



MACHINE LEARNING  
University Master's Degree in Intelligent Systems

reinforcement learning

Ramón A. Mollineda Cárdenas

# a quote

"When it is not in our power to  
determine what is true, we ought to  
follow what is most probable."

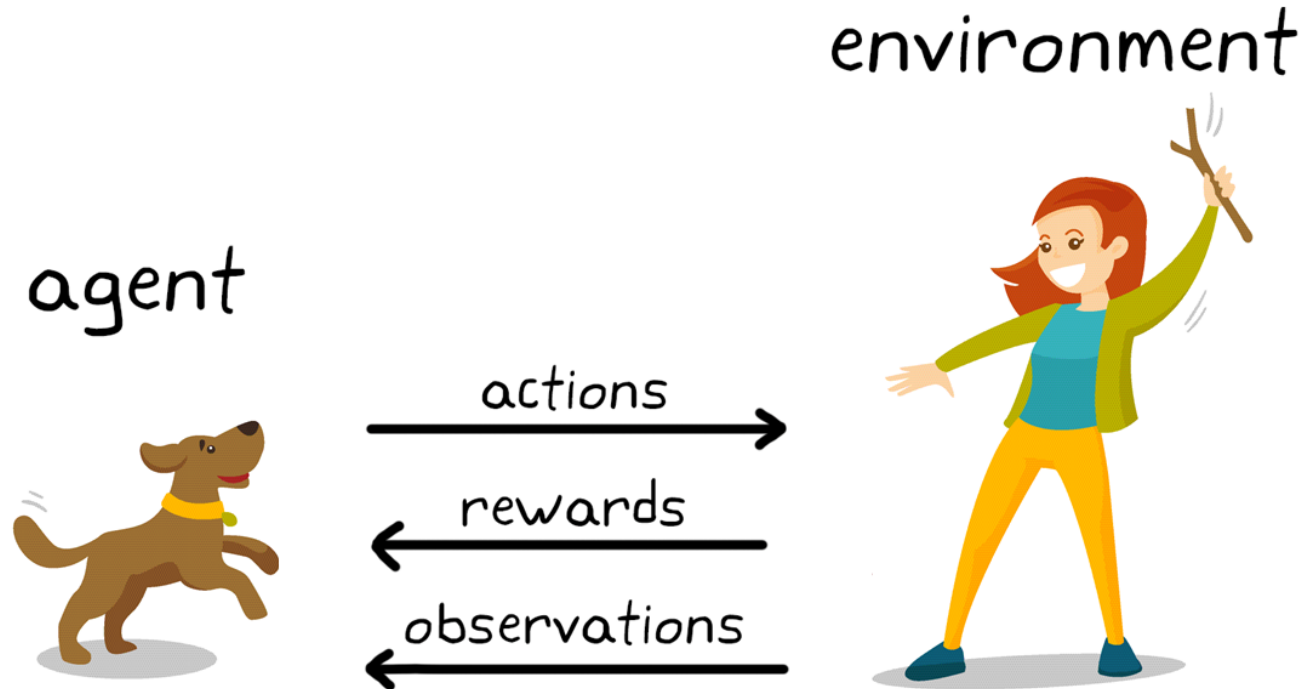
— René Descartes

french philosopher, mathematician and scientist  
1596 – 1650



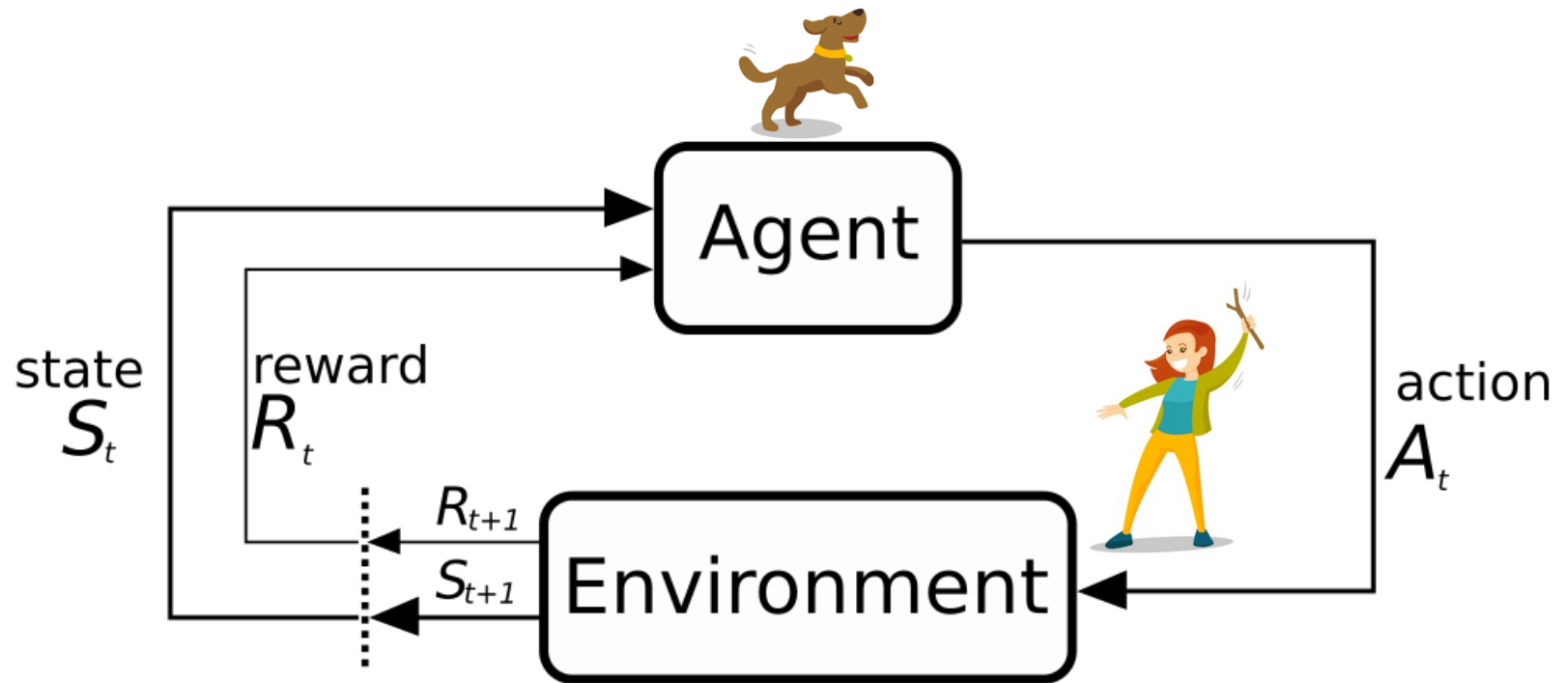
# introduction to reinforcement learning

*use case*



# introduction

## *pipeline*



sample actions, observe rewards, tune the policy

# introduction

## *agent*

- an agent is an entity that interacts with an environment
- it receives information from sensing the environments
- it interacts with the environment using its actuators
- its interactions are based on a policy (agent's brain)

# introduction

## *use case*

A mobile robot decides whether it should enter a new room to collect more trash or try to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly it was able to find the recharger in the past.

goal?

agent?

state?

action?

environment?

short-term rewards?

long-term rewards?

# introduction

*goal*

RL approach involves learning how  
to map states to actions,  
so as to gain  
the highest long-term cumulative reward.

# introduction

## *RL summary*

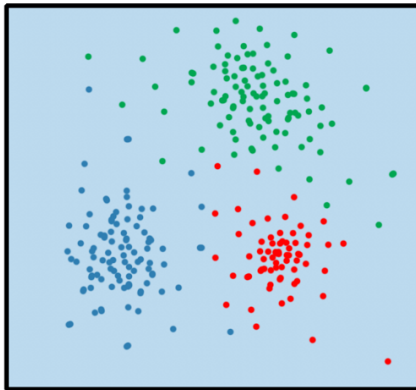
1. The agent observes the environment state (environment observation)
2. The agent decides how to act base on some strategy (policy) that maps states into actions; at the beginning, through trial and error
3. The agent acts according to the policy
4. The agent receives a reward or penalty (based on sensor reading)
5. Learning from the experiences and refining the strategy (policy)
6. Iterate until an **optimal strategy** is found



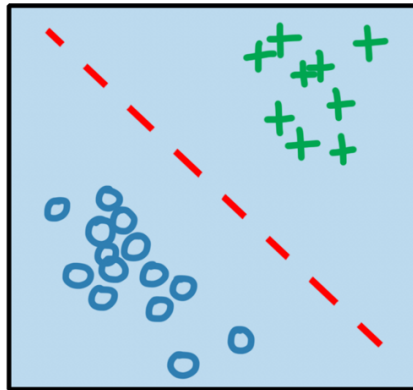
# introduction

## *learning paradigms*

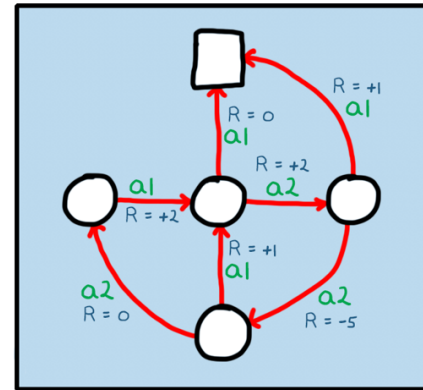
unsupervised  
learning



supervised  
learning



reinforcement  
learning



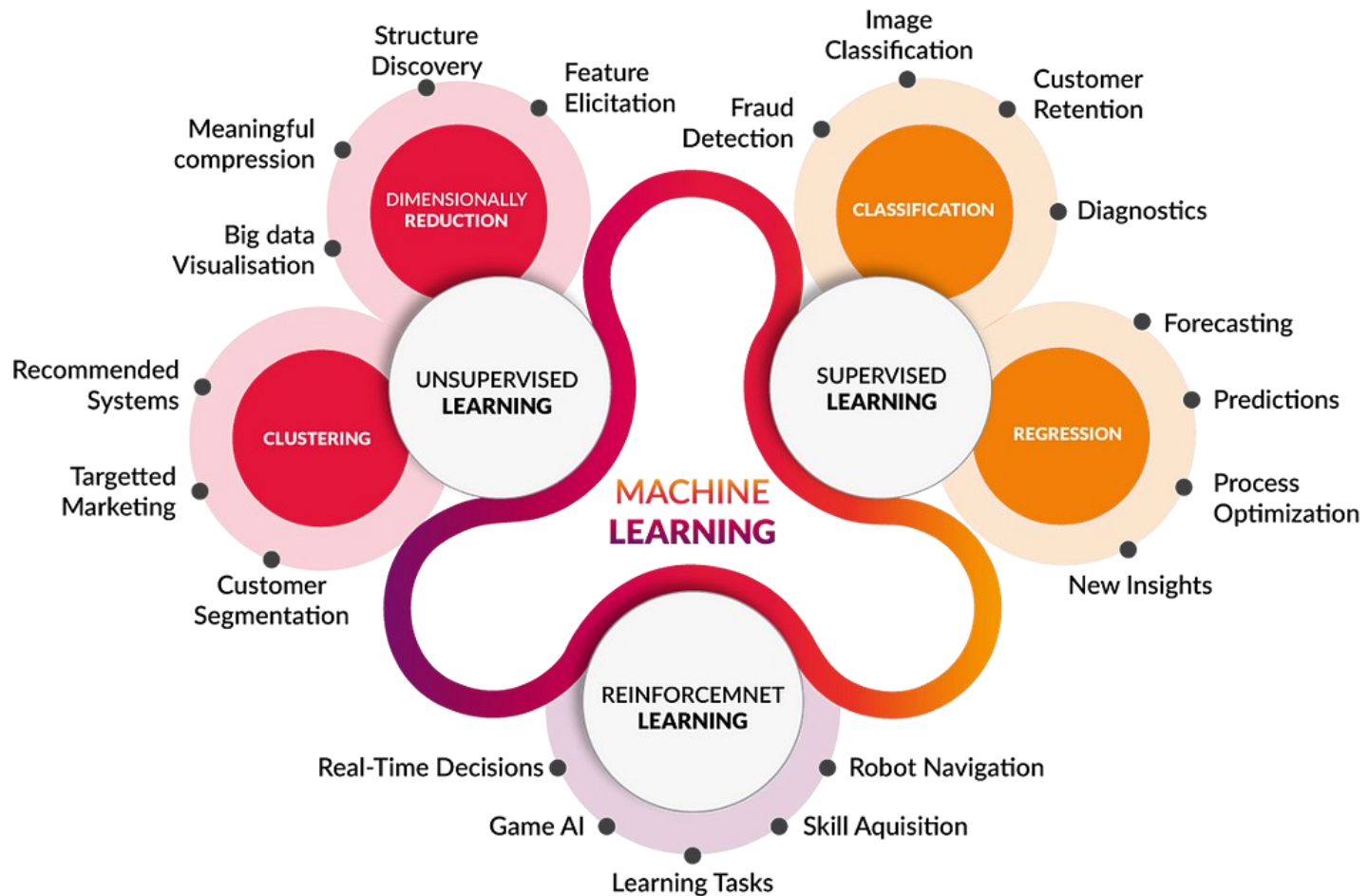
# introduction

## *learning paradigms*

- **unsupervised** learning...
  - discovers similarities between data samples
  - identifies clusters based on similarities
  - is able to detect anomalous data
- **supervised** learning...
  - learns a mapping from instances to data labels
  - requires a labelled / annotated dataset
  - uses labels to guide, correct and monitor the learning process
- **reinforcement** learning...
  - given a state, there exists an optimal action, but it is not known!
  - takes actions based on short- and long-term reward (strategy learning)
  - is guided by sparse and uncertain feedback / supervision
  - does not require collecting, preprocessing and labeling data before training

# introduction

## *learning paradigms*



# introduction

## *key principles*

- it is a closed-loop problem: system's actions affect later inputs.
- actions may affect not only the immediate reward but also the next states and, through that, all subsequent rewards.
- the agent is not taught what action to take on each state (as in supervised learning); instead, it must discover which action maximizes immediate and future rewards (by interacting with the environment).

# introduction

*exploitation – exploration dilemma / trade-off*

exploit known actions

that have proven to be effective (positive rewards)

OR

explore new (unknown) actions

to discover more promising solutions

# introduction

## *key elements*

- a policy  $\pi$ : the brain of a RL agent
  - a mapping from perceived states to actions to be taken
  - a stimulus-response rule
- a reward signal: what is good in an immediate sense
  - $r : S \times A \rightarrow \mathbb{R}$
  - the agent cannot alter  $r$
  - measures the effectiveness/quality of the agent's interaction
- a value function: what is good in the long run (accumulated rewards)
  - maps a state to the expected return (cumulative reward) when starting from that state
  - measures the quality of (some sequence of) actions taken from a state
  - policy tuning is based on value judgments
- a model of the environment: it mimics the behavior of the environment
  - given a state and action, it moves to the next state and provides a reward

# policy

*agent's goal*

objective function over an episode

$$\sum_{t=0}^{\infty} \gamma^t r(s(t), a(t))$$

$t$ : time step

$r$ : reward function

$\gamma$ : discount factor

$s(t)$ : state at a given time step/epoch  $t$

$a(t) = \pi(s(t))$

$$\sum_{t=0}^{\infty} \gamma^t r(s(t), \pi(s(t)))$$

policy

*agent's goal*

**WANTED**

$\pi$

**฿1.13**



policy

*agent's goal*

$$\pi^* = \operatorname{argmax}_{\pi} \sum_{t=0}^{\infty} \gamma^t r(s(t), \pi(s(t)))$$

$$\pi : \mathcal{S} \rightarrow \mathcal{A}$$

$\mathcal{S} \sim$  state space

$\mathcal{A} \sim$  action space

# RL paradigms

**tabular solutions:** the action-value function is represented as an array or a table

**approximate, gradient-descent, solutions:** the state-action spaces involve continuous domains or complex representations (e.g., an image)

# tabular solutions

## *representative methods*

### tabular solutions

- Q-learning: off policy, model-free RL
- State-Action-Reward-State-Action (SARSA): on policy, model-free RL

# Q-learning\*

*off policy, model-free RL algorithm*

- $Q$ -learning approximates the optimal action-value function  $Q^*$  that maximizes the expected value over all successive steps\*\*

$$Q: S \times A \rightarrow \mathbb{R}$$

- $Q$ -learning does not require an environment model (model-free)
- $\pi$  (policy) depends on  $Q$
- $Q$  depends on  $\pi$  ( $\pi$  determines which state-action pair should be updated)
- $Q$ 's update (learning) does not depend on  $\pi$  (off policy)

\* Watkins, C.J.C.H., Dayan, P. Q-learning. *Mach Learn* 8, 279–292 (1992). <https://doi.org/10.1007/BF00992698>.

\*\*  $Q$  has been shown to converge with probability 1 to  $Q^*$  under minimal requirements.

# Q-learning

## *Bellman equation*

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a))$$

where...

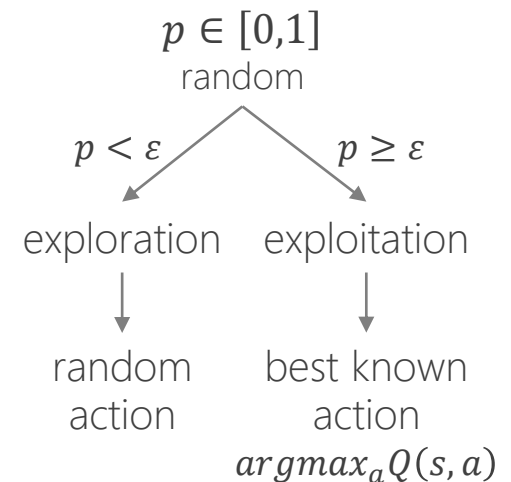
- $a_t = \pi(s_t)$
- $r_{t+1}$  is the reward associated to the state-action pair  $(s_t, a_t)$
- $s_{t+1}$  is the new state after selecting the action  $a_t$  at state  $s_t$
- $\alpha$  is the learning rate ( $0 \leq \alpha \leq 1$ ); the extent to which our  $Q$ -values are being updated in every iteration
- $\gamma$  is the discount factor ( $0 \leq \gamma \leq 1$ ); relevance of future rewards (uncertain)
- $\max_a Q(s_{t+1}, a)$  is the maximum value from state  $s_{t+1}$  (given the current  $Q$ )

# Q-learning

## *off-policy learning*

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left( r_{t+1} + \underbrace{\gamma \cdot \max_a Q(s_{t+1}, a)} \right)$$

- the behavior policy (used to collect data) is different from the policy being optimized
  - $Q$ 's update (learning) does not involve the behavior policy  $\pi$
- $Q$ 's update depends on  $\max_a Q(s_{t+1}, a)$ , a greedy action from  $s_{t+1}$ 
  - it is usually assumed an  $\epsilon$ -greedy behavior policy  $\pi$



# Q-learning

## *Bellman equation: remarks*

- the Bellman equation is a recursion that computes the weighted average of the current value and a new learned value
- an episode of the algorithm ends when state  $s_{t+1}$  is a final state
- action values are finite even when no final state exists (provided that  $\gamma < 1$ )
- for final states  $s_f$ ,  $Q(s_f, a)$  is set to the reward observed and is never updated
- $\alpha$  determines to what extent newly acquired information overrides old information
  - $\alpha = 0$  makes the the agent learn nothing (exclusively exploiting prior knowledge)
  - $\alpha = 1$  makes the agent ignore prior knowledge (to explore new possibilities)
- $\gamma$  determines the importance of future rewards
  - $\gamma = 0$  makes the agent "myopic"
  - $\gamma \geq 1$  the action values might diverge
- Q-learning fails with an infinite (or even large) number of states/actions

# Q-learning

*algorithm*

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$   
Repeat (for each episode):  
    Initialize  $S$   
    Repeat (for each step of episode):  
        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)  
        Take action  $A$ , observe  $R, S'$   
         $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$   
         $S \leftarrow S'$ ;  
    until  $S$  is terminal



# SARSA: State-Action-Reward-State-Action\*

*on policy, model-free RL algorithm*

- SARSA estimates  $Q_{\pi}(s, a)$  for the current agent policy  $\pi$  and for all state-action pairs  $(s, a)$ .

$$Q_{\pi}: S \times A \rightarrow \mathbb{R}$$

- SARSA does not require an environment model (model-free)
- $\pi$  depends on  $Q$ ;  $Q$  depends on  $\pi$  ( $\pi$  determines which state-action pair should be updated), and  $Q$ 's update does depend on  $\pi$  (on policy)
- SARSA explicitly uses the transition from one state-action pair to the next state-action pair (not just the state) in its learning process

\* RUMMERY, G.A.; NIRANJAN, M. On-line Q-learning using connectionist systems. Univ. of Cambridge, Dept. of Engineering, UK, 1994.

# SARSA: State-Action-Reward-State-Action

*Bellman equation*

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1}))$$

where...

- $a_t = \pi(s_t)$ ,  $a_{t+1} = \pi(s_{t+1})$
- $r_{t+1}$  is the reward associated to the state-action pair  $(s_t, a_t)$
- $s_{t+1}$  is the new state after selecting the action  $a_t$  at state  $s_t$
- $\alpha$  is the learning rate ( $0 \leq \alpha \leq 1$ )
- $\gamma$  is the discount factor ( $0 \leq \gamma \leq 1$ )
- $Q(s_{t+1}, a_{t+1})$  is the accumulated reward determines by the current  $\pi$

# SARSA: State-Action-Reward-State-Action

*on-policy learning*

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left( r_{t+1} + \gamma \cdot \underbrace{Q(s_{t+1}, a_{t+1})}_{a_{t+1} = \pi(s_{t+1})} \right)$$

- $Q$ 's update does depend on the current (agent) behavior policy  $\pi$

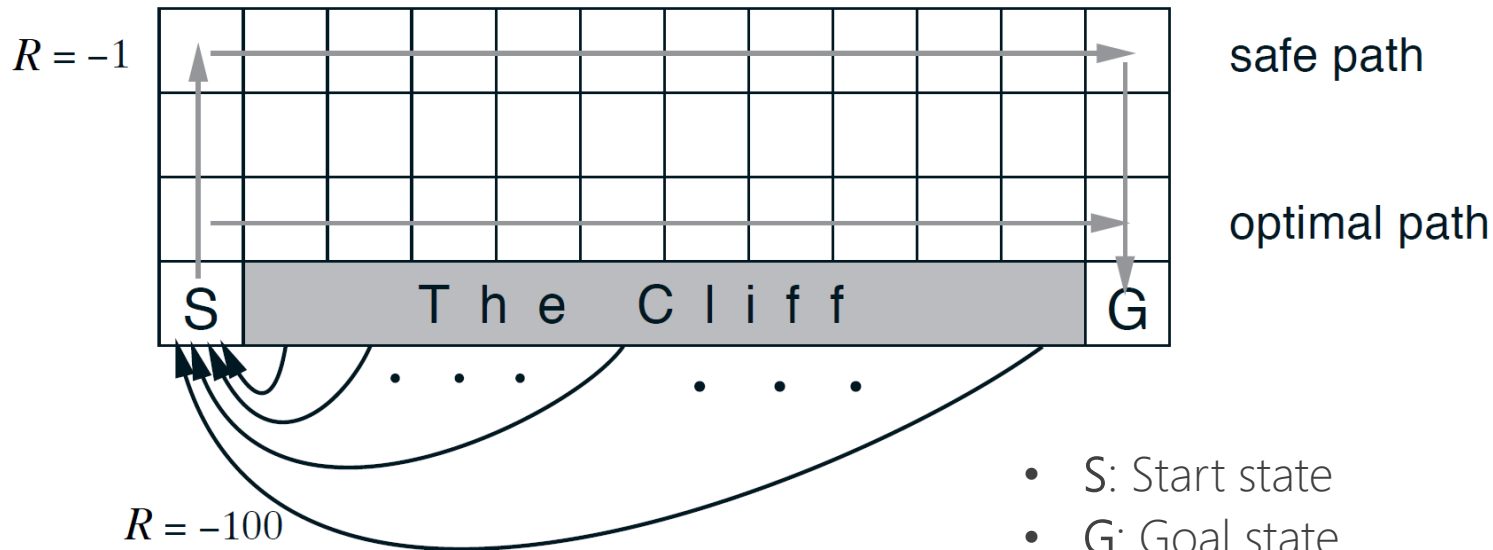
# SARSA: State-Action-Reward-State-Action *algorithm*

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$   
Repeat (for each episode):  
    Initialize  $S$   
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)|  
    Repeat (for each step of episode):  
        Take action  $A$ , observe  $R, S'$   
        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)  
         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$   
         $S \leftarrow S'; A \leftarrow A';$   
    until  $S$  is terminal

# Q-learning versus SARSA

## *Cliff walking example*

which path do you think was learned by each method?

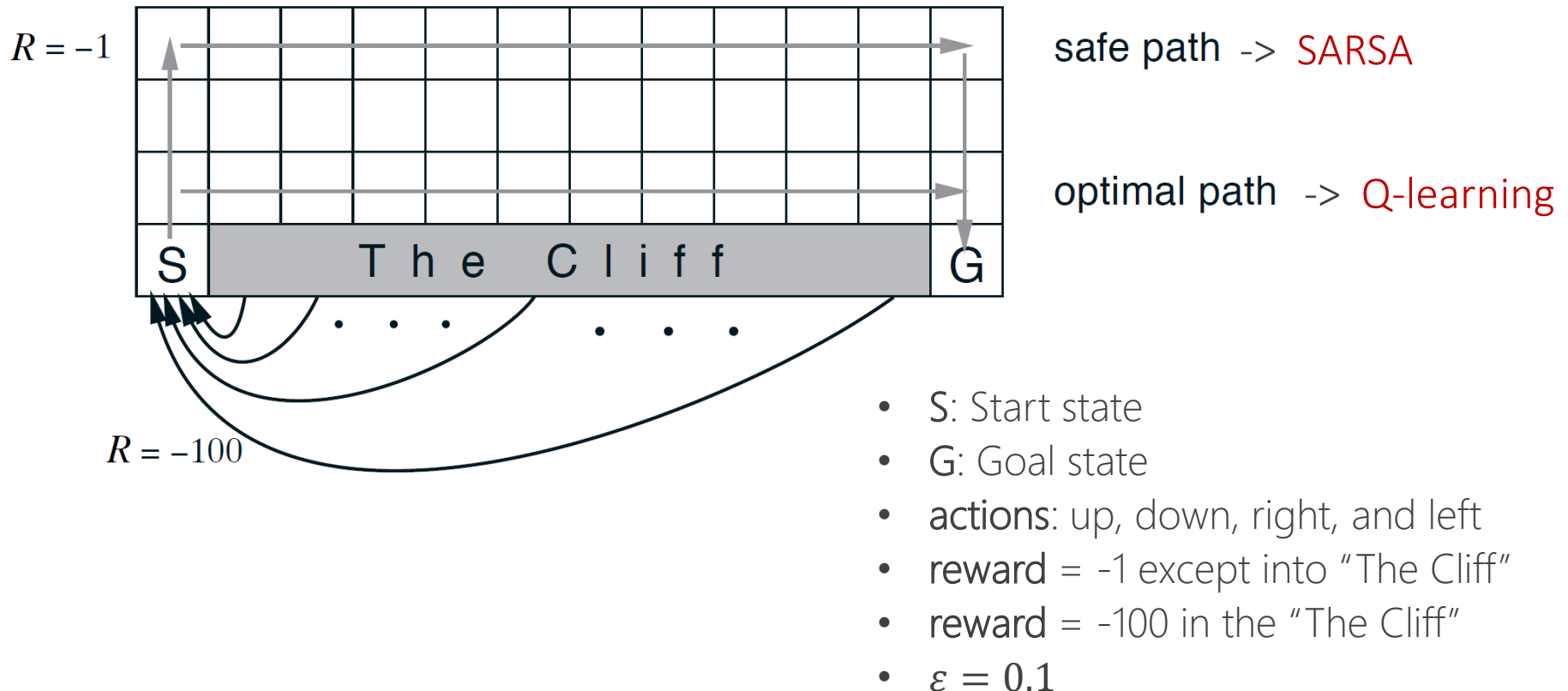


- S: Start state
- G: Goal state
- actions: up, down, right, and left
- reward = -1 except into "The Cliff"
- reward = -100 in the "The Cliff"
- $\epsilon = 0.1$

# Q-learning versus SARSA

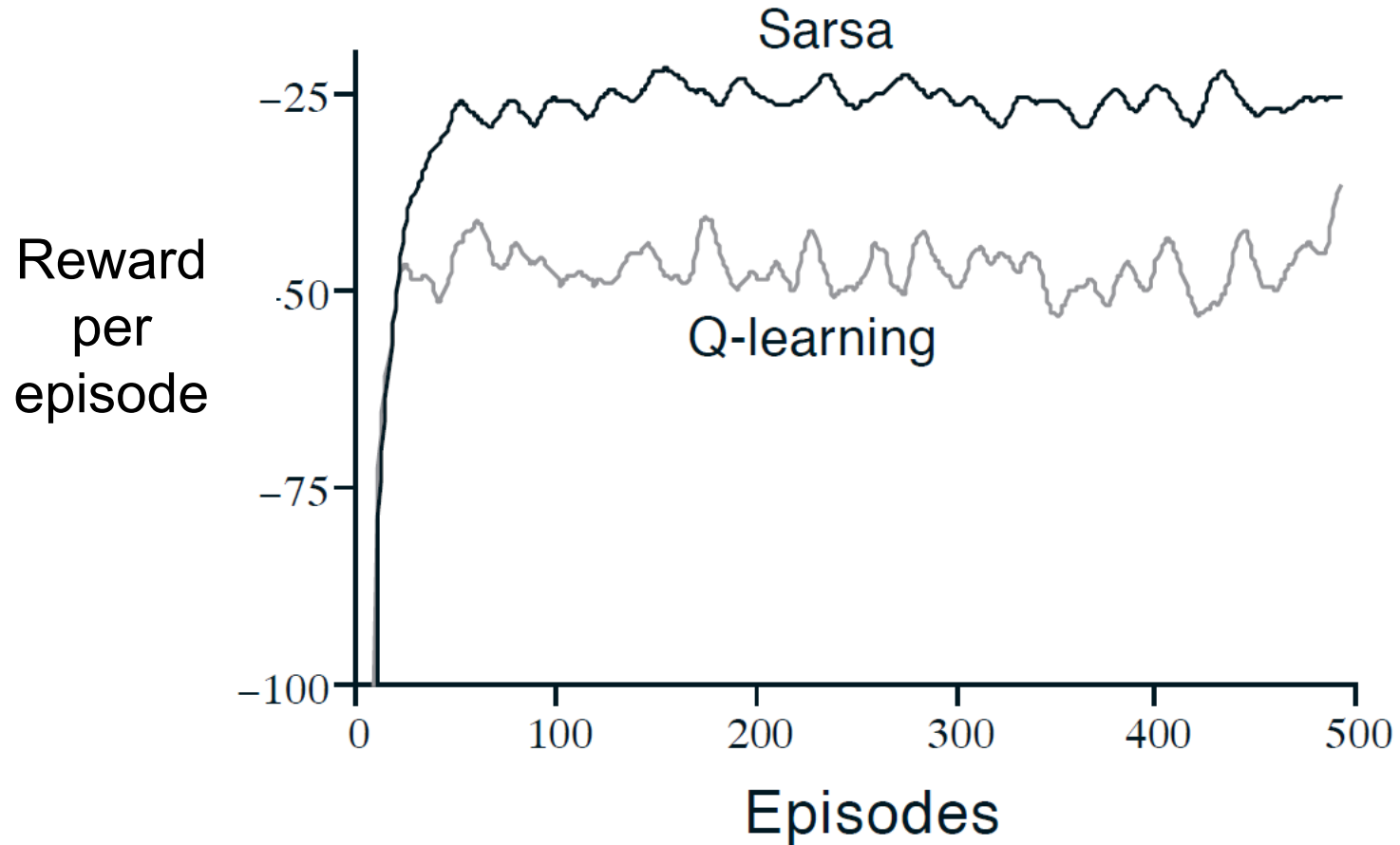
## *Cliff walking example*

which path do you think was learned by each method?



# Q-learning versus SARSA

*Cliff walking example*



# tabular solutions

## *limitations*

- **assumption**: value estimates are represented as a table with one entry for each state or for each state-action pair
- **memory problems**: it is limited to tasks with small numbers of states and actions; if not, there could be memory problems
- **generalization problems**: large tables also need large volumes of data and time to generalize (fill) them properly
- **unobserved states**: in many RL tasks, most states encountered will never have been experienced exactly before

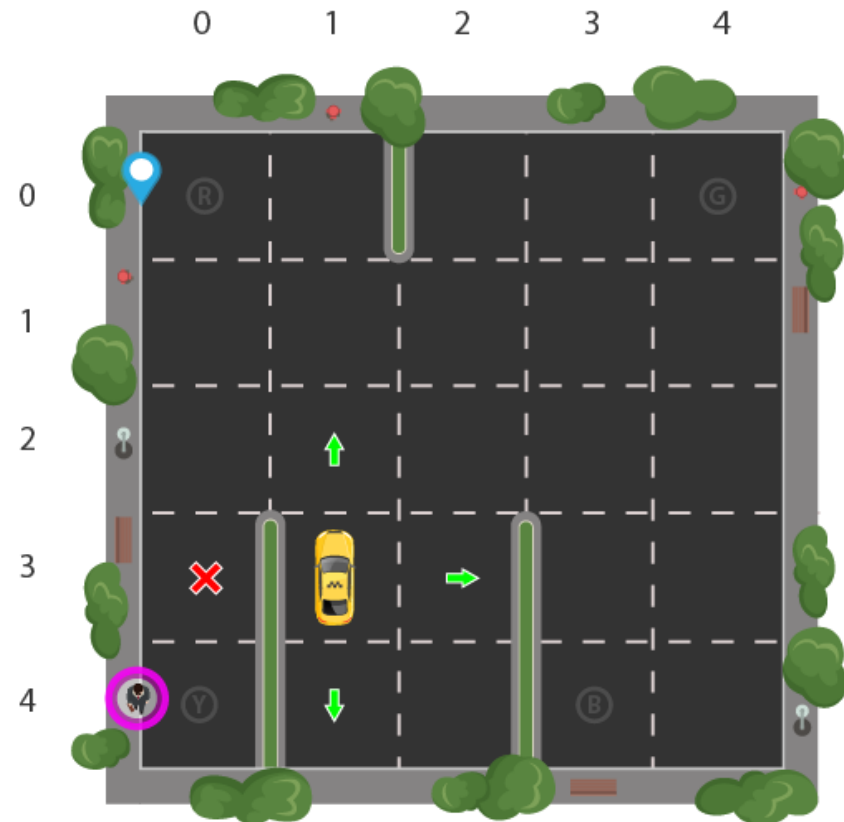


# a self-driving cab

## *case study*

### smart-cab goals are...

- pick up a passenger at one location
- drop off the passenger to the right location
- save time by taking the shortest path
- ensure passenger safety
- comply with rules

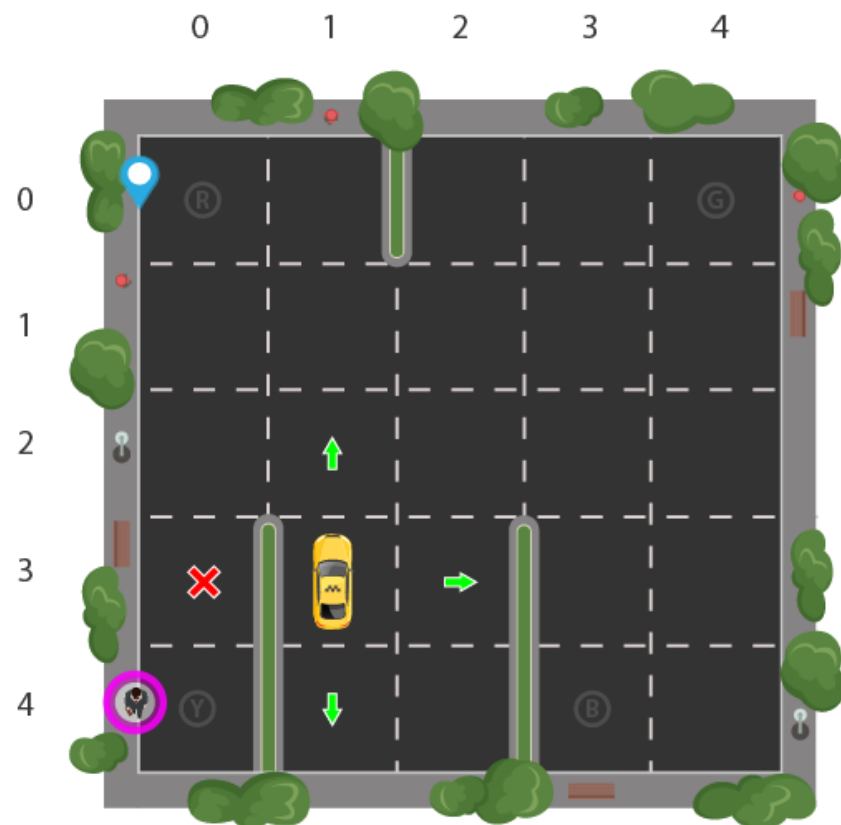


# a self-driving cab

## *state space*

### state space

- taxi locations: a coordinate in a 5 x 5 grid
- passenger locations: 5 = 4 coord. to pick up the passenger (R, G, Y, B) + 1 (taxi location)
- destination locations: 4 (R, G, Y, B)
- state space size =  $5 \times 5 \times 5 \times 4 = 500$  states
- state example: (3, 1, 2, 0) -> 328
  - taxi location: coordinate (3, 1)
  - passenger location: 2 (Y coordinate)
  - destination location: 0 (R coordinate)
  - encoded state:  $328 \in [0, 499]$
  - encoded state =  $((\text{taxi\_row} \times 5 + \text{taxi\_col}) \times 5 + \text{passenger\_location}) \times 4 + \text{destination}$



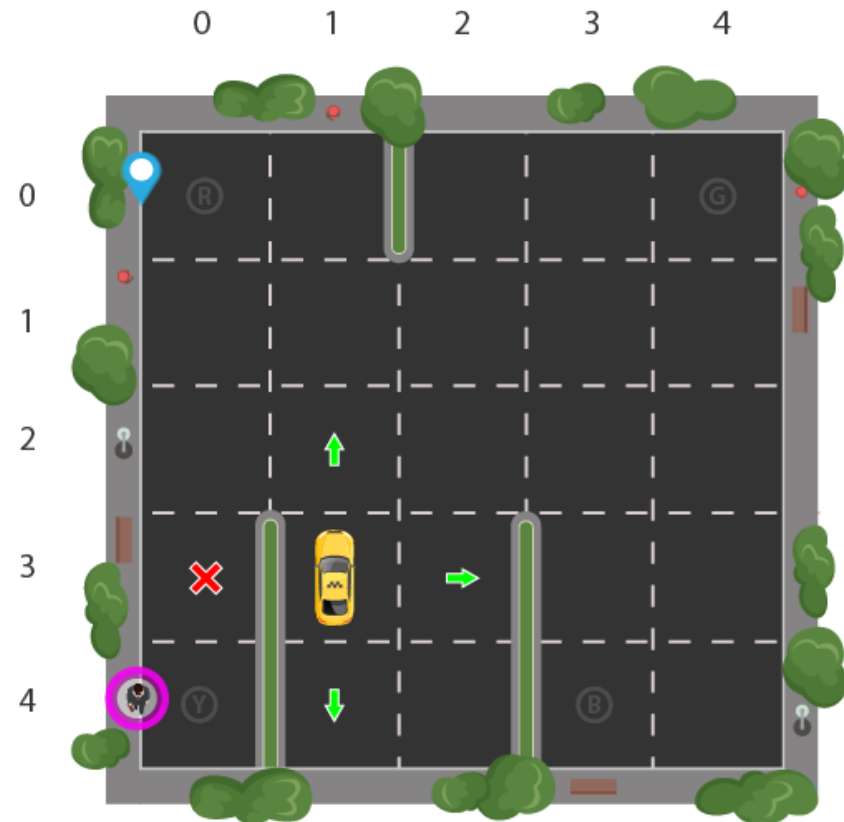
# a self-driving cab

## *action space*

**action space** (set of all the actions)

1. south
2. north
3. east
4. west
5. pickup
6. dropoff

- action space size = 6 actions
- some actions are impossible due to walls
- wall hits are penalized:
  - penalty/reward: -1
  - the taxi does not move anywhere



# a self-driving cab

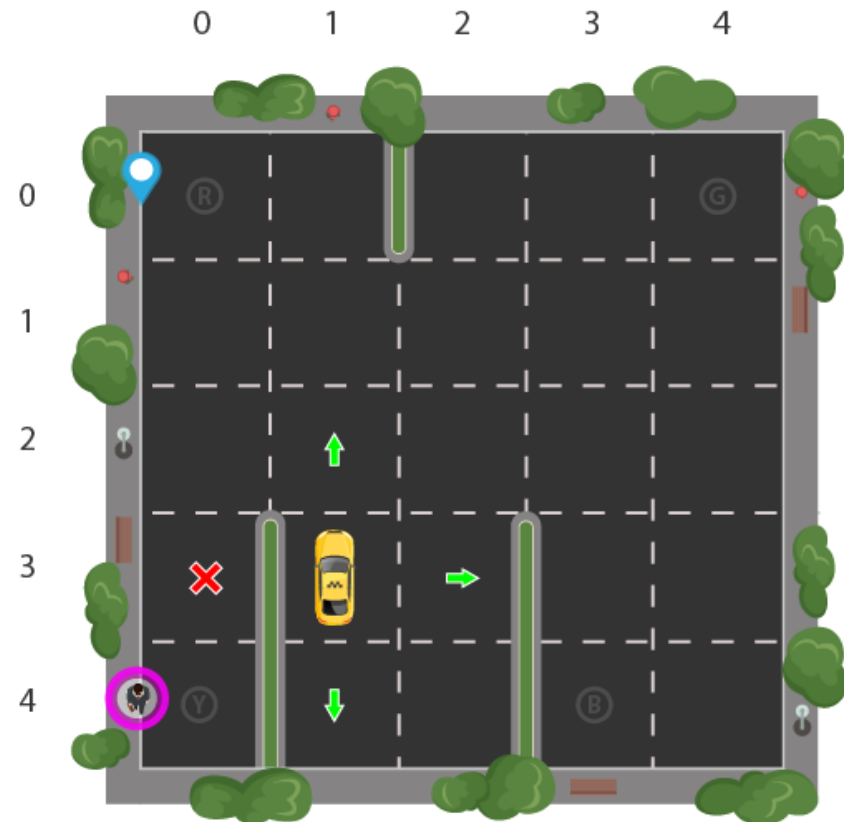
## *rewards*

### rewards

- a high positive reward for a successful dropoff
- a high negative reward for a wrong dropoff
- a slight negative reward for every time-step
- reward table: *State Space* X *Action Space* matrix
- reward table[state] returns a dictionary...  
{action: [(probability, nextstate, reward, done)]}

- rewards at state (row, entry) 328

```
{ 0: [(1.0, 428, -1, False)],  
  1: [(1.0, 228, -1, False)],  
  2: [(1.0, 348, -1, False)],  
  3: [(1.0, 328, -1, False)],  
  4: [(1.0, 328, -10, False)],  
  5: [(1.0, 328, -10, False)]}
```



# a self-driving cab

## *Q-learning*

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	327	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Source: Satwik Kansal, Brendan Martin. Reinforcement Q-Learning from Scratch in Python with OpenAI Gym. LearnDataSci, 2023.

# RL paradigms

tabular solutions: the action-value function is represented as an array or a table

approximate, gradient-descent, solutions: the state-action spaces involve continuous domains or complex representations (e.g., an image)

# approximate solutions

## *motivation*

- **continuous, infinite state/action spaces**: it requires generalization from previously experienced states to ones that have never been seen.
- **online function approximation**
  - it takes earlier examples from previous interactions between the agent and the environment (bootstrap)
  - it learns by bootstrapping from current estimates of the value function
  - generalization from selected examples as in supervised learning (by comparing current function outputs against expectations)

# approximate solutions

## *introduction*

### learning problem (the usual one)

- to learn a policy  $\pi$  based on a state-value function  $v_{\pi}(s)$

### novelty

- the unknown state-value function  $v_{\pi}(s)$  is approximated by a parameterized function  $\hat{v}_{\mathbf{w}}(s)$  with a parameter vector  $\mathbf{w} \in \mathbb{R}^n$
- $\hat{v}_{\mathbf{w}}(s)$  can be a deep artificial neural network, with  $\mathbf{w}$  being the vector of connection weights (deep reinforcement learning)



# approximate solutions

## *state value function approximation*

**assumption:** there exist  $v_\pi(s_t)$  that computes the true value from a state  $s_t$  at time step  $t$  under the policy  $\pi$ ,  $\forall t = 1, 2, 3, \dots$

**goal:** to find  $w$  such that  $\hat{v}_w(s_t)$  best approximates  $v_\pi(s_t)$

**gradient-descent** method:

$$w_{t+1} = w_t - \frac{1}{2} \alpha \nabla [v_\pi(s_t) - \hat{v}_{w_t}(s_t)]^2$$
$$= w_t + \alpha [v_\pi(s_t) - \hat{v}_{w_t}(s_t)] \nabla \hat{v}_{w_t}(s_t)$$

the change is proportional to the negative gradient of the example's squared error

under certain conditions, the method converges to a local minimum

learning rate  
step-size param.

$\left( \frac{\partial \hat{v}_{w_t}(s_t)}{\partial w_{t,1}}, \frac{\partial \hat{v}_{w_t}(s_t)}{\partial w_{t,2}}, \dots, \frac{\partial \hat{v}_{w_t}(s_t)}{\partial w_{t,n}} \right)$

# approximate solutions

## *state value function approximation*

what if  $v_\pi(s_t)$  is unknown? the true value does not exist

alternative: a rough/noisy approximation  $v_t$  of  $v_\pi(s_t)$ ; for example:

$$v_t = r_{t+1} + \gamma \hat{v}_{w_t}(s_{t+1})$$

$$s_{t+1}, r_{t+1} \leftarrow env(s_t, \pi(s_t))$$

gradient-descent method:

$$w_{t+1} = w_t + \alpha [v_t - \hat{v}_{w_t}(s_t)] \nabla \hat{v}_{w_t}(s_t)$$

# approximate solutions

## *deep reinforcement learning*

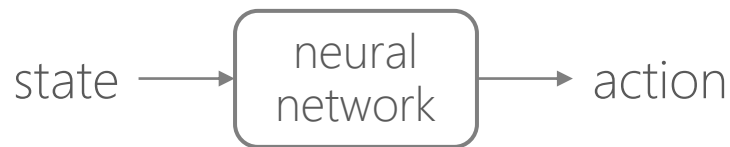
neural networks are function approximators

neural networks are particularly useful in RL when  
the state space or action space are too large  
to be completely known

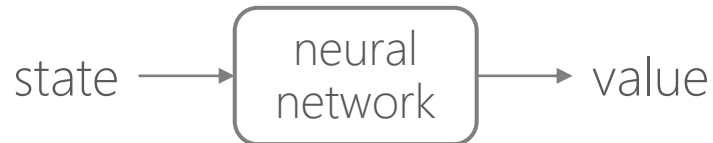
# approximate solutions

## *deep reinforcement learning*

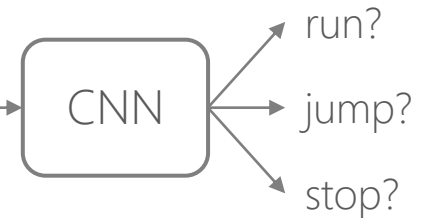
policy  $a = \hat{\pi}_w(s)$



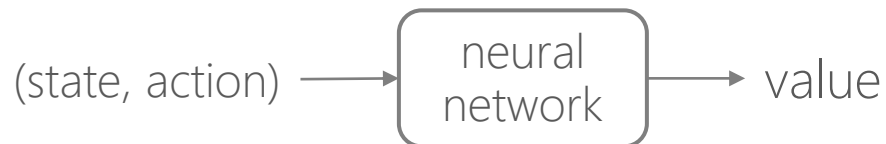
value function  $v = \hat{v}_w(s)$



source: flickr.com



Q-value function  $v = \hat{v}_w(s, a)$



# approximate solutions

## *actor-critic methods*

### previous methods

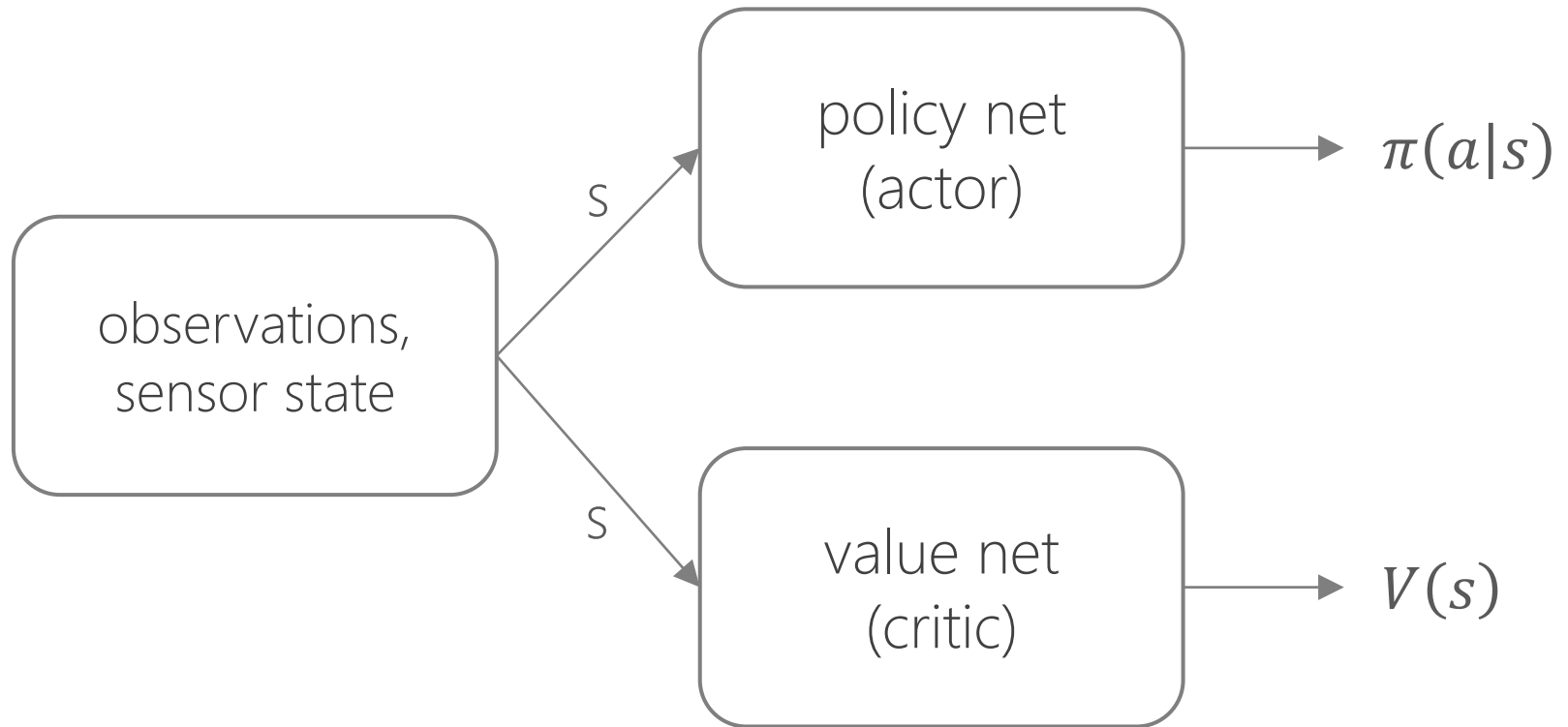
have policy implementations that directly use the value function of states or state-action pairs in choosing an action (e.g.,  $\epsilon$ -greedy)

### actor-critic methods

have a separate memory structure to explicitly represent the policy independent of the value function

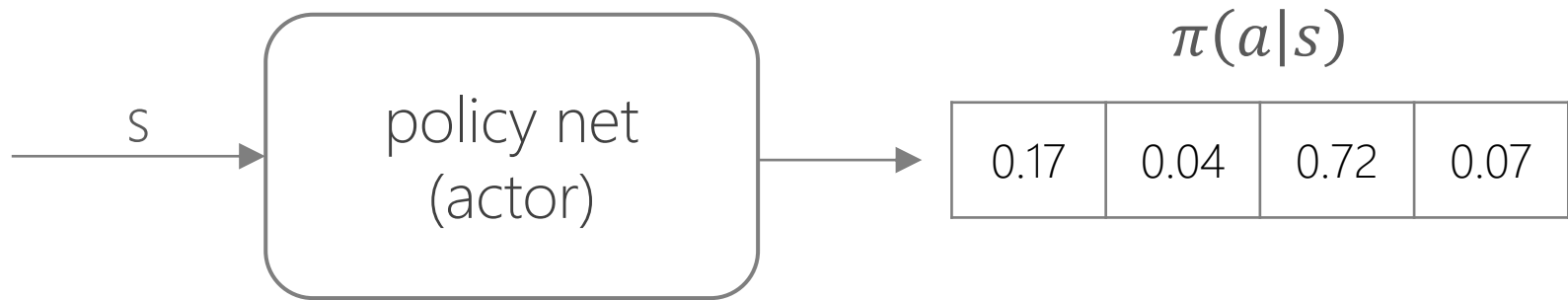
# actor-critic methods

## *introduction*



# actor-critic methods

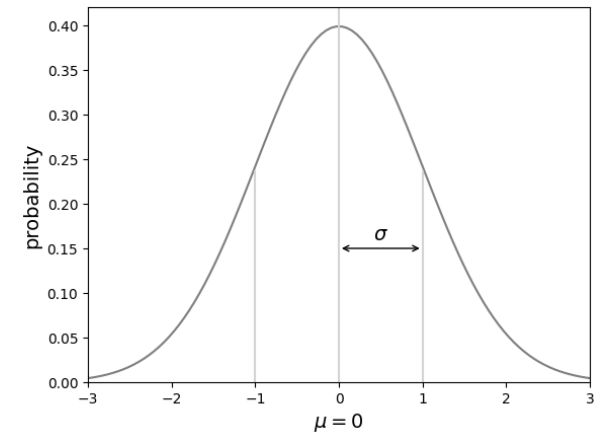
## *discrete policy*



probability distribution  
over the action space

# actor-critic methods

## *continuous policy*



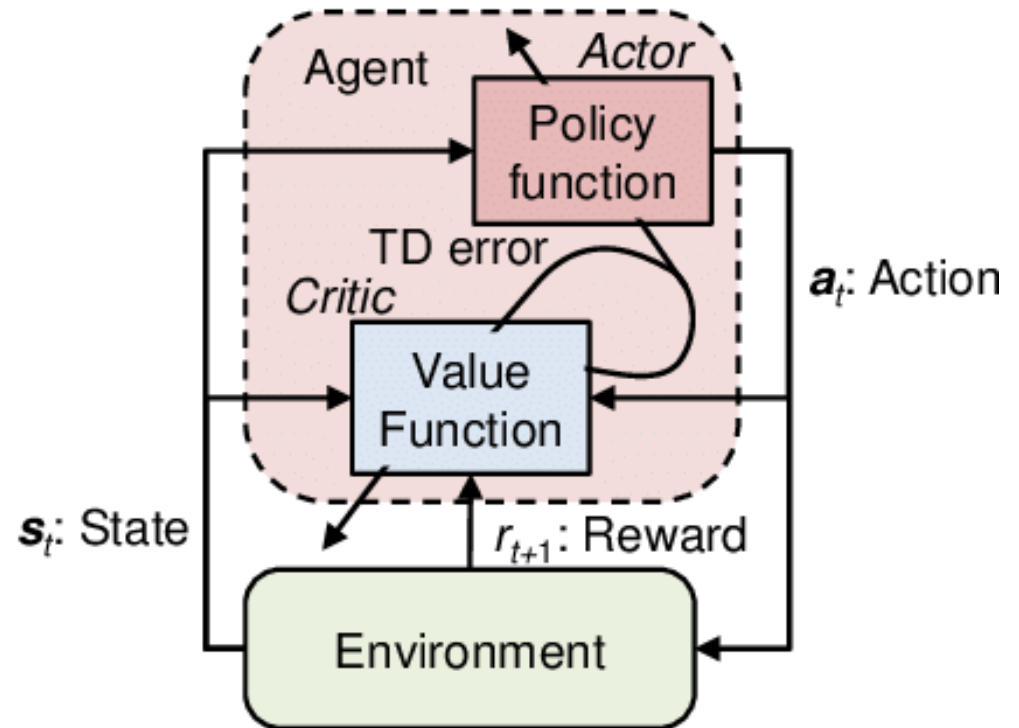


# actor-critic methods

## *architecture*

**actor** (policy): select actions

**critic** (value): criticizes the actions made by the actor



Source: Fuji, Taiki & Ito, Kiyoto & Matsumoto, Kohsei & Yano, Kazuo. (2018). Deep Multi-Agent Reinforcement Learning using DNN-Weight Evolution to Optimize Supply Chain Performance. 10.24251/HICSS.2018.157.

# actor-critic methods

## *architecture*

- the **critique error** (a scalar) drives learning in both actor and critic
- after each action  $\mathbf{a}_t$ , the critic evaluates the new state  $\mathbf{s}_t$  to determine whether things have gone better or worse than expected.

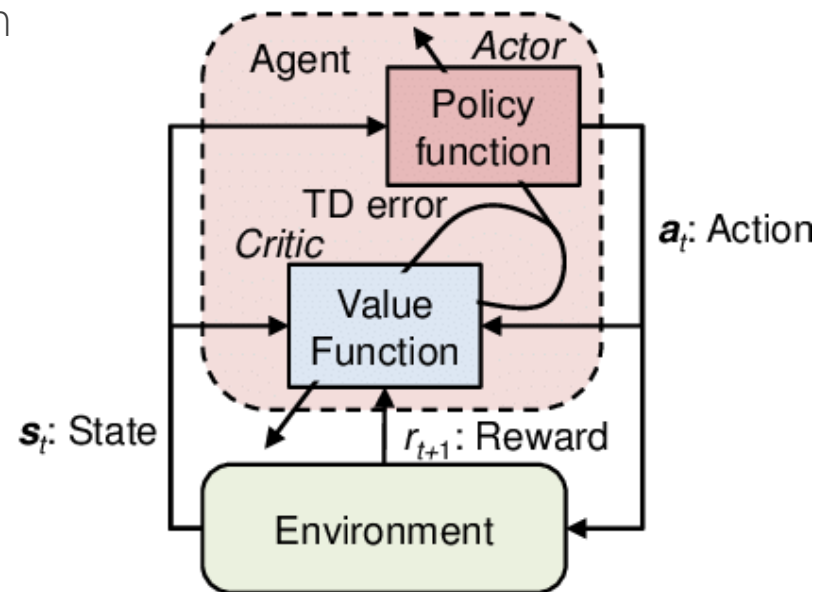
- critique error:

$$\delta_t = r_{t+1} + \gamma v_t(s_{t+1}) - v(s_t)$$

- where...

$v_t$  is the critique value at time  $t$

$v$  is an expected state value



# Deep Deterministic Policy Gradient (DDPG)

*an actor-critic method*

## learning overview

- $Q$ -function (critic) and policy  $\mu$  (actor) are deep neural network models
- critic and actor models are learned iteratively and alternately
- learning involves off-policy data (a policy different from the target)
- the actor model is a deterministic policy optimized using the critic's evaluation

## scope overview

- DDPG is an off-policy algorithm:
  - the behavior policy (used to collect data) is different from the target policy
  - the behavior model (policy) changes continuously
- DDPG can only be used for environments with continuous action spaces
- DDPG uses an exploratory policy by adding noise to actions chosen by the actor

# Deep Deterministic Policy Gradient (DDPG)

*two key tricks*

## experience replay buffer

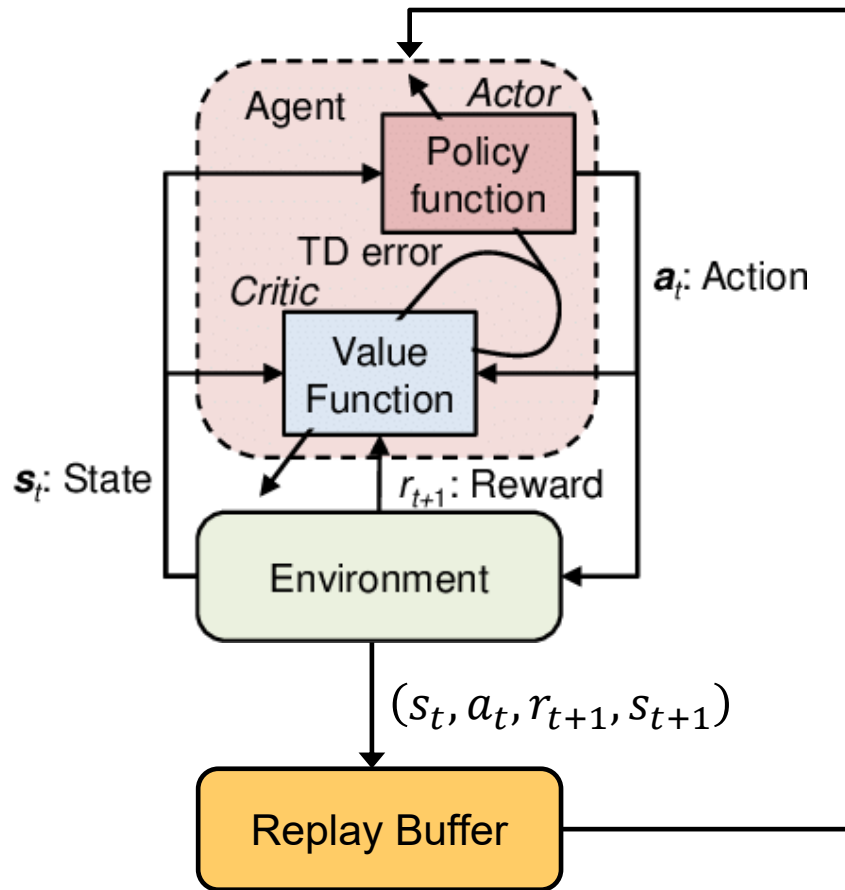
- the replay buffer ( $\mathcal{R}$ ) is the set of previous experiences  $(s_t, a_t, r_{t+1}, s_{t+1})$
- $\mathcal{R}$  should be as large and diverse as possible (useful for learning)
- a random minibatch  $\mathcal{B}_t$  of  $N$  transitions is sampled from  $\mathcal{R}$  at each learning step  $t$
- $\mathcal{B}_t$  is used to update both the  $Q$ -function and the policy networks

## target networks (intended to make the loss minimization stable)

- lagging versions of the  $Q$ -function and the policy (current) networks
- target networks do not change when current networks are being updated
- target networks provide more stable expectations for current networks updates
- target network are updated once after current network update by Polyak average ([+](#))

# Deep Deterministic Policy Gradient (DDPG)

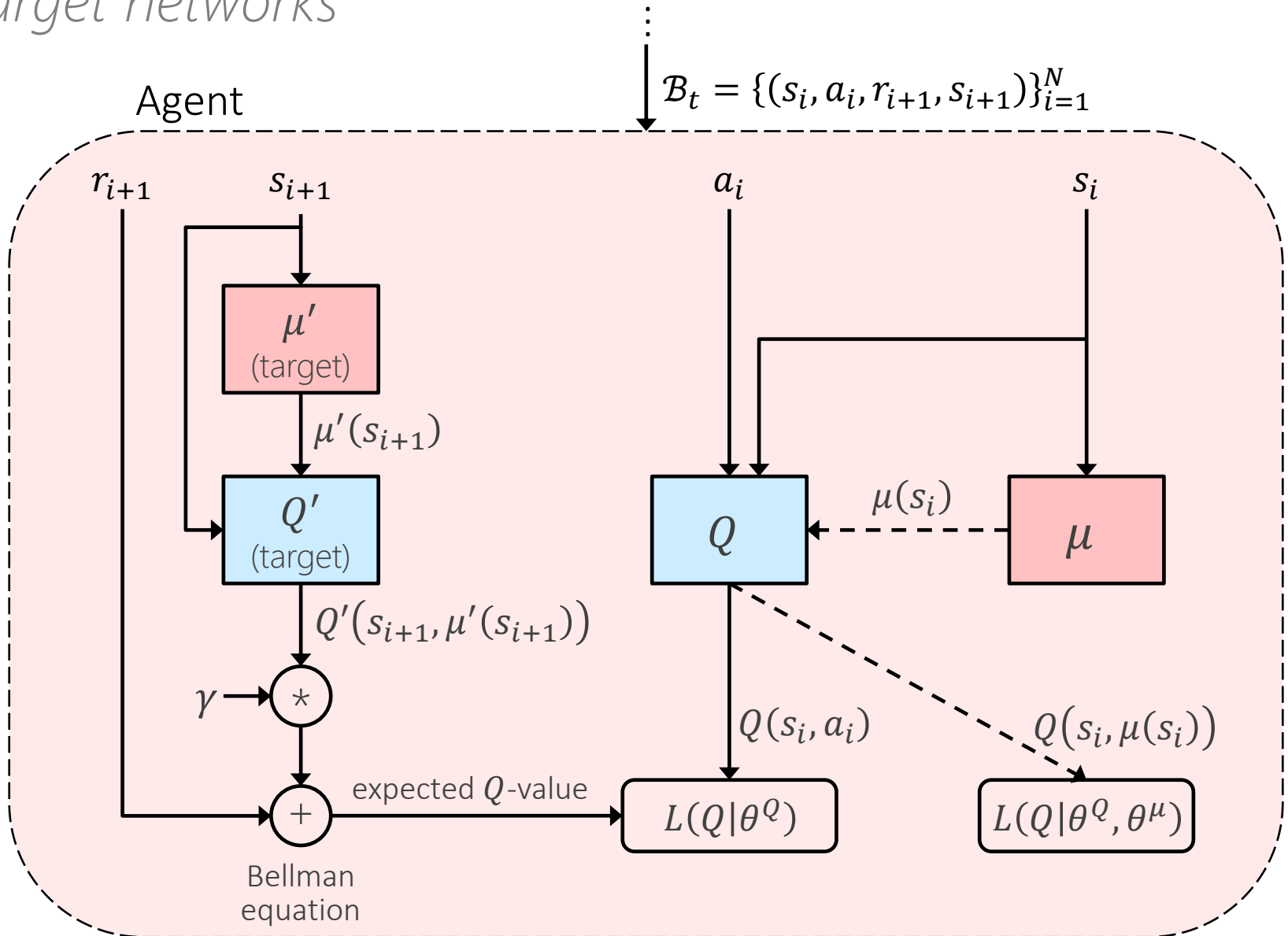
*experience replay buffer*



$$\mathcal{B}_t = \{(s_i, a_i, r_{i+1}, s_{i+1})\}_{i=1}^N$$

# Deep Deterministic Policy Gradient (DDPG)

*target networks*



# Deep Deterministic Policy Gradient *algorithm*

---

**Algorithm 1** DDPG algorithm

---

Randomly initialize critic network  $Q(s, a|\theta^Q)$  and actor  $\mu(s|\theta^\mu)$  with weights  $\theta^Q$  and  $\theta^\mu$ .

Initialize target network  $Q'$  and  $\mu'$  with weights  $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer  $R$

**for** episode = 1, M **do**

    Initialize a random process  $\mathcal{N}$  for action exploration

    Receive initial observation state  $s_1$

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

        Select action  $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$  according to the current policy and exploration noise

        Execute action  $a_t$  and observe reward  $r_t$  and observe new state  $s_{t+1}$

        Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $R$

        Sample a random minibatch of  $N$  transitions  $(s_i, a_i, r_i, s_{i+1})$  from  $R$

        Set  $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

        Update critic by minimizing the loss:  $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

**end for**

**end for**

---

# bootstrapping

## overview

- **bootstrap method**: statistical technique for estimating quantities about a population by averaging estimates from multiple small data samples (sets).
- **a data sample** is constructed by drawing individual observations from a large data sample with replacement
- **an observation** can thus be included in a given sample more than once
- **bootstrapping usually perform better** than nonbootstrapping methods
- **bootstrapping methods are of great interest** to RL



# bootstrapping

## *algorithm*

1. Choose a number of bootstrap samples (iterations) to perform
2. Choose a bootstrap sample size
3. For each bootstrap sample
  - a. Draw a sample with replacement with the chosen size
  - b. Calculate the statistic on the sample
4. Calculate the mean of the sample statistics

# summary

- **RL goal**: to learn how to map each state to the most promising action
- **exploitation/exploration dilemma**: exploit known actions or explore new actions
- **RL elements**: a policy  $\pi$ , a reward signal, a value function, an environment model
- **tabular solutions**: suitable for finite and small spaces of state and action
- **gradient-descent solutions**: suitable for continuous and infinite spaces of st & act
- **off-policy learning**: learning does not depend on the agent policy  $\pi$
- **on-policy learning**: learning does depend on the agent policy  $\pi$
- **Deep Deterministic Policy Gradient** (DDPG) is an off-policy actor-critic method
- **actor-critic method**: policy and value functions are fully independent