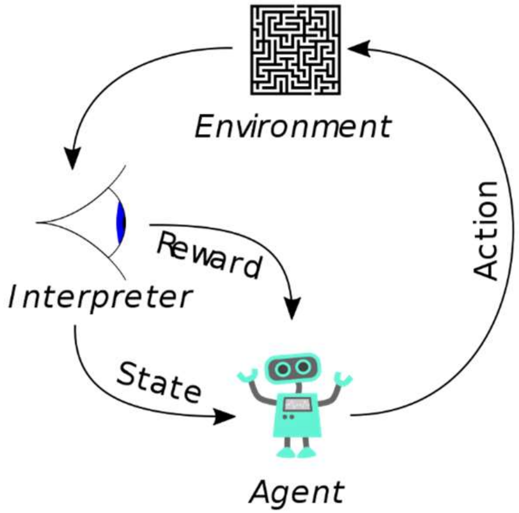
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Robotics / Intelligent Robotics
Introduction to Reinforcement Learning

Luís Paulo Reis, Nuno Lau, Henrique L. Cardoso, Armando Sousa

lpreis@fe.up.pt
Director of LIACC – Artificial Intelligence and Computer Science Lab.
Associate Professor at DEI/FEUP – Informatics Engineering Department, Faculty of Engineering of the University of Porto, Portugal
President of APPIA – Portuguese Association for Artificial Intelligence

What is Reinforcement Learning?

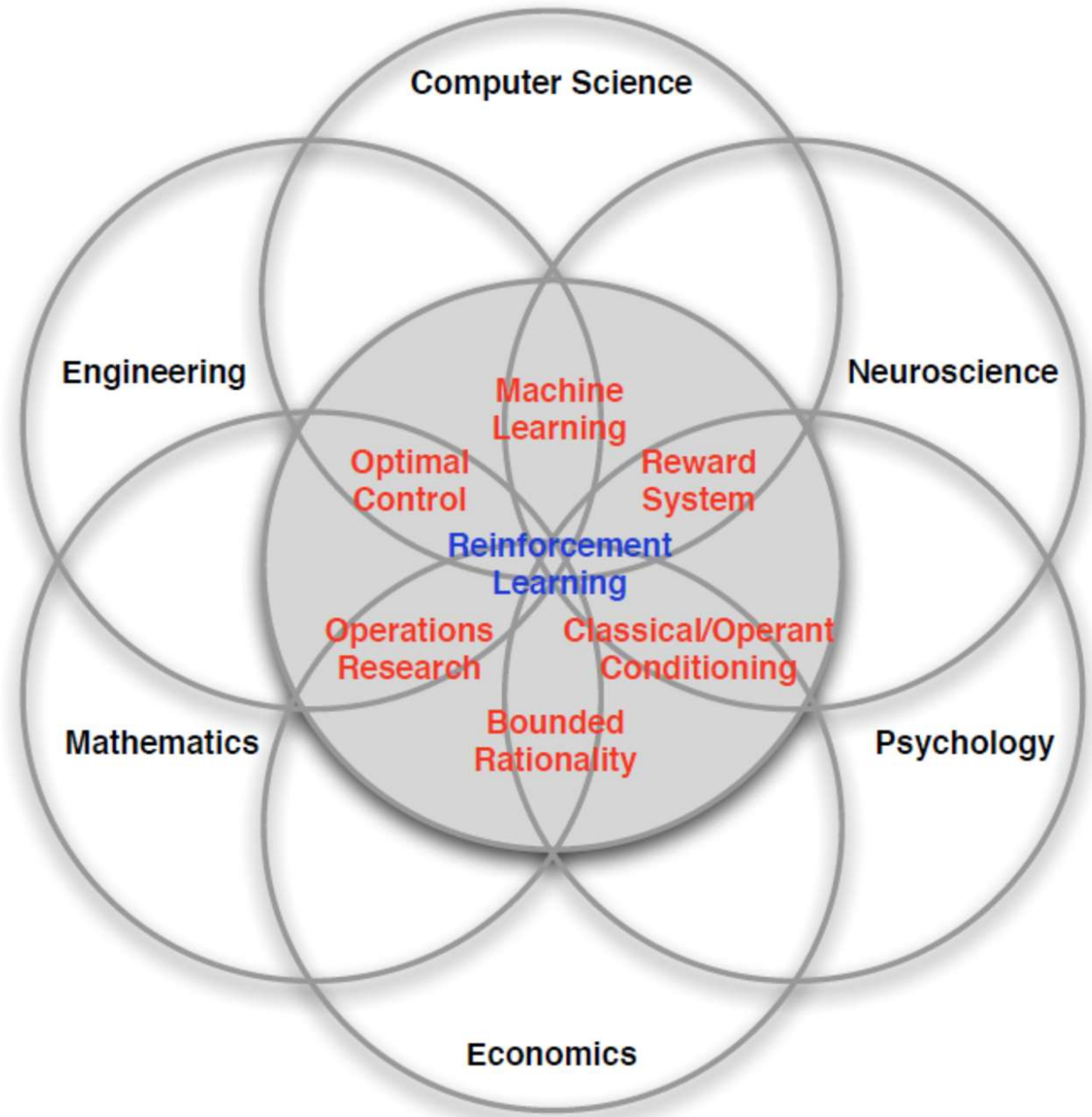


- **Reinforcement Learning (RL)** is focused on goal-directed learning from interaction

- RL is learning what to do – how to map situations to actions – so as to maximize a numerical return signal
  - The learner is not told which actions to take: it must discover which actions yield the most **return** by trying them
  - **return** is the sum of rewards for a given sequence of actions
  - Typically, actions may affect not only immediate reward but also the next situation and subsequent rewards

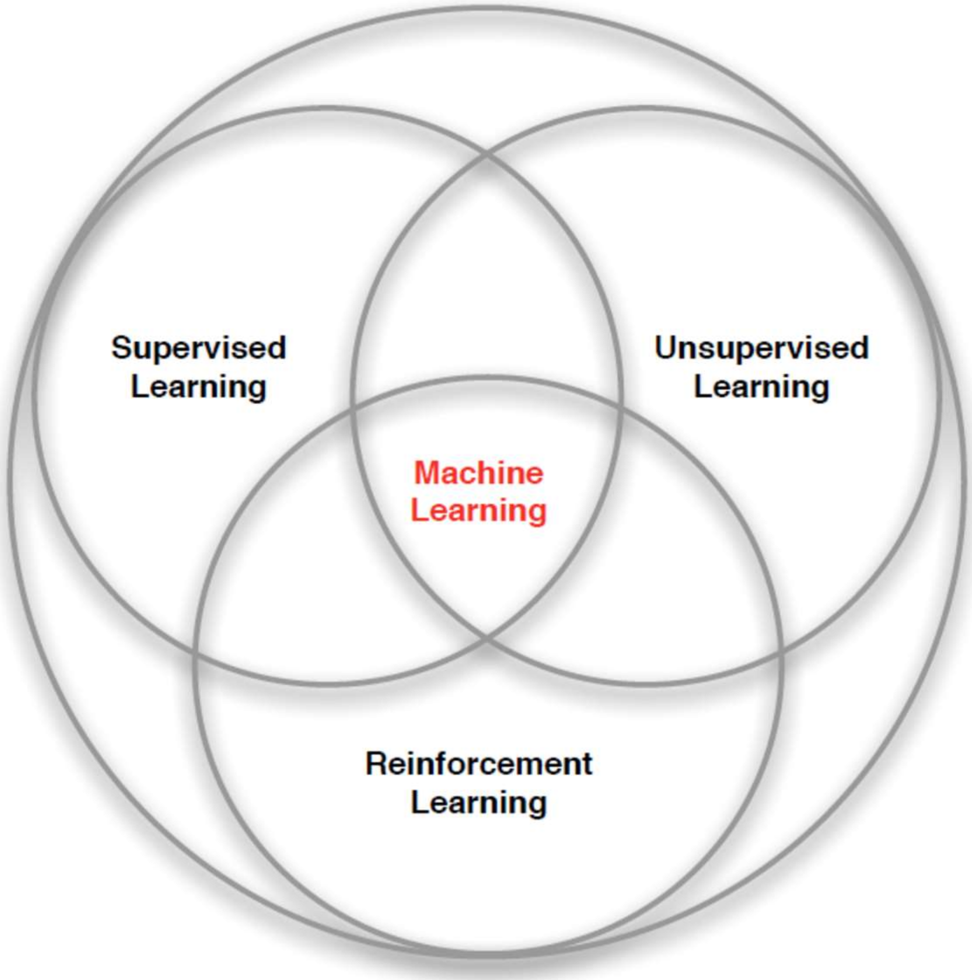
- The exploration-exploitation tradeoff
  - Agent must prefer actions that it knows to be effective – *exploit*
  - But to discover such actions, it has to try actions not selected before – *explore*

What is Reinforcement Learning?



David Silver, RL slides

Branches of Machine Learning



David Silver, RL slides

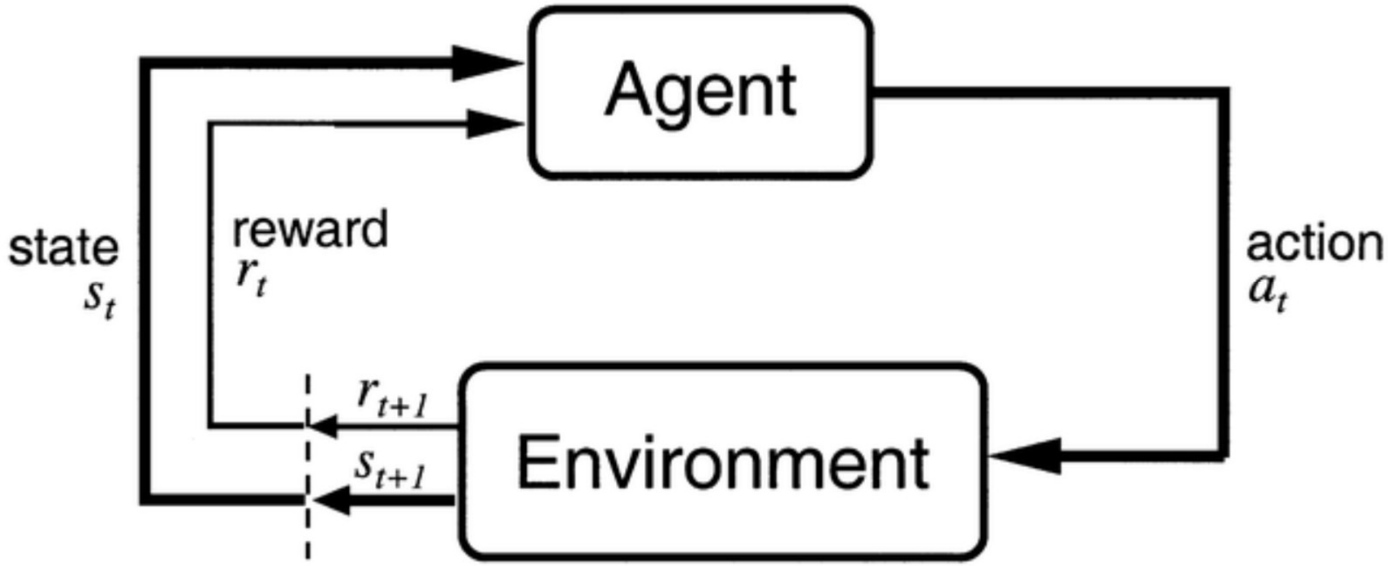
RL vs (Un)Supervised Learning

- Different from **supervised learning**
  - In interactive problems it is impractical to obtain examples of desired behavior
  - In uncharted territory, an agent must learn from its own experience
- Different from **unsupervised learning**
  - RL is trying to maximize a reward signal, not trying to find hidden structure in collections of unlabeled data

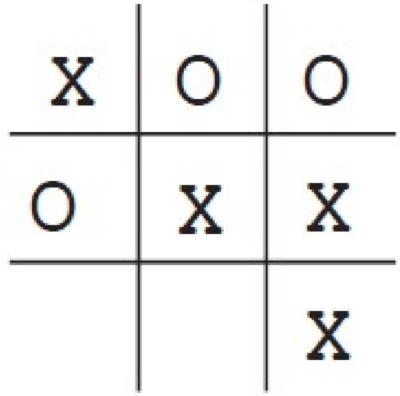
- RL explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment
  - Creating a behavior model while applying it in the environment

- RL is the closest form of ML to the kind of learning humans do

Reinforcement Learning model



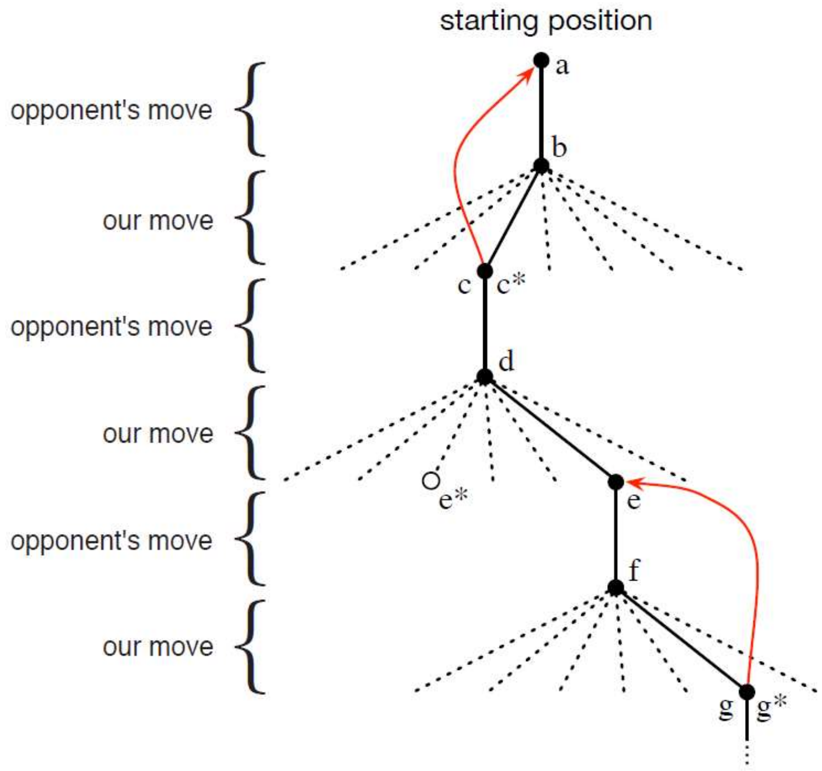
Learning to Play Tic-Tac-Toe



- Rule-based approach
  - Need to hardcode rules for each possible situations that might arise in a game
- Minimax
  - Assumes a particular way of playing by the opponent
- Dynamic programming can compute an optimal solution for any opponent
  - But requires as input a complete specification of that opponent (state/action probabilities)
- Can we obtain such information from experience?
  - Play many games against the opponent!

Learning to Play Tic-Tac-Toe

- States
  - Possible configurations of the board
- Actions
  - Possible moves to make
- Policy
  - Which action should I play in each state?
- Reward
  - How good was the chosen action?



Elements of RL

- **Policy π**
  - How should the agent behave over time?
  - A policy is a (possibly stochastic) mapping from perceived states to actions
- **Reward signal r**
  - Defines the goal of the RL problem
  - On each time step, the environment sends a reward to the RL agent – the agent's goal is to maximize the total reward (return) received over the long run
- **Value function v**
  - Specifies what is good in the long run
  - The value of a state is the total amount of reward an agent can expect to accumulate from that state onwards (it takes into account future rewards)

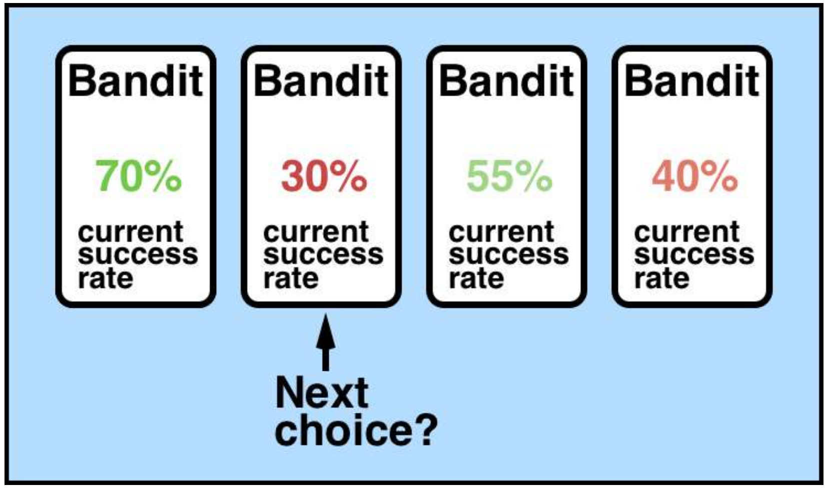
→ We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest return

Bandit Problems

- A simple setting with a single state

- *K*-armed bandit problem
  - There are *k* different actions
  - After each action a numerical reward is received from a stationary probability distribution
  - Each action has a *value* – its expected or mean reward, not known by the agent: $q_*(a) \doteq \mathbb{E}[R_t | A_t = a]$
  - The agent *estimates*, at time step t, the value of an action a: $Q_t(a)$

Bandit Problems



- Selecting *greedy* actions (whose estimated value is greatest): exploiting
- Selecting non-greedy actions: exploring
  - Improve estimates of non-greedy actions' value
- Lower reward in the short run (during *exploration*), but higher in the long run – after discovering the best actions, we can *exploit* them many times

Estimating Action Values

- Sample average:

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}$$

- Update rule:

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$$

$$\boxed{NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]}$$

  - The target indicates a desirable direction in which to move
  - The *step-size* parameter changes from time step to time step

- Giving more weight to recent rewards – constant *step-size* parameter:

$$Q_{n+1} \doteq Q_n + \alpha[R_n - Q_n]$$

where $\alpha \in (0,1]$

Action Selection

- *Greedy* action selection (always exploits): $A_t \doteq \underset{a}{\operatorname{argmax}}\, Q_t(a)$

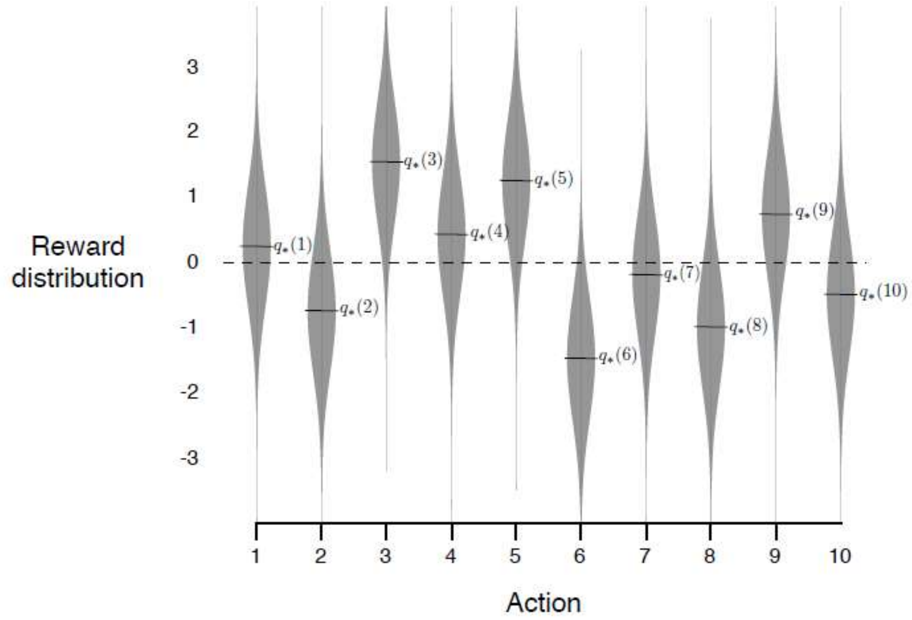
- *ε-greedy* action selection: behave greedily most of the time, but with small probability ε select randomly from among all the actions
  - $Q_t(a)$ will converge to $q_*(a)$ if a is selected sufficiently often

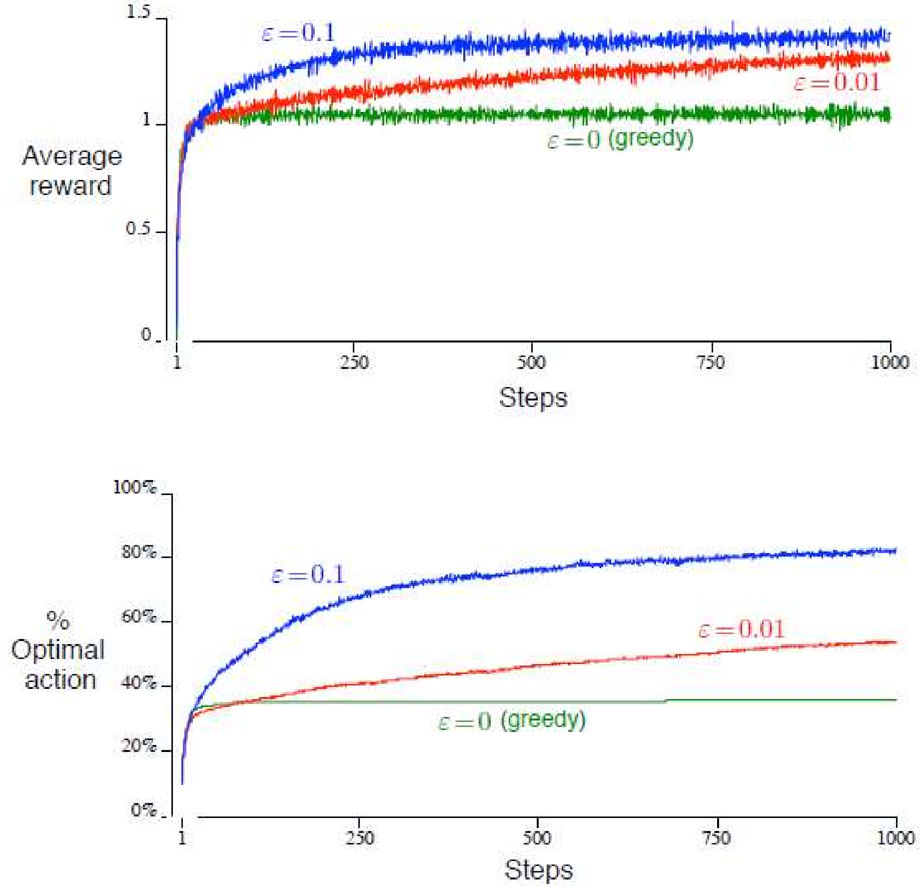
- *Soft-max* action selection (Boltzmann distribution):

$$\Pr\{A_t = a\} \doteq \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{k} e^{Q_t(b)/\tau}}$$

where τ is a temperature parameter: if high, actions will tend to be equiprobable; if low, action values matter more; if $\tau \to 0$, then we have greedy action selection

The 10-armed Testbed





Markov Decision Processes

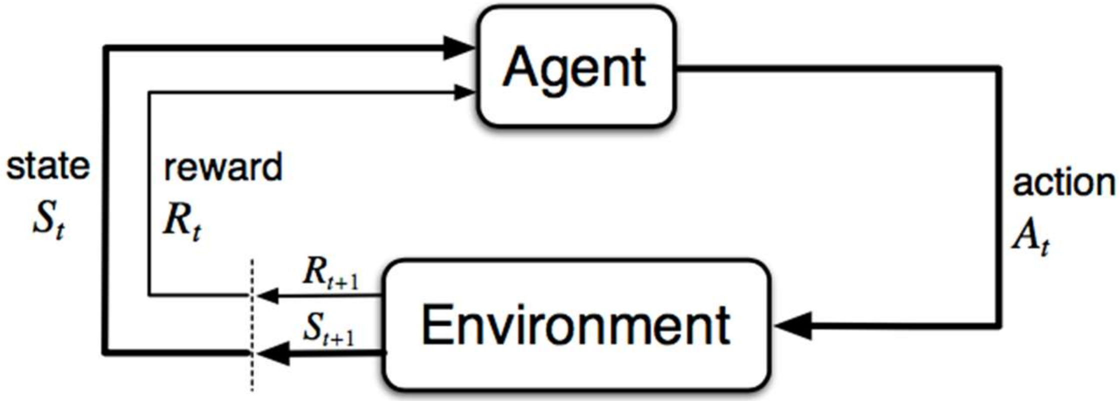
- In the general setting we have many states

- Markov Decision Processes (MDP) are a classical formalization of sequential decision making
  - Actions influence not just immediate rewards, but also subsequent situations (states) and thus future rewards

- A finite MDP, is defined by:
  - a set of states, S
  - a set of actions, A(s)
  - a state transition model, p(s,a,s’) -> [0,1]
  - a reward function, R(s,a,s’) -> r

- In MDPs we estimate the value $q_*(s,a)$

Agent-Environment Interface



- Dynamics of the MDP:

$$p(s',r|s,a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

  - The probability of each possible value for s' and r depends only in the immediately preceding state s and action a
  - The state must include all relevant information about the past agent-environment interaction – Markov property

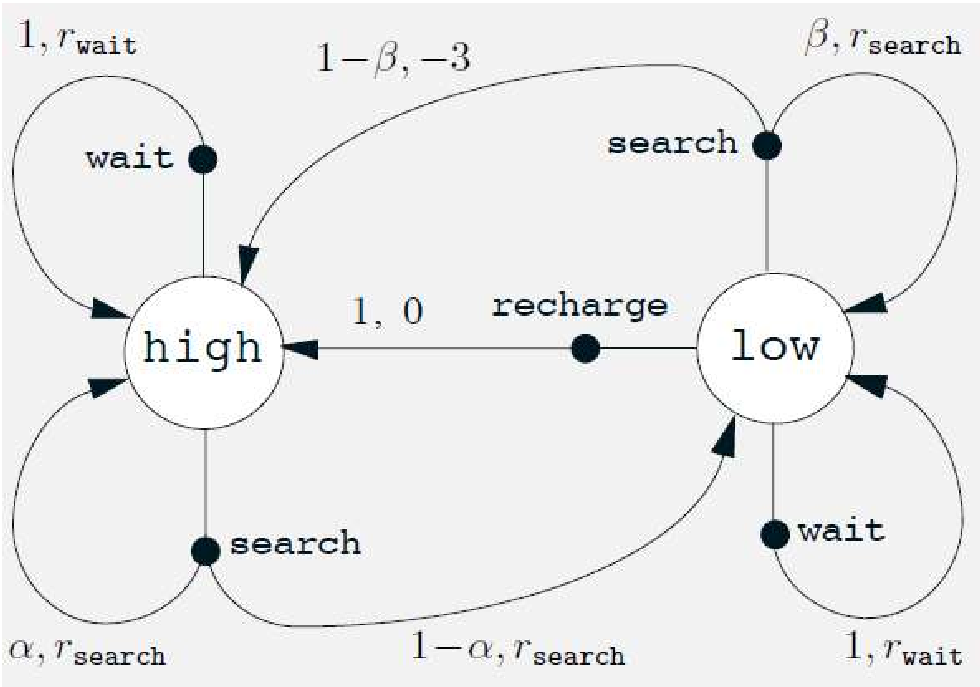
Example: Recycling Robot

- A robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) go to home base and recharge
- Searching is better (higher probability of getting a can) but runs down battery; if out of battery, the robot has to be rescued
- Decisions made on the basis of current energy level: high, low
- Reward is zero except when getting a can, and negative if out of battery

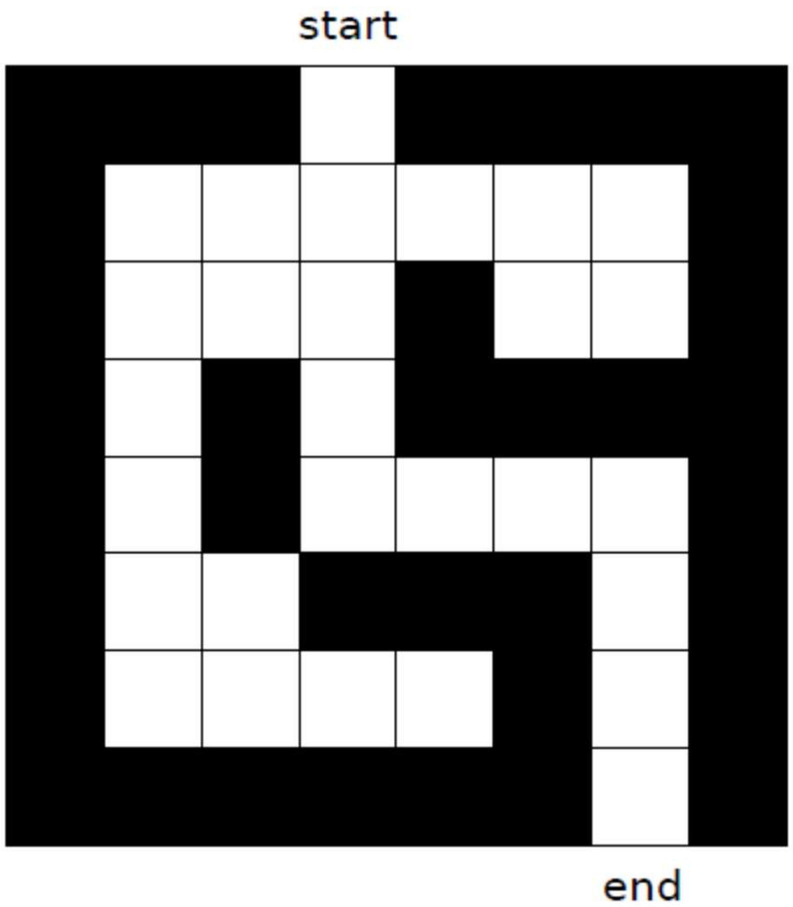
$$\mathcal{S} = \{high, low\}$$
$$\mathcal{A}(high) = \{search, wait\}$$
$$\mathcal{A}(low) = \{search, wait, recharge\}$$
$$r_{search} > r_{wait}$$

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1 - \alpha$	$r_{\texttt{search}}$
low	search	high	$1 - \beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	-

Example: Recycling Robot

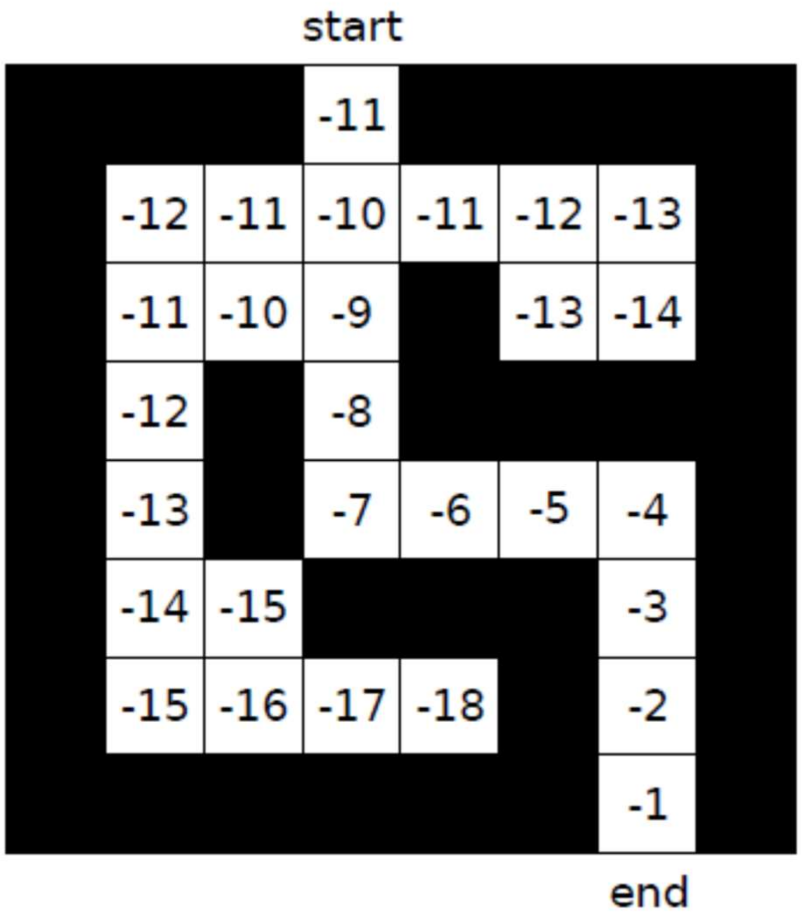


Example: maze



- Reward=-1 for each move
- Value function of optimal policy?

Example: maze



- Value function of optimal policy
- How to determine it?

Goals and Rewards

- A reward signal is used to define the goal of the agent
  - Learning to walk: reward proportional to the robot's forward motion
  - Learning to Escape from a maze: reward -1 for any state prior to escape (encourage escaping as quickly as possible)
  - Learning to find empty cans for recycling: reward of 0 most of the time, +1 for each can collected
  - Learning to play checkers or chess: reward +1 for winning, -1 for losing, and 0 for drawing and nonterminal positions

- Provide rewards in such a way that by maximizing them the agent will also achieve our goal
  - The agent's goal is to maximize the cumulative reward it receives in the long run

→The reward signal is a way of communicating to the robot *what* you want it to achieve, not *how*

Returns and Episodes

- Agent wants to maximize expected return

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

- Episodic tasks: when the agent-environment interaction breaks naturally into subsequences – episodes
  - From a starting state to a terminal state
  - Followed by a reset to another starting state, chosen independently of how the previous episode ended

- Continuing tasks do not break naturally into identifiable episodes (*e.g.,* on-going process-control)
  - Problem with calculating G_t:
    - $T = \infty$
    - G_t could also be infinite (if rewards are positive at each time step)

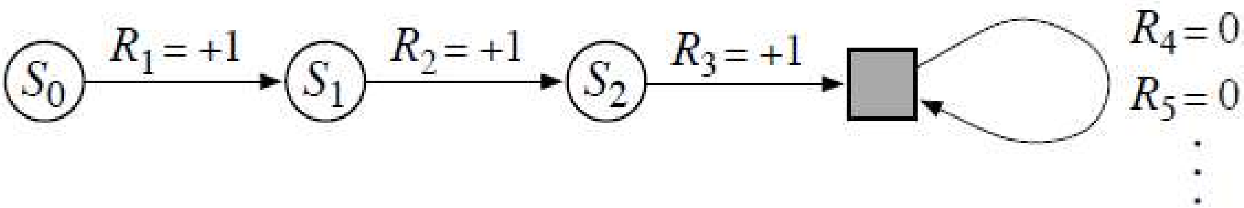
Returns and Episodes

- Adding discounting: agent wants to maximize expected discounted return

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \qquad\qquad G_t \doteq R_{t+1} + \gamma G_{t+1}$$

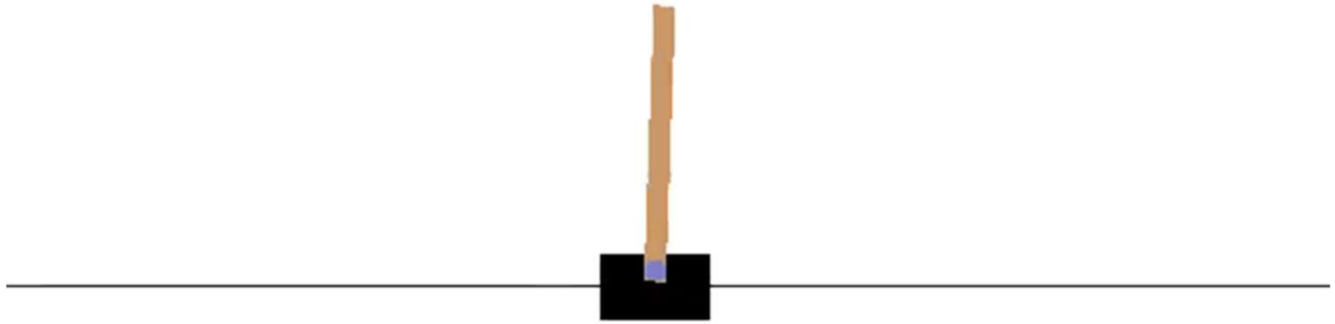
  - $0 \leq \gamma \leq 1$ is the discount rate
    - If $\gamma = 0$ the agent is “myopic” (only immediate reward matters)
    - As γ approaches 1, the agent becomes more farsighted (strongly considers future rewards)
  - G_t is now finite, even if summing an infinite number of terms

- Applicable also to episodic tasks, if we consider a final absorbing state:



Example: Pole Balancing

- Move a cart so as to keep a pole from falling over
  - Failure if the pole falls past a given angle or if the cart runs off the track
  - The pole is reset to vertical after each failure



- Episodic task: reward +1 except when failure
  - return is the number of steps until failure

- Continuing task, using discounting: reward –1 on each failure and 0 otherwise
  - return is $-\gamma^K$, where K is the number of steps before failure

Policies and Value Functions

- RL algorithms involve estimating value functions
  - How good (in terms of expected return) is it to be in a given state?
  - How good is it to perform a given action in a given state?

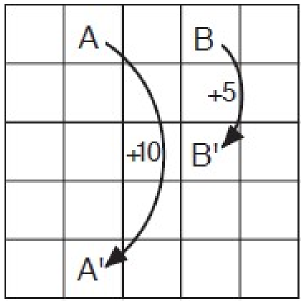
- Future rewards depend on the choice of actions
  - Value functions are defined with respect to policies (ways of acting)

- Policy: a mapping from states to probabilities of selecting each possible action
  - $\pi(a|s) = \Pr(A_t = a|S_t = s)$

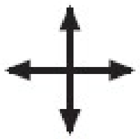
$\rightarrow$RL methods specify how the policy (*i.e.*, the probability distribution over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$) is changed with experience

Policies and Value Functions

- State-value function $v_\pi(s)$
  - Expected return when starting in s and following π thereafter
  - $v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$
- Action-value function $q_\pi(s, a)$
  - Expected return when taking action a in state s, and following π thereafter
  - $q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$
- The value functions v_π and q_π can be estimated from experience
  - $v_\pi(s) \doteq \sum_{s'} p(s, \pi(s), s')[r(s, \pi(s), s') + \gamma v_\pi(s')]$ (Bellman equation)
- Example: using a random policy, with $\gamma = 0.9$:



Gridworld



Actions

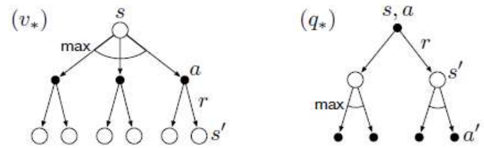
- Off-grid actions have no effect, with $r = -1$
- Any action from A gets to A', with $r = +10$
- Any action from B gets to B', with $r = +5$

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

v_π

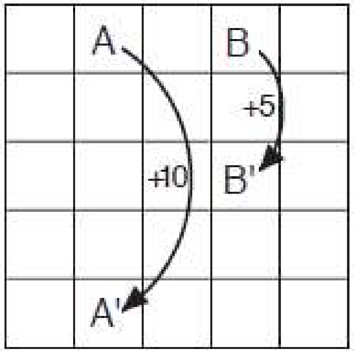
Optimal Policy and Value Function

- Optimal policy π_*: expected return is greater than any other policy
  - Optimal state-value function: $v_*(s) \doteq \max_{\pi} v_\pi(s)$
  - Optimal action-value function: $q_*(s,a) \doteq \max_{\pi} q_\pi(s,a)$

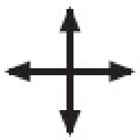


- Once we know v_* or q_*, the optimal policy is greedy

- Example:



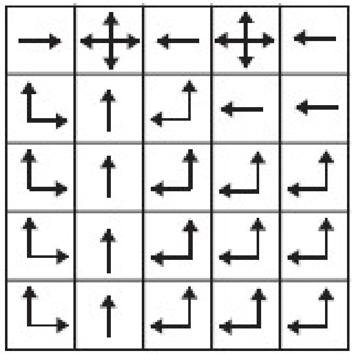
Gridworld



Actions

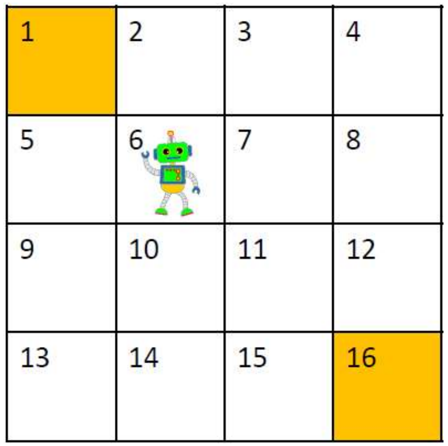
22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

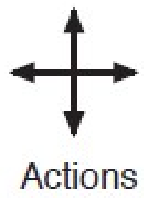
v_*



π_*

Example Grid World





- A bot is required to traverse a grid of 4×4 dimensions to reach its goal (1 or 16)
- There are 2 terminal states (1 and 16) and 14 non-terminal states (2 to 15)
- Each step is associated with a reward of -1
- Consider a random policy: at every state, the probability of every action {up, down, left, right} is 0.25
- Initialize v_1 for the random policy with all 0s

Example Grid World: Policy Evaluation

- Turning Bellman equation into an update:

$$v_{k+1}(s) \doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k S_{t+1} \, | S_t = s]$$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$$\begin{aligned} v_1(6) &= \sum_{a \in \{u,d,l,r\}} \pi(a|6) \sum_{s',r} p(s',r|6,a)[r + \gamma v_0(s')] \\ &= \sum_{a \in \{u,d,l,r\}} \underbrace{\pi(a|6)}_{= 0.25 \ \forall a} \sum_{s'} p(s'|6,a)[\underbrace{r}_{=-1} + \gamma \underbrace{v_0(s')}_{= 0 \ \forall s'}] \\ &= 0.25 * \{-p(2|6,u) - p(10|6,d) - p(5|6,l) - p(7|6,r)\} \\ &= 0.25 * \{-1 - 1 - 1 - 1\} \\ &= -1 \end{aligned}$$

$$\Rightarrow v_1(6) = -1$$

- For non-terminal states, $v_1(s) = -1$
- For terminal states, $p(s'|s,a) = 0$ (and hence $v_k(1) = v_k(6) = 0$, for all k)

$v_1 =$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

Example Grid World: Policy Evaluation

- Step 2, with discount factor $\gamma = 1$

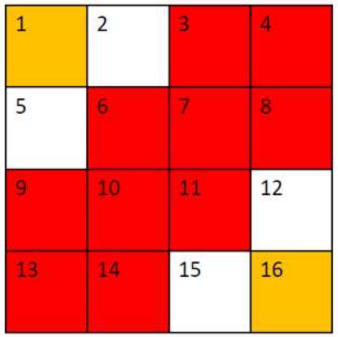
$$v_2(6) = \sum_{\underbrace{a \in \{u,d,l,r\}}} \underbrace{\pi(a|6)}_{= 0.25\ \forall a} \sum_{s'} p(s'|6,a)[\underbrace{r}_{=-1} + \gamma \underbrace{v_1(s')}_{=\begin{cases} -1, s' \in S \\ 0, s' \in S^+ \backslash S \end{cases}}]$$

$$= 0.25 * \{p(2|6,u)[-1-\gamma] + p(10|6,d)[-1-\gamma] + p(5|6,l)[-1-\gamma] + p(7|6,r)[-1-\gamma]\}$$

$$\overset{\gamma=1}{=} 0.25 * \{-2-2-2-2\}$$

$$= -2$$

- For all red states, $v_2(s) = -2$



- For the other states (2, 5, 12, 15):

$$v_2(2) = \sum_{a \in \{u,d,l,r\}} \underbrace{\pi(a|2)}_{= 0.25\ \forall a} \sum_{s'} p(s'|2,a)[\underbrace{r}_{=-1} + \gamma \underbrace{v_1(s')}_{=\begin{cases} -1, s' \in S \\ 0, s' \in S^+ \backslash S \end{cases}}]$$

$$= 0.25 * \{p(2|2,u)[-1-\gamma] + p(6|2,d)[-1-\gamma] + p(1|2,l)[-1-\gamma*0] + p(3|2,r)[-1-\gamma]\}$$

$$\overset{\gamma=1}{=} 0.25 * \{-2-2-1-2\}$$

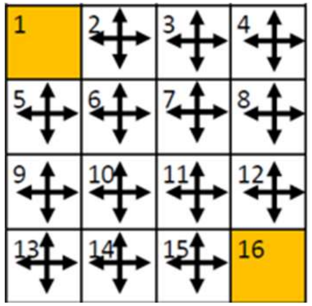
$$= -1.75$$

$$\Rightarrow v_2(2) = -1.75$$

$v_2 =$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

Example Grid World: Policy Evaluation



Random policy

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

...

$k = 10$

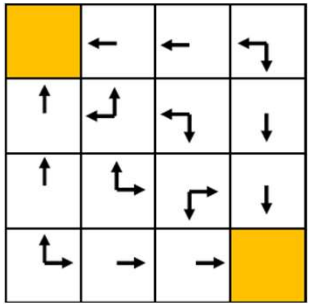
0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

...

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

$\leftarrow v_\pi$


Optimal policy

Policy Evaluation

Algorithm 1 Policy Iteration (Policy Evaluation)

Require: $V(s)$ arbitrarily initialized, $\forall s \in S$

repeat
    repeat
        $\Delta \leftarrow 0$
        for all $s \in S$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s'} P(s, \pi(s), s')[r(s, \pi(s), s') + \gamma V(s')]$
            $\Delta \leftarrow max(\Delta, |v - V(s)|)$
        end for
    until $\Delta \leq \theta$ ▷ a small positive value
    ...
until StablePolicy == `True`

Policy Improvement

Algorithm 2 Policy Iteration (Policy Improvement)

Require: $V(s)$ arbitrarily initialized, $\forall s \in S$

repeat
    $\dots$
    StablePolicy $\leftarrow$ `True`
    for all $s \in S$ **do**
        $b \leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s,a,s')[r(s,a,s') + \gamma V(s')]$
        if $b \neq \pi(s)$ **then**
            StablePolicy $\leftarrow$ `False`
        end if
    end for
until StablePolicy == `True`

Value Iteration

Algorithm 3 Value Iteration

Require: $V(s)$ arbitrarily initialized, $\forall s \in S$

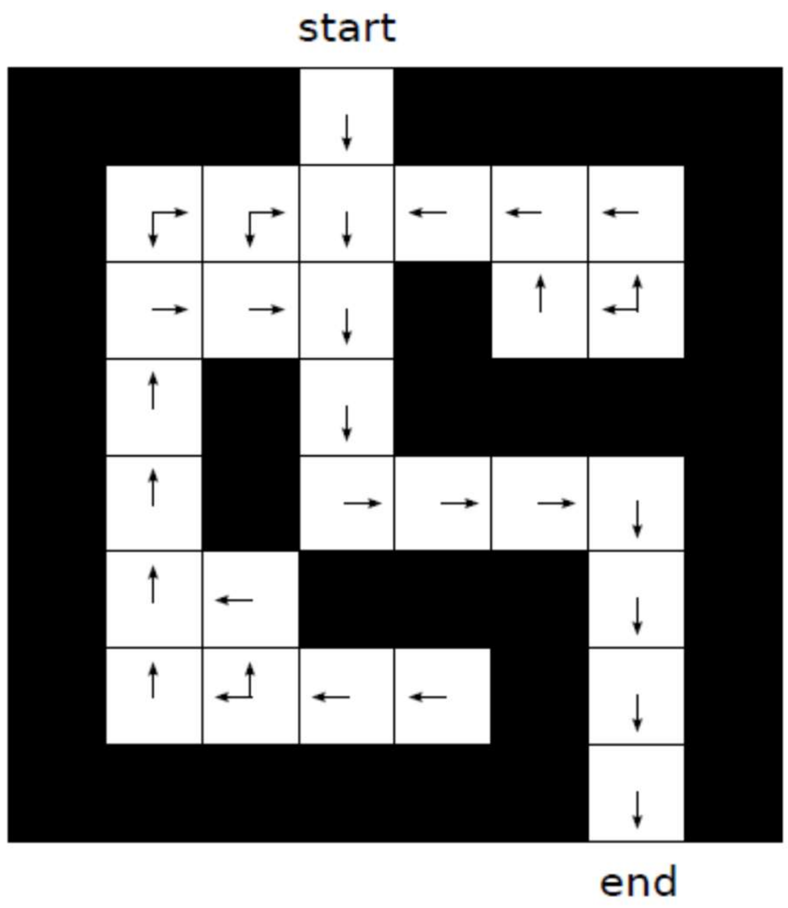
repeat
    $\Delta \leftarrow 0$
    for all $s \in S$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow max_a \sum_{s'} P(s, a, s')[r(s, a, s') + \gamma V(s')]$
        $\Delta \leftarrow max(\Delta, \|v - V(s)\|)$
    end for
until $\Delta \leq \theta$ ▷ a small positive value

Optimal Policy from Value Function

- How to determine the policy from the Value Function?
- Extract a policy!
- Policy maps states to actions

$$\pi(s) = \arg\max_a \sum_{s'} P(s,a,s')[r(s,a,s') + \gamma V(s')]$$

Example: Maze Optimal Policy



Approximation

- Optimal policies are computationally costly to find – we can only approximate
  - In tasks with small, finite state sets: tabular methods
  - Otherwise: function approximation using a more compact parameterized function representation (*e.g.* using neural networks)

→The online nature of RL allows us to *put more effort into learning to make decisions for frequently encountered states*

Temporal-Difference Learning

- TD methods update estimates based on immediately observed reward and state

- Update rule:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

- Because TD bases its update in part on an existing estimate, it is a *bootstrapping* method

Tabular TD(0) for estimating v_π

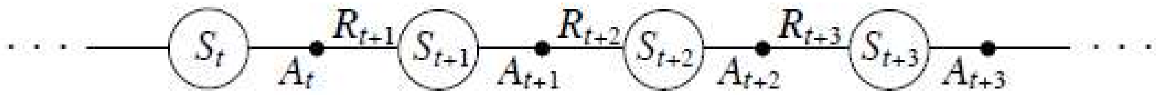
Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until S is terminal

Temporal-Difference Learning

- TD vs Dynamic Programming methods
  - TD methods do not require a model of the environment's dynamics (rewards and next-state probability distributions)
- TD vs Monte Carlo methods
  - TD methods are naturally implemented in an online, fully incremental fashion, while MC methods must wait until the end of an episode
    - Useful if episodes are very long, or in continuing tasks (that have no episodes at all)
- Usually, TD methods converge faster than MC methods on stochastic tasks

Sarsa: On-policy TD Control



- Update rule:

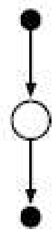
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

  - This rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal



Sarsa

- Converges to optimal policy and action-value function if all state-action pairs are visited infinitely and policy converges to greedy (*e.g.* using ε-greedy with $\varepsilon = 1/t$)

Q-learning: Off-policy TD Control

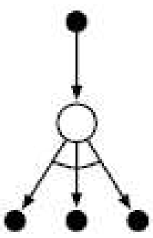
- Update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

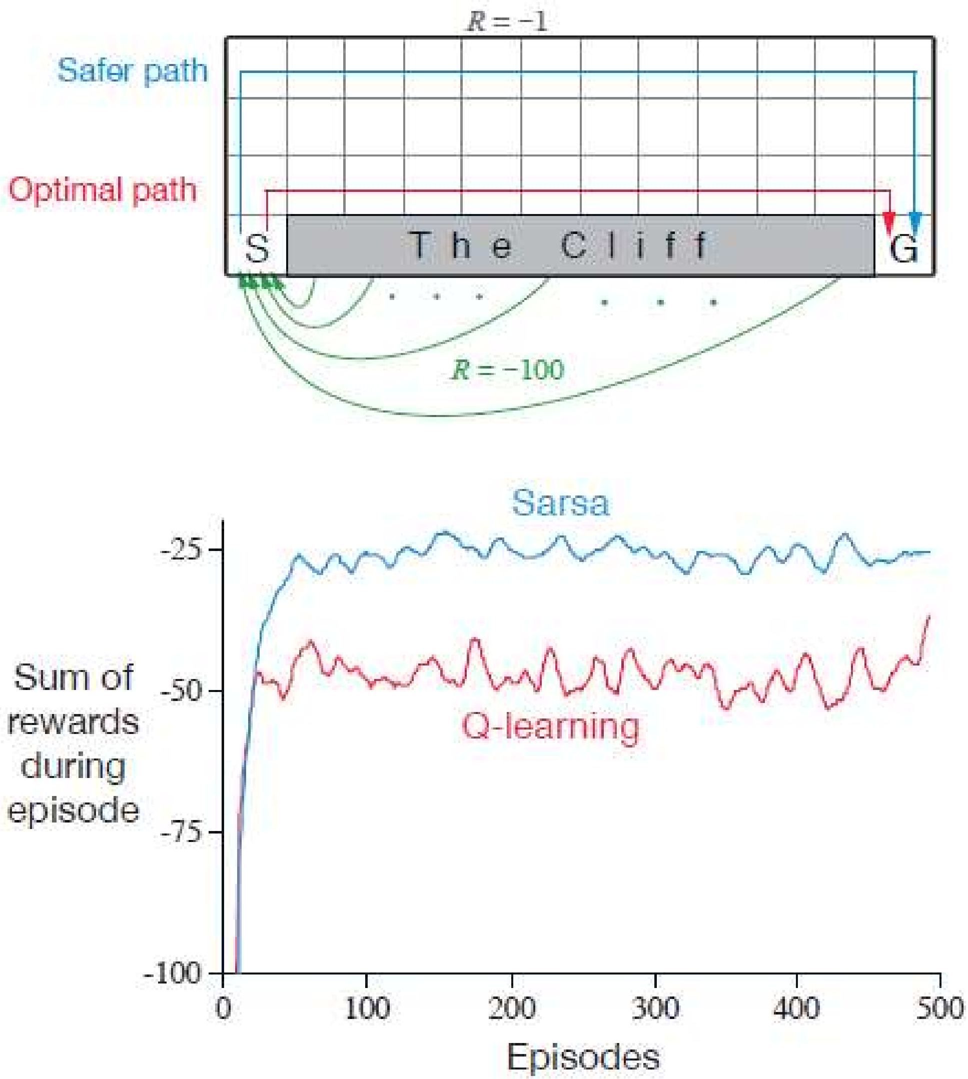
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal



Q-learning

- The learned action-value function Q directly approximates q_*, independently of the policy being followed

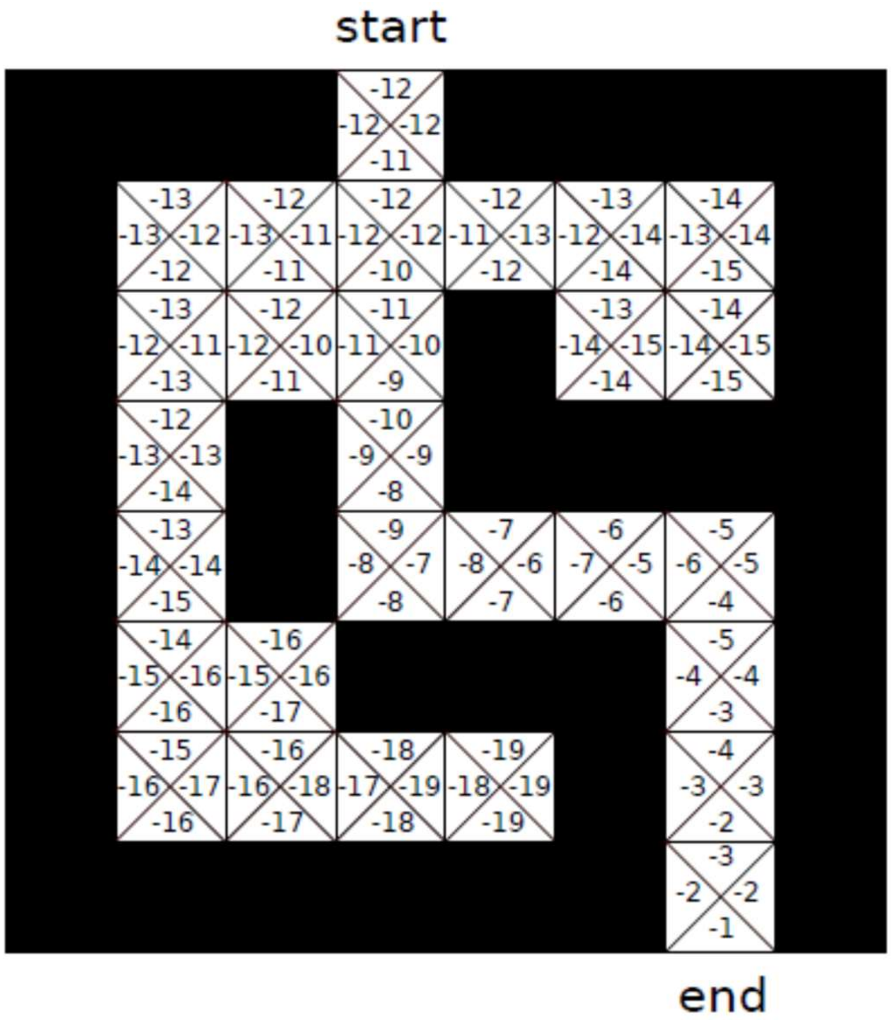
Example: Cliff Walking



- Sarsa and Q-learning with ε-greedy action selection ($\varepsilon = 0.1$)
  - Q-learning learns values for the optimal policy
  - Sarsa takes action selection into account and learns the longer but safer path
  - Given exploration, Q-learning occasionally falls off the cliff, hence the lower online performance

- If ε is gradually reduced, both methods converge to the optimal policy

Example: Maze



Classes of Reinforcement Learning

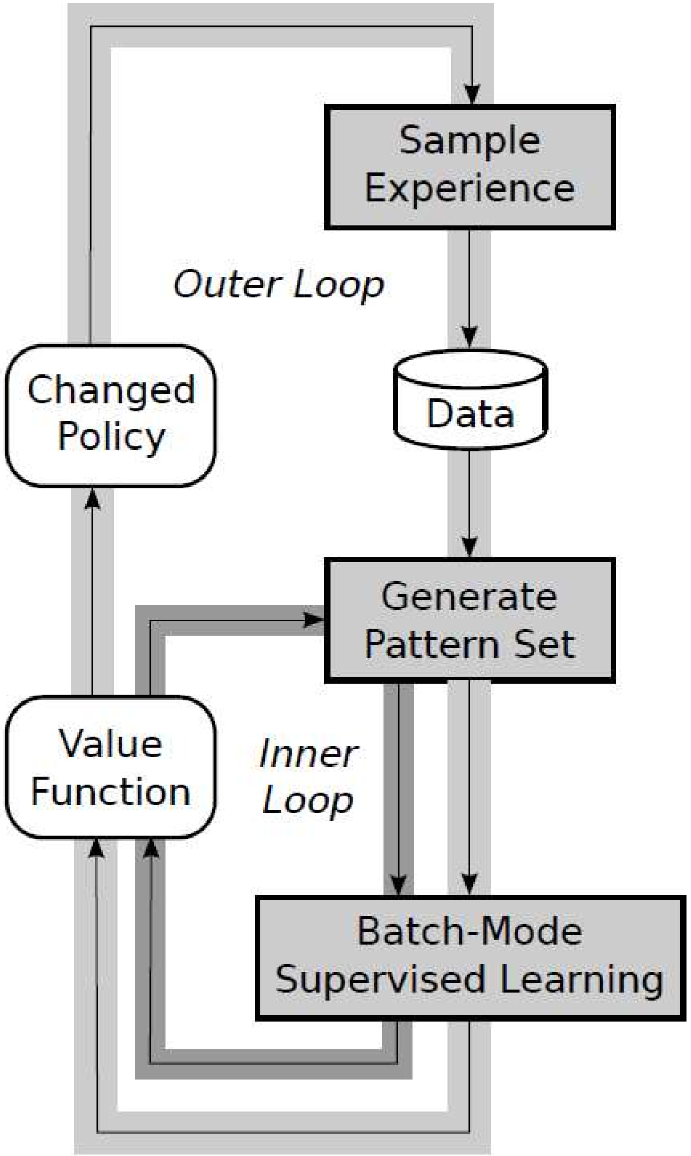
Three **main RL classes** of methods

- **Value Function based** methods
  - No policy representation
  - Policy obtained by evaluating the value function directly
- **Policy Search** methods
  - No value function
  - Optimization of a parametrized policy directly on policy-space
- **Actor-Critic** methods
  - Value function (critic)
  - Explicit Policy representation (actor)

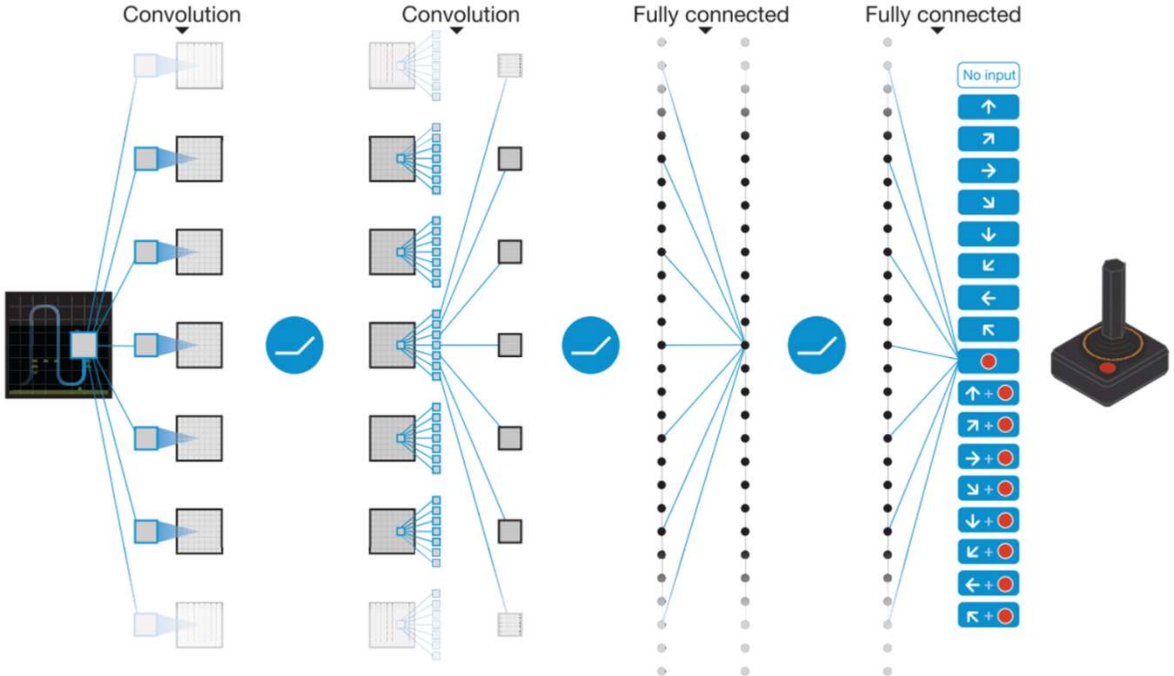
- **Batch RL is a sub-class of Value Function based methods**

Batch Reinforcement Learning

- **Batch RL** estimates value functions by processing **a set of interactions**
- The value function is updated synchronously
- Application of function approximators
- Collected experience is **not discarded**
- Data **efficient**
- Fit $\bar{Q}_i = r_i + \gamma \max_b \hat{Q}(s_{i+1}, b)$

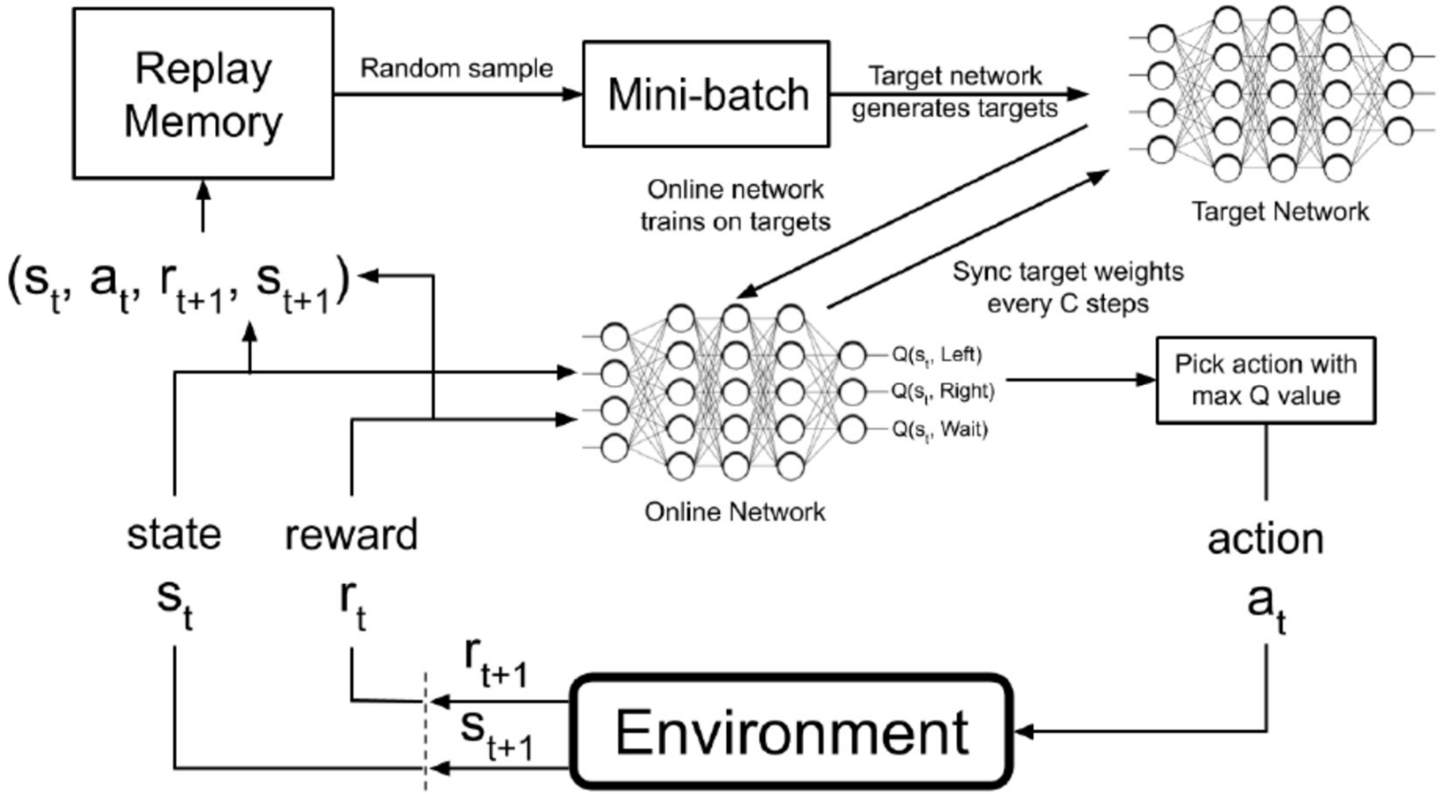


Deep Q Network - DQN



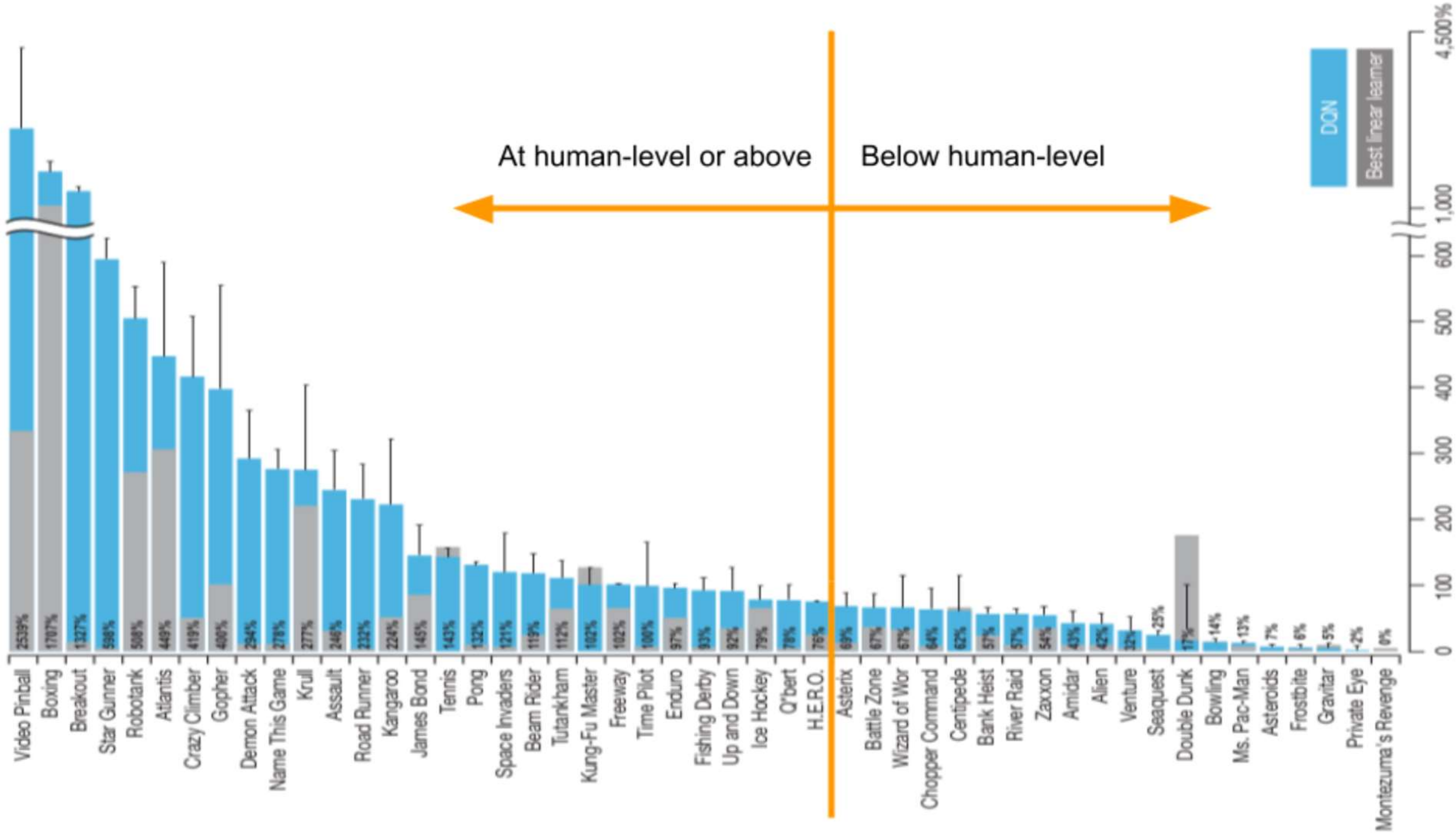
- Input: raw pixels (84x84x4 image)
- Output: Q values for all possible actions

Deep Q Network - DQN



$$\bar{Q}_i = r_i + \gamma \max_b \hat{Q}(s_{i+1}, b)$$

Deep Q Network - DQN



Open AI Gym

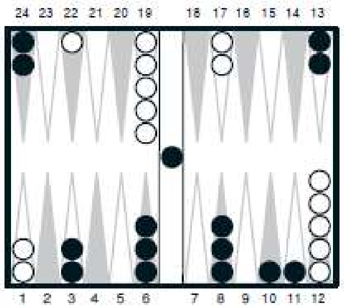
```
import gym

# load the environment
env = gym.make('Example')

# perform N episodes
while True:

    # prepare the environment for the next episode
    obs = env.reset()

    # flag for episode completion
    done = False

    # run the episode
    while not done:

        # choose how to act based on the current state
        # usually the output of a neural network
        action = env.action_space.sample()

        # advance the simulation by one timestep
        # by interacting with the world
        obs, reward, done, info = env.step(action)

# cleanup
env.close()
```

RL Algorithms

Algorithm	Description	Policy	Action Space	State Space
Monte Carlo	Every visit to Monte Carlo	Either	Discrete	Discrete
Q-learning	State–action–reward–state	Off-policy	Discrete	Discrete
SARSA	State–action–reward–state–action	On-policy	Discrete	Discrete
Q-learning - Lambda	Q-learning with eligibility traces	Off-policy	Discrete	Discrete
SARSA - Lambda	SARSA with eligibility traces	On-policy	Discrete	Discrete
DQN [Mnih et al., 2013]	Deep Q Network	Off-policy	Discrete	Continuous
DDPG [Lillicrap et al., 2016]	Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous
A3C [Mnih et al., 2016]	Asynchronous Advantage Actor-Critic	On-policy	Continuous	Continuous
NAF [Gu et al., 2016]	Q-Learning with Normalized Advantage Functions	Off-policy	Continuous	Continuous
TRPO [Schulman et al., 2015]	Trust Region Policy Optimization	On-policy	Continuous	Continuous
PPO [Schulman et al., 2017]	Proximal Policy Optimization	On-policy	Continuous	Continuous
TD3 [Fujimoto et al., 2018]	Twin Delayed Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous
SAC [Haarnoja et al., 2018]	Soft Actor-Critic	Off-policy	Continuous	Continuous

RL in Games

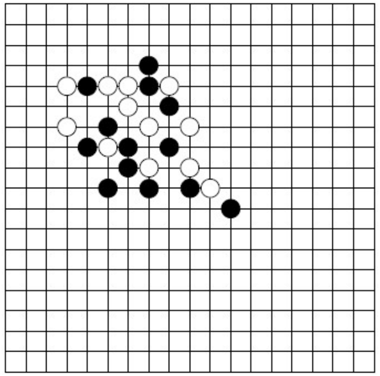
- TD-Gammon [Tesauro, 1995]
  - Neural Network trained with self-play reinforcement learning

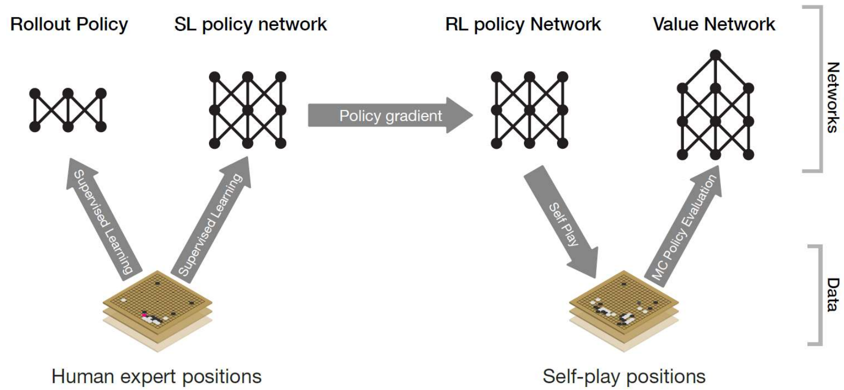


- Atari 2600 Games [DeepMind, 2013]
  - Learn control policies directly from high-dimensional sensory input using reinforcement learning
  - Input is raw pixels and output is a value function estimating future rewards

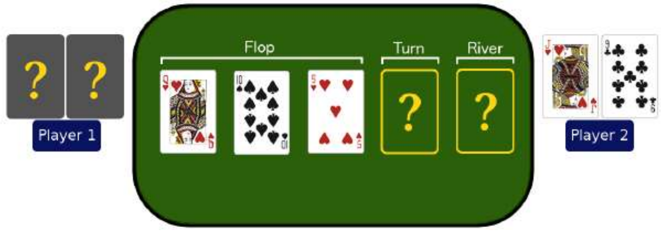
RL in Games

- AlphaGo [Google DeepMind, 2016]
  - Convolutional Neural Networks trained with human expert data
  - Deep Reinforcement Learning with fictitious self-play



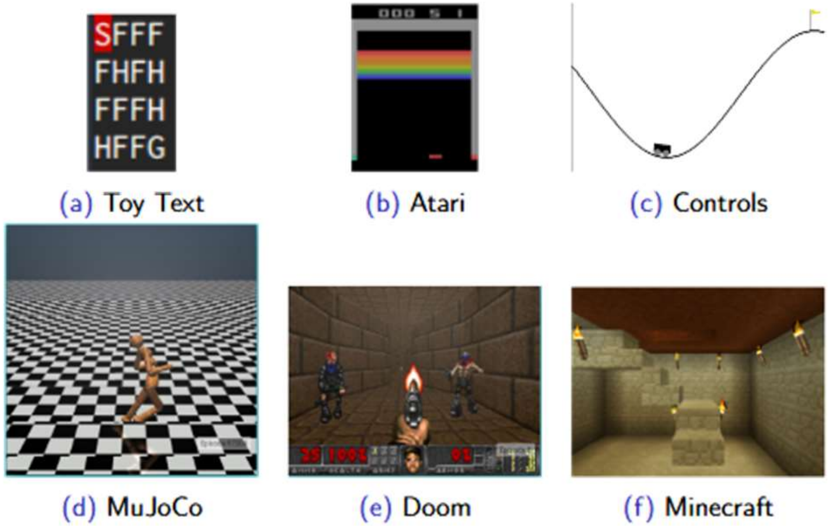


- Poker: Heads-Up Limit Texas Hold'em – NFSP [UCL, 2016]
  - Deep Reinforcement Learning with fictitious self-play
  - No prior knowledge



OpenAI Gym

- Gym is a toolkit for developing and comparing reinforcement learning algorithms
- The gym library is a collection of test problems with a shared interface — environments — that you can use to work out your RL algorithms

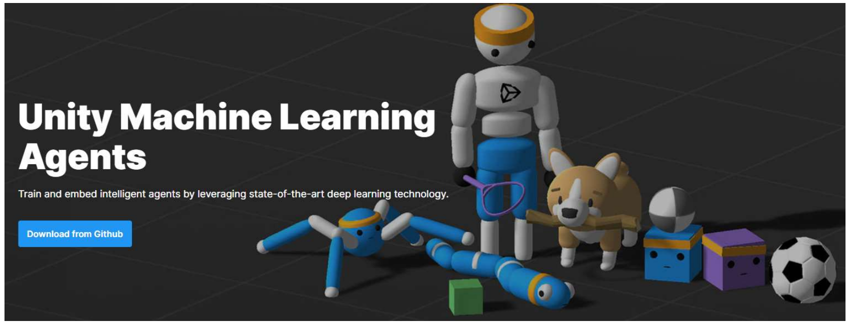


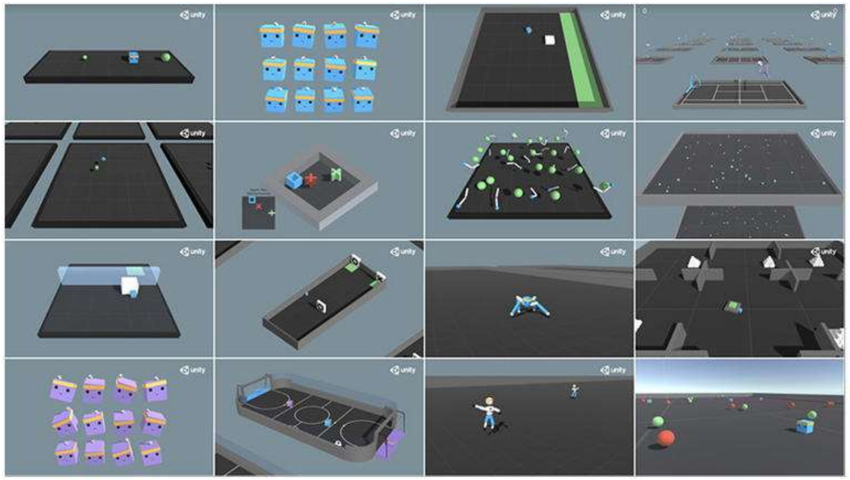
(a) Toy Text (b) Atari (c) Controls (d) MuJoCo (e) Doom (f) Minecraft

- OpenAI Baselines is a set of high-quality implementations of RL algorithms
  - See also Stable Baselines

Unity ML-Agents

- With Unity Machine Learning Agents ([ML-Agents]), you teach intelligent agents through a combination of deep reinforcement learning and imitation learning





Further Reading

- Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning – An Introduction*, 2nd ed., The MIT Press: Chap. 1-3, 6

- Simple Tutorial Videos for Deep RL:

  Introduction on Reinforcement Learning and Deep RL:

  https://www.youtube.com/watch?v=JgvyzIkgxF0

  PPO – Proximal Policy Optimization (PPO):

  https://www.youtube.com/watch?v=5P7I-xPq8u8

Conclusions

- RL enables to learn intelligent behavior in complex environments
- Large number of algorithms and approaches
- Amazing results in vintage Atari Games
- Stunning results of AlphaGo and AlphaZero
- Very promising results in Robotics
- Very fast evolution in the last few years

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Robotics / Intelligent Robotics
Introduction to Reinforcement Learning

Luís Paulo Reis, Nuno Lau, Henrique L. Cardoso, Armando Sousa

lpreis@fe.up.pt
Director of LIACC – Artificial Intelligence and Computer Science Lab.
Associate Professor at DEI/FEUP – Informatics Engineering Department, Faculty of Engineering of the University of Porto, Portugal
President of APPIA – Portuguese Association for Artificial Intelligence