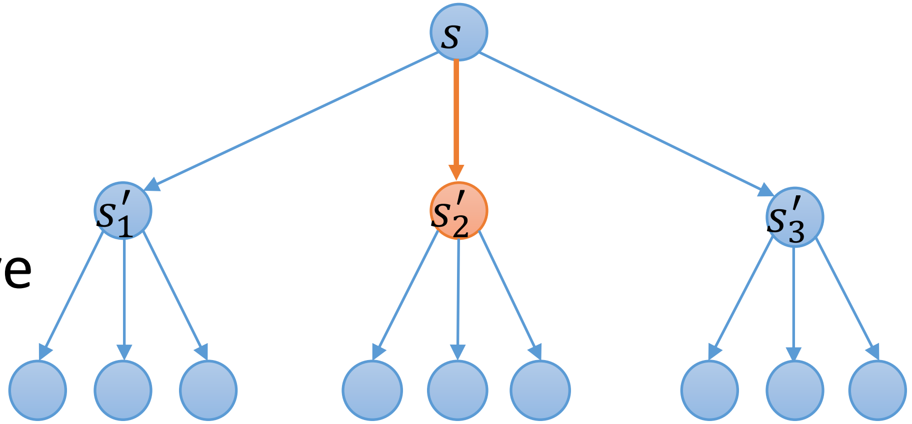


Temporal-Difference (TD) Policy Evaluation

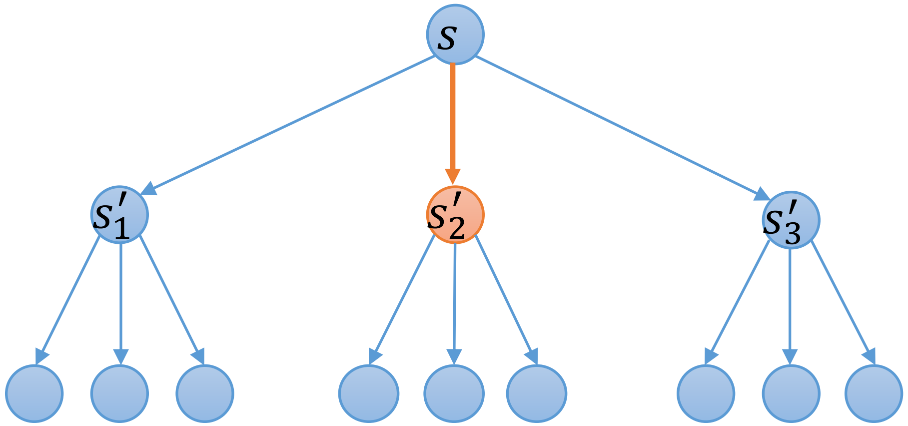
- Temporal-difference: a class of policy evaluation techniques TD(λ)
- Most basic version: TD(0)
  - From any state s, apply policy $a = \pi(s)$ for one time step, obtain reward $r(s, a)$
  - Get to next state s', and estimate return from then on using Q function
    - Note: next action is also from the same policy, $a' = \pi(s')$
    - $Q(s, a) \leftarrow Q(s, a) + \alpha\big(r(s, a) + \gamma Q(s', a') - Q(s, a)\big)$
  - Repeat for many episodes to obtain $Q(s, a)$ estimates at many states s and actions a

Temporal-Difference (TD) Policy Evaluation

- Most basic version: TD(0)
  $$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$$
- Advantages:
  - Online algorithm: Q can be updated during an episode
  - Does not require complete episodes
- Disadvantages:
  - System may not be Markov
  - Initial Q can be very bad and Q may never improve enough



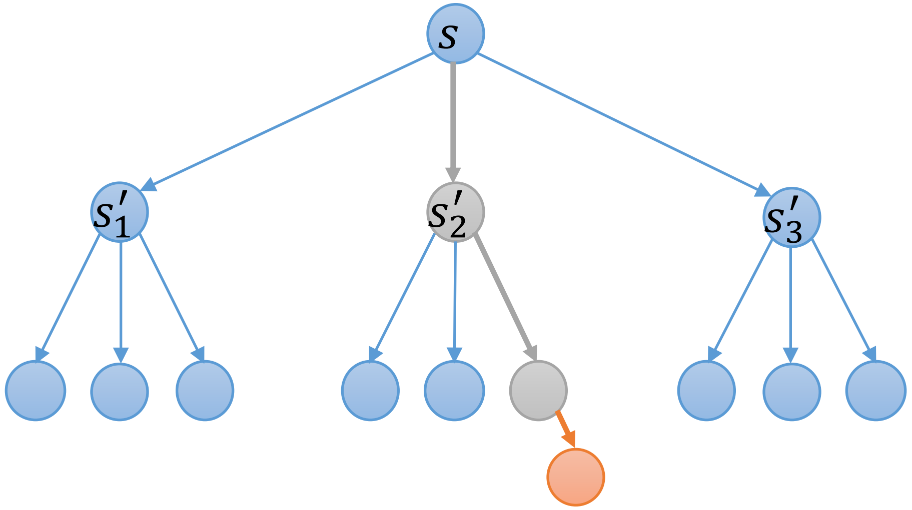
n-step TD



- TD: Look ahead one step
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$

- n-step TD: look ahead n steps

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(\underbrace{r(s,a) + \gamma r(s_{+1},a_{+1}) + \cdots \gamma^{n-1} r\big(s_{+(n-1)},a_{+(n-1)}\big) + \gamma^n Q(s_{+n},a_{+n})}_{:= R_n} - Q(s,a)\right)$$



- MC: Look ahead until the end of the episode

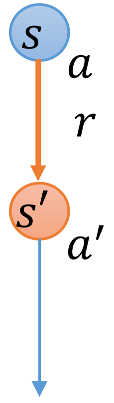
TD(λ)

- n-step return estimate:
  - $R_n = r(s,a) + \gamma r(s_{+1}, a_{+1}) + \cdots \gamma^{n-1} r\big(s_{+(n-1)}, a_{+(n-1)}\big) + \gamma^n Q(s_{+n}, a_{+n})$

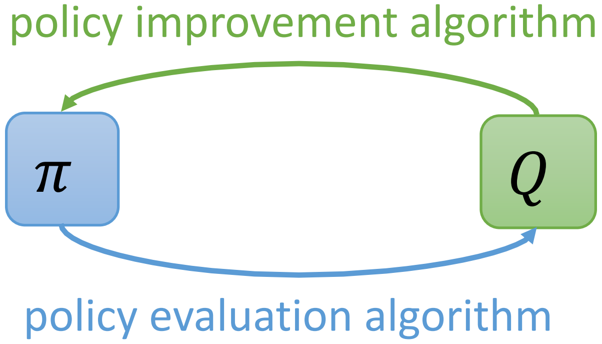
- λ-return: weighted average of different n-step returns
  - Weights: $(1-\lambda)\lambda^{n-1}$
  - Estimated return: $(1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_n$
  - Small λ → near-future rewards are more important
  - Large λ → far-future rewards are more important

- TD(λ) policy evaluation:
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\big((1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_n - Q(s,a)\big)$

SARSA Algorithm



- Start with initial policy π and value function V or Q
- Use ϵ-greedy policy to update Q: $a, a' \sim \pi(s)$, π is ϵ-greedy
  - Repeat for many episodes:
    - $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$
- New policy π is derived from new Q
  - ϵ-greedy policy
    - With probability ϵ, choose random control
    - With probability $1-\epsilon$, choose $a = \arg\max_{a'}\{Q(s,a')\}$
- If $\epsilon, \alpha \propto \frac{1}{k}$, then $Q(s,a) \to Q_{\pi^*}(s,a)$



On-Policy and Off-Policy Learning

- From SARSA:
  - Use ϵ-greedy policy to update Q: $a, a' \sim \pi(s)$, π is ϵ-greedy
    - Repeat for many episodes: $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$

- "**Behaviour policy**": policy used to collect rewards -- $a \sim \pi_B(s)$
- "**Target policy**": policy used to estimate future rewards -- $a' \sim \pi_T(s)$

- "**On-policy learning**": $\pi_B = \pi_T$
  - SARSA is an on-policy learning algorithm

- "**Off-policy learning**": $\pi_B \neq \pi_T$

  $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$, where $a \sim \pi_B(s), a' \sim \pi_T(s)$

Off-Policy Learning

- Off-policy learning: Behaviour and target policies are different
  $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r(s,a) + \gamma Q(s',a') - Q(s,a)\big)$, where $a \sim \pi_B(s), a' \sim \pi_T(s)$

- Advantages:
  - Learn from observing another agent (eg. human) execute a different policy
  - Learn from experience generated from old policies
  - Improve two policies at once, while following one policy

- Example: Q-Learning algorithm
  - π_B is ϵ-greedy with respect to Q
  - π_T is greedy with respect to Q

Q-Learning Algorithm

- Start with initial policy π and value function V or Q
- Update Q:
  - Repeat for many episodes with ϵ-greedy policy $a \sim \pi_B(s)$:
    - $Q(s,a) \leftarrow Q(s,a) + \alpha\left(r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$
- Both the ϵ-greedy π_B and the greedy π_T are derived from Q

- If $\epsilon, \alpha = \frac{1}{k}$, then $Q(s,a) \rightarrow Q_{\pi^*}(s,a)$

Function Approximation

- So far, $Q(s, a)$ is stored in a multi-dimensional array
  - Model-free, but cannot solve large problems

- Parametrize value functions with parameters (or weights) w
  - $\hat{Q}(s, a; w) \approx Q(s, a)$
  - Update parameters w using MC- or TD-based learning
  - Hopefully, Q is generalizable to different states s and actions a

Fitting to a Known Q_π

- Fit $\hat{Q}(s, a; w)$ to $Q_\pi(s, a)$

$$\underset{w}{\text{minimize}} \left\| Q_\pi(S, A) - \hat{Q}(S, A; w) \right\|_2^2$$

  - Training data: $\{(s_i, a_i), Q_\pi(s_i, a_i)\}$
  - The collection of states and actions in training data is denoted S and A

- Gradient with respect to w:
  - $\frac{\partial}{\partial w} \left\| \hat{Q}(S, A; w) - Q_\pi(S, A) \right\|_2^2 = 2 \left(Q_\pi(S, A) - \hat{Q}(S, A; w) \right) \frac{\partial \hat{Q}(S, A; w)}{\partial w}$

- Gradient descent:
  - $w \leftarrow w - \alpha \left(Q_\pi(S, A) - \hat{Q}(S, A; w) \right) \frac{\partial \hat{Q}(S, A; w)}{\partial w}$
  - In practice, use stochastic gradient descent

Monte-Carlo Incremental Weight Updates

- First-visit MC policy evaluation
  - At the first time t that s is visited in an episode,
    - Increment $N(s,a) \leftarrow N(s,a) + 1$
    - Record return $S(s,a) \leftarrow S(s,a) + \sum \gamma^t r(s_t, a_t)$
    - Repeat for many episodes
  - Estimate action-value function: $R(s,a) = \frac{S(s,a)}{N(s,a)} \approx Q(s,a)$
- Above procedure produces “training data” $\{S, A, R\}$
  - Storing a set of S, A, R, etc. is called “**experience replay**”
  - This is as opposed to updating w as data is being collected
- Update weights:
  - $w \leftarrow w - \alpha \left(R - \hat{Q}(S, A; w) \right) \frac{\partial \hat{Q}(S,A;w)}{\partial w}$
- Guaranteed to converge to local optimum

Temporal-Difference Incremental Weight Updates

- Most basic version: TD(0)
  - From any state s, apply policy $a = \pi(s)$ for one time step, obtain reward $r(s, a)$
  - Get to next state s', and estimate return from then on using Q function
    - $Q(s, a) \leftarrow Q(s, a) + \alpha\big(r(s, a) + \gamma Q(s', a') - Q(s, a)\big)$
  - Repeat for many episodes to obtain $Q(s, a)$ estimates at many states s and actions a
- Above procedure produces a collection of current and next states and actions, S, A, R, S', A'
- Update weights using TD target:
  - $w \leftarrow w - \alpha\left(R + \gamma \hat{Q}(S', A'; w) - \hat{Q}(S, A; w)\right)\frac{\partial \hat{Q}(S,A;w)}{\partial w}$
- Not always guaranteed to converge to local minimum

Q-Learning With Function Approximation

Goal: Given a set of weights w^{-}, find the next set of weights w in $\hat{Q}(s, a; w)$

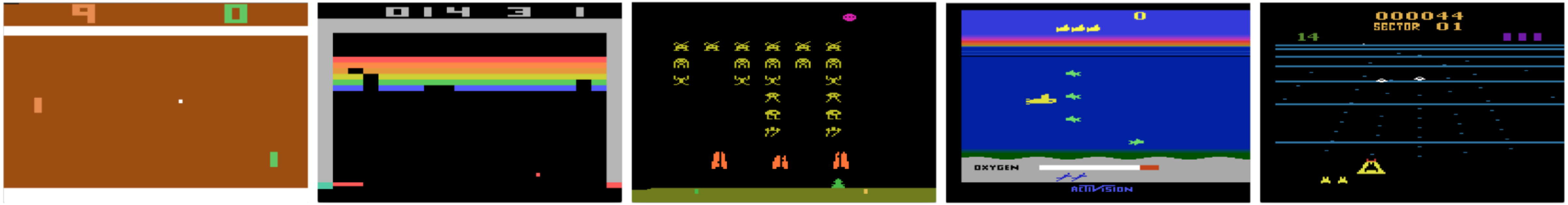
1. From any state s, apply ϵ-greedy policy with respect to $\hat{Q}(s, a; w^{-})$
   - This produces a collection S, A, R, S'
2. Sample from the above collection to obtain a smaller data set $\tilde{S}, \tilde{A}, \tilde{R}, \tilde{S}'$
3. Update weights using stochastic gradient descent

$$\underset{w}{\text{minimize}} \left\| \tilde{R} + \gamma \max_{a'} \hat{Q}\left(\tilde{S}', a'; w^{-}\right) - \hat{Q}\left(\tilde{S}, \tilde{A}; w\right) \right\|_2^2$$

- Use deep Q-network (DQN) for $\hat{Q}\left(\tilde{S}, \tilde{A}; w\right)$ → deep Q-learning

Deep Q-Learning Example: Atari Games

- Minh et al. “Playing Atari with Deep Reinforcement Learning,” 2013



- States: pixels from last few frames
- Actions: controls in the game
- Reward: game score
- Deep Q network: convolutional and fully connected layers

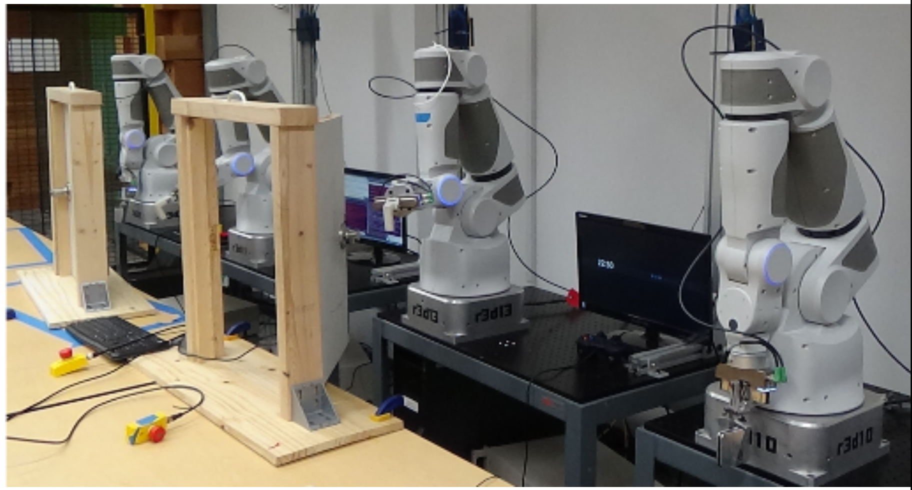
Starting out - 10 minutes of training

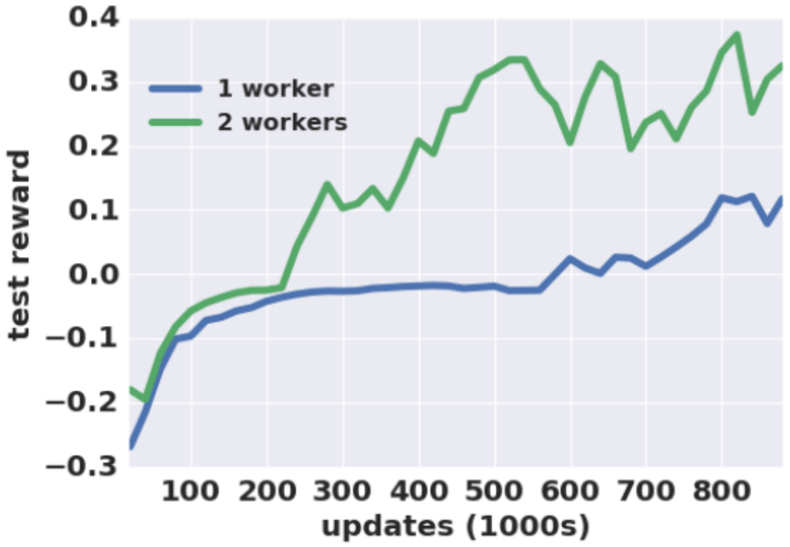
The algorithm tries to hit the ball back, but it is yet too clumsy to manage.

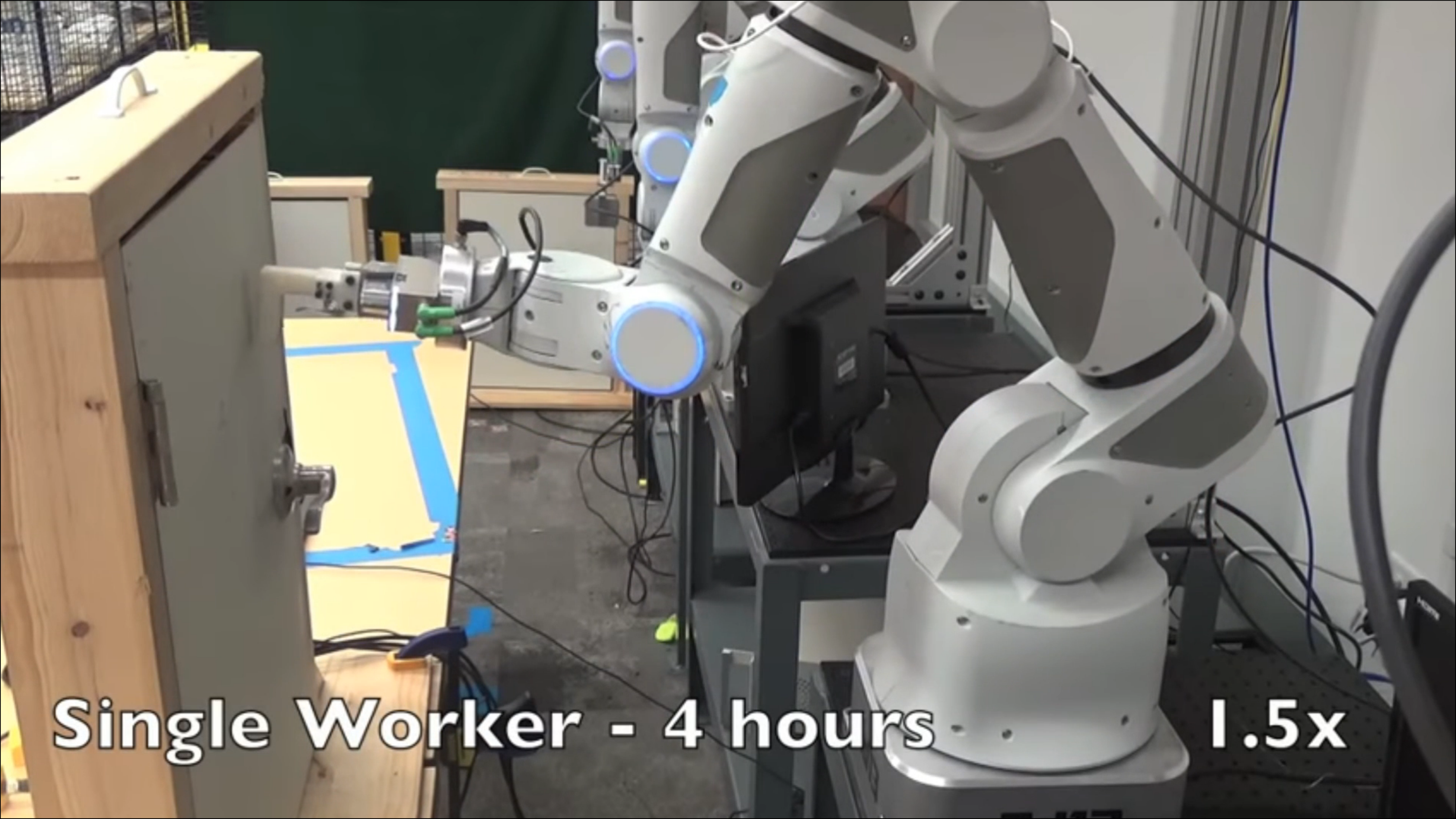
Deep Q-Learning: Robotics Example

- Gu et al. "Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates," 2017.

- States: joint angles, end-effector positions, and their time derivatives, target position
- Actions: joint velocities of arm, torque of fingers
- Task: open door, pick up object and place it elsewhere
- Deep Q network: two fully connected hidden layers, 100 units each
- Main challenge: use multiple robots to learn at the same time and share knowledge






Single Worker - 4 hours
1.5x