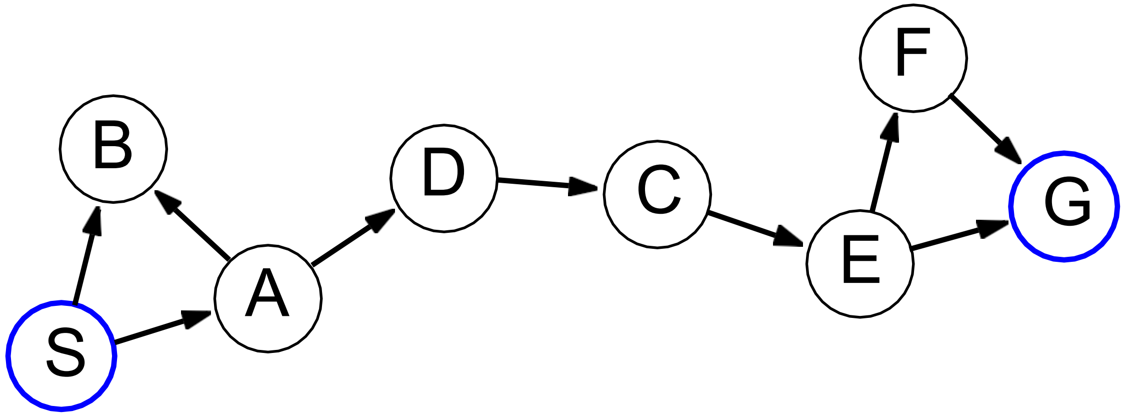
人工智能技术与应用

Markov Decision Process

2019.5.28

So far: search problems



state s, action a $\xrightarrow{\textbf{deterministic}}$ state Succ(s, a)

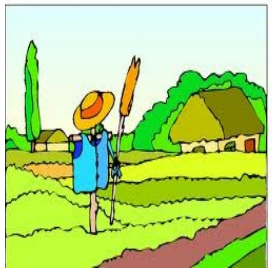
Uncertainty in the real world

$$\begin{tikzcd}
 & \text{state } s'_1 \\
\text{state } s\text{, action } a \arrow[ur] \arrow[dr] \arrow[r, phantom, "\textbf{random}"] & \\
 & \text{state } s'_2
\end{tikzcd}$$

Applications

Robotics: decide where to move, but actuators can fail, hit unseen obstacles, etc.

Resource allocation: decide what to produce, don't know the customer demand for various products



Agriculture: decide what to plant, but don't know weather and thus crop yield

Volcano crossing

		-50	20
		-50	
2			

Roadmap

Markov decision process

Policy evaluation

Value iteration

Dice game

Example: dice game

For each round $r = 1, 2, \ldots$

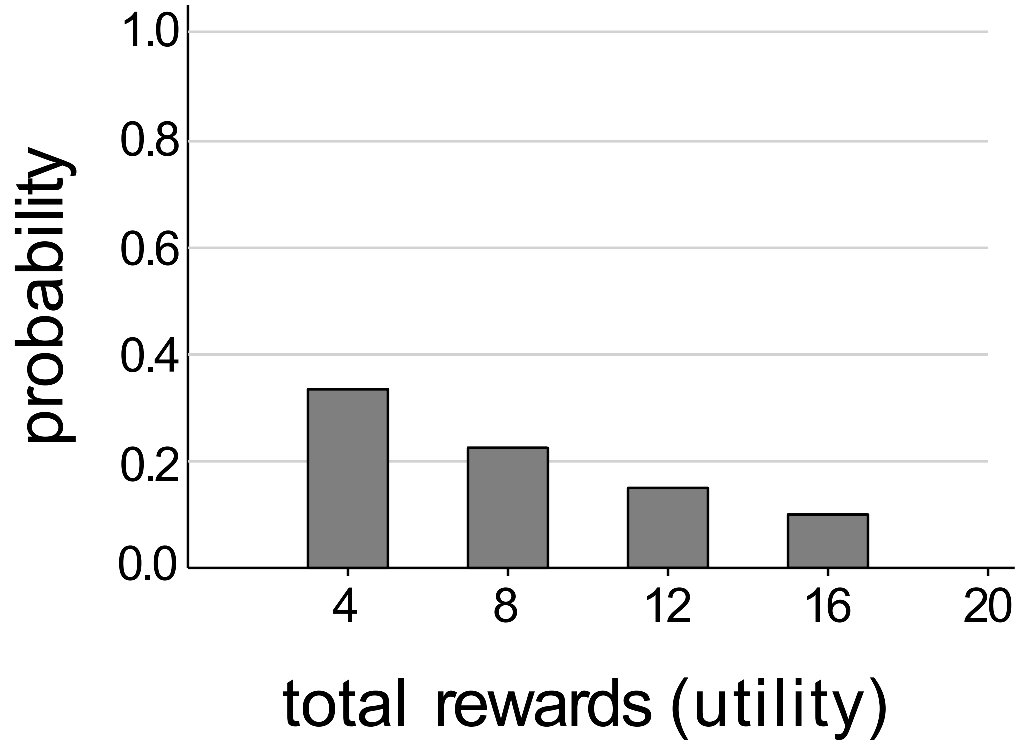
- You choose stay or quit.
- If quit, you get $\$10$ and we end the game.
- If stay, you get $\$4$ and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.

Start Stay Quit

Dice: Rewards: 0

Rewards

If follow policy ”stay”:

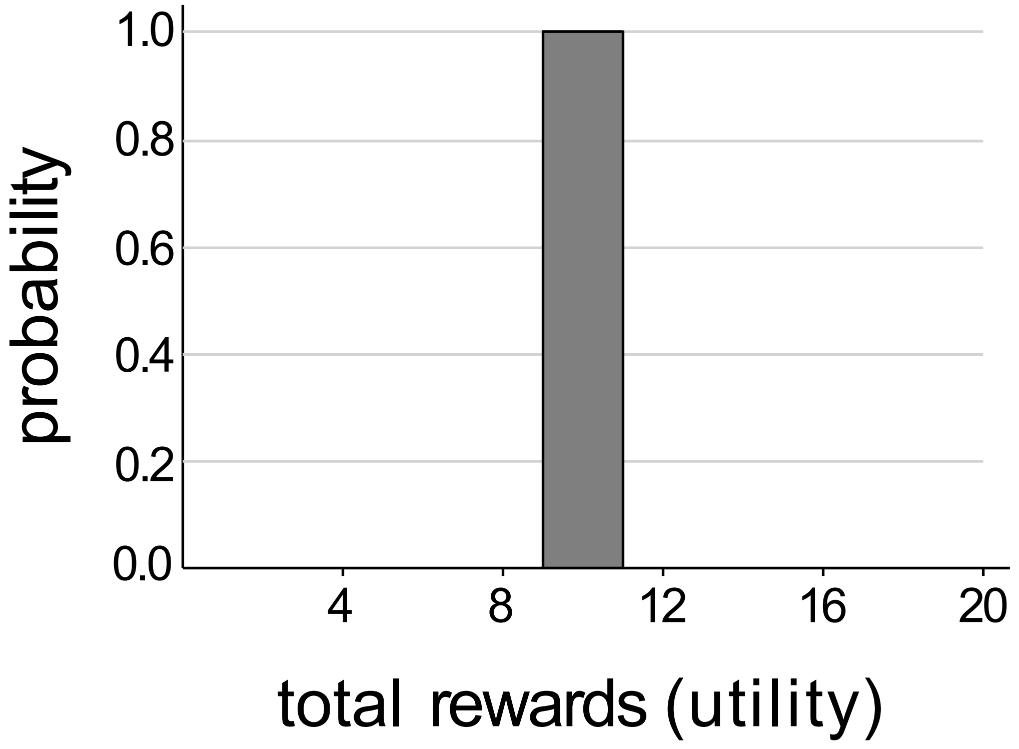


Expected utility:

$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \cdots = 12$$

Rewards

If follow policy ”quit”:



Expected utility:

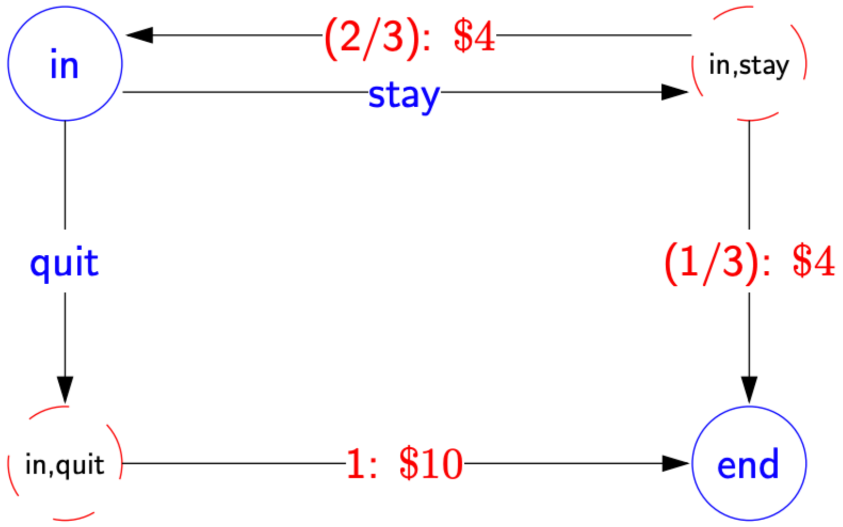
$$1(10) = 10$$

MDP for dice game

Example: dice game

For each round $r = 1, 2, \ldots$

- You choose stay or quit.
- If quit, you get $\$10$ and we end the game.
- If stay, you get $\$4$ and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.



Markov decision process

Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

$T(s, a, s')$: probability of s' if take action a in state s

Reward(s, a, s'): reward for the transition (s, a, s')

IsEnd(s): whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

Search problems

Definition: search problem

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

Succ(s, a): where we end up if take action a in state s

Cost(s, a): cost for taking action a in state s IsEnd(s):

whether at end

- Succ(s, a) $\Rightarrow$ $T(s, a, s')$
- Cost(s, a) $\Rightarrow$ Reward(s, a, s')

Transitions

Definition: transition probabilities

The **transition probabilities** $T(s, a, s')$ specify the probability of ending up in state s' if taken action a in state s.

Example: transition probabilities

s	a	s'	$T(s, a, s')$
in	quit	end	1
in	stay	in	2/3
in	stay	end	1/3

Probabilities sum to one

Example: transition probabilities

s	a	s'	$T(s, a, s')$
in	quit	end	1
in	stay	in	2/3
in	stay	end	1/3

For each state s and action a:

$$\sum_{s' \in \text{States}} T(s, a, s') = 1$$

Successors: s' such that $T(s, a, s') > 0$

What is a solution?

Search problem: path (sequence of actions)

MDP:

Definition: policy

A **policy** π is a mapping from each state $s \in$ States to an action $a \in$ Actions(s).

Example: volcano crossing

s	$\pi(s)$
(1,1)	S
(2,1)	E
(3,1)	N
...	...

Roadmap

Markov decision process

Policy evaluation

Value iteration

Evaluating a policy

Definition: utility

Following a policy yields a **random path**.
The **utility** of a policy is the (discounted) sum of the rewards on the path (this is a random quantity).

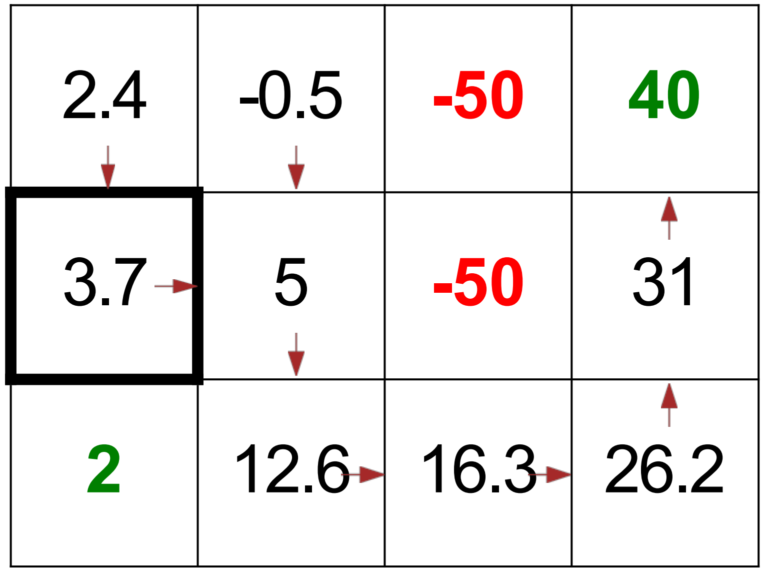
Path	Utility
[in; stay, 4, end]	4
[in; stay, 4, in; stay, 4, in; stay, 4, end]	12
[in; stay, 4, in; stay, 4, end]	8
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	16
...	...

Definition: value (expected utility)

The **value** of a policy is the **expected** utility.

Value: 12

Evaluating a policy: volcano crossing



a	*r*	*s*
		(2,1)
E	-0.1	(2,2)
S	-0.1	(3,2)
E	-0.1	(3,3)
E	-50.1	(2,3)

Value: 3.73

Utility: -36.79

Discounting

Definition: utility

Path: $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \ldots$ (action, reward, new state).

The **utility** with discount γ is

$$u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \cdots$$

Discount $\gamma = 1$ (save for the future):

[stay, stay, stay, stay]: $4 + 4 + 4 + 4 = 16$

Discount $\gamma = 0$ (live in the moment):

[stay, stay, stay, stay]: $4 + 0 \cdot (4 + \cdots) = 4$

Discount $\gamma = 0.5$ (balanced life):

[stay, stay, stay, stay]: $4 + \frac{1}{2} \cdot 4 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 4 = 7.5$

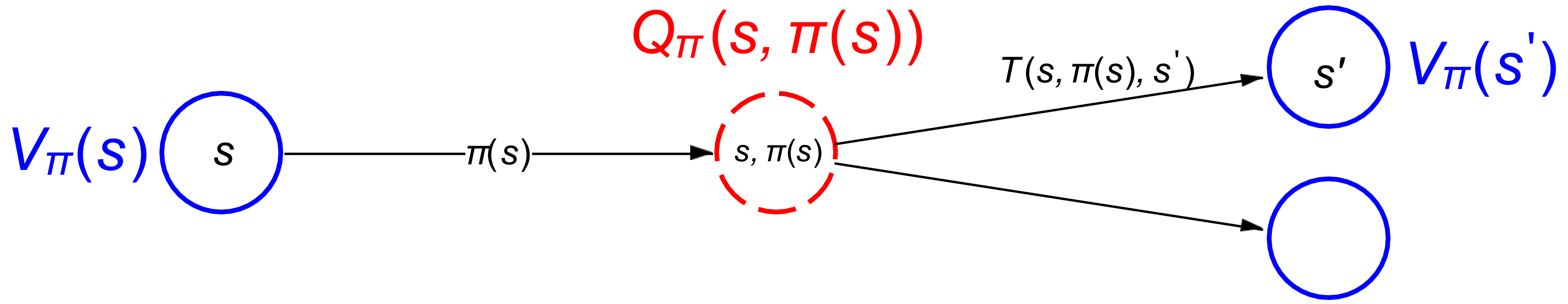
Policy evaluation

Definition: value of a policy

Let $V_\pi(s)$ be the expected utility received by following policy π from state s.

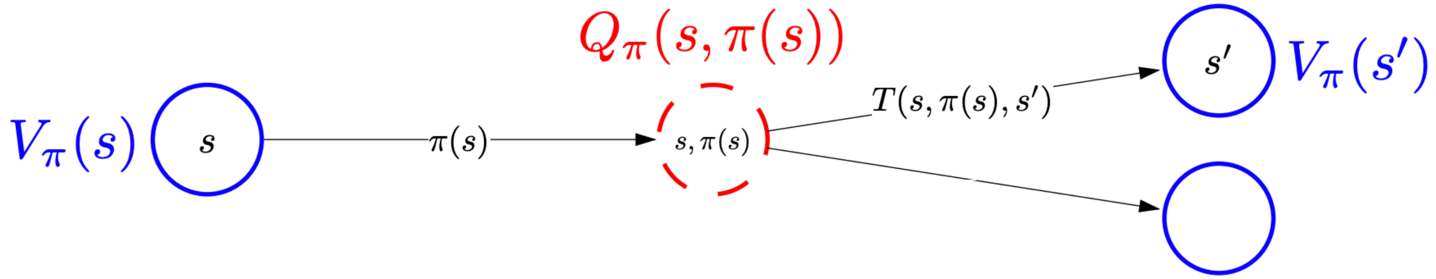
Definition: Q-value of a policy

Let $Q_\pi(s, a)$ be the expected utility of taking action a from state s, and then following policy π.



Policy evaluation

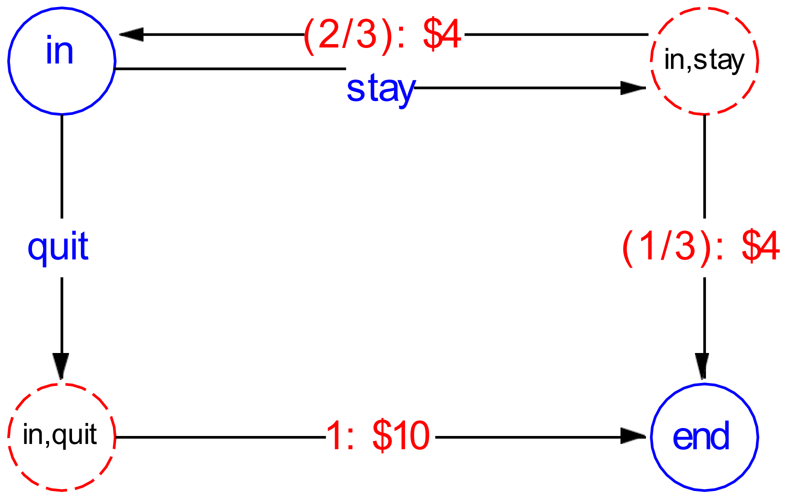
Plan: define recurrences relating value and Q-value



$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s,\pi(s)) & \text{otherwise.} \end{cases}$$

$$Q_\pi(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_\pi(s')]$$

Dice game



(assume $\gamma = 1$)

Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi(\text{end}) = 0$$

$$V_\pi(\text{in}) = \frac{1}{3}(4 + V_\pi(\text{end})) + \frac{2}{3}(4 + 1 \cdot V_\pi(\text{in}))$$

In this case, can solve in closed form:

$$V_\pi(\text{in}) = 12$$

Policy evaluation

Key idea: iterative algorithm

Start with arbitrary policy values and repeatedly apply recurrences to converge to true values.

Algorithm: policy evaluation

```
Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states $s$.
For iteration $t = 1, \dots, t_{\text{PE}}$:
    For each state $s$:
        $V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$
```

Policy evaluation computation

$$\boxed{V_{\pi}^{(t)}(s)}$$

iteration t

state s

0	-0.1	-0.2	0.7	1.1	1.6	1.9	2.2	2.4	2.6
0	-0.1	1.8	1.8	2.2	2.4	2.7	2.8	3	3.1
0	4	4	4	4	4	4	4	4	4
0	-0.1	1.8	1.8	2.2	2.4	2.7	2.8	3	3.1
0	-0.1	-0.2	0.7	1.1	1.6	1.9	2.2	2.4	2.6

Policy evaluation implementation

How many iterations (t_{PE})? Repeat until values don't change much:

$$\max_{s \in \mathrm{States}} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

Don't store $V_\pi^{(t)}$ for each iteration t, need only last two:

$$V_\pi^{(t)} \text{ and } V_\pi^{(t-1)}$$

Complexity

Algorithm: policy evaluation

```
Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states $s$.
For iteration $t = 1, \dots, t_{\text{PE}}$:
    For each state $s$:
        $V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s,\pi(s),s')[\text{Reward}(s,\pi(s),s') + \gamma V_\pi^{(t-1)}(s')]}_{Q^{(t-1)}(s,\pi(s))}$
```

MDP complexity

S states

A actions per state

S' successors (number of s' with $T(s,a,s') > 0$)

Time: $O(t_{\text{PE}} S S')$

Policy evaluation on dice game

Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi^{(t)}(\text{end}) = 0$$

$$V_\pi^{(t)}(\text{in}) = \tfrac{1}{3}(4 + V_\pi^{(t-1)}(\text{end})) + \tfrac{2}{3}(4 + V_\pi^{(t-1)}(\text{in}))$$

s	end	in	
$V_\pi^{(t)}$	0.00	12.00	(t = 100 iterations)

Converges to $V_\pi(\text{in}) = 12$.

Summary so far

- MDP: graph with states, chance nodes, transition probabilities, rewards
- Policy: mapping from state to action (solution to MDP)
- Value of policy: expected utility over random paths
- Policy evaluation: iterative algorithm to compute value of policy

Roadmap

Markov decision process

Policy evaluation

Value iteration

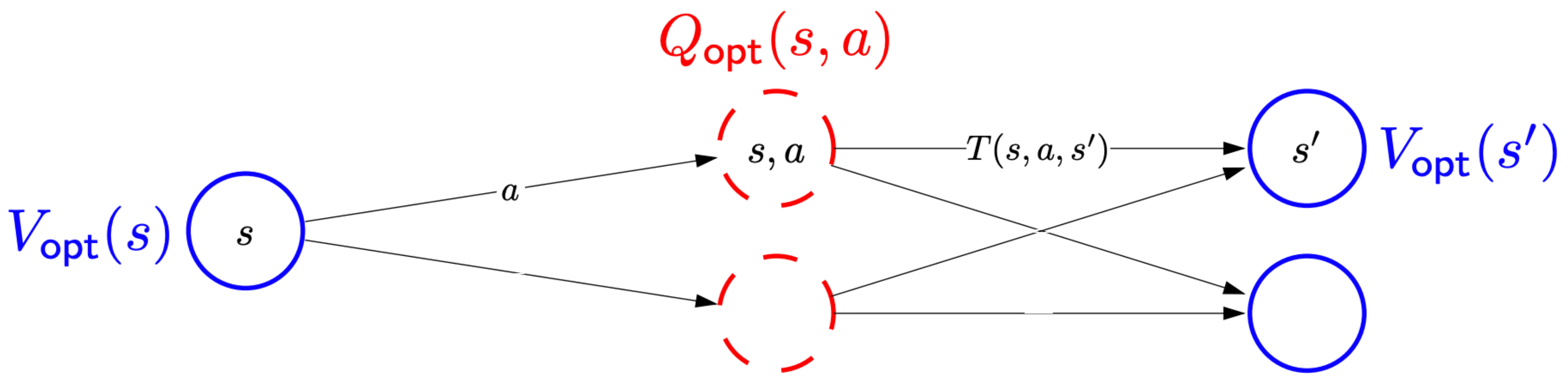
Optimal value and policy

Goal: try to get directly at maximum expected utility

Definition: optimal value

The **optimal value** $V_{\text{opt}}(s)$ is the maximum value attained by any policy.

Optimal values and Q-values



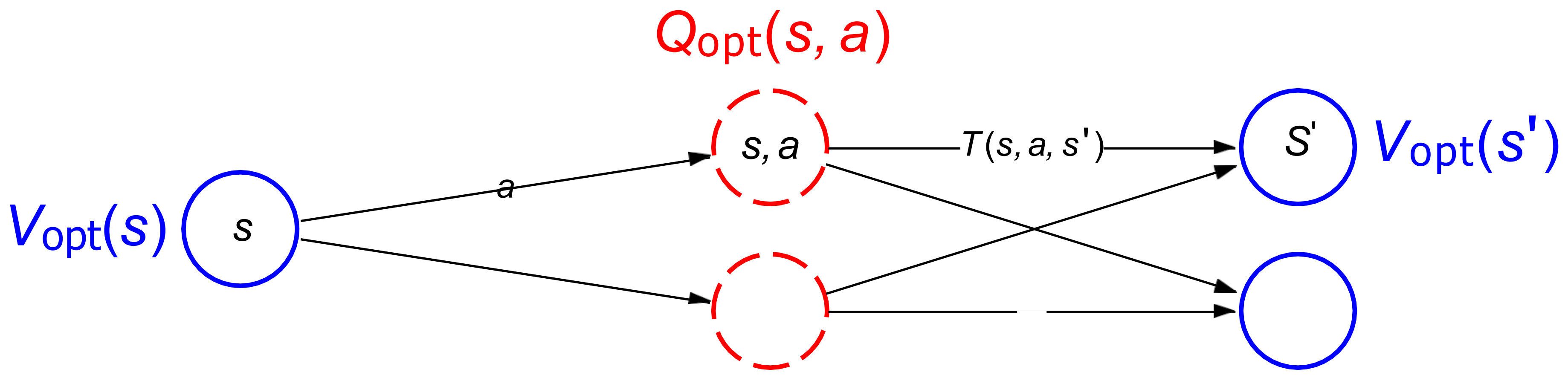
Optimal value if take action a in state s:

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')].$$

Optimal value from state s:

$$V_{\text{opt}}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s,a) & \text{otherwise.} \end{cases}$$

Optimal policies



Given Q_{opt}, read off the optimal policy:

$$\pi_{\text{opt}}(s) = \arg\max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$$

Value iteration

Algorithm: value iteration [Bellman, 1957]

```
Initialize $V_{\text{opt}}^{(0)}(s) \leftarrow 0$ for all states $s$.
For iteration $t = 1, \dots, t_{\text{VI}}$:
    For each state $s$:
        $V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s,a)}$
```

Time: $O(t_{\text{VI}} S A S')$

Value iteration: dice game

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	12.00	(t = 100 iterations)
$\pi_{\text{opt}}(s)$	-	stay	

Convergence

Theorem: convergence

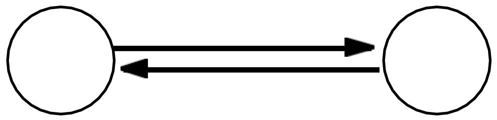
Suppose either

- discount $\gamma < 1$, or
- MDP graph is acyclic.

Then value iteration converges to the correct answer.

Example: non-convergence

discount $\gamma = 1$, zero rewards



Summary of algorithms

- Policy evaluation: (MDP, π) $\rightarrow$ V_π

- Value iteration: MDP $\rightarrow$ $(V_{\mathrm{opt}}, \pi_{\mathrm{opt}})$

Unifying idea

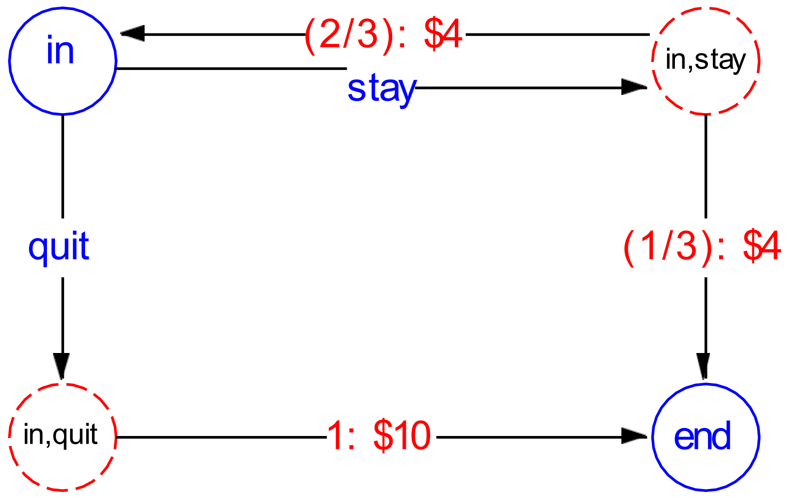
Algorithms:

- Search DP computes FutureCost(s)
- Policy evaluation computes policy value $V_\pi(s)$
- Value iteration computes optimal value $V_{\text{opt}}(s)$

Recipe:

- Write down recurrence (e.g., $V_\pi(s) = \cdots V_\pi(s^j) \cdots$)
- Turn into iterative algorithm (replace mathematical equality with assignment operator)

Review: MDPs



Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

$T(s, a, s')$: probability of s' if take action a in state s

Reward(s, a, s'): reward for the transition (s, a, s')

IsEnd(s): whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

Review: MDPs

- Following a **policy** π produces a path (**episode**)

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$$

- **Value** function $V_\pi(s)$: expected utility if follow π from state s

$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

- **Q-value** function $Q_\pi(s, a)$: expected utility if first take action a from state s and then follow π

$$Q_\pi(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_\pi(s')]$$

Unknown transitions and rewards

Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

IsEnd(s): whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

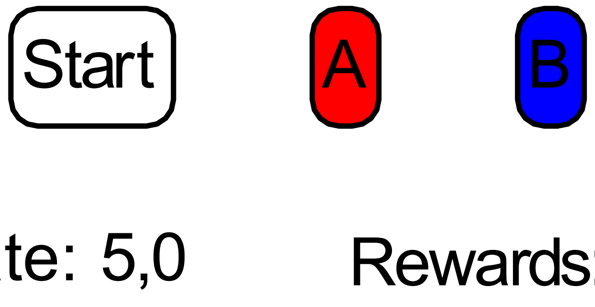
reinforcement learning!

Mystery game

Example: mystery buttons

For each round $r = 1, 2, \ldots$

- You choose A or B.
- You move to a new state and get some rewards.



Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

From MDPs to reinforcement learning

visit wpclipart for original image
http://www.wpclipart.com/people/male/thinker

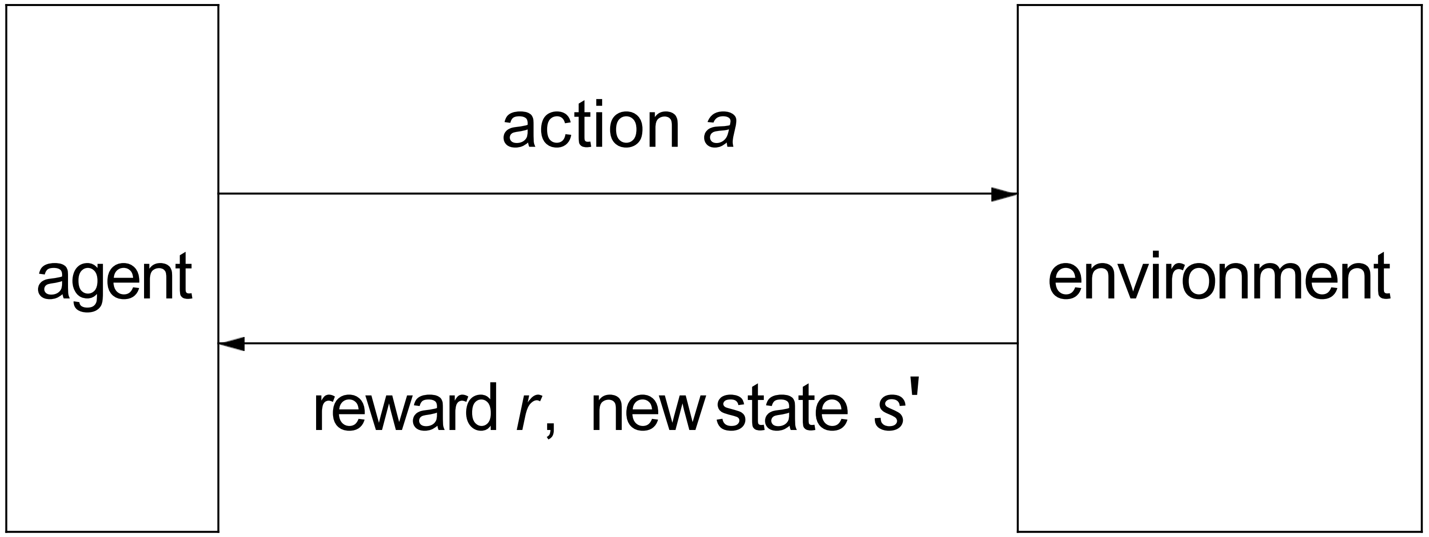
Markov decision process (offline)

- Have mental model of how the world works.
- Find policy to collect maximum rewards.

Reinforcement learning (online)

- Don’t know how the world works.
- Perform actions in the world to find out and collect rewards.

Reinforcement learning framework



Algorithm: reinforcement learning template

```
For $t = 1, 2, 3, \ldots$
    Choose action $a_t = \pi_{\text{act}}(s_{t-1})$ (**how?**)
    Receive reward $r_t$ and observe new state $s_t$
    Update parameters (**how?**)
```

Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

Model-based Monte Carlo

Data: $s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$

Key idea: model-based learning

Estimate the MDP: $T(s, a, s')$ and Reward(s, a, s')

Transitions:

$$T(s, a, s') = \frac{\#\text{ times } (s, a, s') \text{ occurs}}{\#\text{ times } (s, a) \text{ occurs}}$$

Rewards:

$$\text{Reward}(s, a, s') = \text{average of } r \text{ in } (s, a, r, s')$$

Model-based Monte Carlo

Example: model-based Monte Carlo

Data (following policy π):

S1; A, 3, **S2**; B, 0, **S1**; A, 5, **S1**; A, 7, **S1**

Estimates:

$T(S1, A, S1) = \frac{2}{3}$

$T(S1, A, S2) = \frac{1}{3}$

$\text{Reward}(S1, A, S1) = \frac{1}{2}(5 + 7) = 6$

$\text{Reward}(S1, A, S2) = 3$

Estimates converge to true values (under certain conditions)

Problem

Data (following policy π):

S1; A, 3, **S2**; B, 0, **S1**; A, 5, **S1**; A, 7, **S1**

Problem: won't even see (s, a) if $a \neq \pi(s)$

Key idea: exploration

To do reinforcement learning, need to explore the state space.

Solution: need π to **explore** explicitly (more on this later)

From model-based to model-free

$$\hat{Q}_{\text{opt}}(s,a) = \sum_{s'} \hat{T}(s,a,s')[\widehat{\text{Reward}}(s,a,s') + \gamma \hat{V}_{\text{opt}}(s')]$$

All that matters for prediction is (estimate of) $Q_{\text{opt}}(s,a)$

Key idea: model-free learning

Try to estimate $Q_{\text{opt}}(s,a)$ directly.

Model-free Monte Carlo

Data (following policy π):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$$

Recall:

$Q_\pi(s, a)$ is expected utility starting at s, first taking action a, and then following policy π

Utility:

$$u_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \cdots$$

Estimate:

$Q_\pi(s, a) =$ average of u_t where $s_{t-1} = s, a_t = a$

Model-free Monte Carlo

Example: model-free Monte Carlo

Data:

S1; A, 3, **S2**; B, 0, **S1**; A, 5, **S1**; A, 7, **S1**

Estimates (assume $\gamma = 1$):

$$\hat{Q}_\pi(S1, A) = \frac{1}{3}(15 + 12 + 7) \approx 11.33$$

Caveat: converges, but still need to follow π that explores

Note: we are estimating Q_π now, not Q_{opt}

total rewards resulting from (s, a)

Model-free Monte Carlo (equivalences)

Data (following policy π):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$$

Original formulation

$\hat{Q}_\pi(s,a) =$ average of u_t where $s_{t-1} = s, a_t = a$

Equivalent formulation (convex combination)

On each (s,a,u):

$$\eta = \frac{1}{1+(\#\text{ updates to }(s,a))}$$

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta u$$

Interpolation view

Model-free Monte Carlo (equivalences)

Equivalent formulation (convex combination)

On each (s, a, u):

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$$

Equivalent formulation (stochastic gradient)

On each (s, a, u):

$$\hat{Q}_\pi(s, a) \leftarrow \hat{Q}_\pi(s, a) - \eta[\underbrace{\hat{Q}_\pi(s, a)}_{\text{prediction}} - \underbrace{u}_{\text{target}}]$$

Implied objective: least squares regression

$$(\hat{Q}_\pi(s, a) - u)^2$$

stochastic gradient descent view

Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

SARSA

Data (following policy π):

$$s_0;\ a_1, r_1, s_1;\ a_2, r_2, s_2;\ a_3, r_3, s_3;\ \ldots;\ a_n, r_n, s_n$$

Algorithm: model-free Monte Carlo updates

When receive (s, a, u):

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta \underbrace{u}_{\text{data}}$$

Algorithm: SARSA

When receive (s, a, r, s', a'):

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta[\underbrace{r}_{\text{data}} + \gamma \underbrace{\hat{Q}_\pi(s',a')}_{\text{estimate}}]$$

Comparison

$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$

Key idea: bootstrapping

SARSA uses estimate $Q_\pi(s, a)$ instead of just raw data u.

- u is only based on one path, so could have large variance, need to wait until end
- $Q_\pi(s', a')$ based on estimate, which is more stable, update imme-diately

Question

Which of the following algorithms allows you to estimate $Q_{\text{opt}}(s, a)$ (select all that apply)?

model-based Monte Carlo

model-free Monte Carlo

SARSA

Q-learning

Problem: model-free Monte Carlo and SARSA only estimate Q_π, but want Q_{opt} to act optimally

Output	**MDP**	**reinforcement learning**
Q_π	policy evaluation	model-free Monte Carlo, SARSA
Q_{opt}	value iteration	**Q-learning**

Q-learning

MDP recurrence:

$$Q_{\text{opt}}(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s')]$$

Algorithm: Q-learning [Watkins/Dayan, 1992]

On each (s, a, r, s'):

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow (1 - \eta)\underbrace{\hat{Q}_{\text{opt}}(s, a)}_{\text{prediction}} + \eta\underbrace{(r + \gamma\hat{V}_{\text{opt}}(s'))}_{\text{target}}]$$

Recall: $\hat{V}_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a')$

Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

Exploration

Algorithm: reinforcement learning template

```
For $t = 1, 2, 3, \ldots$
    Choose action $a_t = \pi_{\text{act}}(s_{t-1})$ (how?)
    Receive reward $r_t$ and observe new state $s_t$
    Update parameters (how?)
```

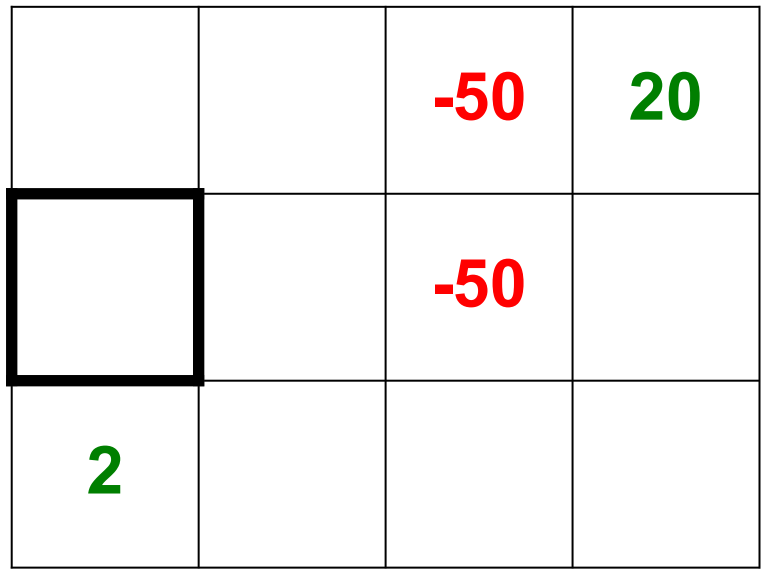
$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$

Which **exploration policy** π_{act} to use?

No exploration, all exploitation

Attempt 1: Set $\pi_{\text{act}}(s) = \arg\max_{a \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s, a)$

Problem: $\hat{Q}_{\text{opt}}(s, a)$ estimates are inaccurate, **too greedy**!



No exploitation, all exploration

Attempt 2: Set $\pi_{\text{act}}(s)$ = random from Actions(s)

Problem: average utility is low because exploration is **not guided**

Exploration/exploitation tradeoff

Key idea: balance

Need to balance **exploration** and **exploitation**.

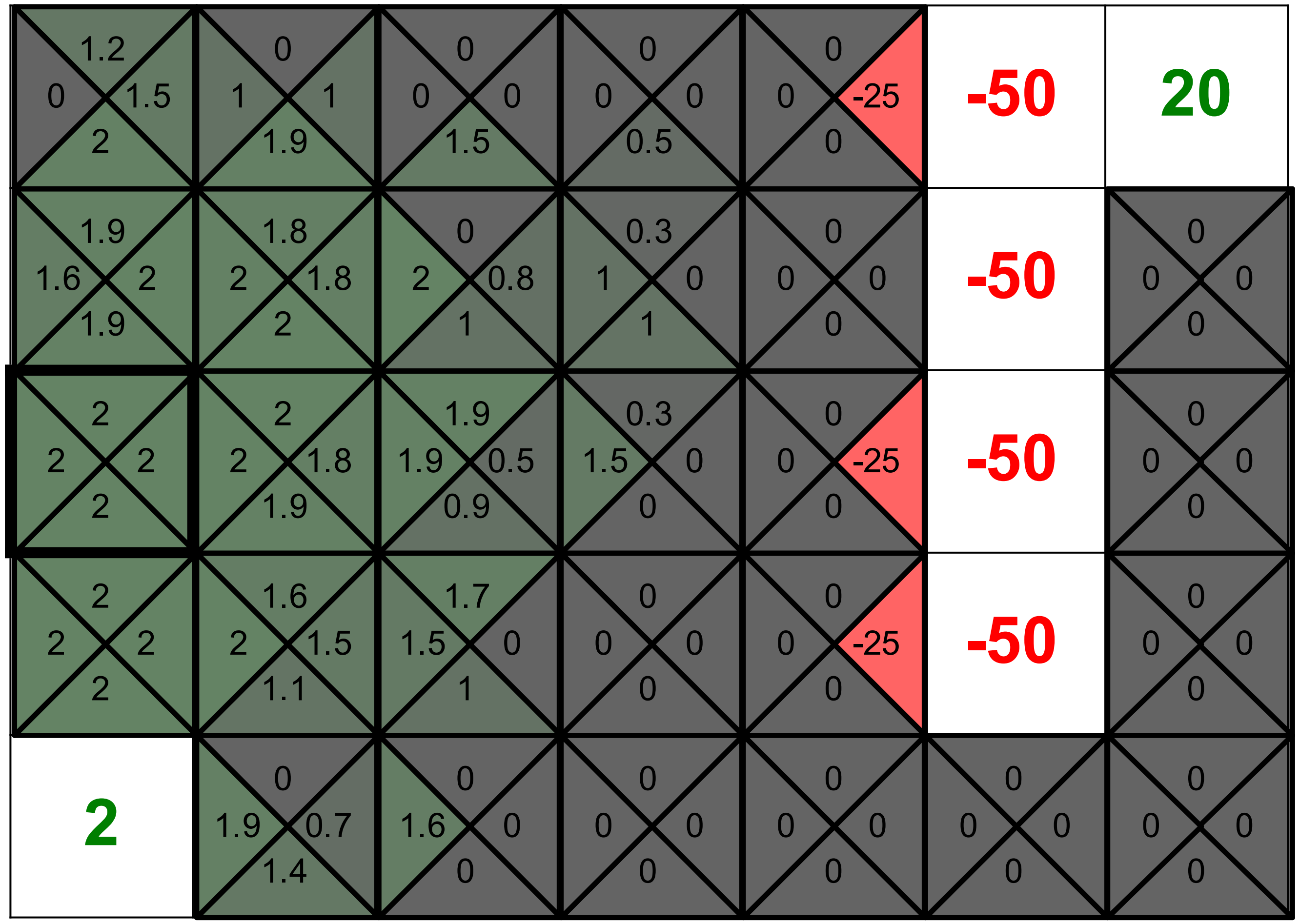
Algorithm: epsilon-greedy policy

$$\pi_{\text{act}}(s) = \begin{cases} \arg\max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s,a) & \text{probability } 1-\epsilon, \\ \text{random from Actions}(s) & \text{probability } \epsilon. \end{cases}$$

Examples from life: restaurants, routes, research

Generalization

Problem: large state spaces, hard to explore



Q-learning

Stochastic gradient update:

$$\hat{Q}_{\text{opt}}(s,a) \leftarrow \hat{Q}_{\text{opt}}(s,a) - \eta[\underbrace{\hat{Q}_{\text{opt}}(s,a)}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}]$$

This is **rote learning**: every $\hat{Q}_{\text{opt}}(s,a)$ has a different value

Problem: doesn't generalize to unseen states/actions

Function approximation

Key idea: linear regression model

Define **features** $\varphi(s, a)$ and **weights** $\mathbf{w}$:

$$Q_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \varphi(s, a)$$

Example: features for volcano crossing

$\varphi_1(s, a) = \mathbf{1}[a = \mathrm{W}]$ $\qquad$ $\varphi_7(s, a) = \mathbf{1}[s = (5, *)]$

$\varphi_2(s, a) = \mathbf{1}[a = \mathrm{E}]$ $\qquad$ $\varphi_8(s, a) = \mathbf{1}[s = (*, 6)]$

... $\qquad$ ...

Function approximation

Algorithm: Q-learning with function approximation

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}]\phi(s, a)$$

Implied objective function:

$$(\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}})^2$$

Covering the unknown

Epsilon-greedy: balance the exploration/exploitation tradeoff

Function approximation: can generalize to unseen states

Summary so far

- Online setting: learn and take actions in the real world!
- Exploration/exploitation tradeoff
- Monte Carlo: estimate transitions, rewards, Q-values from data
  (approximating an expectation with a sample)
- Bootstrapping: update towards target that depends on estimate rather than just raw data
  (using the model predictions to update itself)

Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

Amount of supervision

Supervised learning (e.g., Perceptron)

input x, output y

offline: get all data

Reinforcement learning (e.g., Q-learning)

state-action-reward-state (s, a, r, s')

online: actively choose actions to get data

Unsupervised learning (e.g., k-means)

inputs x

offline: get all data

more supervision

$\downarrow$

less supervision

Challenges in reinforcement learning

Binary classification (sentiment classification, SVMs):

- Stateless, full feedback

Reinforcement learning (flying helicopters, Q-learning):

- Stateful, partial feedback

Key idea: partial feedback

Only learn about actions you take.

Key idea: state

Rewards depend on previous actions ⇒ can have delayed rewards.

States and information

	stateless	state
full feedback	supervised learning (binary classification)	supervised learning (structured prediction)
partial feedback	multi-armed bandits	$\boxed{\text{reinforcement learning}}$

Deep reinforcement learning

just use a neural network for $Q_{\text{opt}}(s, a)$

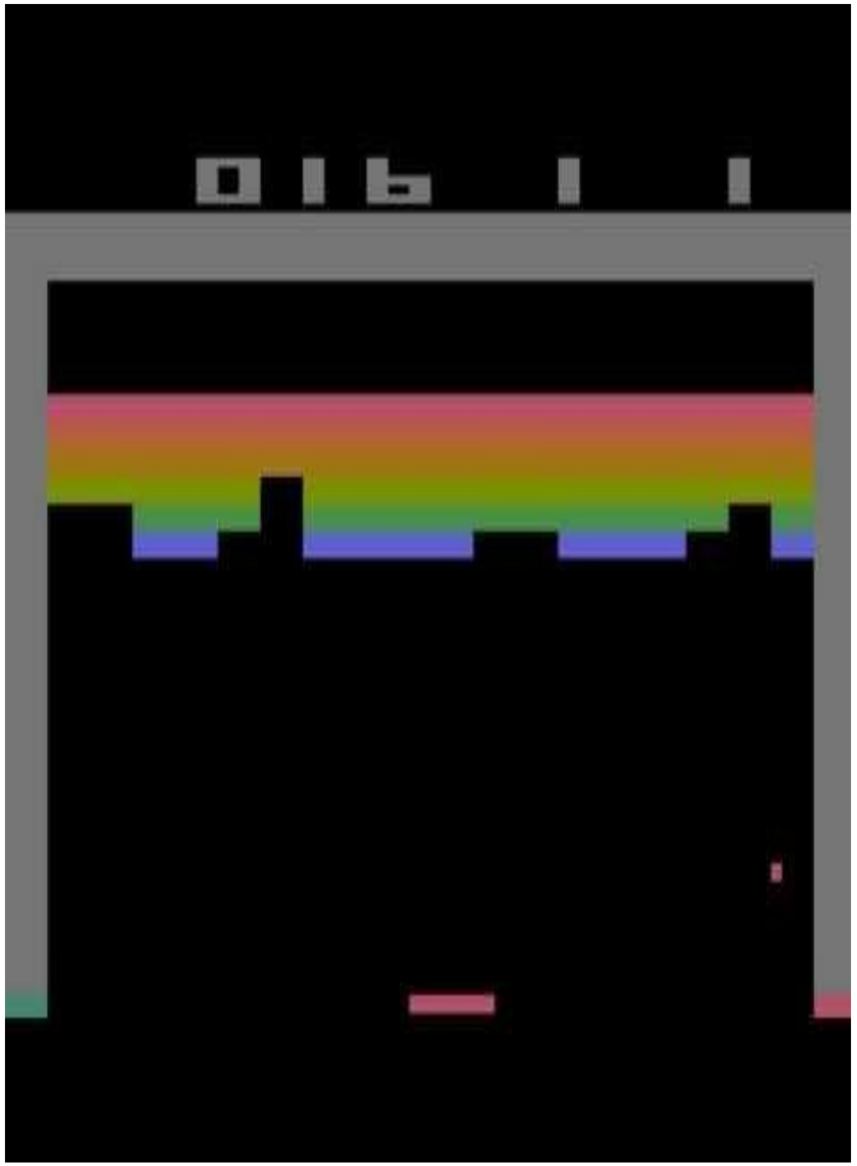
Playing Atari [Google DeepMind, 2013]:

- last 4 frames (images) $\Rightarrow$ 3-layer NN $\Rightarrow$ keystroke
- ϵ-greedy, train over 10M frames with 1M replay memory
- Human-level performance on some games (breakout), less good on others (space invaders)

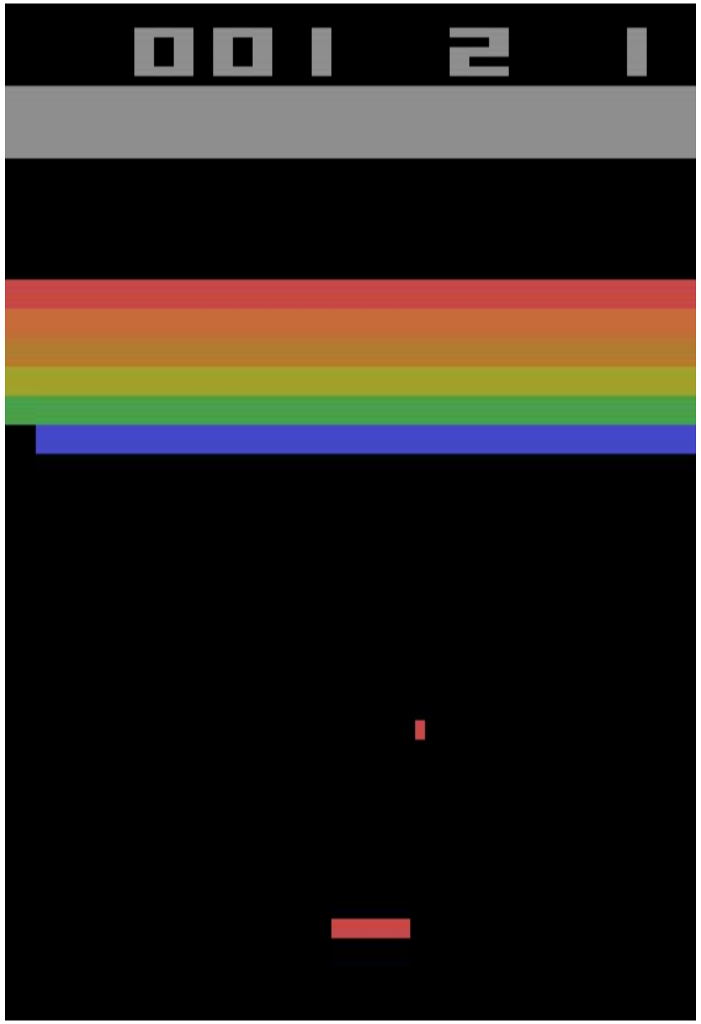
Breakout Game Description

Formally:

- *Actions*
  - move_paddle_left
  - move_paddle_right
  - do_not_move_paddle
- *Rewards*
  - If ball hits brick, reward = 1
  - Otherwise, reward = 0
- *End condition*
  - If ball falls off the screen, game ends



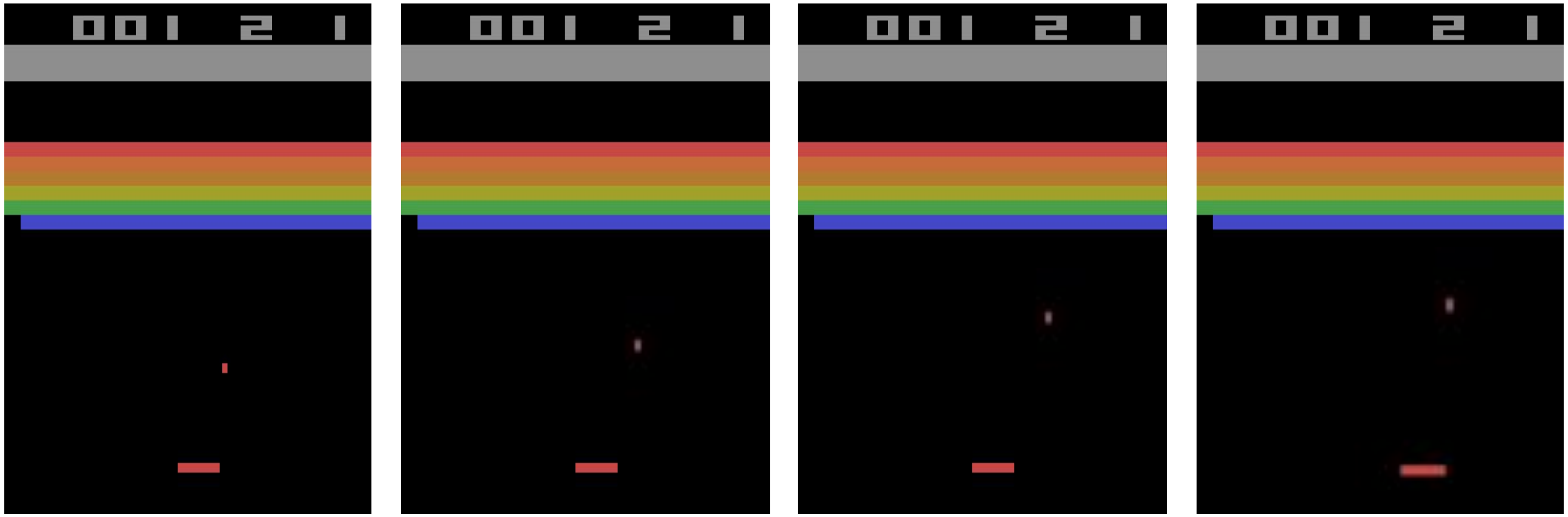
Finding a state representation



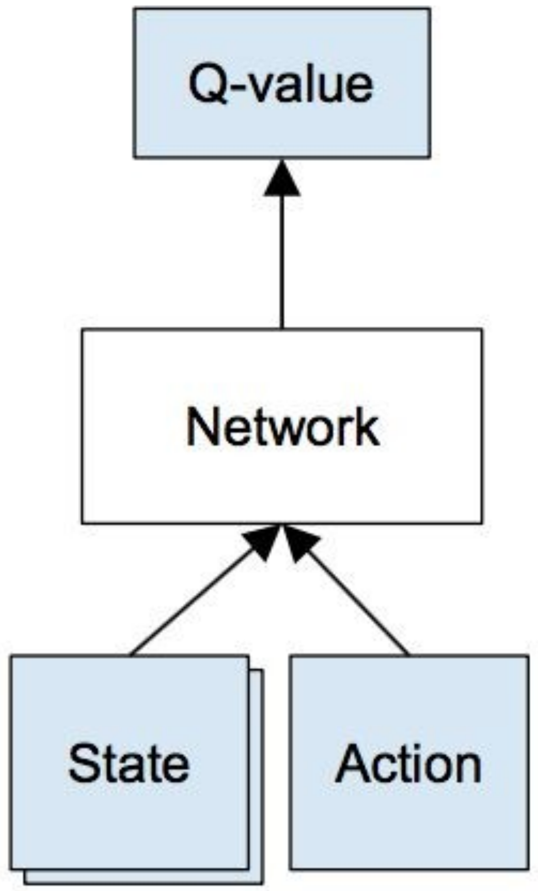
Consider this frame.

- Can you capture information like direction of the ball?
- Can you capture velocity?

Use a small number of consecutive frames for each state.



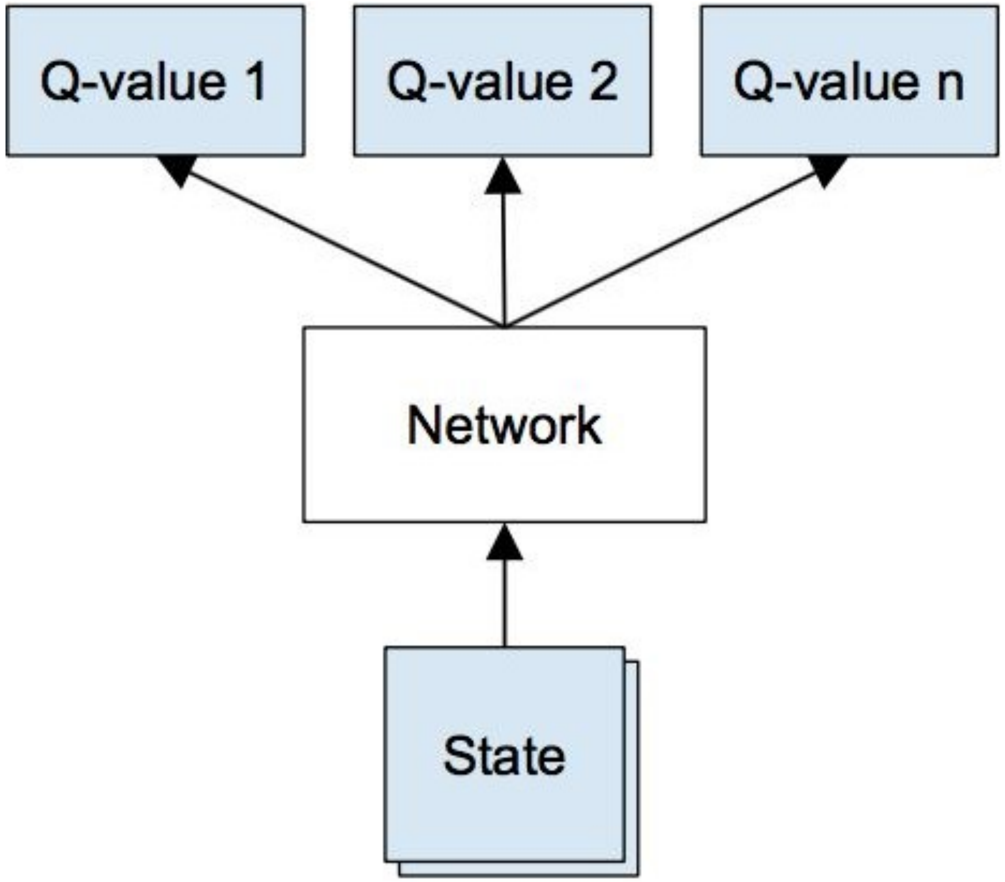
Neural Networks as Q(s, a) approximators



- State and action pair passed as inputs to a neural network.
- Neural network predicts the Q-value for the input action.

Can we make this even more efficient?

Neural Networks as Q(s, a) approximators



- State is the only input into the neural network.
- Network outputs a Q-value for every possible action.
- Action corresponding to the highest Q-value is chosen.

Training Deep-Q-Networks

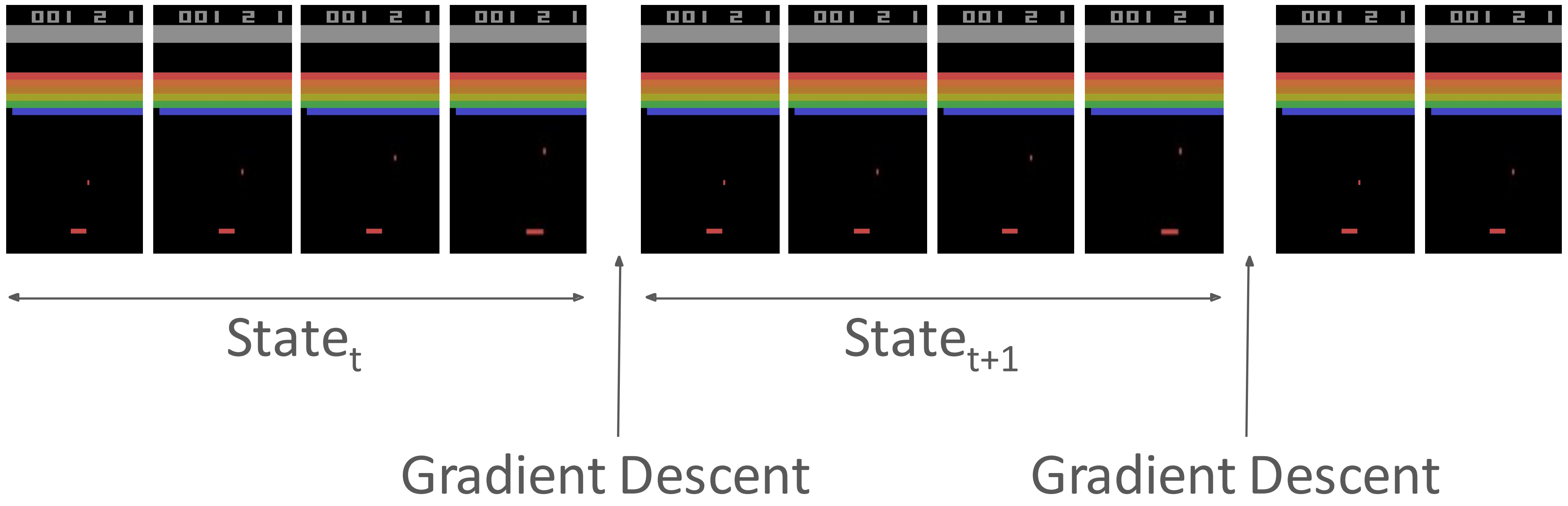
- Initialize weights randomly.
- Loop:
    - Obtain current state (s)
    - Run Neural Network on s to obtain Q-value for every action.
    - Execute action (a) that maximizes Q-value.
    - Obtain reward (r) and new state (s’).
    - Perform gradient descent on Q-learning loss using (s, a, r, s’)

Training Deep-Q-Networks

- Initialize weights randomly.
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action.
  - Execute action (a) that maximizes Q-value.
  - Obtain reward (r) and new state (s').
  - Perform gradient descent on Q-learning loss using (s, a, r, s')

Unstable and inefficient under current data ordering!

Training DQNs can be difficult



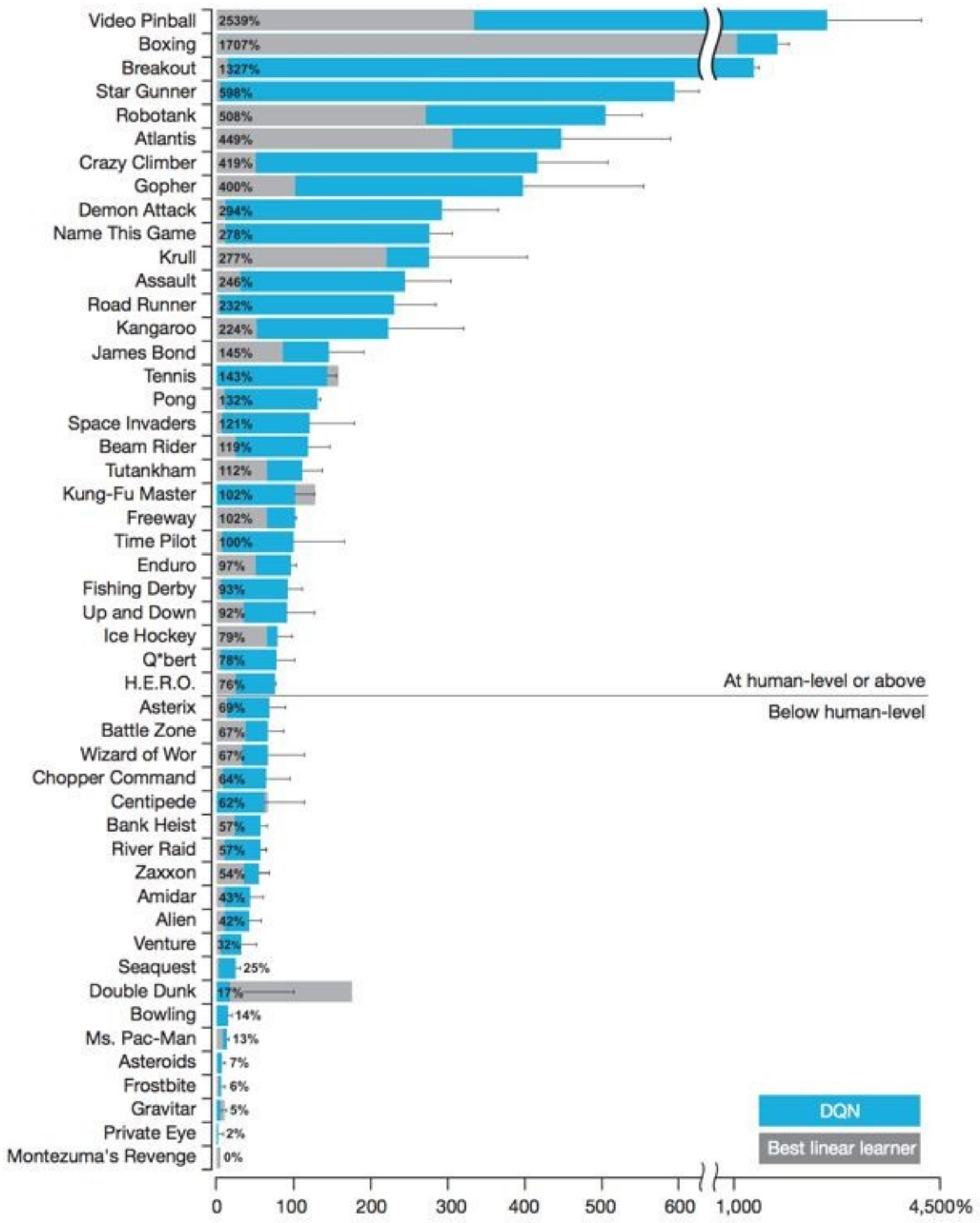
- State$_t$ is highly correlated to State$_{t+1}$
- Gradient descent after consecutive steps ⇒ correlated updates

How do we fix this? Randomly sample states for updates.

Training Deep-Q-Networks

Experience Replay

- Initialize weights randomly.
- Initialize memory (D) with capacity N.
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action.
  - Execute action (a) that maximizes Q-value.
  - Obtain reward (r) and new state (s’).
  - Store (s, a, r, s’) in D
  - Randomly sample $(s, a, r, s')_D$ from D
  - Perform gradient descent on Q-learning loss using $(s, a, r, s')_D$



Comparison of the DQN agent with the best RL methods in the literature

The performance of DQN is normalized w.r.t. A professional human games tester (that is, 100% level) and random play (that is, 0% level).

Source: Mnih et al. (2015)

Deep reinforcement learning

- Policy gradient: train a policy $\pi(a \mid s)$ (say, a neural network) to directly maximize expected reward

- Google DeepMind's AlphaGo (2016)

- Andrej Karpathy's blog post

http://karpathy.github.io/2016/05/31/rl

Applications

Autonomous helicopters: control helicopter to do maneu-vers in the air

Backgammon: TD-Gammon plays 1-2 million games against itself, human-level performance

Elevator scheduling; send which elevators to which floors to maximize throughput of building

Managing datacenters; actions: bring up and shut down machine to minimize time/cost