IMPERIAL

February 24, 2026

Reinforcement learning

Part 1: Basic concepts

Haozhe Tian
Pietro Ferraro, Homayoun Hamedmoghadam

Background

Reinforcement Learning (RL from now on) aims to solve Sequential Decision-making Problems by interacting with the environment.

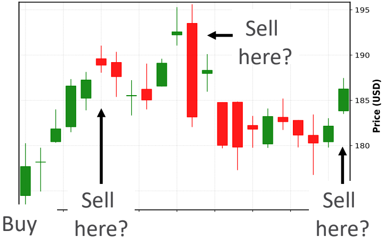
Very old topic - its been around since the 1950's and is closely related to control theory.

Some amazing successes:

- Playing any *Atari* game;
- AlphaGo mastering Go and beating the world champion;
- Tesla's Autopilot.
- Google's AI chip design.

Sequential Decision-making Problems

- **Sequential**: each decision affects future observations
- **Non-greediness**: balancing of long-term and immediate rewards

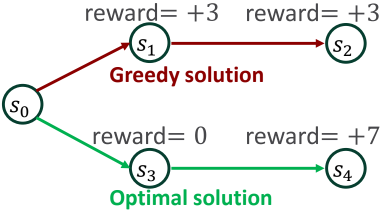


Buy

Sell here?

Sell here?

Sell here?



reward= +3 reward= +3

s_0 s_1 s_2

Greedy solution

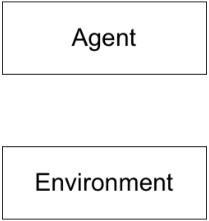
reward= 0 reward= +7

s_3 s_4

Optimal solution

Background

RL algorithms learn policies by probing an environment and seeing how well or poorly they performed on it.

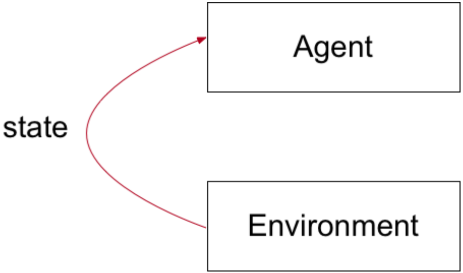


Agent

Environment

Background

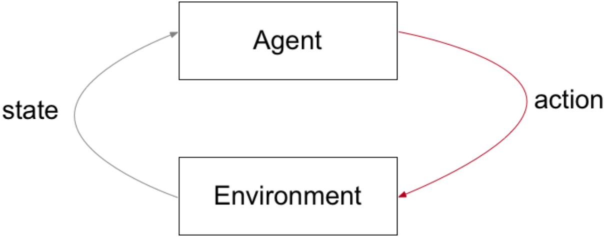
More specifically an agent, observes the environment...



IMPERIAL

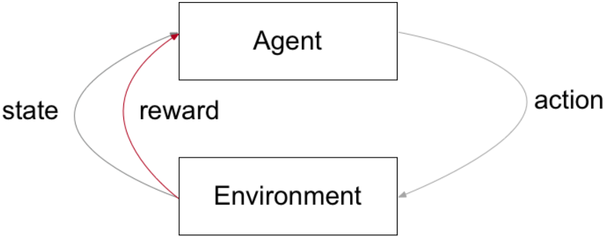
Background

…takes an action…



Background

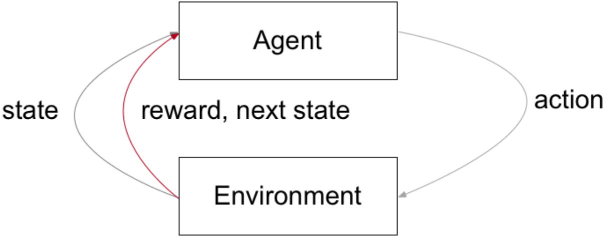
…finds out how well it did…



(Agent changes the environment and/or the environment changes on its own.)

Background

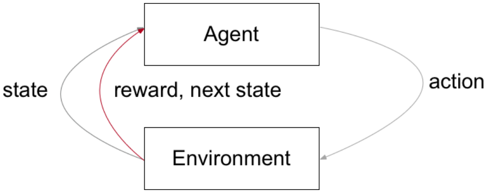
…and then the process repeats.



Background

In this framework there are many questions to be answered:

- Which state am I in?
- How do I decide which actions to take?
- Should I favour early rewards or later rewards?



Markov Decision Processes

To answer the previous question, we start by modeling this state, action, reward, next state, as a Markov Decision Process (MDP).

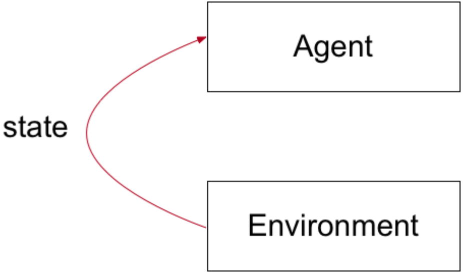
Markov Decision Processes

A MDP is characterised by:

- **State space** S; Action space A; Transition probability P; Reward function R.

Initial state $s_0 \sim \rho^0(s_0)$, where ρ^0 is the initial-state distribution.

At each step t, the agent observes a state $s_t \in S$



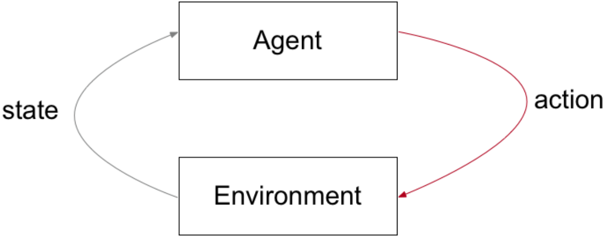
IMPERIAL

Markov Decision Processes

A MDP is characterised by:

- State space S; **Action space** A;
  Transition probability P; Reward function R.

At each step t, the agent takes an action $a_t \in A$



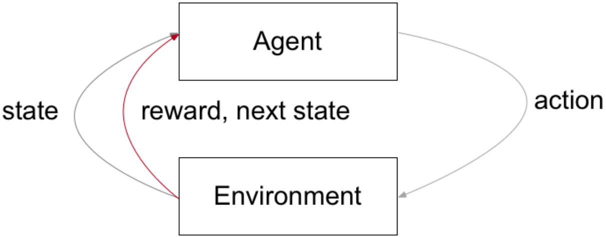
IMPERIAL

Markov Decision Processes

A MDP is characterised by:

- State space S; Action space A;
  Transition probability P; Reward function R.

The agent goes into a new state depending on a transition probability $P(s' \mid s, a)$

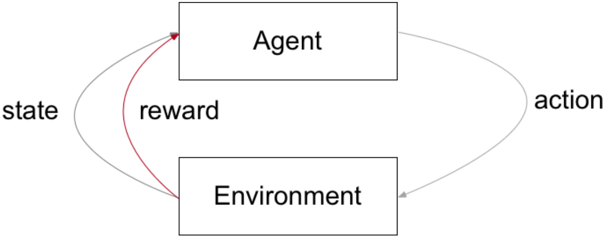


Markov Decision Processes

A MDP is characterised by:

- State space S; Action space A; Transition probability P; **Reward function** R.

By taking action a at state s, the agent receives a reward $R(s, a)$. We sometimes also denote the reward at step t as R_t.



IMPERIAL

MDP: Optimization objective

- We need to measure how “good” a policy is.
- we need to balance immediate and future rewards.
- We do this by defining the total **return**, and introducing the *discount factor* $\gamma \in (0, 1]$ into it:

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i} \tag{1}$$

- Discount is intuitive and mathematically convenient (bounded for infinite-horizon):

$$G_t \leq R_{\max} \sum_{i=0}^{\infty} \gamma^i = R_{\max} \lim_{n \to \infty} \frac{1 - \gamma^{n+1}}{1 - \gamma} = \frac{R_{\max}}{1 - \gamma}. \tag{2}$$

MDP: Optimization objective

- So, if $\gamma \rightarrow 0$, this means that we only care for the immediate return as:

$$G_t \rightarrow R_t \tag{3}$$

- On the other hand, if $\gamma = 1$, we are assuming that each future return will be as important as if it was present:

$$G_t = \sum_{i=0}^{\infty} R_{t+i} \tag{4}$$

MDP: Sumamry

To recap, an MDP can be characterized by the following elements:

- A state space S;
- An action space A;
- An initial-state distribution ρ^0;
- A transition probability $P : S \times A \times S \longrightarrow [0, 1]$;
- A reward function $R : S \times A \longrightarrow \mathbb{R}$.

These definitions may vary slightly across authors and textbooks, but the core idea remains the same: **the next state depends only on the current state and the action taken**.

MDP: Relationship with MP

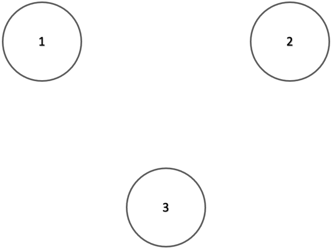
In an MDP, **the next state depends only on the current state and the action taken**.

Some of you might have noticed the similarity between Markov Processes (MP) and Markov Decision Processes (MDPs). The reason being, the latter is an extension of the former.

MDP Example

Consider a simple example:

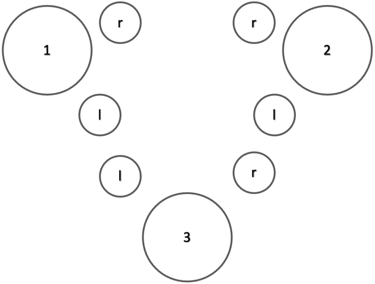
- States $S = \{1, 2, 3\}$



MDP Example

Consider a simple example:

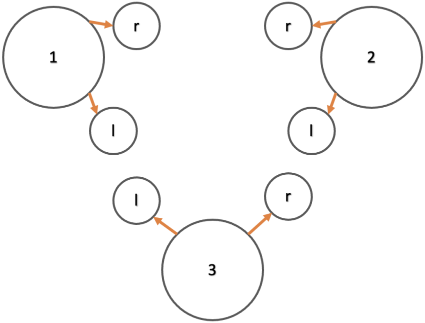
- Let's add the actions $A = \{l, r\}$



MDP Example

Consider a simple example:

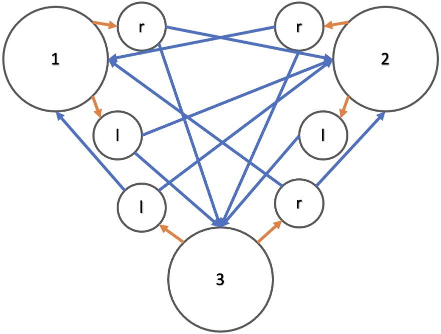
- Possible decisions at each state



MDP Example

Consider a simple example:

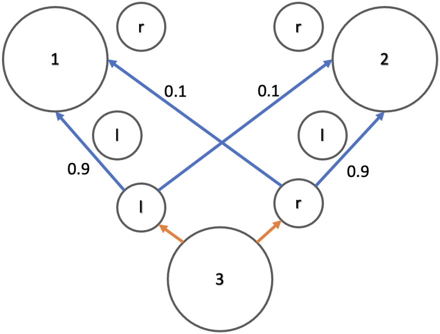
- Possible transitions between states



MDP Example

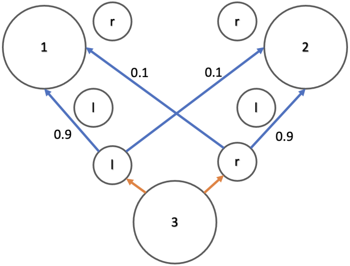
Consider a simple example:

- For a specific state and all possible actions in a stochastic environment...



MDP Example: Transition probability

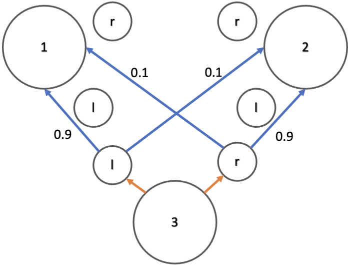
If we look at the transition probabilities, *given a specific action*, that looks very much like a Markov process. For example, assume I always move left:



$$P_l = \begin{bmatrix} 0 & 0.1 & 0.9 \\ 0.1 & 0 & 0.9 \\ 0.9 & 0.1 & 0 \end{bmatrix}$$

MDP Example: Transition probability

Now, let's assume I always move right:

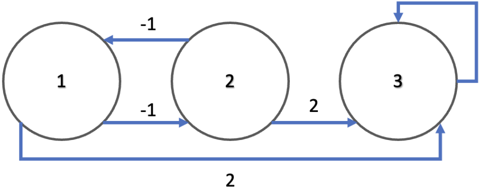


$$P_r = \begin{bmatrix} 0 & 0.9 & 0.1 \\ 0.9 & 0 & 0.1 \\ 0.1 & 0.9 & 0 \end{bmatrix}$$

An MDP can be thought of as a series of MCs, conditioned on the action you take at each state. A slightly more sophisticated way of looking at it is that a MC is defined by a $|S| \times |S|$ matrix, whereas a MDP is defined by a $|A| \times |S| \times |S|$ tensor.

Policy

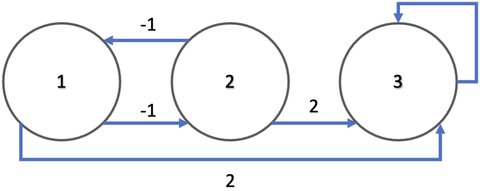
Policy is how we navigate within an MDP by choosing actions. Let's consider the following example and assume that we randomly start at a state and our goal is to reach state 3.



Reward function R(s, a):

$$R(s' = 1) = -1$$
$$R(s' = 2) = -1$$
$$R(s' = 3) = 2$$

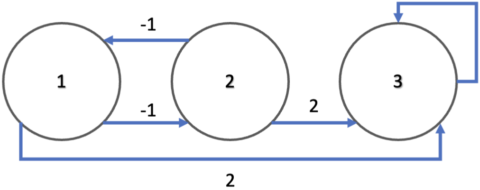
Policy



The transition matrices associated to each move are as follows

$$P_l = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad P_r = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Policy



A policy is a function $\pi : S \longrightarrow A$. In colloquial terms: how will I behave if I am in this specific state?

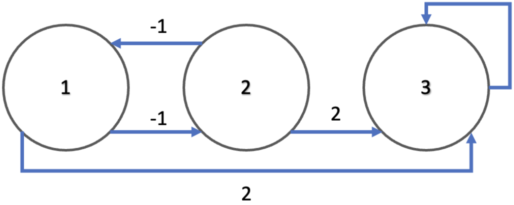
For example, a policy might be:

$$\pi(1) = r, \quad \pi(2) = l, \quad \pi(3) = l$$

Optimal policy

$$\pi(1) = r, \quad \pi(2) = l, \quad \pi(3) = l$$

Is this a good policy?



What would the optimal policy be?

Optimal policy

- What does it mean for a policy to be optimal?
- How do we even evaluate a policy?

Remember this?

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i}$$

The goal in RL is to maximize the **expected return**.

Optimal policy

A **trajectory** is a sequence of states and actions in the environment:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{5}$$

Let's write the return for such a trajectory as:

$$G(\tau) = \sum_{i=0}^{\infty} \gamma^i R_i, \quad \text{with} \quad R_i = R(s_i, a_i) \tag{6}$$

The probability of a trajectory is:

$$P^{\tau}(\tau \mid s_0, \pi) = \rho^0(s_0) \prod_{i=0}^{\infty} P(s_{i+1} \mid s_i, \pi(s_i)) \tag{7}$$

where $\rho^0(.)$ is the initial-state distribution.

Optimal policy

The expected return for a given policy will be:

$$\mathbb{E}_{\tau \sim P^\tau}[G(\tau)] = \int_\tau P^\tau(\tau \mid s_0, \pi) \cdot G(\tau) \tag{8}$$

Now we can formulate the main problem in RL in a compact form:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_{\tau \sim P^\tau}[G(\tau)] \tag{9}$$

which is finding the **optimal policy** π^*.

Value function

We evaluate how “good” a policy is for each state, using the **value function**:

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim P^{\tau}}\left[G(\tau) | s_0 = s\right] \tag{10}$$

This represents the **expected return from a state under the policy** π. You can intuitively think of this equation as:

What would be (on average) the total return if I start in state s and I follow the policy π?

IMPERIAL

Bellman Equation

Importantly, $V^{\pi}(s)$ obeys the Bellman Equation:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V^{\pi}(s') \tag{11}$$

- From the definition $\left(\mathbb{E}_{\tau \sim P^{\tau}}[G(\tau) \mid s_0 = s]\right)$, $V^{\pi}(s)$ depends on statistics over entire trajectories $s, s', s'', \dots,$ making it difficult to compute.
- The Bellman equation calculates $V^{\pi}(s)$ recursively: It depends only on the immediate reward at s and the value of the next state s'. This local relationship enables iterative computation.

IMPERIAL

Bellman Equation: Derivation

$$
\begin{aligned}
V^\pi(s) &= \mathbb{E}_{\tau\sim P^\tau}\left[G(\tau)|s_0=s\right] \\
&= \mathbb{E}_{\tau\sim P^\tau}\left[\sum_{i=0}^{\infty}\gamma^i R(s_i,a_i)|s_0=s\right] = \mathbb{E}_{\tau\sim P^\tau}\left[R(s_0,a_0)+\sum_{i=1}^{\infty}\gamma^i R(s_i,a_i)|s_0=s\right] \\
&= R(s,\pi(s)) + \mathbb{E}_{\tau\sim P^\tau}\left[\sum_{i=1}^{\infty}\gamma^i R(s_i,a_i)|s_0=s\right] \\
&= R(s,\pi(s)) + \gamma\sum_{s'}P(s'\mid s,\pi(s))\left[\underbrace{\mathbb{E}_{\tau\sim P^\tau}\left[\sum_{i=0}^{\infty}\gamma^i R(s_i,a_i)|s_0=s'\right]}_{V^\pi(s')}\right] \\
&= R(s,\pi(s)) + \gamma\sum_{s'}P(s'\mid s,\pi(s))V^\pi(s')
\end{aligned}
$$

IMPERIAL

February 24, 2026

Reinforcement learning

Part 2: Value Iteration and Deep Q-Learning

Haozhe Tian
Pietro Ferraro, Homayoun Hamedmoghadam

Recap

- We want to model decision processes
  - In order to do so we make use of Markov Decision Processes
- We want to evaluate how good a certain policy π is
  - In order to do so, we use the value function $V^{\pi}(s)$ and the Bellman Equation, defined as:

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim P^{\tau}} \left[G(\tau) | s_0 = s \right]$$

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V^{\pi}(s')$$

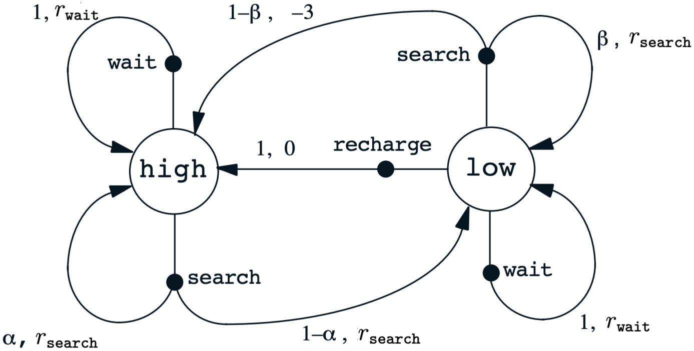
Bellman equation: Recycling robot[1]

- The recycling robot can **search** the office for empty cans, **wait** for someone to bring an empty can, and **recharge** at its base if the battery is low.
- Searching collects more cans but drains the battery.
- If the battery is drained, the robot needs rescue and recharge.

[1] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. Vol. 1, no. 1. Cambridge: MIT press, 1998.

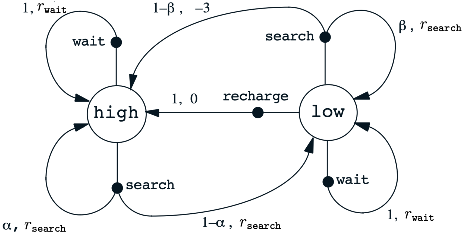
Recycling robot

Here we consider the simple environment with only two states of low/high battery level; $S = \{l, h\}$.



IMPERIAL

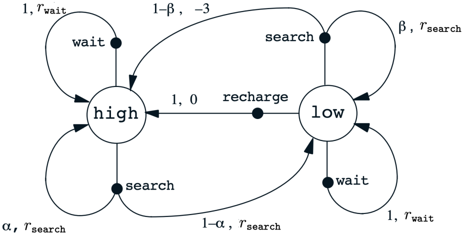
Recycling robot



Let's calculate the value function for a simple deterministic policy: $\pi(\mathbf{l}) = \mathbf{r}$, $\pi(\mathbf{h}) = \mathbf{s}$, and discount factor $\gamma = 0.5$:

$$V^{\pi}(\mathbf{l}) = R(\mathbf{l}, \mathbf{r}) + \gamma \sum_{s'} P_r(s, s') V^{\pi}(s') = \gamma V^{\pi}(\mathbf{h})$$

Recycling robot



$\pi(\mathbf{l}) = \mathbf{r}$, $\pi(\mathbf{h}) = \mathbf{s}$, and discount factor $\gamma = 0.5$:

$$
\begin{aligned}
V^{\pi}(\mathbf{h}) &= R(\mathbf{h}, \mathbf{s}) + \gamma \sum_{s'} P_r(s, s') V^{\pi}(s') \\
&= r_{search} + \gamma[\alpha V^{\pi}(\mathbf{h}) + (1 - \alpha) V^{\pi}(\mathbf{l})]
\end{aligned}
$$

Recycling robot

$\pi(\mathbf{l}) = \mathbf{r}$, $\pi(\mathbf{h}) = \mathbf{s}$, and $\gamma = 0.5$:

$$\begin{cases} V^\pi(\mathbf{l}) = \gamma V^\pi(\mathbf{h}) \\ V^\pi(\mathbf{h}) = r_{search} + \gamma\left[\alpha V^\pi(\mathbf{h}) + (1-\alpha)V^\pi(\mathbf{l})\right] \end{cases}$$

Solving the system of equations, we get:

$$V^\pi(\mathbf{h}) = \frac{r_{search}}{1 - \gamma\alpha - \gamma^2 + \gamma^2\alpha} = \frac{r_{search}}{0.75 - 0.25\alpha}, V^\pi(\mathbf{l}) = \frac{r_{search}}{1.5 - 0.5\alpha}$$

Policy evaluation

The previous problem is easy to solve analytically. But what if there are 10 states? What about 1000? What about 10^{10}?

- In that case, it is difficult or impossible to solve the problem using a system of linear equations.
- The solution is to use approximate iterative methods from Dynamic Programming that represent the starting point for most RL algorithms.

Policy evaluation

We make use of the Bellman equation to find a converging estimate of the value function:

Algorithm 1 Policy evaluation

```
**Inputs:** $\pi, P, R, \gamma, \epsilon$; **Output:** $V^{\pi}(s)$ for all $s \in S$
Set $k \leftarrow 0$ and initialize $V_0^{\pi}(s) \leftarrow 0$ for all $s \in S$
**repeat**
    $k \leftarrow k + 1, \quad \Delta \leftarrow 0$
    **for all** $s \in S$ **do**
        $V_k^{\pi}(s) \leftarrow R(s, \pi(s)) + \gamma \sum_{s' \in S} P(s' \mid s, \pi(s))\, V_{k-1}^{\pi}(s')$
        $\Delta \leftarrow \max(\Delta, |V_k^{\pi}(s) - V_{k-1}^{\pi}(s)|)$
    **end for**
**until** $\Delta < \epsilon$
```

Policy evaluation

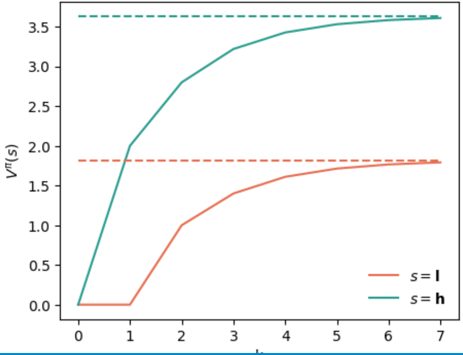
This is called *Bootstrapping*. We are using estimates of the value function to estimate the value function. This might sound weird but there are two reasons this works:

- The Bellman equation (better to say "operator") is a contraction mapping (γ-contraction).
  This means that $||V^{\pi}_{k+1}(s) - V^{\pi}_{k}(s)|| \leq \gamma ||V^{\pi}_{k}(s) - V^{\pi}_{k-1}(s)||$.
  This implies that at some point in the future the algorithm will converge to *something*.
- The true value $V_{\pi}(s)$ is a fixed point of the Bellman operator.
  This means that this *something* the previous algorithm converges to is $V_{\pi}(s)$.

Policy evaluation: Recycling robot

Let's assume for the recycling robot: $\alpha, \beta = 0.8$, $\gamma = 0.5$, and $r_{search} = 3$, $r_{wait} = 2$, and run Algorithm 1 on the policy we chose earlier ($\pi(\mathbf{l}) = \mathbf{r}, \pi(\mathbf{h}) = \mathbf{s}$).

Dashed lines show the manual calculations from Slide 42.



Policy Improvement

Alright, we can find $V_\pi(s)$. But what we really care about is to find a good policy, right? Specifically, in the previous part we said that we want to find the *optimal* policy.

Using the value function we can write this as:

$$\pi^* = \operatorname*{argmax}_{\pi} V^{\pi}(s), \quad \forall s \in S \tag{12}$$

Action-value function

To find the optimal policy, we can use the help of an equation similar to the value function:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau \sim P^{\tau}}[G(\tau) | s_0 = s, a_0 = a] \tag{13}$$

Action-value function is the value of taking action a in state s and then follow the policy π.

Action-Value function

Action-value function also obeys the Bellman Equation:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) Q^\pi(s', \pi(s')) \tag{14}$$

Note the connection to value function:

$$Q^\pi(s,\pi(s)) = V^\pi(s), \tag{15}$$

which allows us to rewrite Q in terms of value function V:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^\pi(s'). \tag{16}$$

Action-Value function

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P(s' \mid s, \pi(s)) V^{\pi}(s')$$
$$Q^{\pi}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^{\pi}(s')$$

The way to think about Q is the following:

What would be (on average) the total return if I start in state s, take action a, and follow the policy π from there on?

This is very useful: it gives us a way to evaluate if taking a is *better* or *worse* than following policy π

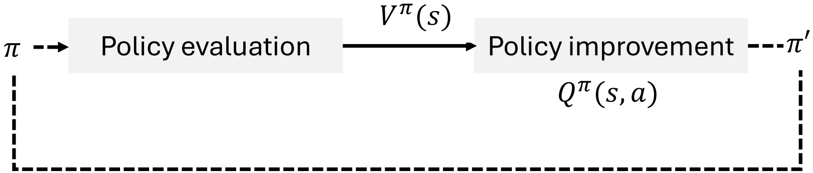
Policy Improvement

Algorithm 2 Policy improvement

```
**Inputs:** $V^\pi, \pi, P, R, \gamma$; **Output:** $\pi'$
Initialize $Q^\pi(s, a) \leftarrow 0$ for all $(s, a) \in S \times A$
**for all** $s \in S$ **do**
    **for all** $a \in A$ **do**
        $Q^\pi(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a)\, V^\pi(s')$
    **end for**
    $\pi'(s) \leftarrow \operatorname*{argmax}_a Q^\pi(s, a)$
**end for**
```

Policy Iteration: Finding the best policy

If π' (from Alg. 2) differs from the input π, Alg. 1 is ran to get the new value function and then again back to Alg. 2, and this is repeated until convergence: $\pi' = \pi$.



π

Policy evaluation

$V^\pi(s)$

Policy improvement

$Q^\pi(s,a)$

π'

Recap

- We use MDPs to model a specific system;
- We can decide on a policy π
- Using Policy evaluation and Policy improvement we can find the best possible policy π^*
- These algorithms are all based on the Bellman Equation

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P(s' \mid s, \pi(s)) V^\pi(s')$$

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^\pi(s')$$

Reinforcement Learning

Alright, so if we can find the optimal policy for every possible MDP, why don't we do it?

One big problem is that we have made a huge assumption so far. Can you point it out?

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P(s' \mid s, \pi(s)) V^{\pi}(s')$$

$$Q^{\pi}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^{\pi}(s')$$

Reinforcement Learning

- Policy evaluation and Policy improvement require $P(\cdot)$, therefore is termed **Dynamic Programming**.
- **Reinforcement Learning (RL)** aims to find $\pi^\star$ without knowing $P(\cdot)$.

In the past, all RL methods were based on the Bellman equations. Recently, new algorithms, e.g., *Policy-gradient* methods, have become popular. All of them try to maximise $\mathbb{E}_{s\sim\rho_0}[V^\pi(s)]$ by the right policy π.

A famous example: Q-learning

Q-learning assumes that we have NO idea about the dynamics of the system $P(s' \mid s, a)$.

We start with an arbitrary Q-function $Q(s, a)$. Our policy is to choose $\pi(s) = \text{argmax}_a Q(s, a)$ for all states.

At state s, we choose action $a = \pi(s)$, we obtain reward r, and we go on to state s'. Then we update the function $Q(s, a)$:

$$Q(s,a) \leftarrow \underbrace{Q(s,a)}_{\text{Current value}} + \alpha \underbrace{\left(r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right)}_{\text{Temporal difference}}$$

A famous example: Q-learning

$$Q(s,a) \leftarrow \underbrace{Q(s,a)}_{\text{Current value}} + \alpha \underbrace{\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)}_{\text{Temporal difference}}$$

The Q-learning update brings $Q(s,a)$ close to $r + \gamma \max_{a'} Q(s',a')$. On average, since we are sampling (s,a,r,s') with π, this becomes a sample-based estimate of:

$$\mathbb{E}_{a=\pi(s),r=R(s,a),s'\sim P(s'|s,a)}\left[r + \gamma \max_{a'} Q(s',a')\right]$$

It can be proven that Q-learning will converge (given enough sampling) to the same exact policy as in Dynamic Programming.

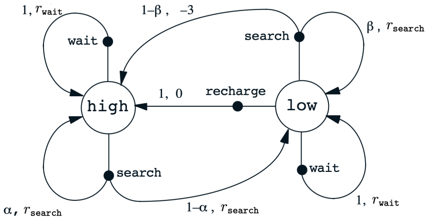
Q-learning example: Recycling robot

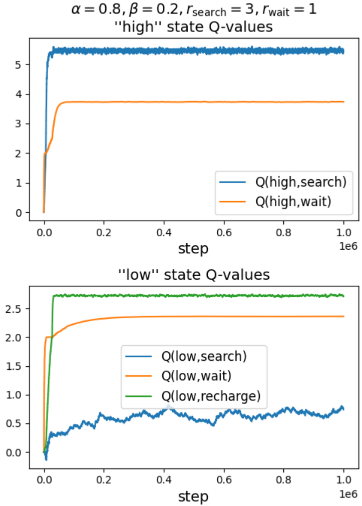
```
alpha, beta, r_s, r_w = 0.7, 0.6, 2, -0.2
gamma = 0.5

def robot_step(s, a):
    match (s, a):
        case (’high’, ’wait’):
            return (’high’, r_w)
        case (’high’, ’search’):
            return (’high’, r_s) if random.random() < alpha\
                               else (’low’, r_s)
        case (’low’, ’wait’):
            return (’low’, r_w)
        case (’low’, ’search’):
            return (’low’, r_s) if random.random() < beta\
                               else (’high’, -3)
        case (’low’, ’recharge’):
            return (’high’, 0)
```

Q-learning example: Recycling robot

```
Q = {’low’: {’wait’: 0, ’search’: 0, ’recharge’: 0},
     ’high’: {’wait’: 0, ’search’: 0}}
epsilon = 0.05
lr = 0.005

s = ’high’
for _ in range(1000000):
    # epsilon-greedy policy
    if random.random() < epsilon:
        a = random.choice(list(Q[s])) # explore
    else:
        a = max(Q[s], key=Q[s].get) # exploit
    # move in the environment
    s_next, r = robot_step(s, a)
    # Q-learning update
    Q[s][a] = Q[s][a] + \
        lr * (r + gamma * max(Q[s_next].values()) - Q[s][a])
    s = s_next
```

Q-learning example: Recycling robot





IMPERIAL

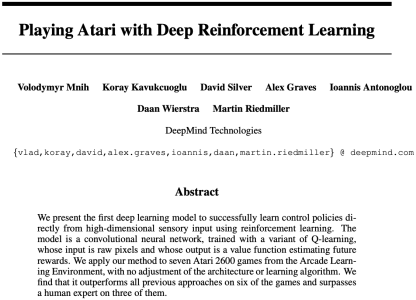
RL with function approximation

Q-learning stores a table of state-action pairs, which becomes exponentially large as the number of states and actions increase, and becomes intractable for continuous states and actions.

For a long time, people have tried to solve that by using approximators of the Q-functions. That is, instead of using a table, use a neural network that receives (s, a) as input and predicts $Q^{\pi}(s, a)$. In this case, convergence to the optimum is no longer guaranteed. Neural networks are also hard to train given a sparse learning signal (as in RL).

RL with function approximation

[Minh et al.] introduced the replay buffer, which trains neural networks using a batch of randomly sampled historical transitions (s, a, r, s'). This stabilizes training and enables CNN-based Q networks to be optimized using a Q-learning–based objective.



Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**
Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.