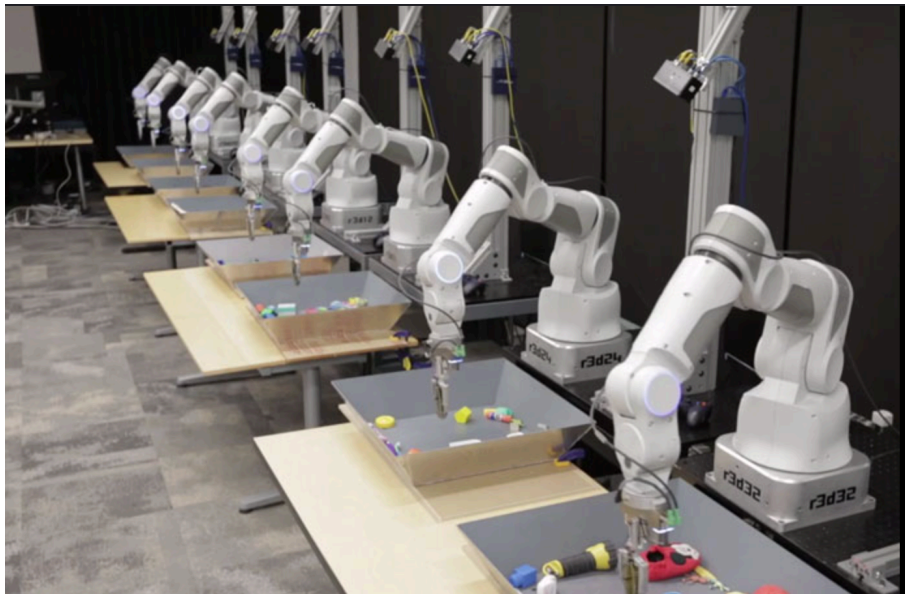
Reinforcement Learning (Part 1/2)

Alexandre Bergel
DCC - University of Chile
http://bergel.eu
01-12-2025

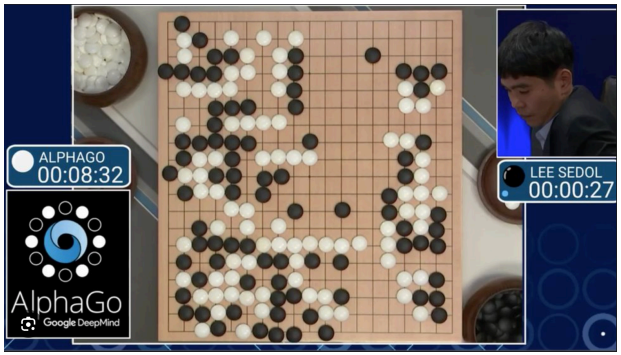


ChatGPT


LEAGUE OF LEGENDS


StarCraft II


DOUBLE KILL!


ALPHAGO
00:08:32
LEE SEDOL
00:00:27
AlphaGo
Google DeepMind

Goal of today

Provide an *overview of Reinforcement Learning*

See a *generic implementation* of Reinforcement Learning

Provide the necessary knowledge to use Reinforcement Learning *in another context*

Outline

1. Overview of Reinforcement Learning
2. Building a RL environment

Outline

1. **Overview of Reinforcement Learning**
2. Building a RL environment

Reinforcement learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability *to automatically learn and improve* from *experience* without being explicitly programmed

Reinforcement learning (RL) is an area of machine learning

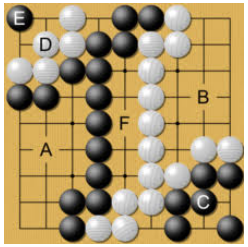
RL is about taking a *suitable action* to *maximize reward* in *a particular situation*

Reinforcement learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability *to automatically learn and improve* from *experience* without being explicitly programmed

Reinforcement learning (RL) is an area of machine learning

RL is about taking a *suitable action* to *maximize reward* in *a particular situation*



Where should I play in order to strengthen my situation?

Reinforcement learning

2017

The game begins with a model that know *very few* about the game of go (just the rules)

It then plays *games against itself*

More games the model play, *better it is at predicting moves* and identifying which players will win

Characteristics of Reinforcement Learning

What makes RL different from other machine learning paradigms?

There is no supervisor, only *a reward signal*

Feedback is delayed, not instantaneous

Time really matters (ie. sequentiality of actions)

Agent's *actions affect the subsequent data* it receives

Reward

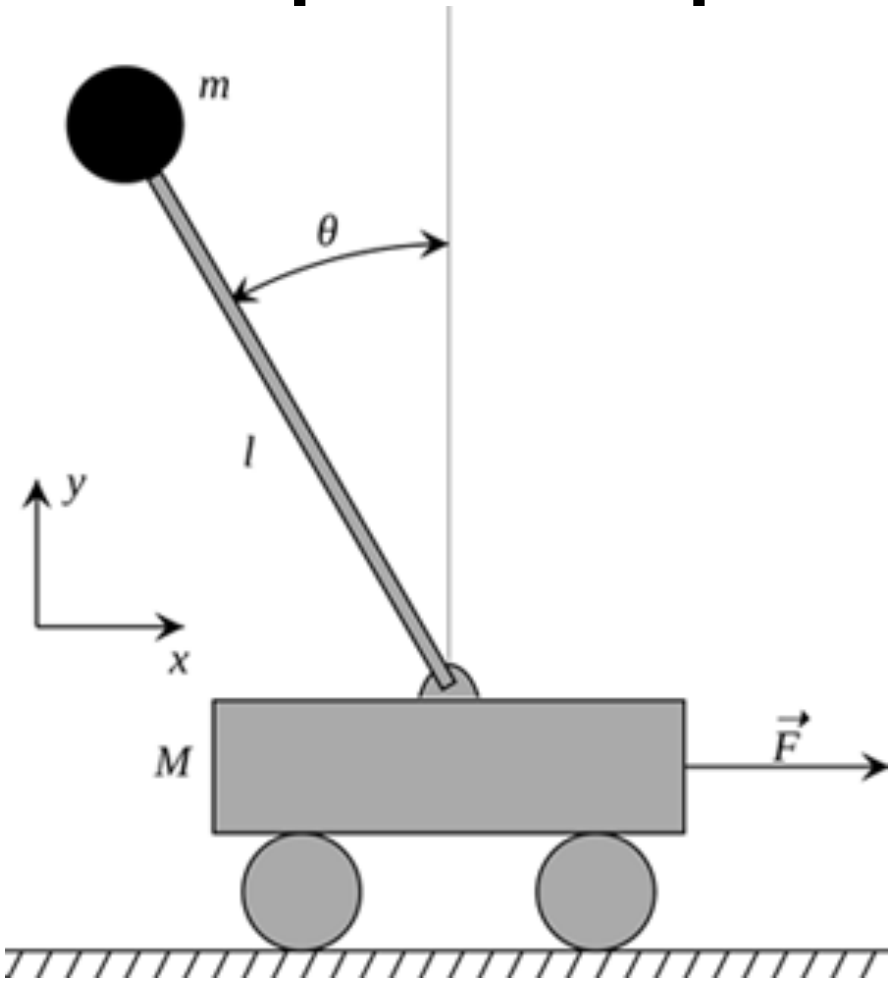
A *reward* R_t is a *scalar feedback signal* (a plain number)

Indicates how well agent is doing at step t

A *high reward* means the agent made a *good action*

The agent's job is to *maximize the cumulative reward* (i.e., sum of the reward across multiple actions)

Example - Car-pole balancing



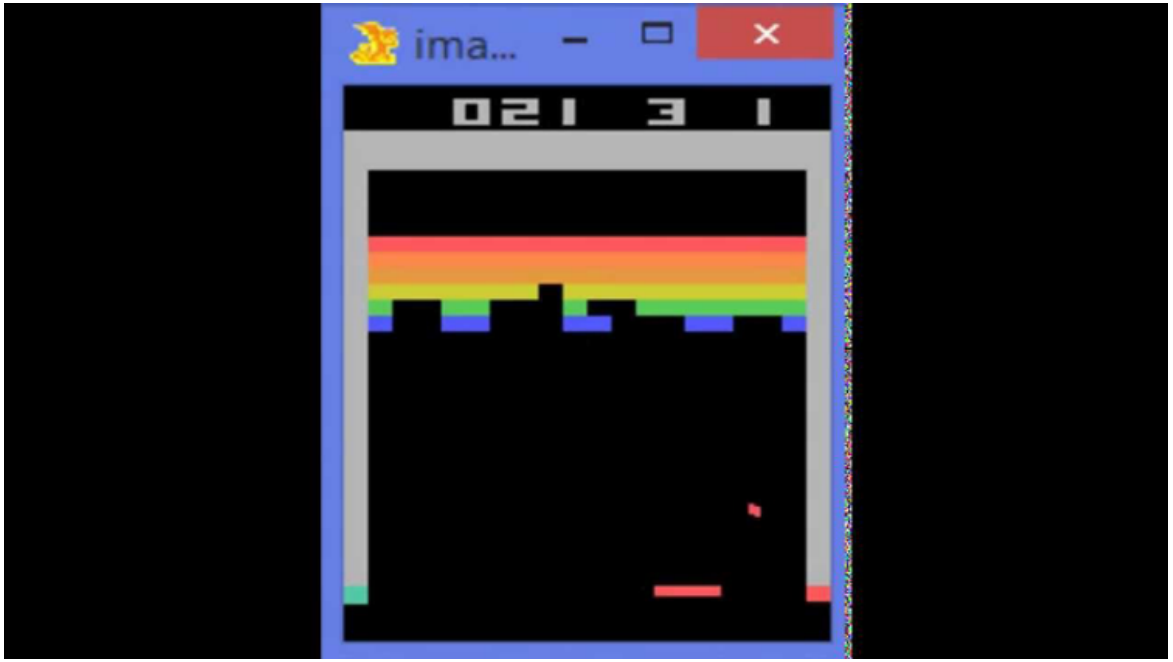
Goal: balance the pole on top of a moving cart

State: angle, angular speed, position, horizontal velocity

Actions: horizontal force to the cart

Reward: 1 at each time step if the pole is upright

Atari games



Atari games are very popular because you cannot efficiently extract a set of data to train a model (e.g., deep learning)

Goal: beat the game with the highest score

State: raw game pixels of the game

Actions: up, down, left, right, etc

Reward: score provided by the game

Doom

Goal: eliminate all opponents

State: raw game pixels of the game

Actions: up, down, left, right, etc

Reward: positive when eliminating an opponent, negative when the agent is eliminated

Training robots for bin packing

Goal: pick a device from a box and put it into a container

State: raw pixels of the real world

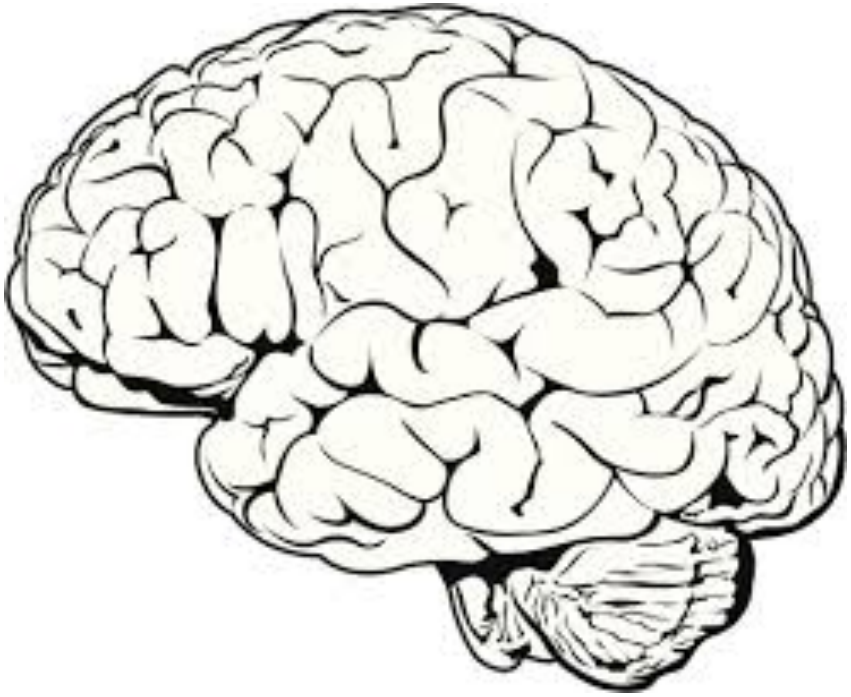
Actions: possible actions of the robot

Reward: positive when placing a device successfully, negative otherwise

Agent

Observation

O_t

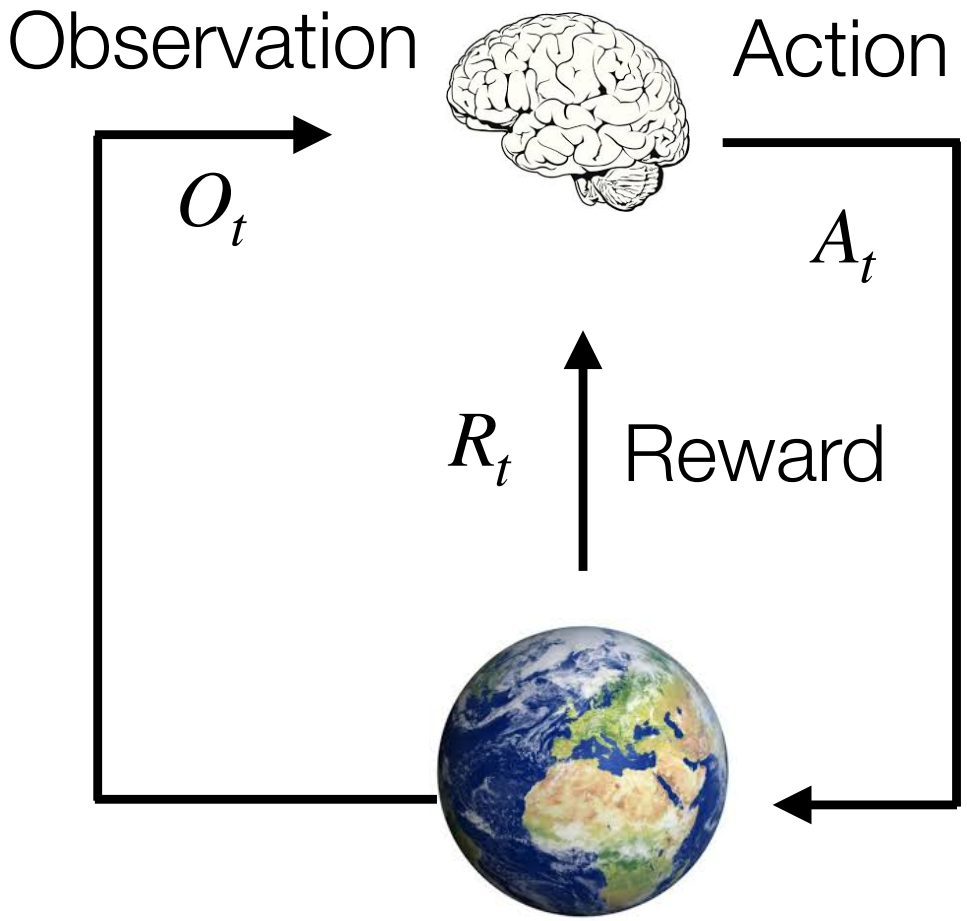


Action

A_t

R_t Reward

Agent and Environment



At each step t the agent:

- Executes actions A_t
- Receives observation O_t
- Receives scalar reward R_t

The environment:

- Receives action A_t
- Emits observation O_{t+1}
- Emits scalar rewards R_{t+1}

Time increments at env. step

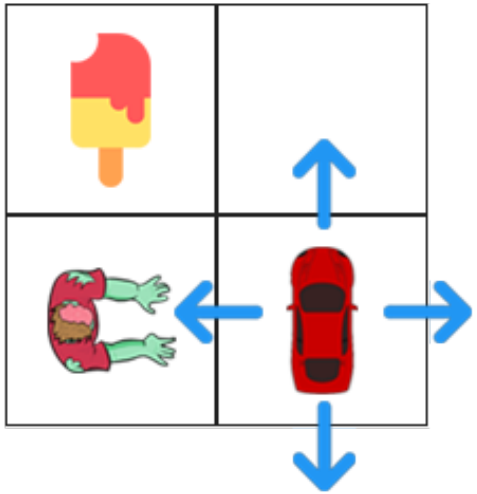
Outline

1. Overview of Reinforcement Learning
2. **Building a RL environment**

Detailed example

The following slides will present a *simple* but *representative example*

A small car will have to *find food* in a very small and simple map while *avoiding zombie*



Building the environment

The initial state looks like this:

```
ZOMBIE = "z"
CAR = "c"
ICE_CREAM = "i"
EMPTY = "*"

grid = [
    [ICE_CREAM, EMPTY],
    [ZOMBIE, CAR]
]

for row in grid:
    print(' '.join(row))
```

Building the environment

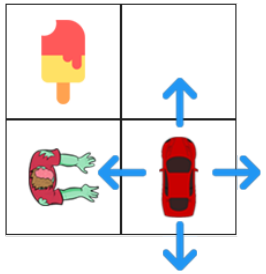
The initial state looks like this:

```
ZOMBIE = "z"
CAR = "c"
ICE_CREAM = "i"
EMPTY = "*"

grid = [
    [ICE_CREAM, EMPTY],
    [ZOMBIE, CAR]
]

for row in grid:
    print(' '.join(row))
```

Define the map in which the robot will live in



Building the environment

The initial state looks like this:

```
ZOMBIE = "z"
CAR = "c"
ICE_CREAM = "i"
EMPTY = "*"

grid = [
    [ICE_CREAM, EMPTY],
    [ZOMBIE, CAR]
]
```

```
for row in grid:
    print(' '.join(row))
```

```
i *
z c
```

Modeling the state

In RL, an agent *observes the world* and learns how to make appropriate decision

An agent *sees the world through a state*

A state is a *complete snapshot* of the world

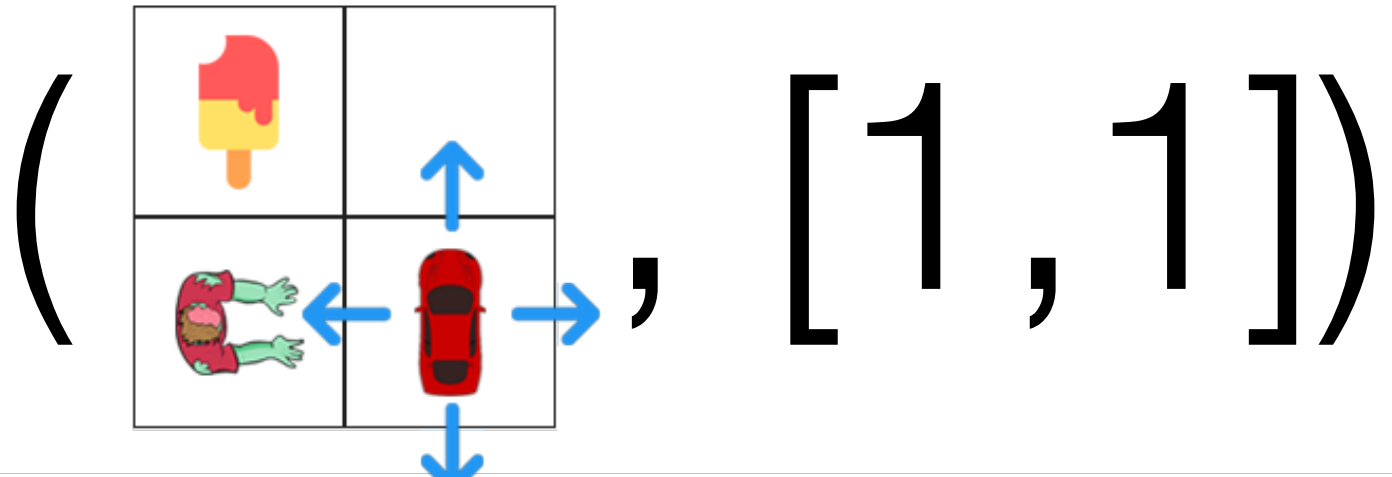
For a given state, the agent learns which decision is best, through multiple exploration

Modeling the state

Our environment state is wrapped in a class that holds the current grid and car position.

A state is *(grid, car position)*

Example of a state:



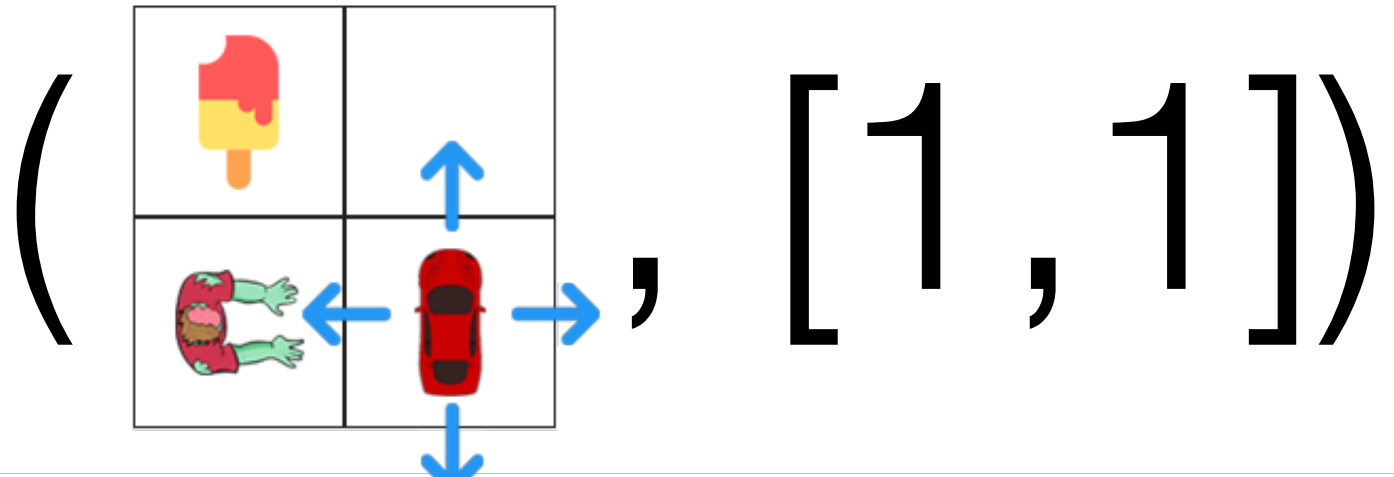
Modeling the state

Our environment state is wrapped in a class that holds

A st

Exam

Note that the grid already knows the car position. But by explicitly providing its position, we do not to search for the car in the grid



Modeling the state

```
class State:

    def __init__(self, grid, car_pos):
        self.grid = grid
        self.car_pos = car_pos

    def __eq__(self, other):
        return isinstance(other, State) and self.grid == other.grid and
self.car_pos == other.car_pos

    def __hash__(self):
        return hash(str(self.grid) + str(self.car_pos))

    def __str__(self):
        return f"State(grid={self.grid}, car_pos={self.car_pos})"
```

Modeling the state

```
class State:
    def __init__(self, grid, car_pos):
        self.grid = grid
        self.car_pos = car_pos

    def __eq__(self, other):
        return isinstance(other, State) and self.grid == other.grid and
self.car_pos == other.car_pos

    def __hash
        return
                                                    os))

    def __str_
        return
                                                    elf.car_pos})"
```

Define the constructor of the class State.
You can create a state as for example:

```
State(grid= [
    [ICE_CREAM, EMPTY],
    [ZOMBIE, CAR]
], car_pos=[1, 1])
```

Modeling the state

```
class State:

    def __init__(self, grid, car_pos):
        self.grid = grid
        self.car_pos = car_pos

    def __eq__(self, other):
        return isinstance(other, State) and self.grid == other.grid and
self.car_pos == other.car_pos

    def __hash__(self):
        return hash(str(self.grid) + str(self.car_pos))
```

RL looks for the best actions for a given state. It is therefore crucial to determine if two states are the same or not. If we are not able to say if two states are the same, then the algorithm will only see new states.
We need a way to say whether two states are equals or not.

Modeling the state

```
class State:

    def __init__(self, grid, car_pos):
        self.grid = grid
        self.car_pos = car_pos

    def __eq__(self, other):
        return isinstance(other, State) and self.grid == other.grid and
self.car_pos == other.car_pos

    def __hash__(self):
        return hash(str(self.grid) + str(self.car_pos))
```

Equality is defined as follows: A state self is equals to an object other if:

- *other is instance of the class State*
- *grid of other is equals of the grid of self*
- *The car position of other is equals to the car position of self*

Modeling the state

A hash value is a number that represents an object. It is a general notion in software engineering and is extremely useful for speeding up data accesses
hash("hello") == 8406516712461168 61
hash("hello") == hash("hel" + "lo")

We will keep grid in a dictionary, and to do so, we must define a hash method. There are many ways to define hash method, we here just pick one that is easy and convenient

```
        return isinstance(other, State) and self.grid == other.grid and
    self.car_pos == other.car_pos

    def __hash__(self):
        return hash(str(self.grid) + str(self.car_pos))

    def __str__(self):
        return f"State(grid={self.grid}, car_pos={self.car_pos})"
```

Modeling the state

```
class State:

    def __init__(self, grid, car_pos):
        self.grid = grid
        self.car_pos = car_pos

    def __eq__(self, other):
        return self.grid == other.grid and self.car_pos == other.car_pos

    def __hash__(self):
        return hash(str(self.grid) + str(self.car_pos))

    def __str__(self):
        return f"State(grid={self.grid}, car_pos={self.car_pos})"
```

Python is very much text oriented. Being able to print objects is useful to understand what is going on. This is very useful when debugging.

Modeling the actions

RL is about making an agent learn the good actions from the bad ones

An *action* is represented as *an integer*

Convenient in many cases (e.g., could be used as an index of an array or a matrix)

Modeling the actions

Each action is a number

That number represents an index in the variable ACTIONS

```
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3

ACTIONS = [UP, DOWN, LEFT, RIGHT]
```

Initial state

RL computes the *cumulative rewards* over a *sequence of actions*

An exploration step is called an *episode*

It is therefore important to have the *same initial state* at each episode

It is not essential, but considerably eases the learning

Initial state

The initial state is formulated as:

```
start_state = State(grid=grid, car_pos=[1, 1])
```

Initial state

Each exploration (episode) will therefore begin with the very same state

```
grid = [
    [ICE_CREAM, EMPTY],
    [ZOMBIE, CAR]
]
```

The initial state is formulated as:

```
start_state = State(grid=grid, car_pos=[1, 1])
```

Moving the car

We need a function to move the car around

```
def new_car_pos(state, action):
    p = deepcopy(state.car_pos)
    if action == UP:
        p[0] = max(0, p[0] - 1)
    elif action == DOWN:
        p[0] = min(len(state.grid) - 1, p[0] + 1)
    elif action == LEFT:
        p[1] = max(0, p[1] - 1)
    elif action == RIGHT:
        p[1] = min(len(state.grid[0]) - 1, p[1] + 1)
    else:
        raise ValueError(f"Unknown action {action}")
    return p
```

Moving the car

We need a function to move the car around

```
def new_car_pos(state, action):
    p = deepcopy(state.car_pos)
    if action == UP:
        p[1] = max(0, p[1] - 1)
    elif action == RIGHT:
        p[1] = min(len(state.grid[0]) - 1, p[1] + 1)
    else:
        raise ValueError(f"Unknown action {action}")
    return p
```

The function gives a state and action as parameter, and returns the new car position

Moving the car

We need a function to move the car around

```
def new_car_pos(state, action):
    p = deepcopy(state.car_pos)
    if action == UP:
        p[0] = max(0, p[0] - 1)
    elif action == DOWN:
        ...
        raise ValueError(f"Unknown action {action}")
    return p
```

A position is defined as a list. E.g., [1,1]
It is important to make sure that we do not do any unwanted side effect.
Making a copy of it prevent us from modifying objects that are used elsewhere

Moving the car

We need a function to move the car around

```
def new_car_pos(state, action):
    p = deepcopy(state.car_pos)
    if action == UP:
        p[0] = max(0, p[0] - 1)
    elif action == DOWN:
        p[0] = min(len(state.grid) - 1, p[0] + 1)
    elif action == LEFT:
        p[1] = max(0, p[1] - 1)
```

The variable p indicates the position of the car. p = [Y,X]
If the car has to go UP, then we reduce by 1 the first element of p
We use max to make sure that the car does not exit the map

Moving the car

We need a function to move the car around

```
def new_car_pos(state, action):
    p = deepcopy(state.car_pos)
    if action == UP:
        p[0] = max(0, p[0] - 1)
    elif action == DOWN:
        p[0] = min(len(state.grid) - 1, p[0] + 1)
    elif action == LEFT:
        p[1] = max(0, p[1] - 1)
    else: raise ValueError("Unknown action {}".format(action))
    return p
```

Similarly, we make sure that the Y coordinate does not exit the bottom of the map

Moving the car

We need a function to move the car around

```
def new_car_pos(state, action):
    p = deepcopy(state.car_pos)
    elif action == DOWN:
        p[0] = min(len(state.grid) - 1, p[0] + 1)
    elif action == LEFT:
        p[1] = max(0, p[1] - 1)
    elif action == RIGHT:
        p[1] = min(len(state.grid[0]) - 1, p[1] + 1)
    else:
        raise ValueError(f"Unknown action {action}")
    return p
```

Similarly, when going right, the car must not exit the map

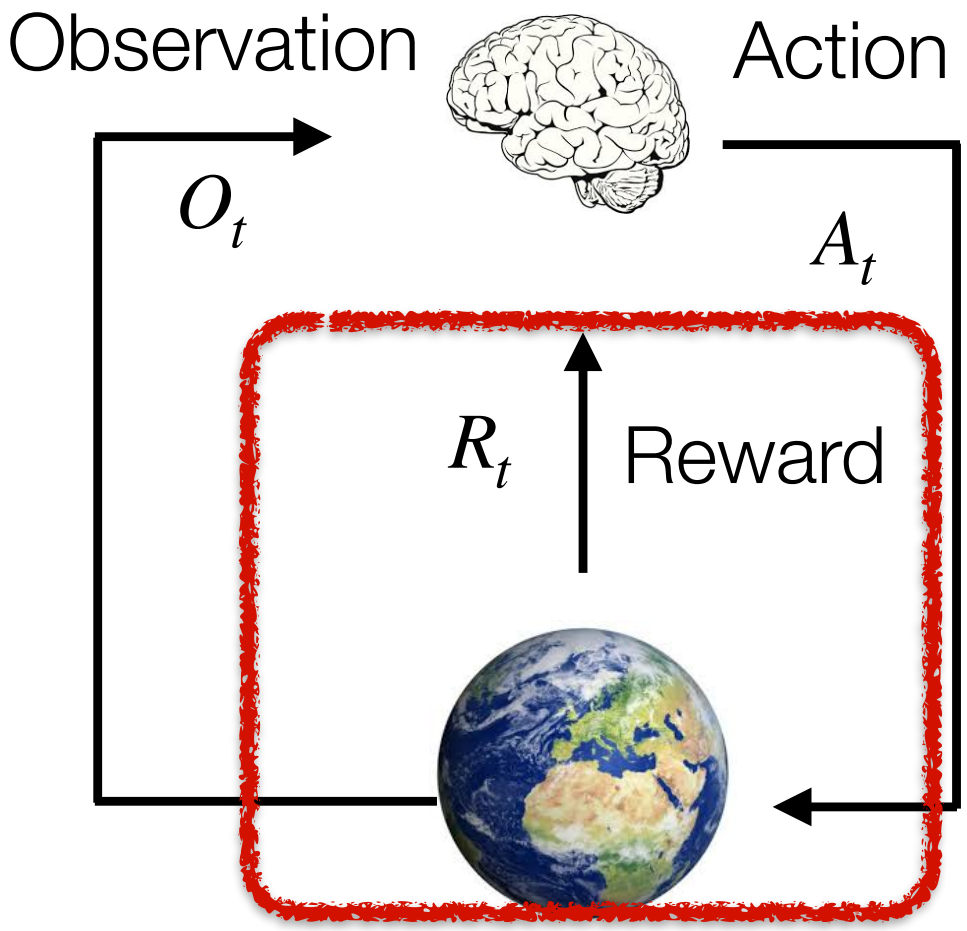
Moving the car

We need a function to move the car around

```
def new_car_pos(state, action):
    p = deepcopy(state.car_pos)
    if action == UP:
        p[0] = max(0, p[0] - 1)
    elif action == DOWN:
        p[0] = min(len(state.grid) - 1, p[0] + 1)
    elif action == LEFT:
        p[1] = max(0, p[1] - 1)
    elif action == RIGHT:
        p[1] = min(len(state.grid[0]) - 1, p[1] + 1)
    else:
        raise ValueError(f"Unknown action {action}")
    return p
```

Return a new car position

Agent and Environment



At each step t the agent:

- Executes actions A_t
- Receives observation O_t
- Receives scalar reward R_t

The environment:

- Receives action A_t
- Emits observation O_{t+1}
- Emits scalar rewards R_{t+1}

Time increments at env. step

Interacting with the environment

Our agent needs a way to *interact with the environment*, in order to to choose an action

We define a *function* that takes the *current state* with an *action*, and *returns new state*, *reward*, and whether or not the *episode has completed*:

```
from copy import deepcopy
def act(state, action):
    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)

    if grid_item == ZOMBIE:
        reward = -100
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == ICE_CREAM:
        reward = 1000
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == EMPTY:
        reward = -1
        is_done = False
        old = state.car_pos
        new_grid[old[0]][old[1]] = EMPTY
        new_grid[p[0]][p[1]] = CAR
…
```

```
from copy import deepcopy
def act(state, action):
    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)
    elif grid_item == ICE_CREAM:
        reward = 1000
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == EMPTY:
        reward = -1
        is_done = False
        old = state.car_pos
        new_grid[old[0]][old[1]] = EMPTY
        new_grid[p[0]][p[1]] = CAR
…
```

This is an essential function that we will use to execute an episode.
The function returns a new state, the reward, and an indication on whether the episode had ended or not.
It is important to indicate whether the episode has ended or not

```
from copy import deepcopy
def act(state, action):
    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)
        reward = -100
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == ICE_CREAM:
        reward = 1000
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == EMPTY:
        reward = -1
        is_done = False
        old = state.car_pos
        new_grid[old[0]][old[1]] = EMPTY
        new_grid[p[0]][p[1]] = CAR
…
```

Obtain a new position of the car

```
from copy import deepcopy
def act(state, action):
    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)

    if grid_item == ZOMBIE:
        reward = -100
        reward = 1000
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == EMPTY:
        reward = -1
        is_done = False
        old = state.car_pos
        new_grid[old[0]][old[1]] = EMPTY
        new_grid[p[0]][p[1]] = CAR
…
```

The function act returns a new state, so, we need to create a new grid. We simply copy the grid from the actual state, and we modify it

```
from copy import deepcopy
def act(state, action):
    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)

    if grid_item == ZOMBIE:
        reward = -100
        is_done = True
        new_grid[p[0]][p[1]] = CAR
    elif grid_item == ICE_CREAM:
        reward = 1000
        is_done = True
        is_done = False
        old = state.car_pos
        new_grid[old[0]][old[1]] = EMPTY
        new_grid[p[0]][p[1]] = CAR
…
```

If the car is on the zombie, then we have a very negative rewards.
The episode is ended, and we update the map

```
from copy import deepcopy
def act(state, action):
    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)

    if grid_item == ZOMBIE:
        reward = -100
        is_done = True
        new_grid[p[0]][p[1]] = CAR
    elif grid_item == ICE_CREAM:
        reward = 1000
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == EMPTY:
        reward = -1
        is_done = False
...
```

If the car is on the ice cream, then we have a positive negative rewards.
The episode is ended, and we update the map

```
from copy import deepcopy
def act(state, action):
    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)
    ...
    reward = 1000
    is_done = True
    new_grid[p[0]][p[1]] += CAR
elif grid_item == EMPTY:
    reward = -1
    is_done = False
    old = state.car_pos
    new_grid[old[0]][old[1]] = EMPTY
    new_grid[p[0]][p[1]] = CAR
...
```

If the car is on an empty cell, then we have a little negative reward, we are not done, and we update the map.
The little negative reward is useful to indicate that the car should not choose a long path.

This may happens if we are on the border of the map

…

```
elif grid_item == CAR:
    reward = -1
    is_done = False
else:
    raise ValueError(f"Unknown grid item {grid_item}")

return State(grid=new_grid, car_pos=p), reward, is_done
```

…

Return a tuple with the new state, the reward, and a boolean indicating if the episode is over or not

```
        is_done = raise
    else:
        raise ValueError(f"Unknown grid item {grid_item}")
return State(grid=new_grid, car_pos=p), reward, is_done
```

Interacting with the environment

In our case, *an episode starts from the initial state* and ends by either *crashing into a zombie* or *eating the ice cream*

Q-Learning

So far, we have defined the necessary to:

- Build & explore a map
- Create a new state for a given state and action

We still need *to learn from a sequence of actions*

This is exactly what *Q-learning* is about

Q-Learning

Q-learning is one of most popular algorithms for reinforcement learning

The goal is to make an agent learn a *policy*, which tells *an agent what action to take in a particular state*

Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state

Learning to drive with Q-Learning

```
import numpy as np
import random
random.seed(42) # for reproducibility

# How many episodes we are considering? More episodes means more
exploration, and therefore a more efficient model
N_EPISODES = 20

# How many steps, at maximum, per episode? We need to make sure that
this number is large enough for end the episode
MAX_EPISODE_STEPS = 100

MIN_ALPHA = 0.02
alphas = np.linspace(1.0, MIN_ALPHA, N_EPISODES)
gamma = 1.0
eps = 0.2

# Contains the Q-table
q_table = dict()
```

Learning to drive with Q-Learning

We will *decay the learning rate alpha*, at every *episode*. As our agent explores more and more of the environment, he will "believe" that there is not that much left to learn.

Additionally, we put some limits for the number of training episodes and steps.

Learning to drive with Q-Learning

Learning rate is how big you take a *leap in finding optimal policy*

Alpha indicates how much you are *updating the Q value* with each step

Higher alpha means you are updating your Q values in big steps

When the *agent is learning* you should *decay* this to stabilize your model output which eventually converges to an optimal policy

Learning to drive with Q-Learning

We define a function to access the Q-table

```
def q(state, action=None):

    if state not in q_table:
        q_table[state] = np.zeros(len(ACTIONS))

    if action is None:
        return q_table[state]

    return q_table[state][action]
```

dcc
CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Learning to play with Q-learning

Q-table is a matrix [state, action] (4 x 4 in our case)
The table is initialized with zeros
We update and store q-values after each episode
The q-table is a reference for our agent to find the best action to take for each state

table

```
def q(state, action=None):
    if state not in q_table:
        q_table[state] = np.zeros(len(ACTIONS))

    if action is None:
        return q_table[state]

    return q_table[state][action]
```

Learning to drive with Q-Learning

Choosing an action given the current state is simple

Act with random action with some small probability or the best action seen so far (using our Q-table):

```
def choose_action(state):
    if random.uniform(0, 1) < eps:
        return random.choice(ACTIONS)
    else:
        return np.argmax(q(state))
```

Learning to drive with Q-Learning

Choosing an action given the current state is simple

Act with random action with some small probability or the best action seen so far (using our Q-table):

np.argmax returns the index of the maximum value.
e.g., np.argmax([1, -3, 5, 1]) returns 2

```
def
        return random.choice(ACTIONS)
    else:
        return np.argmax(q(state))
```

Learning to drive with Q-Learning

Why our agent uses random actions, sometimes? Remember the environment is unknown, so it has to be explored in some way.

Q-learning algorithm

We then need to train our agent, using the Q-learning algorithm

```
for e in range(N_EPISODES):

    state = start_state
    total_reward = 0
    alpha = alphas[e]

    for _ in range(MAX_EPISODE_STEPS):
        action = choose_action(state)
        next_state, reward, done = act(state, action)
        total_reward += reward

        q(state)[action] = q(state, action) + \
                alpha * (reward + gamma *  np.max(q(next_state)) - q(state, action))
        state = next_state
        if done:
            break
    print(f"Episode {e + 1}: total reward -> {total_reward}")
```

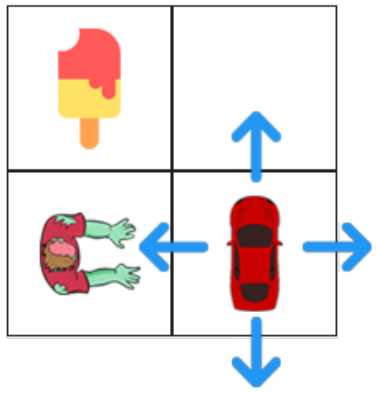
Example of an execution

```
Episode 1: total reward -> 999
Episode 2: total reward -> 998
Episode 3: total reward -> 997
Episode 4: total reward -> 997
Episode 5: total reward -> 999
Episode 6: total reward -> 999
Episode 7: total reward -> 998
Episode 8: total reward -> -100
Episode 9: total reward -> -101
Episode 10: total reward -> 999
Episode 11: total reward -> 999
Episode 12: total reward -> 999
Episode 13: total reward -> 999
Episode 14: total reward -> 999
Episode 15: total reward -> 999
Episode 16: total reward -> 998
Episode 17: total reward -> 999
Episode 18: total reward -> 999
Episode 19: total reward -> 999
Episode 20: total reward -> 999
```

Did our agent learn something?

Let's extract the policy our agent has learned by selecting the action with maximum Q value at each step.

We will do that manually. First, the start_state



Our agent starts here at every new episode

Did our agent learn something?

```
sa = q(start_state)
print(f"up={sa[UP]}, down={sa[DOWN]}, left={sa[LEFT]}, right={sa[RIGHT]}")
```

prints the following

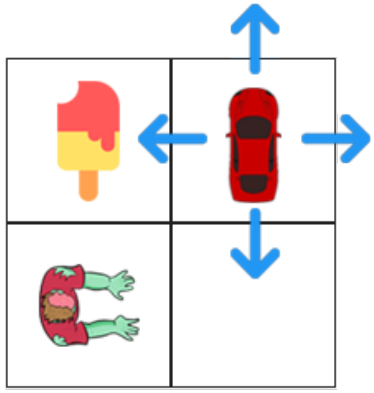
```
up=998.99, down=225.12, left=-85.10, right=586.19
```

UP seems to have the highest Q value.
So, let's take that action:

```
new_state, reward, done = act(start_state, UP)
```

Did our agent learn something?

The new state looks like this:



What is the best thing to do now?

```
sa = q(new_state)
print(f"up={sa[UP]}, down={sa[DOWN]}, left={sa[LEFT]}, right={sa[RIGHT]}")
```

```
up=895.94, down=842.87, left=1000.0, right=967.10
```

Did our agent learn something?

Our agent *does not know anything* about the rules of the game

However, it manages to *avoid zombie* and *get the ice-cream*

Also, it tries to *reach the ice cream* as quickly as possible

The *reward* is the ultimate signal that *drives the learning process*

Conclusion

Overview of Reinforcement Learning

We detailed the *implementation of Q-learning*, a simple and generic algorithm

Today we voluntarily *avoided the formal essence of reinforcement learning*

Tomorrow we will give an overview of it

dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE