UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Advance Topics in Computational Intelligence

Learning to Sail using RL

Written by:
Shivani Patel

18 May 2022

Table of Contents

1. Overview

The objective of this project is to implement a sailboat agent which learns to sail towards a given target, in the prevalent conditions of wind and water, using reinforcement learning. The world is modeled as a custom OpenAI Gym environment[1]. Constructing the world is mainly composed of the following tasks in detail - modeling the physics of sailing for the state transition and experimenting with the reward function in order to arrive at the optimal policy. On the other hand, implementation of the RL algorithm is not in the scope of this project, and hence existing libraries are used[2] for the same. This report covers the custom implementation of the environment, basic theoretical background regarding sailing, sailboat modeling in our virtual world, design of the reward function and the impact & comparison of performance of various RL algorithms from the *stable_baselines* library & final learnt policy.

2. Pre-requisite sailing knowledge

2.1 Parts of a boat

Before diving into the details of how sailboats work, this section would cover the most basic terms from the sailing vocabulary which are required and sufficient to understand this project.

1. **Bow** - front of the boat.
2. **Stern** - back of the boat.
3. **Keel** - a very heavy metal section below the boat for stability. Usually used in boats larger than 20ft (“keel boats”), this prevents the boat from capsizing even when it tilts significantly during sailing.
4. **Rudder** - the piece attached at the stern which could be moved via a wheel (“helm”) or a handle (“tiller”) to move the boat. This is what our boat agent would conrol to change the heading. Also attached to a rudder is the propeller which moves the boat forward under the engine.
5. **Hull** - the portion of the boat that rides both in and on top of the water. This is what is responsible for the water drag.
6. **Mainsail** - literally, the main sail.

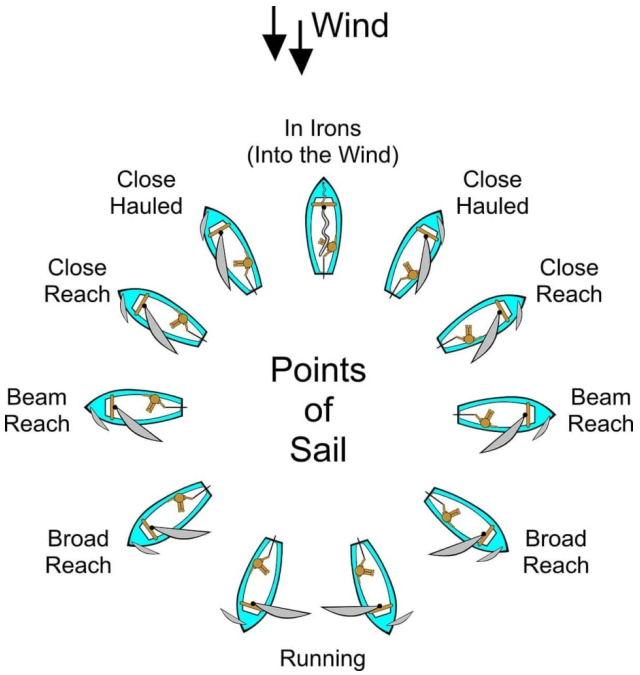
7. **Mast** - the tall rod-like structure in the centre which holds the mainsail.
8. **Boom** - what provides support to the mainsail from below; attached to the mast.
9. **Jib/Genoa** - the top sail at the bow. If the topsail covers the front triangular section completely, it is a jib. If it overlaps the mainsail too (or a ">100% coverage"), it's called a genoa.
10. **Lines** - the ropes on a sailboat, each with a specific function (whether to hoist or reef the sails).

There are, obviously, many more parts which are out of scope and unrequired for this project. Additionally, also important are the below terms:

1. **Port** - left side of the bow. Easy to remember as *left* and *port* both have 4 characters.
2. **Starboard** - right side of the bow.
3. **Windward** - the side where the wind is coming from; facing the wind.
4. **Leeward** - the side which is hidden/shielded from the wind.

2.2 Points of sail

In order to move forward under sails and prevalent environmental conditions, one needs to "trim" or adjust the sails accordingly. Depending on the direction of the wind relative to the bow of the boat, the positions or "points of sail" are the following:



1. **No-go zone/dead zone** - about 90° wide - 45° on either side of the direction of wind. A boat cannot sail in this direction as the sails would flap or "luff" due to the turbulence of the wind on the surface of the sail.
2. **Close haul** - moving slightly away from the headwind (with the wind still coming from the bow) , this is the closest course to the wind that one can sail.
3. **Close reach** - further away from the wind, between 45-90° to the wind coming from the front. The 2 sails are still "close" and thus the wind combined with the curvature of the sail acts as a "pulling" force to make the boat move forward.
4. **Beam reach** - when the wind is abeam (i.e. coming from the sides). This is the position where the boat can sail the fastest due to apparent wind.

5. **Broad reach** - The wind direction is now astern (or coming from behind) at an angle. This is the first point of sail where the propulsion of boat is due to the "push" of the wind.
6. **Running** - directly downwind when the wind is coming from behind. The sails are moved out all the way on either way in order to catch maximum wind and push the boat forward. As is might not be intuitive, this is NOT the fastest direction of movement - as the boat moves in the same direction as the wind, the "apparent wind" felt by the boat is less than the true wind, and hence the pushing force of the wind becomes less and less, the faster the boat moves.

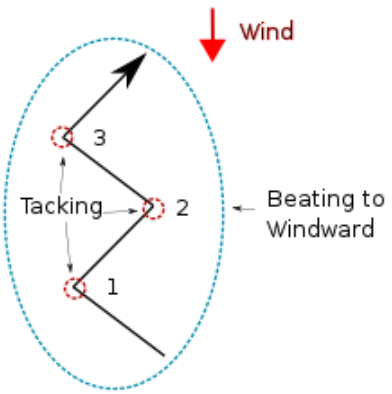
2.3 Sailing maneuvers

The 2 most common sailing maneuvers are tacking and jibing. The course to target would seldom be a straight line, and so would the consistency of the wind. Hence, the boat would be required to be turned at some point in time (or sail!).

- ***Tacking*** is when the bow is moved across the wind direction. Usually undertaken when the current point of sail is close haul (i.e. when the boat is sailing well within 90° to the wind coming from the front).
  Since this move - at some point - involves moving through the no-go zone, it could sometimes happen that the boat might get stuck in the same, especially if the maneuver wasn't carried out fast enough. This is known as *"getting in irons"*. The boat velocity will fall almost to zero - and hence, this is sometimes done deliberately during MoB (man overboard) procedures to stop the boat in emergencies.

- ***Jibing*** is when the stern of the boat is moved across the wind direction. It's carried out slowly, and could be dangerous as the boom (the part of the boat supporting the mainsail from below) would move suddenly from one side to the other. If any personnel is standing in the way, the impact of the move could potentially be fatal. Also, it is carried out when the current point of sail is broad reach (i.e. wind at less than 90° from behind the boat).

With respect to the current project, we would like the sailboat agent to learn tacking on its own - when the target is directly upwind, and we provide an appropriate dead zone, the boat would have to move zig-zag (and hence doing a series of tack) till it reaches the said target. This, in sailing terms, is called *"beating"*:



2.4 Effects of wind[3]

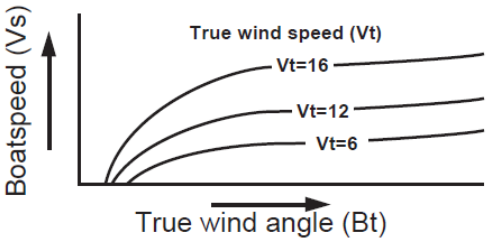
Since sailboats are powered by wind, modeling the effects of wind as closely as possible to the real world would render our agent to behave quite accurately. Of course, alongwith the direction and speed of the wind, the sails need to be trimmed correctly as well - in order to gain maximum boat speed in the prevailing wind conditions, but for the sake of simplicity we'd assume this is always the case.
As mentioned before, the region of 90° (45° each side) of the wind is considered the *dead zone* - in which direction a boat cannot move under sails. Hence, there are 3 factors at play here:

1. **Wind direction θ_w** - measured in radians, headwind from N is considered as 0 radians.
2. **Wind velocity V_w** - speed in nautical miles per hour (kts).
3. **Dead zone θ_{dead}** - measured in radians, this is the area spread equally on both sides of the wind direction, to which the sailboat cannot move under sails. It is usually fixed in real life, but to see its impact on the learnt route, it is made configurable in the project.

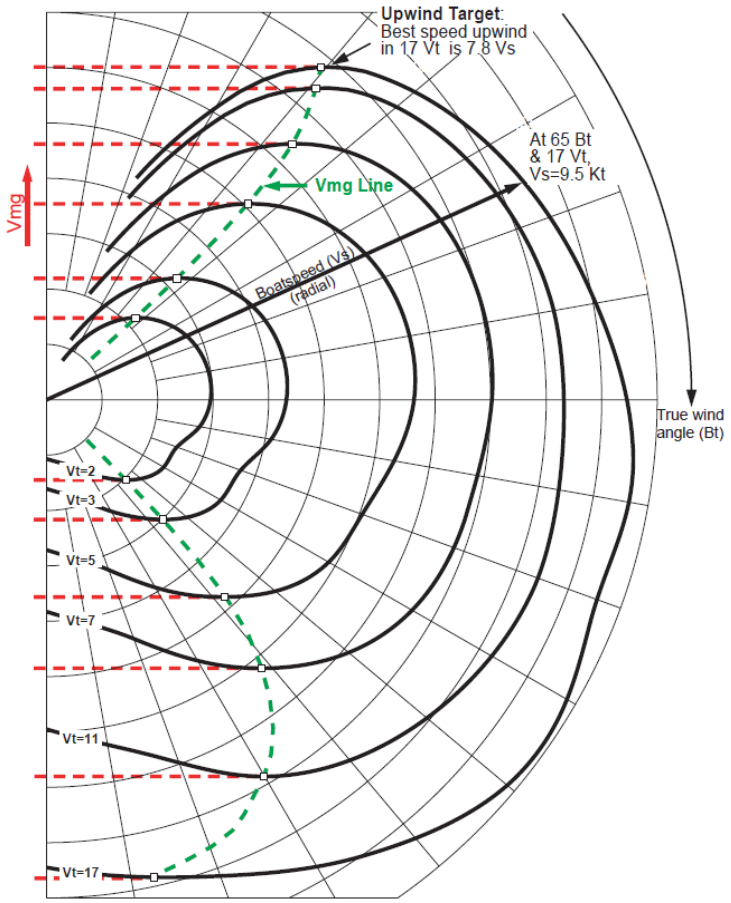
The boat velocity is hence expressed as:

$$V(\theta) = V_w \left[1 - \exp(-(\theta - \theta_w)^2 / \theta_{dead})\right]$$

Where θ is the current heading of the boat in radians with respect to N; clockwise direction to be positive.
However, this is not the most accurate modeling of wind on the boat's velocity. In real life, it is given by *Polars* for the boat[4]. A 'polar' is a database relating a boat's speed to the true wind conditions it is sailing in. If graphed in normal fashion (with a line for each true wind speed), it looks like this:



When graphed in polar coordinates (where boatspeed is the radial distance from the origin and true wind angle is measured from the top), the format provides a graphical solution for V_{mg} ('Velocity' Made Good [to weather or leeward]). Boatspeed and angle for optimum sailing into (or away from) the wind can be determined by inspection. The term 'polar' usually refers to this type of graph:



Ockam polars include additional data that is used to calculate optimal sailing on any point of sail. However, using such complex models in this project is out of scope for now.

3. Implementation details

3.1 OpenAI Gym

The sailing environment has been implemented as a custom one with OpenAI Gym. The reason this is better than creating everything from scratch is that by only implementing the required methods with the necessary contracts, the complex details of the RL algorithm, training etc. is taken care of by using models from existing libraries like Stable Baselines[2].
Following the documentation of OpenAI Gym[5], the following methods are implemented:

1. **__init()__**
   Defining the observation and action spaces, as well as the initialising the variables with the default values.
   The action space is defined as a *gym.spaces* object - a discrete space with 2 possible values 0 and 1. 0 is when the boat turns port-side 0.1 radians, and a 1 is the same on starboard side.
   The observation space is also a *gym.spaces* continuous object - a tuple of size 3 - xy-coordinates, and the heading of the boat θ in radians.

2. **step(action)**

This method defines the transition of one state to another given the action parameter. It does the necessary updates to states and calculates the immediate reward the agent gets in arriving at the next state taking the *action*. Moreover, it also returns a boolean whether the trial is done (successfully or otherwise).
The action can be one of the 2 possible - turn sloghtly left or right, which results in a change of ±0.1 radians. We recalculate the new heading using this, and hence the new velocity according to the equation given in earlier sections. The new xy-coordinates are then calculated according to the new velocity.

The reward function is also defined here as:

- For every step taken, a negative reward of 0.1. This is to promote the agent to find the policy faster.
- If the target is reached - a positive reward of 100.
- If the boat crashes on the sides of the screen, which is termed as "channel_hit" - a negative reward of 10.
- If the boat hits the top or bottom of the screen, which is termed as "target_escaped" - a negative reward of 10.
- If the boat's heading is turned more than 90° on either side (pi/2 radians) - a negative reward of 5. This is to demote the boat moving in circles.
- A positive/negative reward every 10 steps, proportional to moving closer/farther wrt the target respectively.

A successful trial is one in which the boat's current position is within the target (not just the point, entire circular target). An unsuccessful one is when the boat crashes out goes out to the edges.

3. **reset()**
   This method is called at the compeletion of a trial. It moves the agent to (one of) the initial state(s), resets the step counter and returns the initial observation.

4. **render()**
   Used to render the environment to visualise the agent's current state and the transitions at each step. PyGame[6] has been used in this project to render a 600x600 screen denoting the open sea, the boat at the initial position, the target, wind indicator at top-right (direction & velocity) as well as the paths the sailboat agent has taken in the past trials shown as block trails. The trial in which the agent got the maximum reward up until now is denoted as a white path.

5. **close()** [OPTIONAL]
   This one is an optional utility method to close the pygame window. Closing the pygame screen will not impact the agent's learning - it would just not render the visualisation.

3.2 Helper utils

trialrunner.py

This is the runner script which initialises a SailingEnv object with the custom environment class we just created above. Taking the MAX_TEST_TRIALS from the global configuration, for each trial it calls the *render()* and the *step(action)* methods with a **random policy**, while also keeping track of the number of steps in each episode and max reward seen till now, and for which episode number. The highest reward episode is marked with a white trail for visualisation.
Please note that this script uses a random policy, and as actions are sampled randomly, it is only good enough for an initial visualisation of how the boat would sail with no knowledge of the optimum path.

train.py

While calling the individual methods of the custom environment class and keeping track of the rewards works, that is not the most elegant way to do so. Hence, we train the agent with one of the algorithms from Python's *stable_baselines* library[7].
In this project, the following algorithms have been used to train the agent and the corresponding models created have been saved in order to test them later:

1. DQN (Deep Q-Learning)
2. TRPO (Trust Region Policy Optimisation)
3. A2C (Advantage Actor Critic)
4. PPO2 (Proximal Policy Optimization)
5. ACER (Actor Critic with Experience Replay)

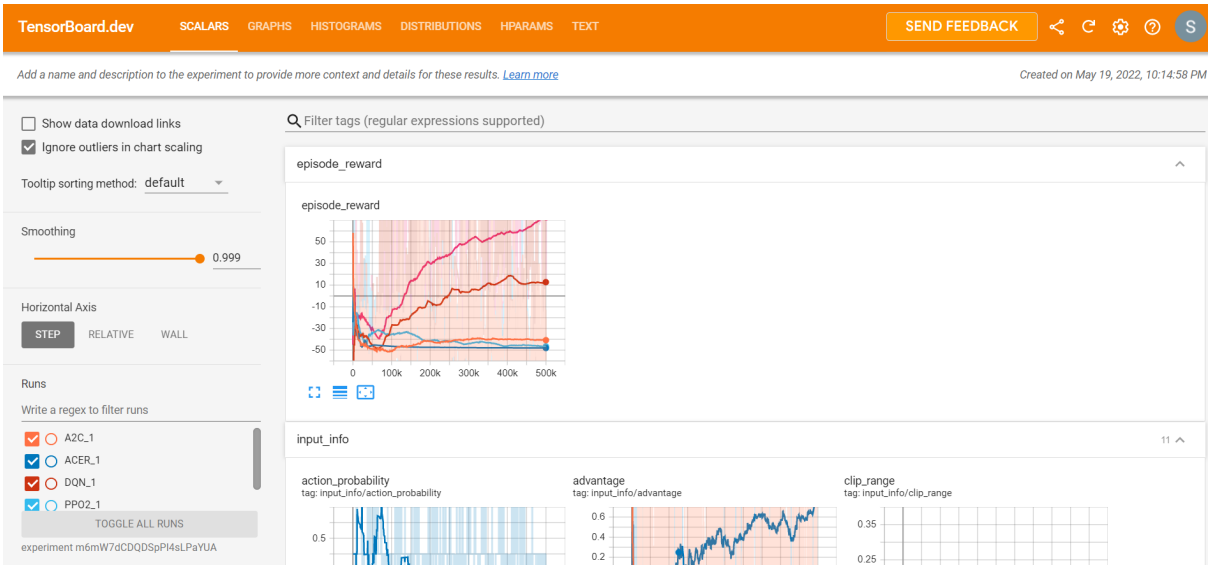
The explanation of the working of the above algorithms is out of scope for this project. Also, not all algorithms from the library could've been used as in our case, we have a discrete action space but a continuous state space (due to the boat's heading *theta*).
Furthermore, the tensorboard integration of the algorithms from stable_baselines have been used to save the logs locally during training - which are later used for a comprehensive comparison. To train individual algos, use: *python -c "import train; train.train(\"TRPO\")"*

test.py

This file loads the created models during training & use the policy learnt to predict the actions. To test individual algos, use: *python -c "import test; test.test(\"TRPO\")"*

compare_algos.py

This is a meta-runner for testing and comparing the 5 algorithms used from the stable_baselines library. For each algorithm, it calls first the train method, then the test script and at the end, uploads all the tensorboard logs created during training to TensorBoard.dev to visualise the performance and results. The link for such a newly created experiment in TensorBoard could be found in the terminal logs, for instance https://tensorboard.dev/experiment/m6mW7dCDQDSpPl4sLPaYUA/#scalars&_smoothingWeight=0.999:



4. Experimental results

4.1 Training & Testing

Training the sailboat agent with all the 5 algorithms for 500,000 timesteps each, took around 2 hours in total. Each training created its tensorboard logs, which were uploaded to create the experiment dashboard, andd the same took <4 minutes. During test, screenshots are created and stored after every 10 timesteps to keep a record of which routes the sailboat agent is taking as it gathers more information on the optimality.

4.2 Rewards

The episode reward per timestep varies a lot. So we applied max smoothing (0.999) to obtain the below graph for all the 5 algorithms. Initially in the learning phase, all the algorithms were getting negative rewards as they were still trying to figure out the best path, and in this process, the sailboat agent is colliding with the boundaries and failing to reach the target. We see that TRPO and DQN were the only ones to have found the target and managed to get a positive reward at the end of the training; and in fact, TRPO performed the best in the later steps - obtaining the highest episodic reward averaged over the timesteps.

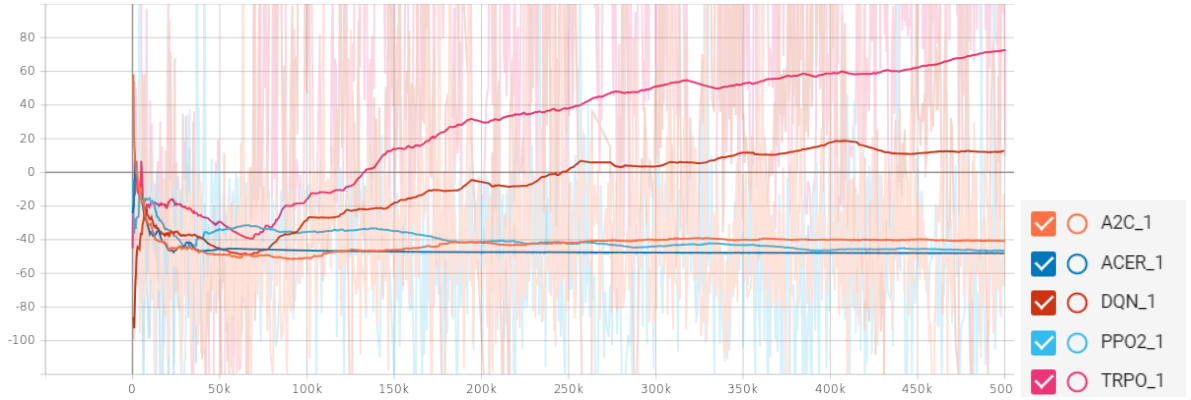


Fig. episode_reward vs timestep

4.3 Loss

Loss in reinforcement learning is the product of the negative log likelihood of the action and the return, which is to be minimised[8]. The losses are high and fluctuating in the beginning (refer the magnified view). As the training progresses, the loss first decreases sharply (but still fluctuating) and then becomes stable towards the end. One interesting observation here is that ACER and PPO2 slightly increase after the initial dip, before becoming stable.

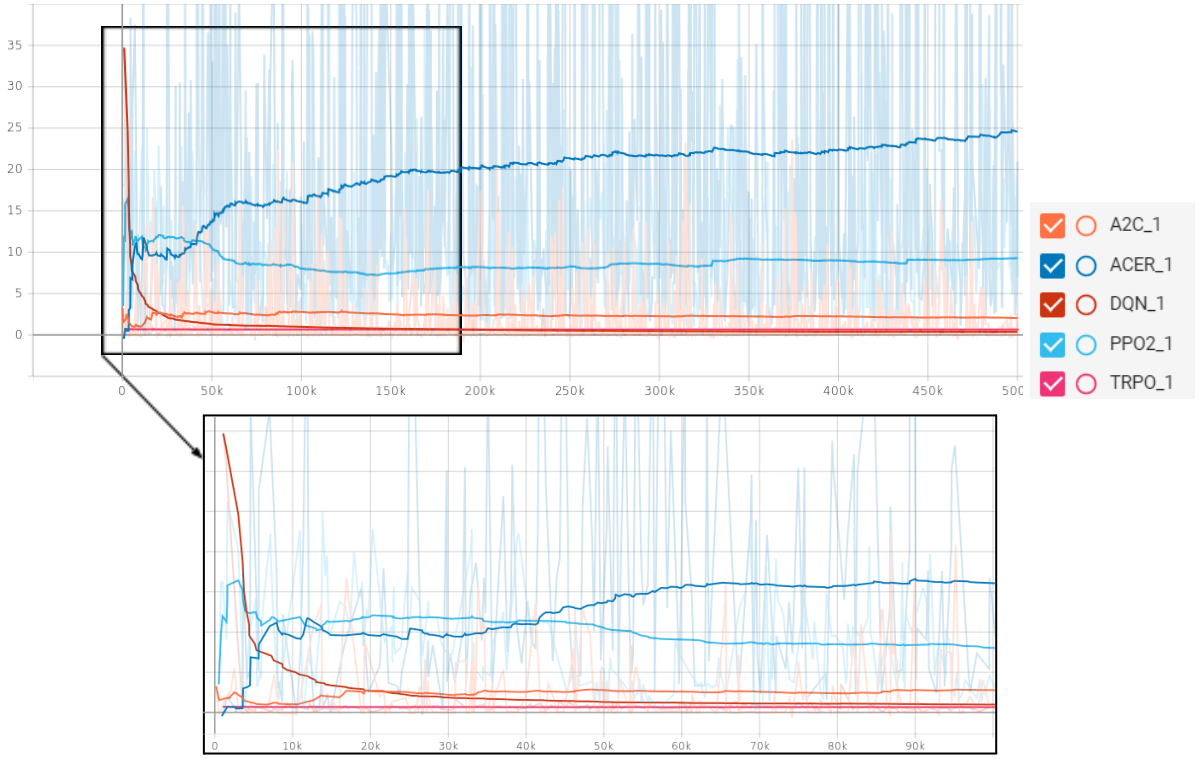


Fig. loss vs timestep

Entropy loss is a clever and simple mechanism to encourage the agent to explore by providing a loss parameter that teaches the network to avoid very confident predictions. As the agent learns, the actions should become more and more predictable - hence a decreasing entropy loss is the expected behaviour, which is also depicted by our sailboat agent. In all algorithms except ACER, this decrease is very, very gradual.

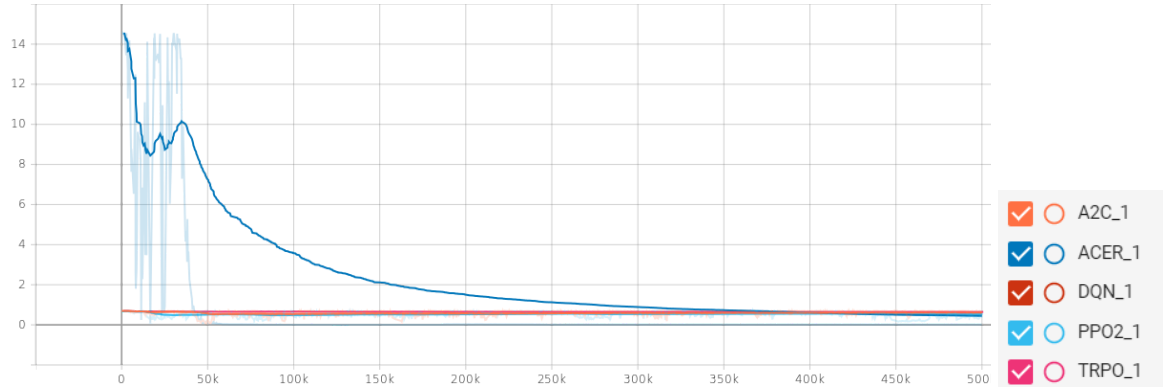


Fig. entropy_loss vs timestep

5. Conclusions

We observe from the rendered training runs that initially, as the sailboat agent is unaware of which actions to take (i.e. which direction to turn to), it takes random paths in order to try to reach the target location, also crashing into the sides in the process:

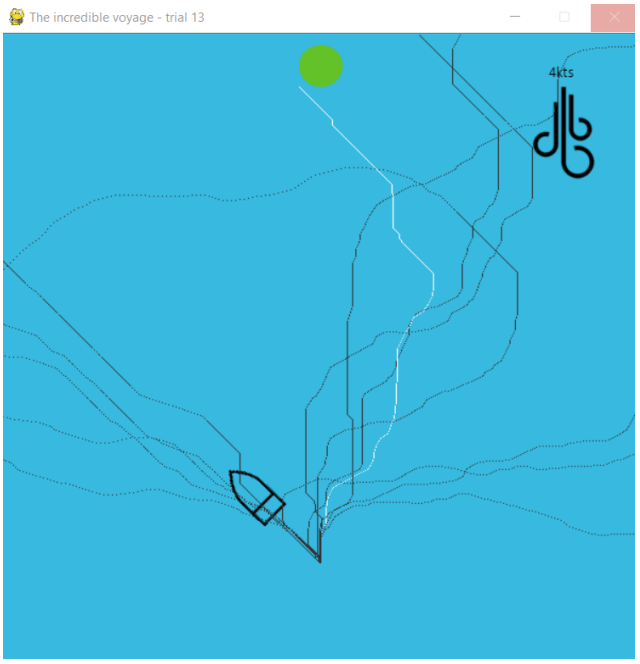


Fig. Random paths chosen by the agent

As it is not able to sail directly upwind, the agent learns to perform the maneuver of tacking - going in a zig-zag way - in order to reach the target directly upwind. More and more often, the boat *beats* instead of choosing random actions. This "learning" is observed the fastest while using TRPO as the algorithm (with *stable_baselines'* default parameters):

Fig. Tacking behavior learnt by sailboat agent (highest reward path showed in white)

6. Further scope

The sailboat model used in this implementation is a far from accurate representation of real-life. As such, the effects of water current, air and water drag should also be added to the velocity equation.
The hyperparameters individual to each of the algorithms could be tweaked and the performance graph plotted in order to find the best <algorithm + configuration> overall. This step was deliberately missed out on in this project as the algorithm was not the centre of concern.
Additions in the action space for a more fine-grained control could include tension in lines (i.e. level of trimming) of sails - which in this case has been assumed to be always at an optimal level. Although, for *real sailboat agent*, this is one additional point of control in order to gain or lose velocity. Lose or "luffing" sails could result in slowing down of the boat - and as such could be learnt by the agent when it needs to avoid an obstacle while cruising at high velocities. Since this is not currently taken into account, the only way the agent can slow down is to move the boat "in irons" - i.e. directly towards the wind.
More global configuration variables could be added to model different boats - size of the hull, sails and weight of the boat could all affect the resistance.

7. References

1. https://medium.com/cloudcraftz/build-a-custom-environment-using-openai-gym-for-reinforcement-learning-56d7a5aa827b
2. https://stable-baselines.readthedocs.io/en/master/guide/custom_env.html
3. http://www.phys.unsw.edu.au/~jw/sailing.html
4. http://www.ockam.com/2013/06/03/what-are-polars/
5. https://gym.openai.com/docs/#environments
6. https://www.pygame.org/docs/
7. https://stable-baselines.readthedocs.io/en/master/guide/algos.html
8. https://ai.stackexchange.com/questions/35023/what-is-the-difference-between-a-loss-function-and-reward-penalty-in-deep-reinfo