Autonomous Underwater Vehicle Control using Reinforcement Learning

Sthithpragya Gupta and Giovanni Napoli

Universitat Jaume I, Castellon de la Plana, Spain

Abstract

This project aims to explore Reinforcement Learning (RL) strategies to control the motion of an Autonomous Underwater Vehicle (AUV) by navigating it via user-specified way-points. The sensor information is supplied to the RL framework which outputs a policy and controls the thruster actuators to reach the desired way-point. The AUV used in the project is Girona 500. The policy generated by the trained RL framework has been simulated in UnderWater Simulator Graphical User Interface (UWSim GUI) allowing the visualization of the AUV motion.

1 Introduction

The domain of Autonomous Underwater Vehicles is vast and challenging. One of the most crucial elements of this domain is Navigation. A common method of having an AUV navigate from the start to the goal position is by specifying a few intermediate way-points along the trajectory which the vehicle should follow. However, there are multiple challenges to the problem of autonomous navigation viz:

- Highly coupled dynamics of the vehicles
- Unknown parameters for the dynamic model
- Nonlinearities arising due to flow and hydraulic resistance
- Limited data about the underwater environment
- Errors due to turbulences and external forces

To face these challenges, a self-learning and decision-taking agent which can refine its behaviour can be employed. Thus, in this study, we have attempted to build a Reinforcement Learning based controller for navigating the AUV through the way-points. Reinforcement Learning (RL) is a branch of machine learning algorithms which aim to learn the action-situation mapping to maximise a numerical reward, without any prior knowledge about the actions[1]. The elements of an RL framework are (fig. 1):

- **Action** A_t - Action executed by the robot at any time-step t
- **Observation** O_t - Outcome associated with executing A_t
- **State** S_t - New state of the robot post action
- **Policy** π - Determines which action to undertake
- **Agent** - The robot (AUV) responsible for learning the policy
- **Environment** - The external environment with which the robot interacts (ocean)

2 Formulating the Control Architecture

The first step towards implementing a control scheme for the AUV was to survey and find out the appropriate Reinforcement Learning techniques suitable for AUV control.

2.1 Value-based vs Policy-based techniques

The entirety of RL techniques can be broadly classified on the following bases:

- **Value-based vs Policy-based techniques** - Classification on the basis of how the technique quantifies the available information and arrives at the optimal policy.

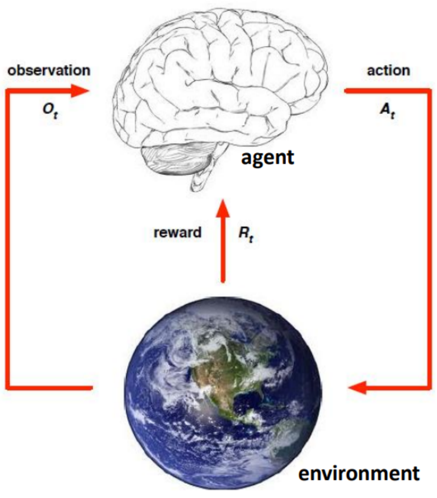


Figure 1: A general RL framework (adapted from slides of David Silver's course)

- **Model-based vs Model-free techniques** - Classification on the basis of whether the technique builds a model of the environment for learning or not
- **On-policy vs Off-policy techniques** - Classification on the basis of whether the technique determines the optimal policy by learning on a static or dynamic data-set

All these classifications cater to different aspects of a technique. However, from among these. the Value vs Policy-based classification is most crucial as it determines first-hand whether the plausible technique may yield satisfactory results for the problem or not.

Value-based techniques work by quantifying how beneficial it is to be in a particular state (or perform a particular action in that state) when trying to reach a goal. To do so, two quantities: *state-value function* and *action-value function* are defined. The state-value function for a state s when following a policy π is the expected reward return for the agent when starting from that state and following the policy. Similarly, the action-value function is defined as the expected reward for the agent when starting from s, taking the action a and the following policy. [1]. By greedily selecting the states (or actions) with the highest values, the optimal policy can be formulated.

Contrary to value-based techniques, policy-based techniques work by directly parametrizing the policy. By introducing these *policy parameters* in the policy to determine the optimal action, the expected reward can be formulated in terms of the parameters. Thus by optimising the policy parameters, we can train the agent to achieve the highest reward. Both the aforementioned techniques have their advantages and disadvantages. However, the advantages of policy-search techniques often outweigh their disadvantages [2].

Advantages of policy-based techniques over value-based ones:

- Faster convergence
- More efficient for models with high-dimensional state and (or) action spaces
- Can also work with continuous state and (or) action spaces
- Can learn stochastic policies

However, these methods also suffer from some disadvantages viz:

- Slower convergence on noisy data as compared to value-based methods
- Unlike value-based methods, policy-based methods may get stuck in a local optima

Since the goal objective is to control the AUV position (state space) by altering the thruster actuation (action space), both spaces being continuous rather than discrete, we need a technique to handle both continuous state and action spaces. Hence the choice of policy-based search methods. Furthermore, we will be implementing a model-free approach as our framework simply determines the most suitable action from the raw input data. Also, learning will be off-policy.

2.2 Gradient descent based direct policy search

The policy in this method is represented via a shallow neural network (fig. 2). The input to the network is the state of the AUV and the output is the action, defining the thruster actuation. The weights of the neural network represent the policy parameters. Stochastic gradient descent is used to compute the gradients, which is followed by backpropagation to update the weights with respect to the gradients [3].

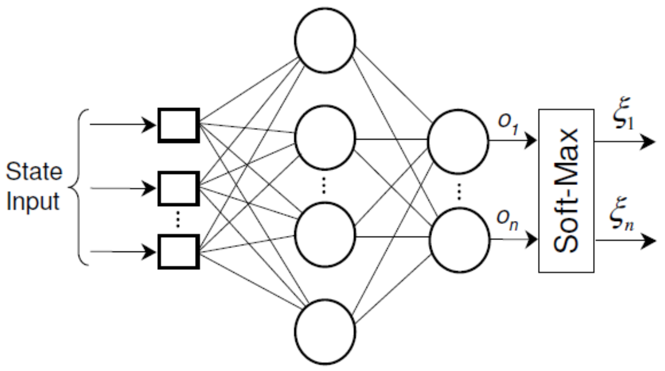


Figure 2: A schematic of the neural network architecture (adapted from [3])

The procedure for the gradient descent based direct policy search is as follows [4] - Initialize the policy parameter vector θ_0, initial state s_0, and the gradient vector $z_0 = 0$. Post initialization, the learning procedure is iterated for T time-steps. The policy is μ which at any time-step t is simply determined by θ_t the policy parameter vector then. For every time-step, do:

1. Generate the control action a_t according to the policy $\mu(\theta_t)$
2. Execute the action and observe the reward obtained $r(s_{t+1})$ for entering the new state s_{t+1}
3. Update the gradient vector ($\beta \epsilon [0, 1)$ is a tunable parameter)

$$z_{t+1} = \beta z_t + \frac{\nabla \mu(\theta_t, s_t)}{\mu(\theta_t, s_t)}$$

4. Update the policy parameter vector (α is the fixed rate learning parameter)

$$\theta_{t+1} = \theta_t + \alpha r(s_{t+1}) z_{t+1}$$

Post this training phase, we have a refined policy to work and implement. While this approach is simple, easy to implement, and works well for cases with a low-dimensional action space, it has some disadvantages:

- The approach is computationally inefficient when handling a large data-set, especially one with high variance
- For cases involving a high dimensional or a continuous action space, the approach is computationally very inefficient [5]
- Prone to get stuck in a local optima while finding the optimal policy

The aforementioned disadvantages prompted us to look for alternatives which could improve upon these disadvantages. To improve upon the method, following initiatives could be undertaken:

- Increasing the number of policy parameters - By doing so, not only would the policy be able to accommodate higher dimensional state and(or) action spaces, but also handle the noise and variance in the data better. This can be achieved by using a Deep Neural Network instead of a shallow one to represent the policy to be learned which is the underlying feature of Deep Reinforcement Learning [6].

- Borrowing elements of value-based approaches - Value-based methods are more adept to handling high-variance data as well as don't get stuck in a local optima while optimising the policy. This can be achieved by employing a hybrid approach known as the Actor-Critic method which has been explored in the next section.

Thus, in the subsequent method, we combine using Deep Neural Network for policy representation with the hybrid Actor-Critic method allowing us to implement Deep Reinforcement Learning using Actor-Critic method.

2.3 Deep Reinforcement Learning using Actor-Critic method

Actor-critic methods aim at combining the strong points of policy-based and value-based methods. The actor is a deep neural network which parametrises the policy (same as before) to arrive at the optimal policy. The weights of the actor neural network represent the policy parameters θ. The critic is a deep neural network which approximates the action-value function and is used to determine whether the action undertaken by the actor was good or bad. It is used to guide on how to update the actor's policy parameters. The weights of the critic neural network represent the action-value parameters w.

The procedure can be summarised as follows [7] - Initialize the policy parameter vector θ_0, initial state s_0, and the action-value parameter vector w_0. Post initialization, the learning procedure is iterated for T time-steps. For every time-step, do:

1. Execute the action a_t and observe the reward obtained $r(s_{t+1})$ for entering the new state s_{t+1}
2. Compute the next plausible action a_{t+1}
3. Compute the new action-value function $Q_{t+1}(w_t, s_{t+1}, a_{t+1})$
4. Compute the local gradient vector δ (γ is the discount factor)

$$\delta = r(s_{t+1}) + \gamma Q_{t+1} - Q_t$$

5. Update the policy parameter vector (α is the fixed rate learning parameter)

$$\theta_{t+1} = \theta_t + \alpha \nabla log\mu(\theta_t) Q_{t+1}$$

6. Update the action-value parameter vector ($\phi(s, a)$ is a critic defined feature vector)

$$w_{t+1} = w_t + \beta\delta\phi(s_t, a_t)$$

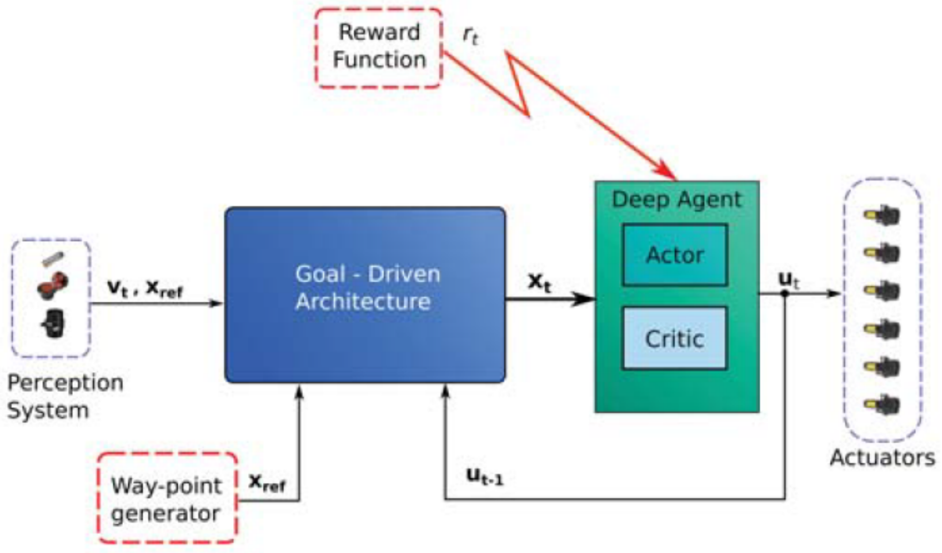


Figure 3: A schematic of the actor-critic architecture for AUV Control (adapted from [8])

Post this training phase, we have fully trained the actor and critic neural networks and have a refined policy to work with and implement. For our use-case (fig. 3), the input to this cumulative network is the state of AUV and the output is the action, defining the thruster actuation[8].

3 Implementing the Architecture

This section deals with implementing the framework for using the Deep Actor-Critic method for AUV control. The framework has been prepared in python using the *Stable Baselines* library, an extension of the OpenAI Gym, developed by robotics lab U2IS (INRIA Flowers team) at ENSTA ParisTech. Furthermore, it would be useful to introduce some terminology. The state s of the AUV is a 12 dimensional vector. It contains the following:

- 3 dimensional Cartesian goal displacement vector - displacement between current position and goal
- 3 dimensional orientation vector - current roll-pitch-yaw orientation of the AUV
- 6 dimensional velocity vector - current velocity of the AUV

As the choice of AUV for the study was Girona500, the action a from any state is a 5 dimensional vector corresponding to the 5 thrusters mounted on Girone500.

3.1 Building the framework

Conventionally, in any study involving usage of an RL agent to control a robot, the general framework is to first train the RL agent on a simulator of the robot and then test it on the actual hardware. However, as in our case the test is to be conducted on UWSim [9] (a simulator itself), we first needed to prepare a basic simulator of UWSim to train our RL agent on. To do so, the key functions from *dynamics.py* ROS node (of the UWSim package) were ported to a separate script which could be executed in conjunction with the training process of the RL agent. The aforementioned ROS node simulates the underwater oceanic environment. The ported script also performs the same function as the aforementioned ROS node which is to determine the forward and inverse dynamics of Girona500. It was used to determine the new state of Girona500 after executing an action as determined by the RL agent (fig. 4).

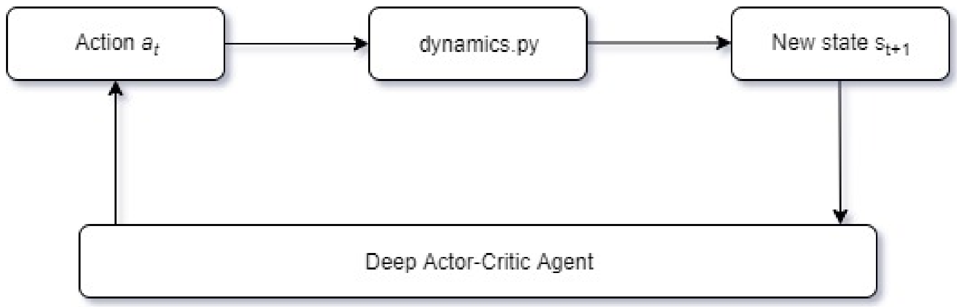


Figure 4: A schematic of the usage of the ported dynamics script

The next crucial step is that of reward allocation. By allocating appropriate rewards, not only the training process can be sped up, but also the quality of results obtained be improved. As the objective of the study is to control only the position of the AUV and not its orientation or any other aspect, for the determination of reward we will be using the Cartesian goal displacement d. We have used the reward allocation scheme presented in [8] which is as follows (fig. 2):

- If $d < \beta$, reward = 10 (favourable)
- If $\beta < d < \beta_{max}$, reward $\epsilon\{-1, -10\}$ (somewhat in the correct region, faces a penalty proportional to distance from the goal)
- If $d > \beta_{max}$, reward = -10 (highly unfavourable, take corrective measures as soon as possible)
- If $d >$ *bound*, reset the training as AUV has gone way off from the desired trajectory

The parameters β, β_{max} and *bound* are tunable parameters which were set via trial and error during the training phase (more in the following part).

3.2 Training the agent

Once the framework was completed, the training phase and parameter tuning began. The actor-critic method which was used to train the agent was the Advantage Actor Critic (A2C) method. It is a synchronous and deterministic implementation of the actor-critic method which utilises multiple workers in

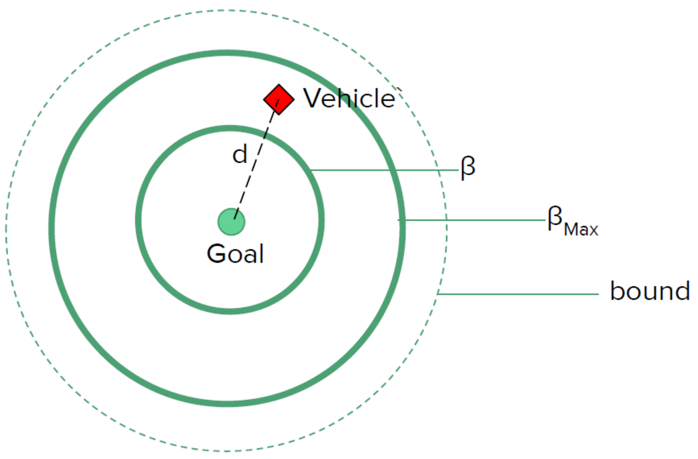


Figure 5: The reward allocation boundaries

parallel environments to speed up training and make the most of the available GPU resources [10].

This was followed by training the agent on the framework following the A2C method for certain time-steps. However, the number of time-steps could not be pre-determined. This, coupled with the uncertainty regarding the β, β_{max} and *bound* parameters meant that we had to proceed with the training in batches. The methodology adopted in [8] stated to initially train the agent for a certain time-steps on relatively forgiving limits for these parameters and then eventually make the β, β_{max} and *bound* parameters more strict. In doing so, we could first train the agent to crudely reach the goal and then bootstrap from the learnt behaviour to further learn more sophisticated, and refined behaviour. For our case, this translated to reaching the goal way-point with higher accuracy. After trial and error, we arrived at the following training regime for the agent:

1. First train for 1,000,000 time-steps with $\beta = 200cm$ and $\beta_{max} = 435cm$
2. Next train for another 1,000,000 time-steps with $\beta = 175cm$ and $\beta_{max} = 410cm$
3. Next train for another 1,000,000 time-steps with $\beta = 150cm$ and $\beta_{max} = 385cm$
4. Next train for another 1,000,000 time-steps with $\beta = 100cm$ and $\beta_{max} = 335cm$
5. Next train for another 1,000,000 time-steps with $\beta = 50cm$ and $\beta_{max} = 285cm$
6. Next train for another 1,000,000 time-steps with $\beta = 25cm$ and $\beta_{max} = 250cm$

The *bound* was set to be 435cm all throughout. If the goal displacement crossed this limit, the state of the AUV (goal displacement, pose, velocity) were reset to random values and the training proceeded. While the limits could be made even more stricter, with each training pass, the time taken for training on finer β and β_{max} increased significantly. Thus, we decided to limit the training to 6,000,000 time-steps in total.

4 Testing the RL agent in UWSim - Interfacing with ROS

As stated before, we simulated the trained RL agent using UnderWater Simulator Graphical User Interface (UWSim GUI), that allowed us to visualize and test the AUV motion. Our initial idea was to simply add a node to the original UWSim dynamics package which contained the trained RL agent (fig. 6). This node subscribed to `/g500/pose` topic, evaluated the current state of Girona500 and computed the action or thruster actuation. This thruster actuation would then be published to the `/g500/thrusters_input` topic. However, we soon faced problems with this approach regarding:

- mismatching versions of python
- state-vector that had to be fed to the RL agent
- changing the basic scene

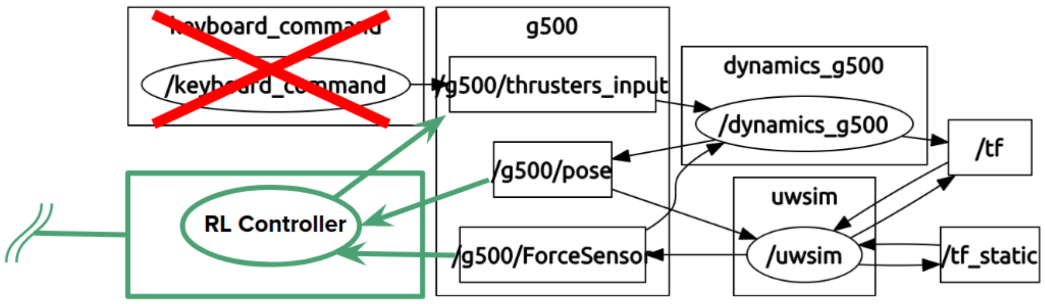


Figure 6: The initially planned ROS implementation containing a node for the trained RL agent

4.1 Mismatching versions of Python

As previously stated, we used the *Stable Baselines* library (to train and access the agent), available only in Python3, and the UWSim ROS package, which is available only in Python2. Consequently, we could not directly interface the two. We could not convert one of the two into the other version of python. So, instead of writing a single code, we opted to write two different codes: one running in Python2 and the other in Python3 to bridge the flow of information.

- **Python2**: This program aims to create a ROS Node that subscribes to `/g500/pose` and publishes on `/g500/thrusters_input`.
- **Python3**: This program uses the RL library to access the agent and evaluates the distances, and computes the forces that must be actuated by the thrusters of Girona500.

At this point, we wanted to use the code in Python3 like a 'service' for the Python2 code. For better comprehension, let's see the pseudo-code:

- **Python2** gets the information from `/g500/pose`;
- **Python2** calls the Python3 code passing the Pose information;
  - **Python3** evaluates the costs and computes the thrusters forces;
  - **Python3** sends back to the Python2 code the thrusters forces;
- **Python2** publishes on `/g500/thrusters_input`;
- **Python2** waits for a new message from `/g500/pose`.

The problem with this implementation was that it was not feasible to launch a Python3 code from a Python2 program. So, we made as 'service' the ROS Node code (in Python2) that can be called by the RL Algorithm program (in Python3). Again, for a better comprehension we show the algorithm of the two working programs:

- **Python3** evaluates the thrusters;
- **Python3** calls the Python2 code;
  - **Python2** publishes on `/g500/thrusters_input`;
  - **Python2** gets the info from the Pose;
  - **Python2** calls back the Python3 code and shares the information;
- **Python3** starts again

All the aforementioned interfacing has been accumulated into a single node which is `/RL_interface` (fig. 7). This node is initialised with the remainder UWSim dynamics package.

4.2 State-Vector

The State-Vector s (containing information about Girona500's goal displacement, pose, velocity) that must be fed to the trained RL agent, needs to have a 6-dimensional velocity vector. However, the current UWSim implementation only allowed for a 3-dimensional velocity vector containing the linear velocities along the coordinate axis. To overcome this, we had to feed the angular velocities as well. We decided to leave out the angular velocity about the roll direction since it cannot be actuated and its magnitude is negligible. This left us with having to add the two angular velocities (along with the pitch and yaw). So, we changed the `/dynamic` node to also publish `/g500/pdot` and we subscribed to that from the Python2 code. So finally the ROSgraph is the one shown in (fig. 7).

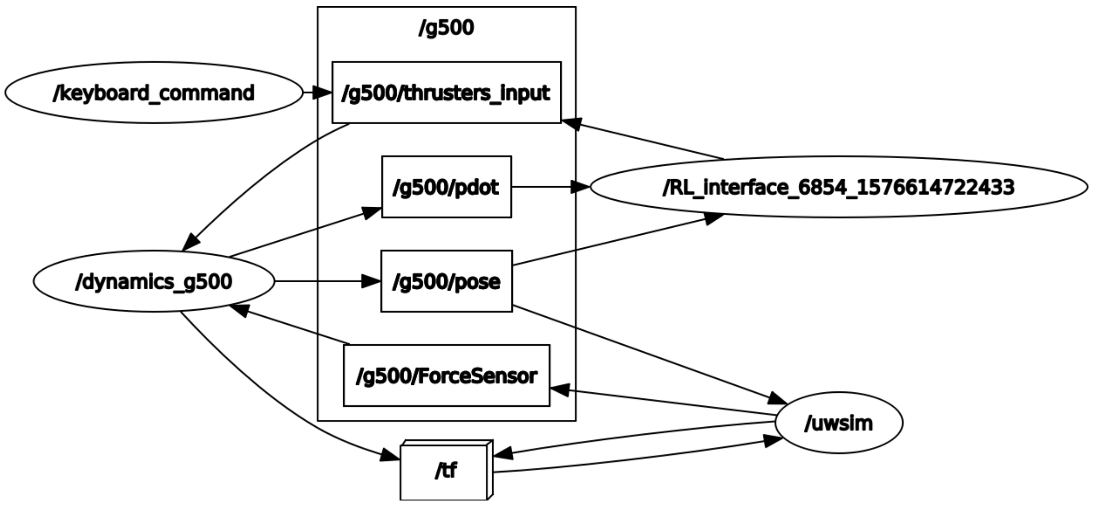
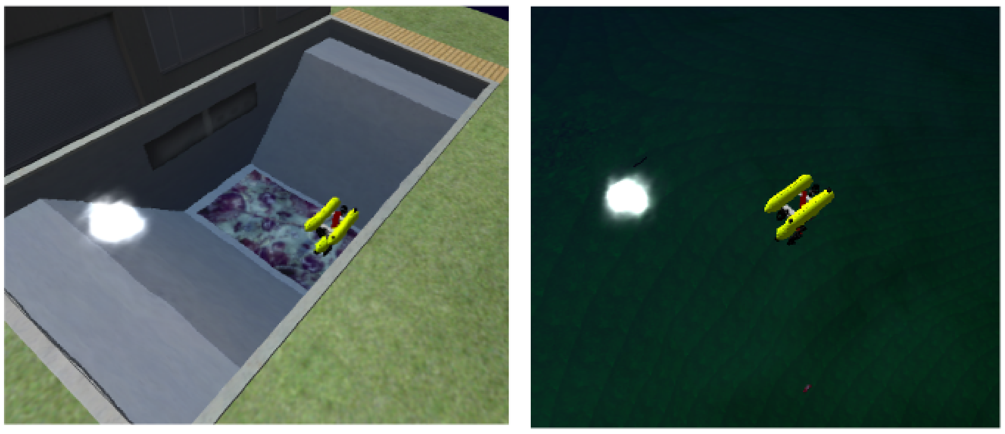


Figure 7: ROS Graph of UWSim running with `RL_Interface`

4.3 Changing the scene

As you can see in (fig. 8) there are two different scenes. The figure on the left represents the basic scene given by the UWSim launch file, while the one on the right is a modified scene. In fact, in the very first simulations, the robot was not able to reach the goal because it was located outside the swimming pool. For this reason, we used an open space terrain called `sea_floor_without_ship`.



(a) Basic scene given by UWSim (b) Modified scene

Figure 8: Basic and modified scenes

5 Results

The trained RL agent was tested in the UWSim interface for various combinations of β, β_{max} and *bound* before arriving at the ones mentioned in section 3.2. Wrongly tuning these parameters could be very disastrous as we learned that the RL agent may learn very wrong behaviour. To test the trained agent, we placed a way-point at (2,2,0) position (in red) with the start position of Girona500 AUV being (0,0,0). The intermediate trajectory of Girona500 is traced in red. The results can be categorized in various cases:

1. **Training does not converge -** For wrongly set parameters, the training of the RL agent did not converge. The agent was not able to learn a definitive behaviour which could produce a policy to maximise the reward. When tested, this resulted in an oscillating behaviour learned by the agent (fig. 9). The AUV would move towards the goal and then back towards the start position in a loop.

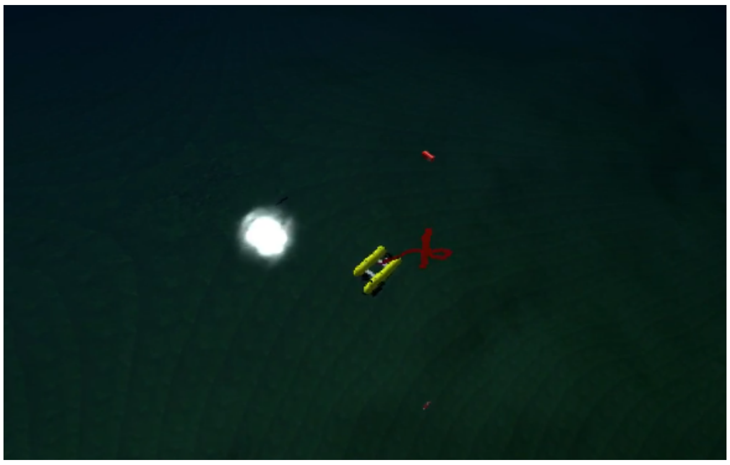


Figure 9: The behaviour learned by the agent for wrongly tuned parameters

2. **Training converges but calibration error during testing -** After multiple iterations, the β, β_{max} and *bound* parameters were tuned such that the training of the model started converging. The cumulative rewards for the AUV indicated that it was able to reach the goal in the training environment. However, when this trained agent was deployed for testing in UWSim, we observed that it moved in direction diametrically opposite to the goal (fig. 10). We suspected this was being caused by a mismatch in the axis orientation between the configuration assumed during the testing and training phases.

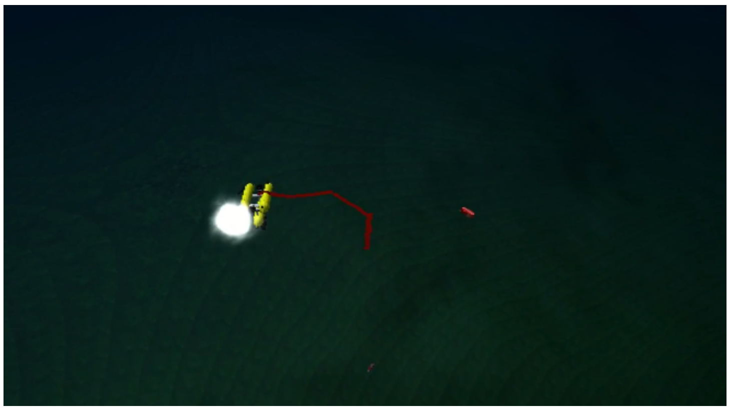


Figure 10: The behaviour learned by the agent for well tuned parameters, with a calibration error

3. **The correct scenario** - Post adjustment, the trained RL agent was demonstrating the ideal behaviour when being tested with UWSim. The AUV started from the initial position in direction of the goal and eventually reached it with an accuracy of 40cm or 0.4m (fig. 11). This is quite close to the trained behaviour, where the agent navigates to the way-point with an accuracy of 25cm (β was set to 25cm in the finale of the training phase).

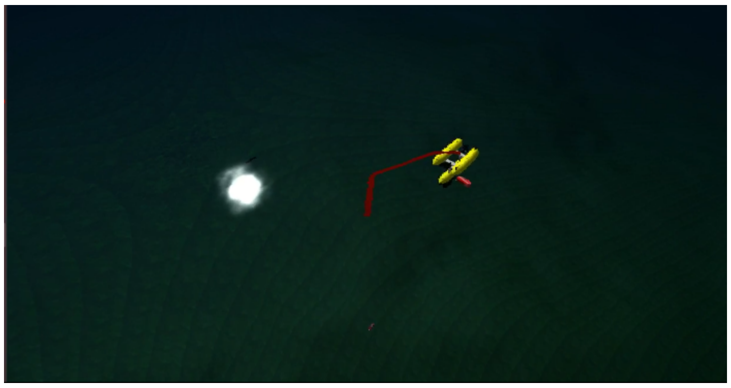


Figure 11: The correct behaviour learned by the agent for well tuned parameters

6 Conclusions

The goal of this project was the surveying and implementation of a RL strategy to control the motion of an AUV for navigation purpose. To achieve this, we trained a Deep Actor-Critic RL framework and tested it in a simulator called UWSim. The key challenges were encountered while implementing the training framework, interfacing ROS and *Stable Baselines* libraries, and tuning the training parameters that had to be set. The program was tested in three different settings: the first used poorly tuned parameters, so the behaviour was oscillating and unsatisfactory; the second used well tuned parameters and the agent's training converged, but the AUV went in the wrong direction; the third experiment was tested reorienting the axes, in this way the AUV followed the right path and reached the goal. Therefore, observing the results, we can conclude that the aim of the project has been achieved. A possible future work could involve the improvement of the navigation accuracy by training the agent on more capable hardware for more time-steps and stricter β, β_{max} and *bound* parameters.

References

[1] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, MIT Press, 1998. URL `http://www.cs.ualberta.ca/ sutton/book/the-book.html`

[2] R. S. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: Proceedings of the 12th International Conference on Neural Information Processing Systems, NIPS99, MIT Press, Cambridge, MA, USA, 1999, p. 10571063.

[3] A. El-Fakdi, M. Carreras, Policy gradient based reinforcement learning for real autonomous underwater cable tracking, in: 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008, pp. 3635–3640. doi:10.1109/IROS.2008.4650873.

[4] A. El-Fakdi, M. Carreras, N. Palomeras, P. Ridao, Autonomous underwater vehicle control using reinforcement learning policy search methods, in: Europe Oceans 2005, Vol. 2, 2005, pp. 793–798 Vol. 2. doi:10.1109/OCEANSE.2005.1513157.

[5] R. Yu, Z. Shi, C. Huang, T. Li, Q. Ma, Deep reinforcement learning based optimal trajectory tracking control of autonomous underwater vehicle, in: 2017 36th Chinese Control Conference (CCC), 2017, pp. 4958–4965. doi:10.23919/ChiCC.2017.8028138.

[6] *V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602.*
URL `https://arxiv.org/pdf/1312.5602.pdf`

[7] *V. Konda, J. N. Tsitsiklis, Actor-critic algorithms, Ph.D. thesis, USA, aAI0804543 (2002).*

[8] *I. Carlucho, M. De Paula, S. Wang, B. Menna, Y. Petillot, G. Acosta, Auv position tracking control using end-to-end deep reinforcement learning, 2018, pp. 1–8. doi:10.1109/OCEANS.2018.8604791.*

[9] *M. Prats, J. Prez, J. J. Fernndez, P. J. Sanz, An open source tool for simulation and supervision of underwater intervention missions, in: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2012, pp. 2577–2582. doi:10.1109/IROS.2012.6385788.*

[10] *V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning (2016). arXiv:1602.01783.*