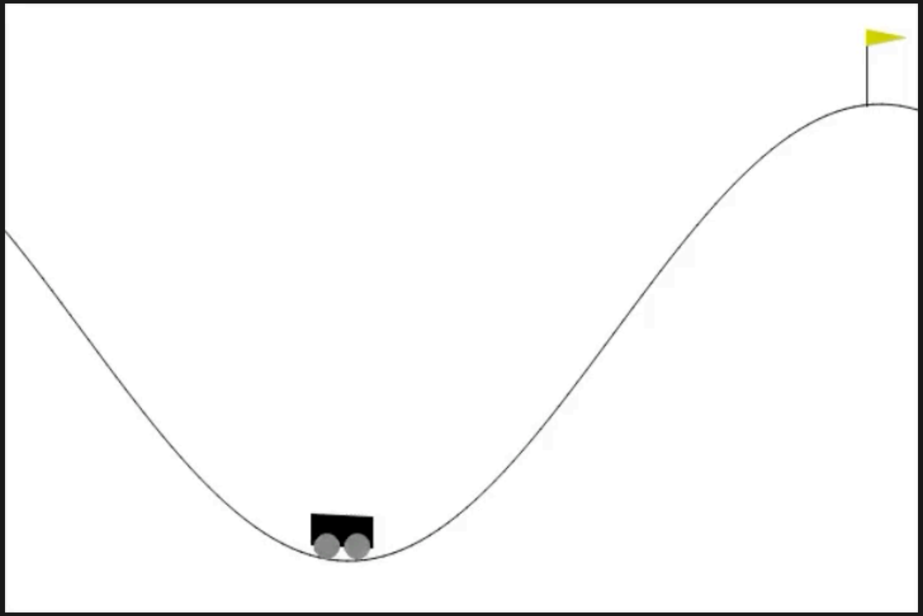
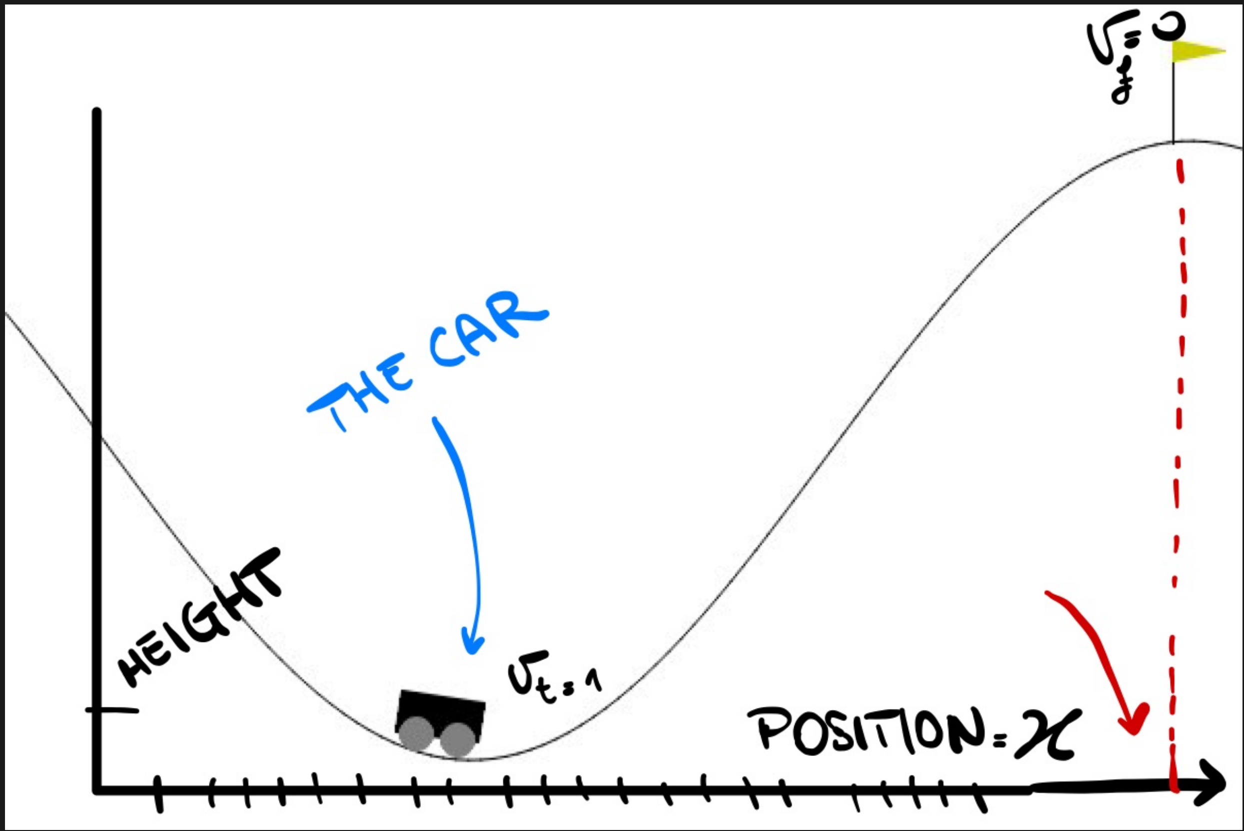
THE MOUNTAIN CAR PROBLEM

Alessio G. & Campagnolo A.



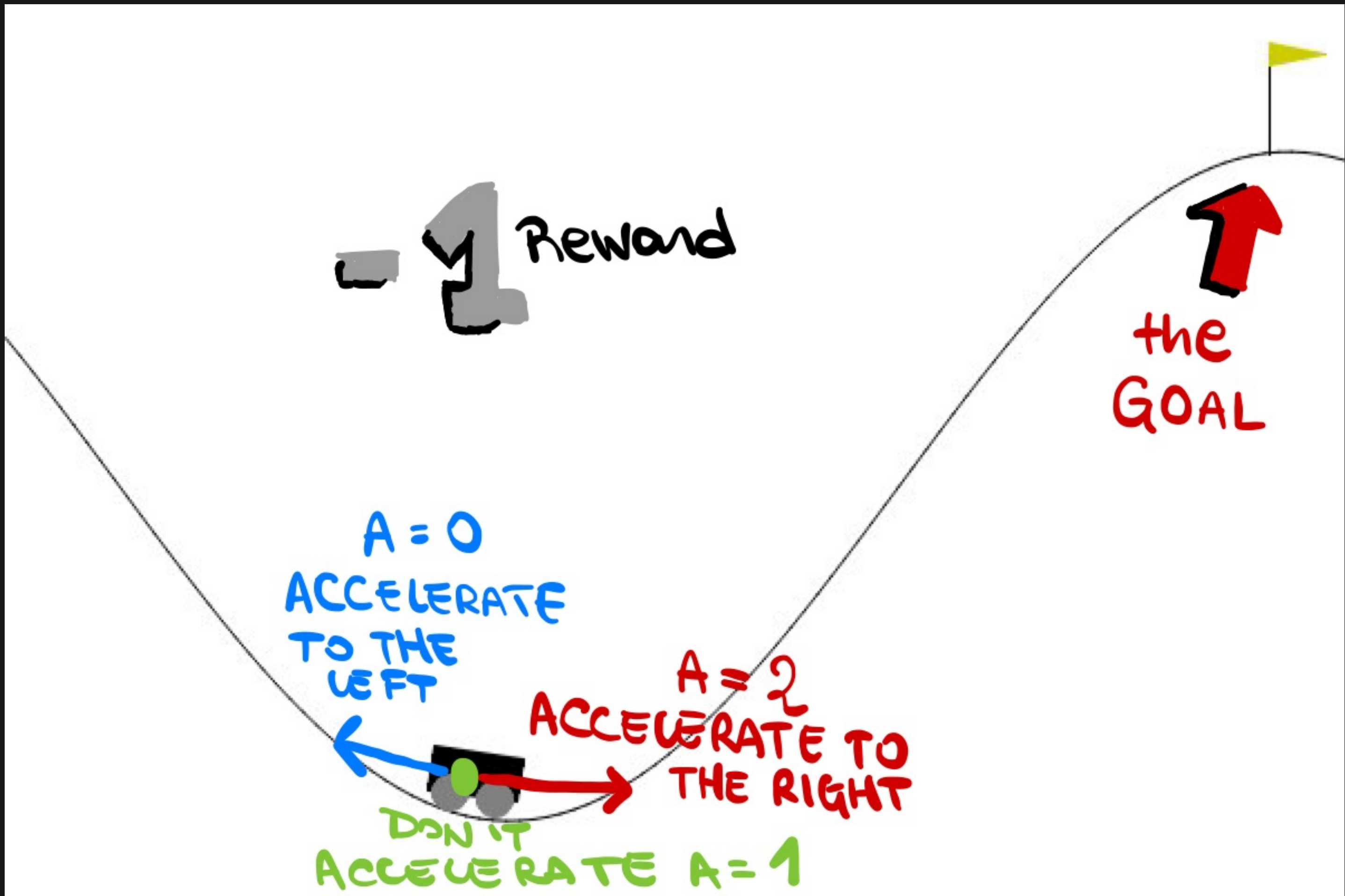
OBSERVATION SPACE



Num	Observation
0	Position of the car along the x-axis
1	Velocity of the car

[redacted]

ACTION SPACE



Num	Observation
0	Accelerate to the left
1	Don't accelerate
2	Accelerate to the right

[redacted]

THE AGENT-ENVIRONMENT INTERFACE

```
env = gym.make("MountainCar-v0")
```

```

class MCE(gym.Env):
    metadata = {
        "render_modes": ["human", "rgb_array"],
        "render_fps": 30,
    }

    def __init__(self, render_mode: Optional[str] = None, goal_velocity=0):
        self.min_position = -1.2
        self.max_position = 0.6
        self.max_speed = 0.07
        self.goal_position = 0.5
        self.goal_velocity = goal_velocity

        self.force = 0.001
```

Property
Action Space
Observation Shape
Observation High
Observation Low

THE EPISODE

The car moves a long the x-axis. The episode ends if either of the following happe

- Termination: The position of the car is greater than or equal to 0.5 (the goal position on t right hill)
- Truncation: The length of the episode is 200.

STEP FUNCTION

Given an action, the mountain car follows the following transition dynamics:

$$\text{Velocity}_{t+1} = \text{Velocity}_t + (\text{action} - 1) \cdot \text{force} - \cos(3 \cdot \text{Position}_t) \cdot \text{g}$$

$$\text{Position}_{t+1} = \text{Position}_t + \text{Velocity}_{t+1}$$

```python
def step(self, action: int):
        assert self.action_space.contains(
            action
        ), f"{action!r} ({type(action)}) invalid"

        position, velocity = self.state
        velocity += (action - 1) * self.force + math.cos(3 * position) * (-self.gravity)
        velocity = np.clip(velocity, -self.max_speed, self.max_speed)
        position += velocity
        position = np.clip(position, self.min_position, self.max_position)
        if position == self.min_position and velocity < 0:
            velocity = 0

        terminated = bool(
            position >= self.goal_position and velocity >= self.goal_velocity
```

[redacted]

STARTING STATE

The position of the car is assigned a uniform random value in [-0.6 , -0.4]. The starting vel
car is always assigned to 0.

```
def reset(self,*, seed: Optional[int] = None, options: Optional[dict] = None):
    super().reset(seed=seed)
    # Note that if you use custom reset bounds, it may lead to out-of-bound
    # state/observations.
    low, high = utils.maybe_parse_reset_bounds(options, -0.6, -0.4)
    self.state = np.array([self.np_random.uniform(low=low, high=high), 0])

    if self.render_mode == "human":
        self.render()
    return np.array(self.state, dtype=np.float32), {}
```

THE SARSA ALGORITHM

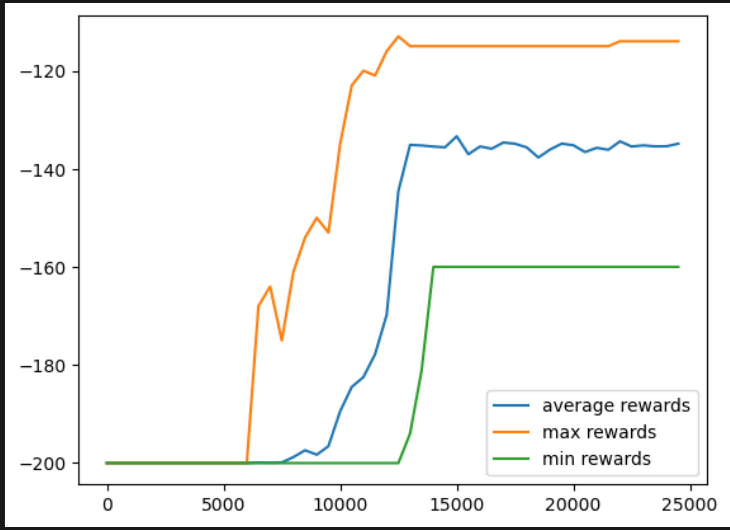
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[r + \gamma Q(s',a')\right]$$

```

for episode in range(EPISODES):
    # initialize
    state,_ = env.reset()
    discrete_state = get_discrete_state(state)
    done = False
    episode_reward = 0

    # epsilon greedy
    action = epsilon_greedy(q_table, discrete_state, epsilon)

    # episode loop
    while not done:
        # new_state and reward
        new_state, reward, done, _, _ = env.step(action)
```

RESULTS SARSA

Parameters used - Average reward per episode - Rendering

```
# parameter setting
LEARNING_RATE = 0.2
DISCOUNT = 0.95
EPISODES = 25000
SHOW_EVERY = 500
FRAMES_EVERY = 5000
epsilon = 1
```



THE Q LEARNING ALGORITHM

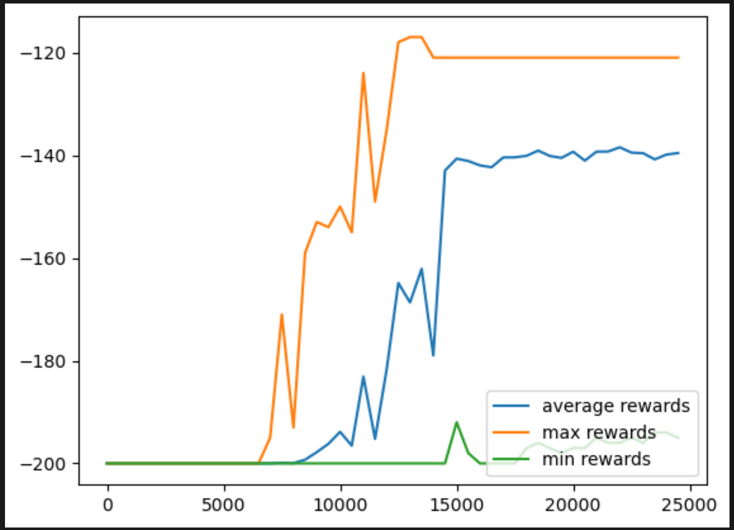
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a')\right]$$

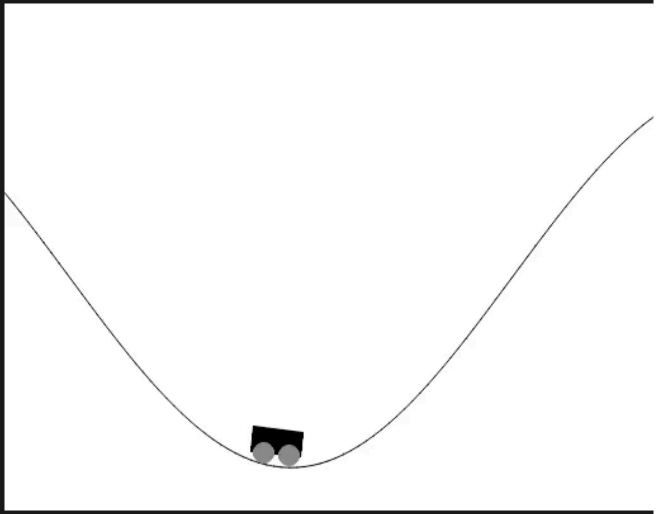
```python

for episode in range(EPISODES):
    state,_ = env.reset()
    discrete_state = get_discrete_state(state)
    done = False
    episode_reward = 0

    if episode % SHOW_EVERY == 0:
        render = True
        print(episode)
    else:
        render = False

    while not done:
        if np.random.random() > epsilon:
```

RESULTS Q LEARNING

Parameters used - Average reward per episode - Rendering

```
# parameter setting
LEARNING_RATE = 0.2
DISCOUNT = 0.95
EPISODES = 25000
SHOW_EVERY = 500
FRAMES_EVERY = 5000
epsilon = 1
```





THE EXPECTED SARSA ALGORITHM

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[r + \gamma \sum_{a} \pi(a|s')Q(s',a) \right]$$

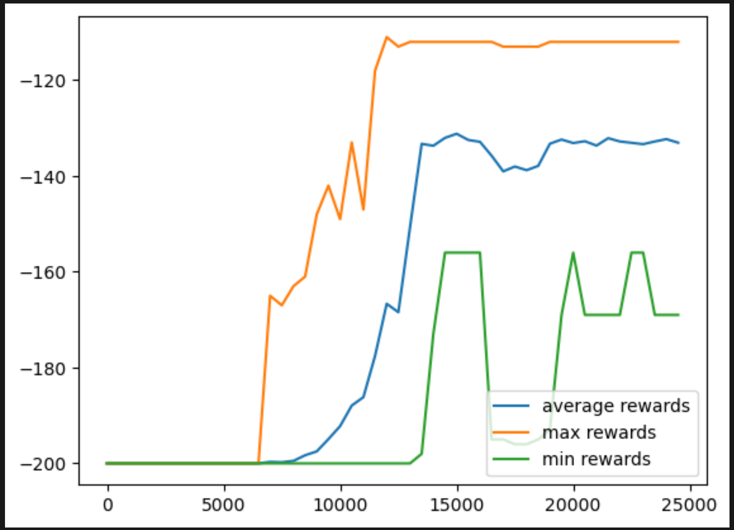
```python

for episode in range(EPISODES):
    # initialize
    state,_ = env.reset()
    discrete_state = get_discrete_state(state)
    done = False
    episode_reward = 0

    # episode loop
    while not done:

        # epsilon greedy
        action = epsilon_greedy(q_table, discrete_state, epsilon)
        # new_state and reward
        new_state, reward, done, _, _ = env.step(action)
```

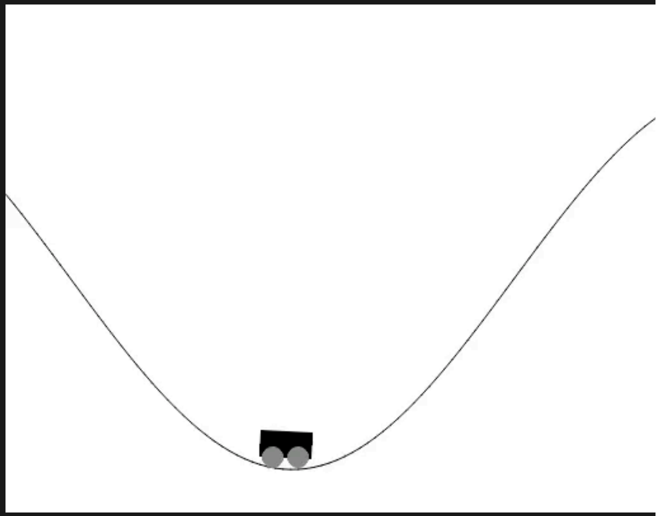
[redacted]

RESULTS EXPECTED SARSA

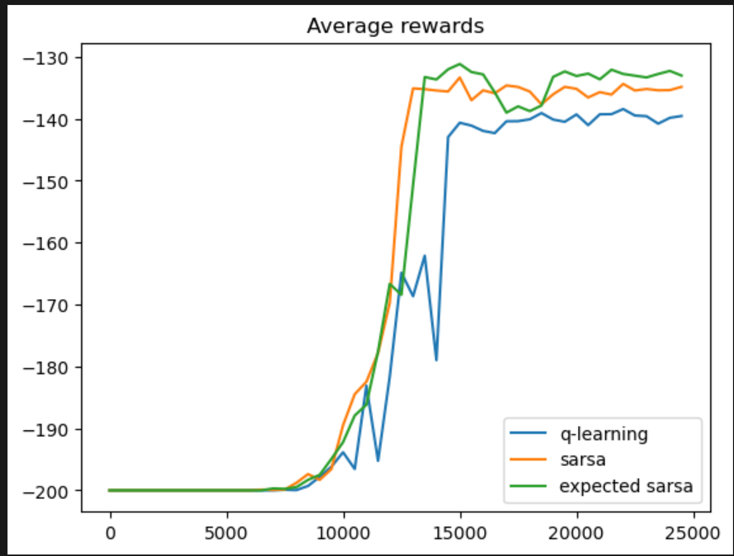
Parameters used - Average reward per episode - Rendering

```
# parameter setting
LEARNING_RATE = 0.2
DISCOUNT = 0.95
EPISODES = 25000
SHOW_EVERY = 500
FRAMES_EVERY = 5000
epsilon = 1
```

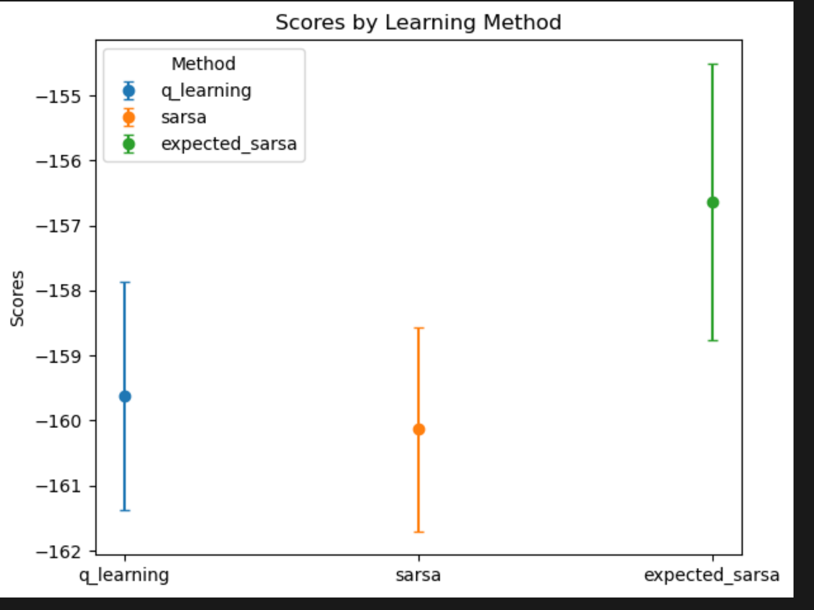




COMPARISON BETWEEN THE ALGORITMS



BOXPLOT


Scores by Learning Method
Method
q_learning
sarsa
expected_sarsa
Scores
−155
−156
−157
−158
−159
−160
−161
−162
q_learning
sarsa
expected_sarsa

IMPROVEMENTS

- Improve reward function
- Maximization bias
- After states methods
- Different trajectory, improve transition dynamics
- Mountain car continuous problem

REFERENCES

- [1] OpenAI Gym
- [2] OpenAI Gym MountainCar-v0
- [3] Richard S. Sutton (2018). "Reinforcement Learning". MIT Press: 119-140

GITHUB

The mountain car discrete problem solved with TD learning

```
import gym
import numpy as np

class RL_Trainer:
    def __init__(self, env_name, learning_rate=0.1, discount=0.95,
        show_every=50, generate_frames=False, frames_every=500,
        epsilon=1, start_epsilon_decaying=1, end_epsilon_decaying=500, discrete_os_size=[20, 20], q_table=
        self.env = gym.make(env_name, render_mode='rgb_array')
        self.learning_rate = learning_rate
        self.discount = discount
        self.show_every = show_every
        self.frames_every = frames_every
        self.epsilon = epsilon
        self.start_epsilon_decaying = start_epsilon_decaying
```

THE END