Università di Bologna

Laurea Magistrale in Informatica

2D America's Cup Match Simulator with Multi-Agent RL

Deep learning models applied to regatta simulations

Gianluca di Giacomo
Alessandro Tomaiuolo
Pietro Sami

Abstract

This project presents a 2D America's Cup match simulator developed as part of the Artificial Intelligence course for the Master's degree in Computer Science at the Università di Bologna, utilizing Multi-Agent Reinforcement Learning (MARL) to solve complex control problems in competitive regatta scenarios. By implementing the Proximal Policy Optimization (PPO) algorithm within a Gymnasium and Petting Zoo framework, autonomous agents are trained to navigate a simplified match race environment.

The core of the model focuses on a reward function designed to balance time efficiency, safety, and adherence to sailing regulations.

To ensure unbiased learning, the environment employs randomized starting positions and shuffled action execution orders. Quantitative results demonstrate significant policy maturity as training progressed from 500,000 to 1,000,000 timesteps, with the success rate increasing from 83.5% to 97.7% and a drastic reduction in collisions and out-of-bounds errors. The final model achieved nearly identical win rates between agents, confirming a robust and fair competitive framework.

Contents

1 Background

While the fields of Reinforcement Learning and Regatta are vast, the following sections will provide a focused overview of the essential concepts utilized in this project without delving into exhaustive theoretical detail.

1.1 Machine Learning

Machine learning is the subset of artificial intelligence focused on algorithms that can "learn" patterns in data and make accurate inferences about new data.

Usually, we categorize machine learning as supervised, unsupervised, and reinforcement learning. In supervised learning, there are labeled data; in unsupervised learning, there are no labeled data; and in reinforcement learning, there are evaluative feedbacks but no supervised signals. Unlike supervised learning, it presents additional challenges like credit assignment, stability, and exploration. [2]

Supervised learning is commonly used for risk assessment, image recognition, predictive analytics, and fraud detection, and comprises several types of algorithms.

The most common unsupervised learning method is cluster analysis, which uses clustering algorithms to categorize data points according to value similarity. Association algorithms allow data scientists to identify associations between data objects inside large databases, facilitating data visualization and dimensionality reduction.

Reinforcement learning algorithms are common in video game development, autonomous driving, robotics, industrial control systems, and Natural Language Processing. [1]

1.2 Deep Learning

Deep learning, or deep neural networks, is a particular machine learning approach, usually for supervised or unsupervised learning, and can be integrated with reinforcement learning for state representation and/or function approximation. Deep learning is in contrast to "shallow" learning. For many machine learning algorithms, e.g., linear regression, logistic regression, and decision trees, we have an input layer and an output layer, and the inputs may be transformed with manual feature engineering before training. In deep learning, between the input and output layers, we have one or more hidden layers. At each layer except the input layer, we compute the input to each unit as the weighted sum of units from the previous layer; then we usually use a nonlinear transformation, or activation function, applied to the input of a unit to obtain a new representation of the input from the previous layer. We have weights on links between units from layer to layer. After computations flow forward from input to output, at the output layer and each hidden layer, we can compute error derivatives backward and backpropagate gradients towards the input layer, so that weights can be updated to optimize some loss function. [2]

1.3 Reinforcement Learning

Reinforcement Learning (RL) is a general class of algorithms in the field of machine learning that aims at allowing an agent to learn how to behave in an environment, where the only feedback consists of a scalar reward signal. RL should not be seen as characterized by a particular class of learning methods, but rather as a learning problem or a paradigm. The goal of the agent is to perform actions that maximize the reward signal in the long run.

To explain the RL process, let's start by imagining a simple scenario of a Regatta, where our agent (a boat) wants to get to the target as fast as possible.

The loop outputs a sequence of state, action, reward, and next state:

- Our agent receives state S_0 from the Environment, first position of the boat.
- Based on that state S_0, the agent takes action A_0 (e.g., the boat will decide to rotate 15 degrees right).
- The environment goes to a new state S_1, new position.
- The environment gives some reward R_1 to the agent; e.g., the boat is now heading towards the target (+1).

The agent's goal is to maximize its cumulative reward, called the expected return.

1.3.1 Markov Decision Process

A Markov process is a way of modeling something that changes over time, step by step, where the future depends only on the present, not on the full history of the past.

The RL process is a Markov Decision Process (MDP), a mathematical model of how a decision-making problem works over time. It describes:

- What situations the agent can be in
- What actions it can take
- How the world changes when the agent acts
- What rewards the agent gets for its actions
- How much the agent cares about the future

1.3.2 Value function and Bellman Equation

Almost all reinforcement learning algorithms involve estimating value functions, which serve as a metric for the expected return, the total amount of reward an agent can expect to accumulate over the long run. Accordingly, value functions are defined with respect to particular ways of acting, called policies Formally, a policy is a mapping from states to probabilities of selecting each possible action. [3]

Value functions are generally categorized into two types: the State-Value Function ($V^{\pi}(s)$), which estimates the expected cumulative reward starting from a state s while following a specific policy π, and the Action-Value Function ($Q^{\pi}(s, a)$), which evaluates the expected return of taking a specific action a in state s.

One fundamental property of value functions is that they satisfy certain recursive properties. For any policy π and any state s value functions can recursively be defined in terms of a so-called Bellman Equation. It denotes that the expected value of state is defined in terms of the immediate reward and values of possible next states weighted by their transition probabilities, and additionally a discount factor γ, which balances the trade-off between immediate gratification and future gains. [4]

The goal for any given MDP is to find a best policy, i.e. the policy that receives the most reward.

1.3.3 Proximal Policy Optimization

Proximal Policy Optimization, or PPO belongs to the family of policy gradient methods, which directly optimize the agent's policy to maximize cumulative rewards. Unlike standard policy gradient methods that can be unstable due to large updates, PPO introduces a "clipped" objective function. This mechanism limits the change a single update can make to the policy, ensuring that the new policy does not deviate too far from the previous one.

1.4 Regatta

A regatta is a competition between boats that involves following a pre-established route defined by a tender. The term comes from the Venetian language, with regatta meaning "contest, contention for mastery". Boats could be rowed, usually with eight rowers per boat, or sailed.

A regatta that involves two or more boats is called a match race.

2 Model

2.1 Environment

The environment is a semplified model of the conditions of a real match race and was built with the following characteristics:

- A **field** of standard size of 400x400 pixels, which is a 2D plane with x and y axes.
- A **boat** that is the agent involved in the training. It is represented by a small triangle, which has a velocity and a heading; both depend on the wind.
- A **target**, which is the objective to reach, represented by a circle with a radius of 20 pixels.

An **episode** consist of a simulation with two boats ($boat_0$ and $boat_1$) competing to reach the target within 250 steps.

In an episode, the only force acting on the agents is the wind, which starts to blow from the North and has small variations in its angle during the race.

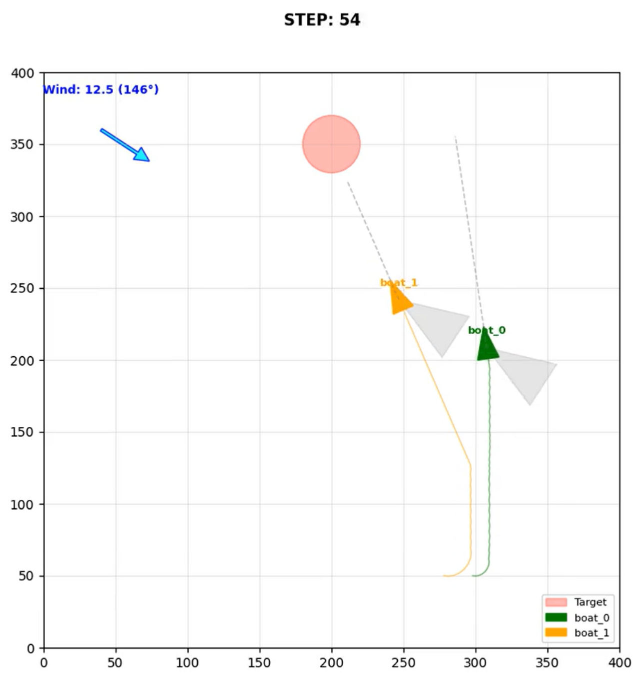

Figure 1: Example of a simulation

The system uses a polar diagram function that takes as input the angle (between the heading of the boat and the wind) and the speed of the wind; then it calculates the speed of the boat. In addition to the speed information, each boat needs a description of the environment, which includes: the distance from the target, the distance from an opponent, and the angle relative to the target.

The environment was developed using the Gymnasium framework and extended for multi-agent support through the PettingZoo library, with SuperSuit utilized to apply essential wrappers for environment preprocessing and compatibility. The training and optimization of the agents are conducted using the Proximal Policy Optimization (PPO) algorithm from Stable Baselines3. PPO serves as the "actor-critic" framework where the actor decides the boat's actions and the critic evaluates the expected return of those decisions.

2.2 Rules

The environment establish the following rules:

- If a boat touches an edge of the field, the episode terminates for the one that touched the edge.
- After 250 steps, the episode is terminated.
- If two boats collide, the episode is terminated.
- If both boats reach the target, the boat that reached it first is the winner and the episode is terminated.

3 Objective

The main objective is to create an environment where an agent learns to reach the target as quickly as possible without colliding with another boat or exiting the field. We design a reward function that encodes the trade-off between time efficiency and safe navigation, with components that create competitiveness. By providing continuous feedback on the agent's progress and imposing strict penalties for failure states, the function guides the learning process toward a policy that maximizes speed and maintaining a safe distance from obstacles (boundaries and other boats).

4 Reward Function

4.1 VMG

It is important to distinguish between boat speed and **VMG**. A boat might be moving at 6 knots through the water, but if it is sailing at a wide angle from the target, its VMG may only be 3.5 knots—or less. It is calculated using the formula:

$$\text{VMG} = \text{Boat Speed} \times \cos(\theta)$$

where θ is the angle between the boat's heading and the wind.

It is important to train the agent to maximize the VMG instead of the classic speed, so at every step the reward is increased by the VMG multiplied by a factor of 0.7.

```
to_target_angle = np.arctan2(self.target[1] - state['y'], self.target[0] -
↪   state['x'])
angle_error = to_target_angle - state['heading']
vmg = state['speed'] * np.cos(angle_error)
rewards[agent] = rewards[agent] + (vmg * 0.7)
```

In order to generate an element of competitiveness, the environment incorporates a **wind shadow mechanism**. This effect occurs when a leading boat obstructs the airflow for trailing opponents, significantly reducing their speed.

The implementation identifies this phenomenon by calculating the relative position of agents compared to the current wind direction. Using a dot product projection, the system determines if an agent is positioned directly downwind of an opponent. If the agent falls within a 20-degree angular cone and remains within a proximity of the boat, it is considered to be inside the shadow box. Under these conditions, the effective wind speed for the affected boat is reduced by 40%, simulating the loss of power.

```
for opp in opponents:
    opp_state = self.boat_states[opp]

    dx = state['x'] - opp_state['x']
    dy = state['y'] - opp_state['y']
    dist = np.hypot(dx,dy)
    wind_vec = -np.array([np.cos(self.wind_direction),
     ↪  np.sin(self.wind_direction)])
    pos_vec = np.array([dx, dy])

    proj = np.dot(wind_vec, pos_vec)

    if proj > 0 and dist < (self.boat_radius * 10):
        cos_angle = np.clip(proj / (dist + 1e-6), -1.0, 1.0)
        angle_diff = np.arccos(cos_angle)
        if angle_diff < np.radians(20):
            current_wind_speed = current_wind_speed * 0.6
```

4.2 Time Penalty

To encourage the agents to reach the target as soon as possible, we added a **time penalty** that decreases the reward function by 0.05 at every step.

```
rewards[agent] -= 0.05
```

4.3 Deadzone penalty

In real sailing, a boat cannot sail directly into the wind; there is a narrow cone of angles around the wind direction where the sails cannot generate forward propulsion. This segment is called the no-go zone or **Deadzone**. We calculate the apparent wind angle by subtracting the boat's heading from the wind direction. If the resulting angle is less than 20, we subtract 0.5 from the reward function at every step.

The penalty is designed to be non-prohibitive. This prevents the agent from becoming 'trapped' in local optima or being discouraged from performing essential turns, while still providing a clear incentive to exit the Deadzone in an efficient manner.

```
apparent_wind = self.wind_direction - state['heading']
wind_angle_rel = np.abs(np.degrees(apparent_wind)) % 360
if wind_angle_rel > 180: wind_angle_rel = 360 - wind_angle_rel
if wind_angle_rel < 20:
rewards[agent] -= 0.5
```

4.4 Overtaking

To simplify the simulation environment, the model implements a restricted subset of competitive sailing regulations. Specifically, right-of-way is determined by each boat's proximity to the target along the x-axis. The agent maintaining the lead position relative to this coordinate is granted priority, allowing it to maintain its course, while other boats are prohibited from initiating overtaking maneuvers.

To detect potential overtaking violations, the environment implements a predictive trajectory model. For each agent, a linear projection is computed based on its current heading and velocity vector. By calculating the determinant of the combined velocity vectors, the system identifies whether the trajectories intersect.

The potential collision is determined by the parameters t and u, representing the number of discrete time steps until intersection for the agent and the opponent, respectively. If an intersection is predicted within a threshold of 10 steps, the environment evaluates the agent's priority status. Agents attempting an illegal maneuver while outside the prioritized zone are penalized to enforce adherence to regatta-specific overtaking constraints.

During the testing phase, we observed that the boats were following the overtaking rules so strictly that they would occasionally dodge the target to avoid a penalty, even when they were in a winning position. This resulted in agents losing their advantage because they prioritized rule-following over reaching the final objective. To resolve this, we implemented a boolean variable to determine if a boat is currently near the target, defined as being within a distance equal to 3 times the target radius. By applying the penalty only when this variable is false, we allow the agents to focus entirely on reaching the target during the final approach.

```
for opp in opponents:
    displacement_opp = self.boat_states[opp]['speed'] * 0.514 * self.dt
    dx_opp = np.cos(self.boat_states[opp]['heading']) * displacement_opp
    dy_opp = np.sin(self.boat_states[opp]['heading']) * displacement_opp
    det = dx_me * dy_opp - dx_opp * dy_me

    x_opp = self.boat_states[opp]['x']
    y_opp = self.boat_states[opp]['y']

    dx_pos = x_opp - state['x']
    dy_pos = y_opp - state['y']

    opp_pos = np.array([x_opp,y_opp])
    opp_dist_to_target = np.linalg.norm(opp_pos - self.target)

    if det != 0:

    t = (dx_pos * dy_opp - dy_pos * dx_opp ) / det
    u = (dx_pos * dy_me - dy_pos * dx_me ) / det
    if 0 <= t <= 10 and 0 <= u <= 10:

    if not state['is_inside_curr'] and not state['is_near_target']:
        rewards[agent] -= 3
    else:
        rewards[agent] -= 0.05
```

4.5 Collisions

As previously stated, any contact between the boats results in the immediate termination of the episode. To deter collisions, a penalty of -200 is assigned to the agents upon such an event. Furthermore, an attention radius—defined as four times the boat radius—is implemented; within this range, agents incur a penalty if the proximity threshold is breached, discouraging dangerously close maneuvers.

```
collision = False
attention_radius = self.boat_radius * 4
agent_lists = [a for a in self.possible_agents]
if len(agent_lists) >= 2:
    boat_0 = agent_lists[0]
    boat_1 = agent_lists[1]
    pos_0 = np.array([self.boat_states[boat_0]['x'],
    ↪   self.boat_states[boat_0]['y']])
    pos_1 = np.array([self.boat_states[boat_1]['x'],
    ↪   self.boat_states[boat_1]['y']])

    distance_between_boats = np.linalg.norm (pos_1 - pos_0)
    if distance_between_boats < (self.boat_radius * 2):
        collision = True

    elif distance_between_boats < attention_radius:
        proximity_penalty = (attention_radius - distance_between_boats)
        / attention_radius * 5.0
        rewards[boat_0] -= proximity_penalty
        rewards[boat_1] -= proximity_penalty

# Penalty for collision
if collision:
    rewards[agent] -= 200.0
    state['finished'] = True
    terminations[agent] = True
```

4.6 Final states

Episode termination and reward distribution are determined by the agents' positions relative to the target and the elapsed time steps.

The first boat to enter the target radius is awarded 100 points and declared the winner. To prevent a single agent from dominating early in the training and obstructing the progress of others, the episode continues after the first arrival. Remaining boats receive a scaled penalty based on their distance to the goal, creating competition but allowing them to still earn a 50-point reward upon finishing. This approach balances the training process, ensuring that all agents have the opportunity to learn the optimal route even if they do not reach the target first.

```
# Target Reached
if dist_to_target < self.target_radius:
    if self.winner == None:
        rewards[agent] += 100.0
        state['finished'] = True
        terminations[agent] = True
        self.winner = agent
    elif agent != self.winner:
        rewards[agent] += 50.0
        state['finished'] = True
        terminations[agent] = True

        for opp in self.agents:
            if opp != agent:
            opp_pos = np.array([self.boat_states[opp]['x'],
             ↪  self.boat_states[opp]['y']])
            opp_dist_to_target = np.linalg.norm(opp_pos - self.target)

            max_penalty = 30.0
            min_penalty = 5.0
            distance_ratio = np.clip(opp_dist_to_target /
            (self.target_radius * 10), 0.0, 1.0)
            scaled_penalty = min_penalty + (max_penalty - min_penalty) *
             ↪  distance_ratio
            rewards[opp] -= scaled_penalty
```

If an agent exits the predefined field boundaries, it incurs a substantial penalty of -200. This penalty is designed to strictly discourage out-of-bounds movement and instead incentivize the agent to explore the navigable environment within the designated area.

```
# Out of bounds
    if not (0 <= state['x'] <= self.field_size and 0 <= state['y'] <=
     ↪  self.field_size):
        rewards[agent] -= 200.0
        state['finished'] = True
        terminations[agent] = True
```

4.7 Personal Best Reward

We introduced a reward mechanism based on personal improvement for agents that do not win the race. Even if a boat is not the first to reach the target, it can still receive positive feedback if it manages to reduce its minimum distance to the objective compared to its personal best. By rewarding this incremental progress, we ensure that all agents remain motivated to explore better routes and improve their strategies, preventing them from becoming discouraged simply because they were outpaced by a faster competitor. This approach is also particularly effective during the early stages of training, when reaching the target is a rare occurrence for agents that are still exploring the environment.

```
# Reward to boats that didn't win if they beat their personal best distance
↪  (closer to target)
if state['finished'] and agent != self.winner and dist_to_target <
↪  self.global_best_distance[agent]:
    rewards[agent] += (self.global_best_distance[agent] - dist_to_target) *
     ↪  0.5
    self.global_best_distance[agent] = dist_to_target

    self.best_distances[agent] = min(self.best_distances[agent],
     ↪  dist_to_target)
```

4.8 Secure training balance

To ensure unbiased learning in the multi-agent environment, the training process must prevent any single agent from gaining a structural advantage. We address this by introducing randomization into the initial state and the execution order of each episode.

First, starting positions are randomized to eliminate spatial bias. While the boats begin the race adjacent to one another, we shuffle their lateral offsets so that neither agent is consistently favored by a specific starting position. Additionally, we randomize the sequence in which agents perform their actions during each step. If one agent were to always move first, the second agent could develop a reactive advantage by observing the first agent's state change. By shuffling the action order, we maintain fairness and ensure that the learned policies are robust.

```
agents_order = self.agents[:]
self.np_random.shuffle(agents_order)

for agent in agents_order:
        ...

spawn_offsets = [0, 20]
self.np_random.shuffle(spawn_offsets)
for i, agent in enumerate(self.agents):
    offset = spawn_offsets[i]
    self.boat_states[agent] = {
                'x': start_x + offset,
                'y': ...,
```

5 Results

Now we present some results obtained using the reward function described in the previous section. A test consists of a series of episodes. For each test, we used the following statistics to evaluate a model:

- **Success Rate**: the percentage of simulations in which victory was achieved divided by the total number of simulations.
- **Wins boat0**: the number of victories for $boat_0$.
- **Wins boat1**: the number of victories for $boat_1$.
- **Collisions**: the number of collisions between the boats.
- **Out of Bounds**: the number of out-of-bounds occurrences (if the boats have exited the field).
- Timeouts: the number of timeouts (if the boats haven't reached the target within 250 steps).

For each boat:

- **Avg VMG**: the average of the VMG during all the simulations.
- **Avg Triple Turns**: the average number of times the boat chooses the same action (apart from the straight one) three times in a row during all the simulations.
- **Start IN**: how many times the boat started inside (the starting point is near the y-coordinates of the target).
- **Start OUT**: how many times the boat started outside (the other boat has the starting point near the y-coordinates of the target).
- **Win (IN)**: number of victories when the start is inside.
- **Win (OUT)**: number of victories when the start is outside.

We trained the model using 500,000 and 1,000,000 timesteps (a timestep is a single interaction between the reinforcement learning agent and the environment). Then we tested the model over 1000 episodes.

5.1 Results with 500,000 timesteps

Using 500,000 timesteps for training, the results are the following:

Table 1: General Results of Validation Test

Metric	Value
Episodes	1000
Success Rate	83.5%
Wins (boat_0)	419 (41.9%)
Wins (boat_1)	416 (41.6%)
Collisions	116
Out of Bounds	48
Timeouts	1

Table 2: Details of Agents' Performance

Agent	Avg VMG	Turns	Start In	Win (In)	Start Out	Win (Out)
boat_0	5.80	2.93	506	201 (40%)	494	218 (44%)
boat_1	5.79	3.11	494	196 (40%)	506	220 (43%)

5.2 Results with 1,000,000 timesteps

Using 1,000,000 timesteps for training, the results are the following:

Table 3: General Results of Validation Test

Metric	Value
Episodes	1000
Success Rate	97.7%
Wins (boat_0)	486 (48.6%)
Wins (boat_1)	491 (49.1%)
Collisions	18
Out of Bounds	4
Timeouts	1

Table 4: Details of Agents' Performance

Agent	Avg VMG	Turns	Start In	Win (In)	Start Out	Win (Out)
boat_0	6.21	2.56	508	251 (49%)	492	235 (48%)
boat_1	6.21	2.5	492	248 (50%)	508	243 (48%)

6 Conclusion

In this project, we successfully developed a 2D sailing simulator driven by Multi-Agent Reinforcement Learning to solve the complex control problems inherent in competitive regattas. By leveraging the Proximal Policy Optimization (PPO) algorithm, we trained autonomous agents to master navigation strategies that balance speed, safety, and adherence to sailing regulations.

The core of our approach lay in the design of a reward function that translated sailing physics into scalar feedback. The integration of Velocity Made Good (VMG) rather than simple speed proved essential for guiding agents toward the target against the wind. Furthermore, the inclusion of environmental penalties—specifically for the "Deadzone," wind shadows, and overtaking violations—forced the agents to learn realistic behaviors. A critical refinement was the introduction of the "Personal Best" reward mechanism, which ensured that trailing agents continued to optimize their paths rather than succumbing to sparse rewards, and the adjustment of overtaking rules near the target, which resolved the issue of agents prioritizing rule-adherence over victory.

The quantitative results demonstrate a clear correlation between training duration and policy maturity. In the transition from 500,000 to 1,000,000 timesteps, the model achieved significant performance gains:

- **Reliability:** The Success Rate increased from 83.5% to 97.7%, indicating that the agents learned to navigate the environment consistently crashing.
- **Safety:** The most notable improvement was the drastic reduction in collisions, dropping from 116 events to just 18. Similarly, out-of-bounds errors fell from 48 to 4, proving that the agents successfully internalized the spatial constraints of the field.
- **Efficiency:** The Average VMG improved from 5.80 to 6.21, suggesting that the agents optimized their tacking angles to maximize upwind velocity.

Finally, the randomization of starting positions and action execution orders successfully mitigated structural bias. The win rates between $boat\_0$ (48.6%) and $boat\_1$ (49.1%) in the final model are nearly identical, confirming that the environment facilitates fair competition where victory is determined by policy quality rather than initial conditions.

Currently, the agents operate under a simplified rule set; however, the model could be expanded to include more intricate right-of-way protocols and sophisticated tactical maneuvers common in professional racing. By progressively incorporating these layers, the reinforcement learning framework could evolve into a powerful tool for analyzing emergent behaviors and testing complex strategies in highly regulated, real-world competitive scenarios.

References

[1] IBM. Types of machine learning.

[2] Yuxi Li. Deep reinforcement learning, 2018.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[4] Marco Wiering and Martijn {Van Otterlo}. *Reinforcement Learning: State of the Art*. Springer, 2012.