Universidade de Aveiro

Wall Following

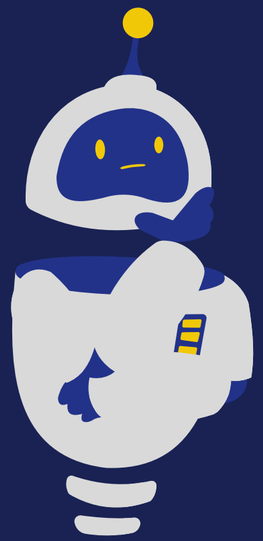
USING THE CIBERRATO SIMULATION ENVIRONMENT

Perception and Control
Assignment 2
2023/2024

David Alexandre Ornelas
nºMec: 109802
MRSI

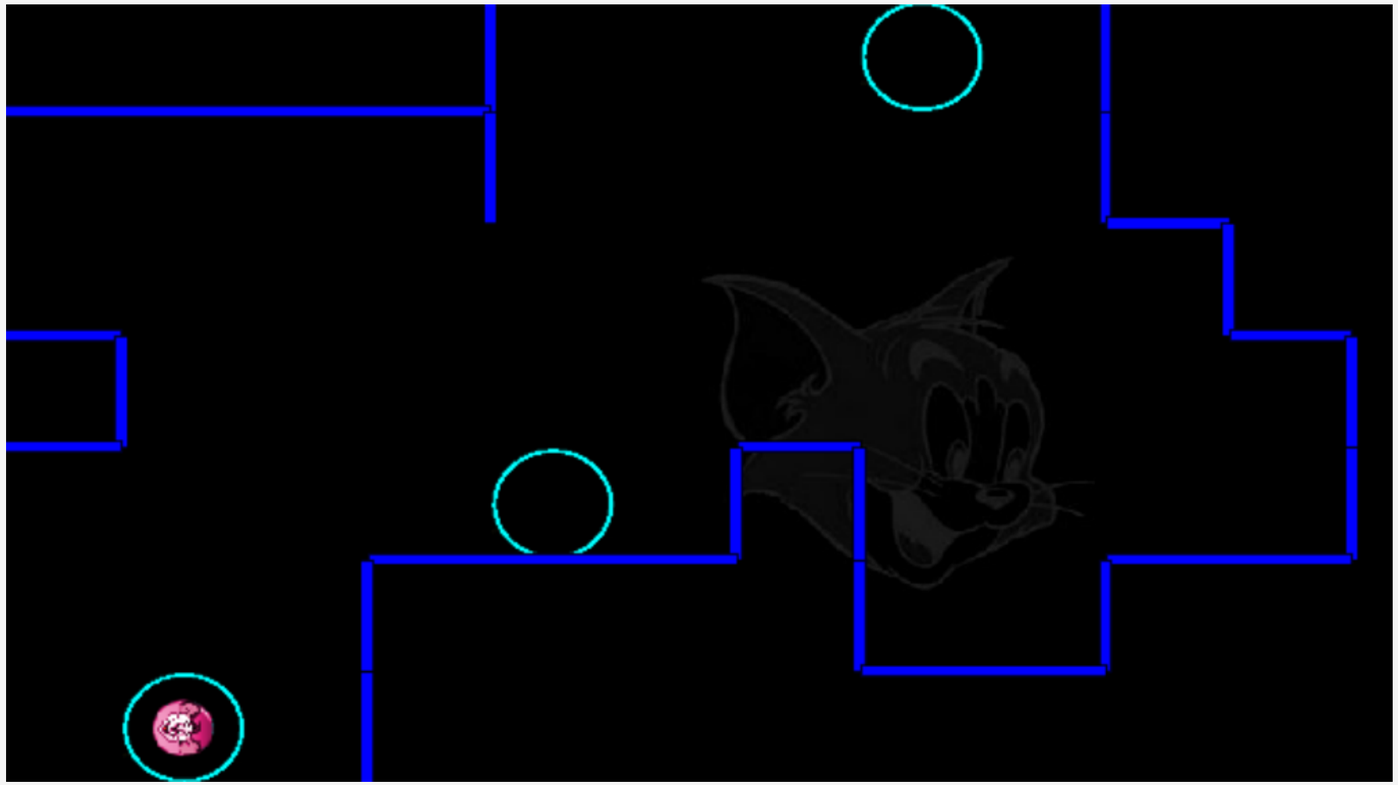
INTRODUCTION

- **Intro:** Develop a mobile robot agent able to follow a wall.
- **Rules:** Follow the wall by keeping it on the right of the robot without deviating too much.
- **Goal:** Develop two agents, one based on a classic control approach and another based on Machine Learning or Fuzzy Control .



ENVIRONMENT

- Navigate on the maze based on obstacle sensors (distance to wall) and ground sensor (checkpoints).

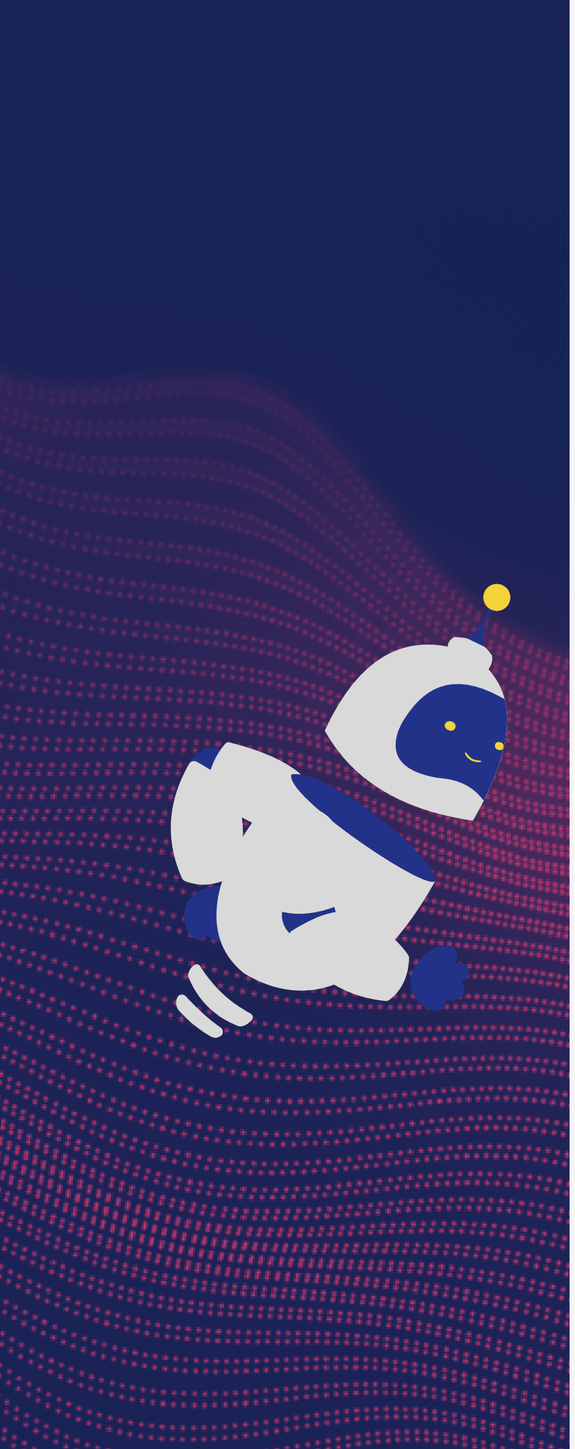


CLASSIC APPROACH

- The classic approach was based on this steps:
  - load sensor data (obstacle sensors)
  - movement model + coordinate retrieval
  - control
  - navigate

- The agent makes decisions each new cell based on the distances to the wall from the obstacle sensors.

- Possible decisions: Move, Rotate, Turn.



MOVEMENT MODEL

- Movement model adapted from the given equations.
- Based on the wheel speed (left/right), the model returns the linear and rotational component with gaussian noise
- Based on estimated pose from both components, the distance from previous coordinate will measure if a new cell is close.
- Retrieve new coordinate when a distance > 1.9 is reached and update current position each new coordinate.

CONTROL

- Distance from robot to wall used:
  - 1/sensor measure - half wall size.
- The orientation is normalized to [-180°,180°].
- The default state is “rw” -> Right wall.
- Each new cell the agent checks the sensors data and chooses a new state (right wall, front wall, no wall).
- When the agent is in state no wall, it turns right, cause there’s no right wall nearby.

CONTROL

- A threshold was defined for both front and right sensor, to check if it’s to close to a wall or not.
- If it detects a front wall or no wall, the agent either rotates left or right depending on sensor data.
- If it just detects the supposed right wall, the agent keeps moving forward updating his position each cell.
- In case it deviates from the wall, it fixes the trajectory by turning slowly on the opposite direction.

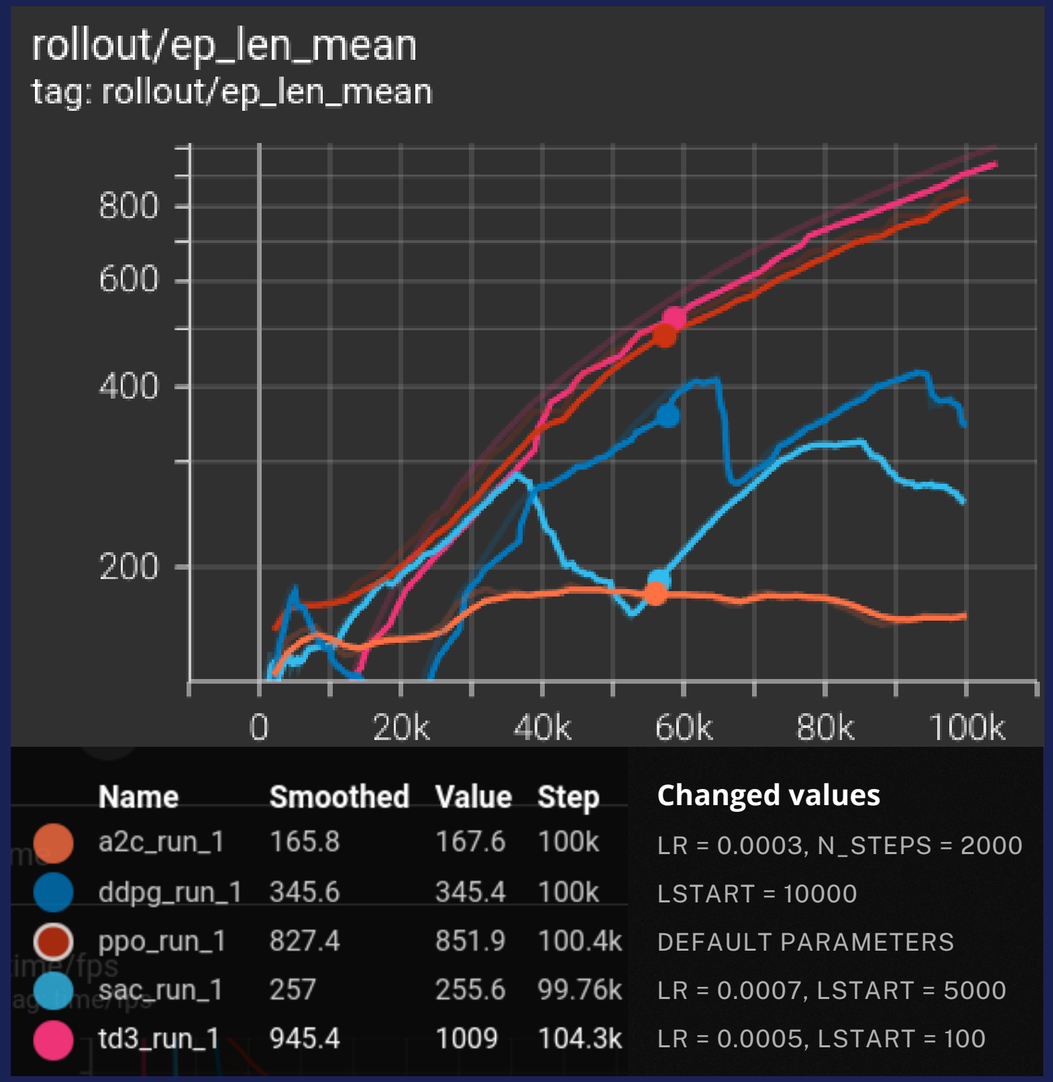
TESTS

- The agent seems to start well updating the trajectory, however after several cells, it starts losing track of its position by updating either too fast or too slow from the expected position.
- This method proved to be unreliable, achieving a score from 100 to 1000, depending on the collisions and losses on trajectory.
- A detailed review from this approach would be necessary to fix the trajectory problems (noise).

MACHINE LEARNING APPROACH

- For this environment, the approach used was based on the example given in class.
- The example uses reinforcement learning models from **Stable Baselines 3**.
- Several **default models** were tested, as a first approach, to understand which algorithms would converge better into optimal scores.
- Models tested: **A2C**, **DDPG**, **DQN**, **PPO**, **SAC**, **TD3**.

MODEL TESTING



- The models were tested with the default parameters from SB3.
- Some parameters were adjusted to avoid bad trainings. For the 1st approach it was adjusted just the learning rate and the learning start.
- PPO and TD3 performed better overall. PPO as default converged smoother than the others.
- PPO was the model chosen to fine-tune to the environment, due to having a more stable training.

PPO MODEL

- PPO performs on-policy training, by sampling data from it's interaction with the environment and updates its policy each iteration.
- Focus on taking the biggest possible improvement step on a policy using the current data, without stepping too far.

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

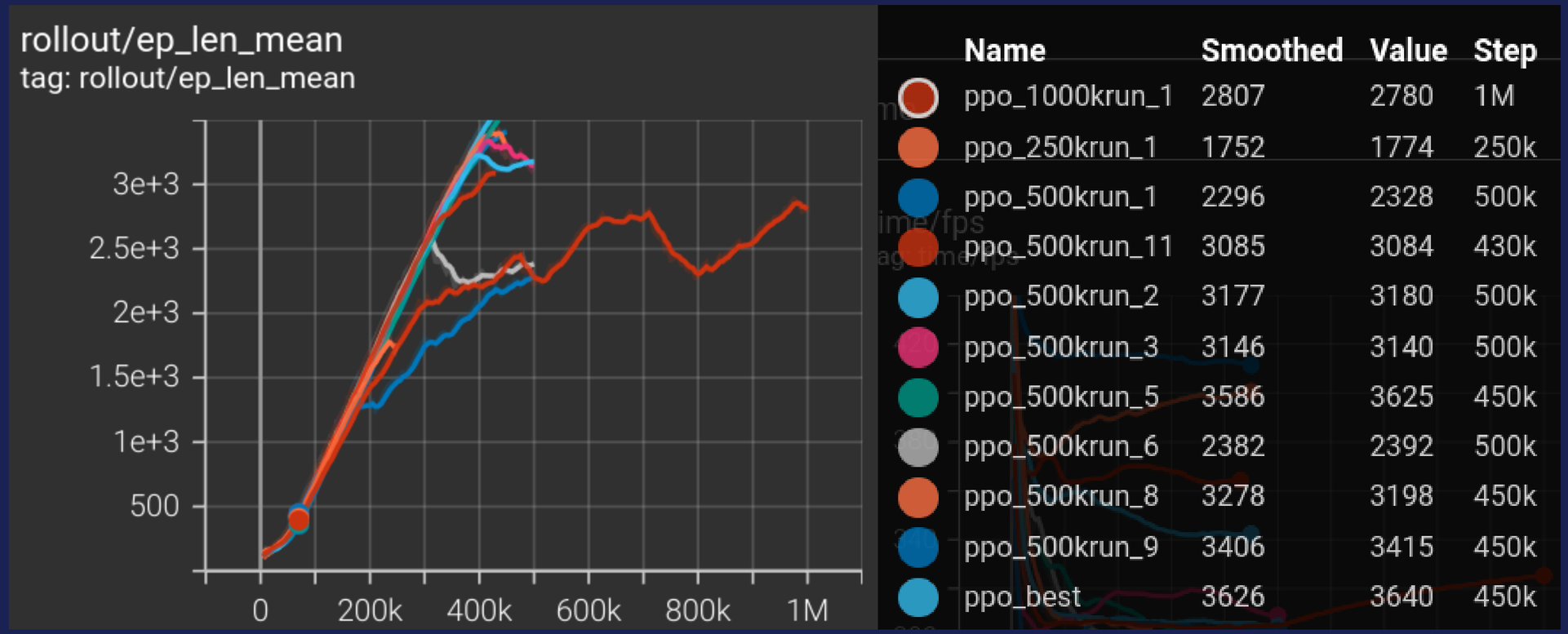
PPO PARAMETERS

- The policy used was the Multi-Layer Perceptron, that consists of 2 hidden layers of 64 for both Actor and Critic, which are fully connected neural networks.
- The major parameters tuned were the learning_rate, nº of steps, batch_size, nº of epochs and lambda for N time steps.
- The number of steps was changed to 5000, to give the agent time to experience before updating policy, with a batch size of 250 (samples at a time).
- The number of epochs was increased to 20, to learn more from his samples.
- Learning rate and lambda were the most important parameters on training.

FINE TUNING

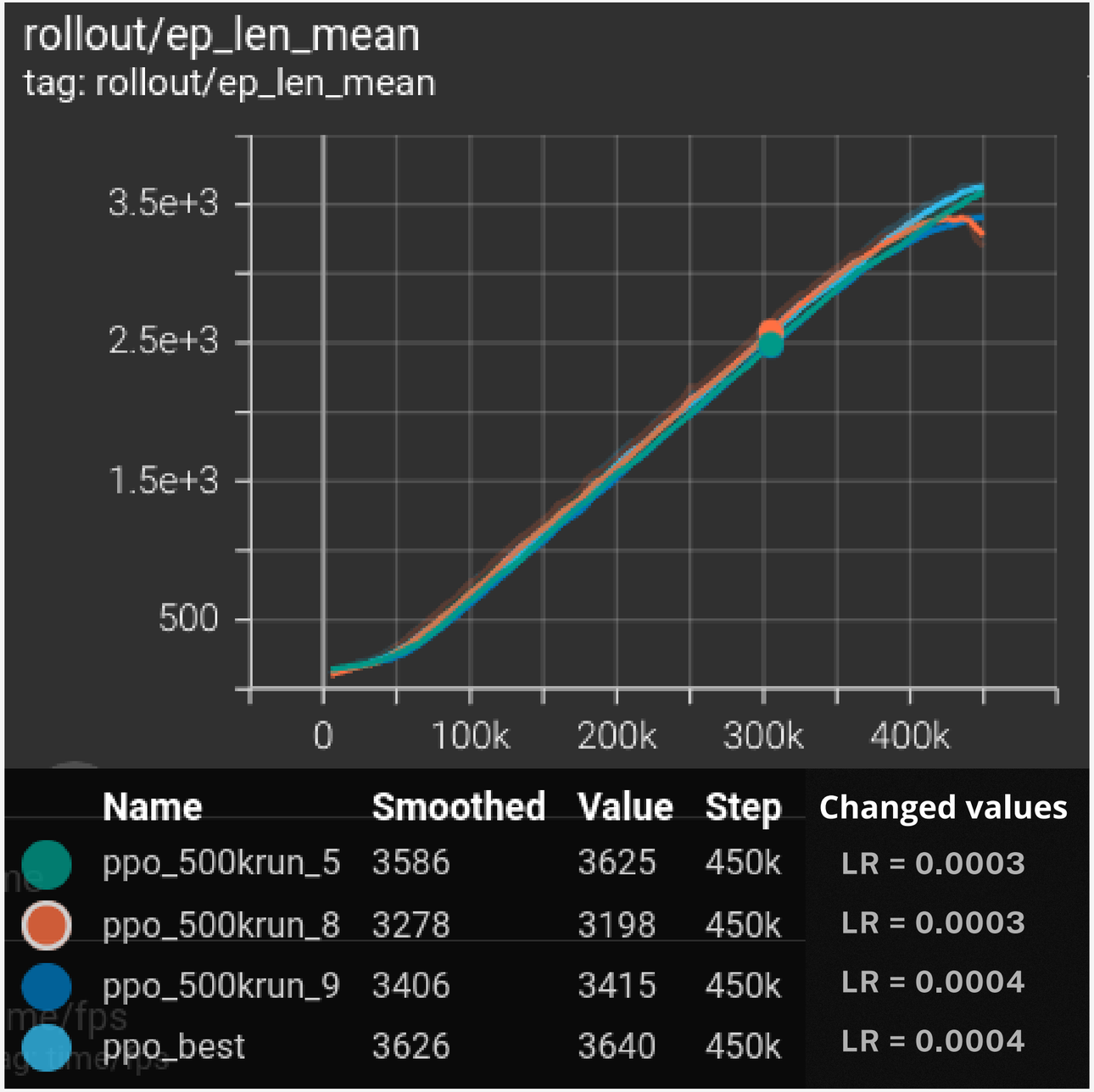
- PPO supports multiple environments however in this case it was not used.
- Several tests were done with different total of time steps, however the best convergence to optimal results was around 400.000 and 500.000, for a learning rate between 0.0003 and 0.0005.
- The value of lambda was increased from 0.95 to 0.99 to focus more on past experiences, since the robot needs to pass each cell close to the wall to score points, otherwise he could miss a cell and get bad convergence due to not scoring on good cells just by missing one cell.
- To keep track of the robot training, besides the viewer from ciberRato, it was also used the tensorboard logs, to keep track of the mean scores and train loss.

TESTS



Name	Smoothed	Value	Step
ppo_1000krun_1	2807	2780	1M
ppo_250krun_1	1752	1774	250k
ppo_500krun_1	2296	2328	500k
ppo_500krun_11	3085	3084	430k
ppo_500krun_2	3177	3180	500k
ppo_500krun_3	3146	3140	500k
ppo_500krun_5	3586	3625	450k
ppo_500krun_6	2382	2392	500k
ppo_500krun_8	3278	3198	450k
ppo_500krun_9	3406	3415	450k
ppo_best	3626	3640	450k

- The models with best performance as previously mentioned were the ones around 400k~500k total time steps, while values around 200k and 1M would lead to worse learning. It was also tested with different learning rates as previously mentioned. However with lambda changes the model performance increased from a mean score of 2000 to 3000.

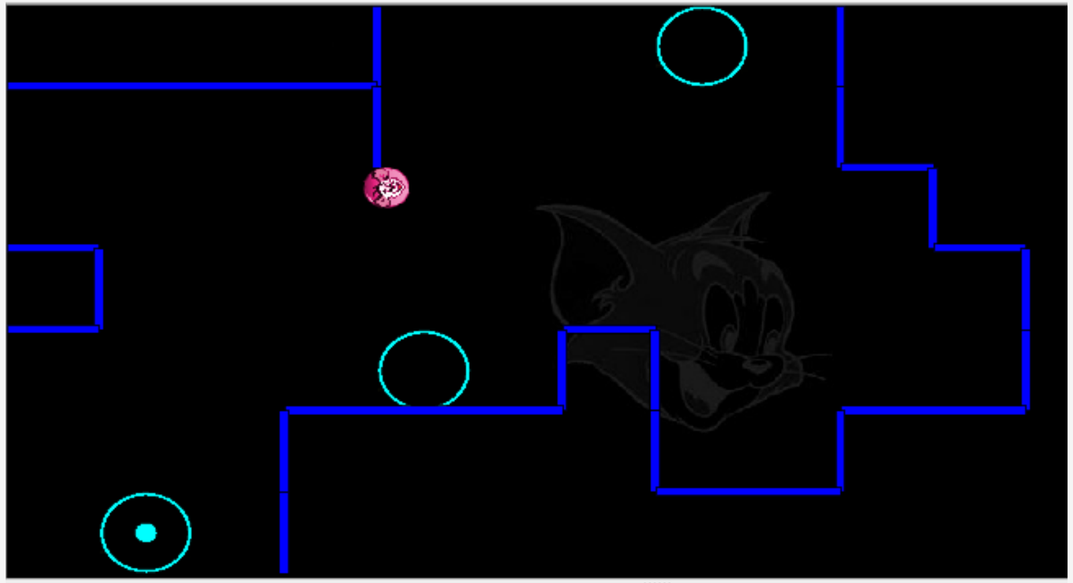


RESULTS

- The best results were all with the same parameters:
  - 450.000 total time steps;
  - nº of steps = 5000;
  - batch size = 250;
  - nº epochs = 20;
  - lambda = 0.99;

- Overall the PPO agent performs the task quite well and with small miss steps.

CONCLUSIONS



- **The PPO agent** sometimes would miss turn on this wall, however it performs the task as intended with a small margin of error, with a mean score of 3600 per simulation.

- **The classic agent:**
  - Several tests were made to improve this, however due to the lack of time, the agent did not achieve the intended performance.
  - In some cases the agent would do smooth laps with 0 or few collisions, but unreliable due to its inconsistency.
  - PPO proved to be more reliable in achieving the intended result.

Universidade de Aveiro

THANK YOU !

REFERENCES

https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

https://spinningup.openai.com/en/latest/algorithms/ppo.html

https://stable-baselines.readthedocs.io/en/master/modules/policies.html

https://stable-baselines3.readthedocs.io/en/master/guide/algos.html

https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html

https://stable-baselines3.readthedocs.io/en/master/guide/tensorboard.html