Local Blind Planner

Lorenzo Jacqueroud, George Adaimi, Alexandre Alahi, *EPFL Switzerland*

***Abstract*—This project aims at implementing a local path follower and an obstacle avoidance method for a simulation using a deep neural network. The path follower is trained with a fully connected network and the obstacle avoidance is based on a reflex approach, implemented with a Temporal Convolutional Network (TCN). The first was tested on a fixed goal at each training session and still needs to be generalized to a random path at each episode, while the TCN was only tested on very simple tasks and will only be able to be fully trained once the path follower has been completed.**

I. Introduction

In the context of the European Rover Challenge, the Xplore association at EPFL is building a rover (Fig. 1) that can perform complex tasks such as explore its surrounding, navigate towards a waypoint and collect and deposit samples. The navigation team at Xplore is in charge of creating an autonomous system that is able to generate a map of its environment and create an optimal path to follow in order to reach a specific goal. To achieve this, various sensors are present on board. A lidar is positioned on top of the rover with which it is able to map the environment. An IMU incorporated directly with the lidar measures the linear and angular accelerations. Four potentiometers positioned on the four joints of the rocker-bogie suspension system [1] retrieve the angle positions of those joints. Finally encoders on each of the wheel's motor allow the wheel velocities to be determined. These are the relevant sensors for the navigation task, which consists in navigating from a starting position to a waypoint. With the Lidar, the rover constructs a map of the environment, and using a path planner algorithm, it generates an optimal path to reach the desired end point, by avoiding obstacles and steep terrain. The problem is that currently, the system is entirely relying on this method for the navigation, and therefore if imprecisions in the measurements are present, it could cause a lot of issues in the path planning and path following algorithms. Examples

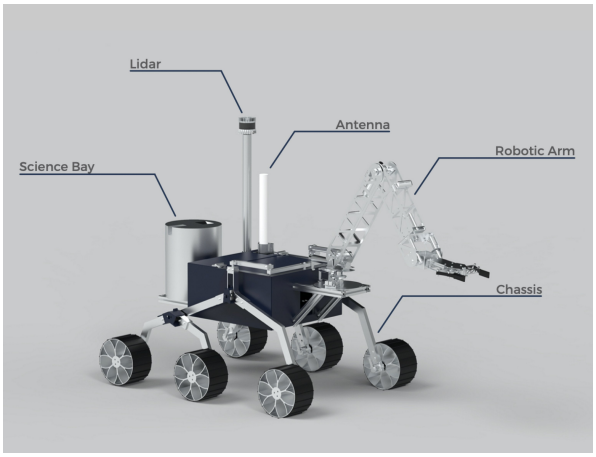

Figure 1: Rendering of the rover showing its different components

of such imprecisions could be either noise on the recorded data, or the Lidar simply recording incorrect data, which could be caused by dust or other disturbances. In addition, this method precisely finds big obstacles and avoids difficult terrain, but it could miss smaller obstacles (i.e. smaller rocks or holes) that the rover has a bit of trouble passing. Therefore the objective is to create a system that is able to reach the final destination by following a generated path, but being able to adapt to the situation in case the rover encounters a problem. In order to do that, reflexes are going to be added to the system, such that when it detects something using the IMU, potentiometers or wheel encoders, it knows how to react. Examples of this, could be that the command is to move forward, but if wheels are not turning, then the rover could be facing an obstacle that it cannot overcome, or if the rover is tilted, it might have fallen in a small hole.

A procedure for detecting problems in a system and solving it, is called Fault Detection Isolation and Recovery (FDIR). It consists in the definition of steps to follow in order to best manage those kind of situations. First comes fault detection, then isolation (isolating the malfunctioning system in

order to avoid extra damage, although this step is not always implemented as it's not necessary in every case) and finally the recovery. This method is often implemented in systems such as spacecrafts [2], other flying vehicles [3] or ground vehicles [4]. For the recovery step in particular, possible solutions consist in a set of predefined rules to follow in case of obstacle detection [5], using an arm to help with the recovery maneuver [6] or training an algorithm to react in the best way possible, for example with a genetic algorithm [7]. Another option is the use of a neural network and Lee et al. [8] were able to achieve very good results on a four legged robot with this approach. They implemented a reflex based approach, where they trained it to learn to step over obstacles in order to follow a general path, by generating a sequence of foot trajectories. The problem definition of the rover is very similar, indeed in both cases the robot has to learn to overcome an unexpected obstacle and continue to follow a path in the optimal way. Therefore the implementation of the algorithm is inspired by the paper from Lee et al. and is done with a neural network trained in simulation. Eventually the algorithm will also be transferred on the real rover. The next sections describe the technical details of the implementation: Section II covers in more details the simulation itself and the setup behind it, Section III describes the neural network architecture, in Section IV the gym environment is presented and Section V explains the algorithm used in the training loop. The procedure with the reasoning behind it and the results obtained are found in Section VI and finally Section VII expands on the next steps and what could be done as a follow up of this project.

II. Simulation

First let's take a look at how the simulation for this project is built. The simulation integrates three different softwares for the entire implementation: Gazebo, ROS (Robot Operating System) and OpenAI Gym.

A. Gazebo simulator

Gazebo is the simulator itself. It can generate a world, defined in a `world` file, with objects in it, such as robots which are defined in `urdf` files. A `urdf` file is used to create the rover, which is built on a structure of links and joints between each part and that determine how they can move with respect to one another. Fig. 2 shows an example of a simulation running, with the rover in a martian-like environment.

B. OpenAI Gym

OpenAI Gym is a toolkit specifically designed for developing reinforcement learning algorithms. The main component of Gym is the environment itself. It is a class which implements some basic possible operations, for example taking a step in the environment by selecting a certain action, or making an observation and obtaining the state and reward after that specific action has been performed. When creating a new environment, one has to define these operations, such as what the actions are and what each of them does. The framework makes it easy to also integrate a reward for each action and a reset of the entire environment to its initial state. It is therefore a very good tool for implementing a reinforcement learning algorithm, indeed an action can be selected from a defined policy and sent to the environment, which then returns the new state and the reward obtained, and those can be used to update the policy.

C. ROS

ROS is a framework for creating multi threading programs, where each script is capable of communicating with the others through a specific channel. The building blocks of ROS are nodes and topics: the nodes are scripts that run in parallel to each other, and these can publish or subscribe to topics. When a node publishes to a topic, it sends data to it, and any node that is subscribed to that same topic will be notified of the new data in the form of a callback function, and can then read what has been sent.
This architecture allows for a very modular and changeable structure. For example, in the case of the rover simulation, different nodes are responsible for generating a movement command, sending the command to each motor wheel, reading the lidar data, computing a path towards a goal point, etc..

D. Integration

The three components are integrated together as shown in Fig. 3. The main script corresponds

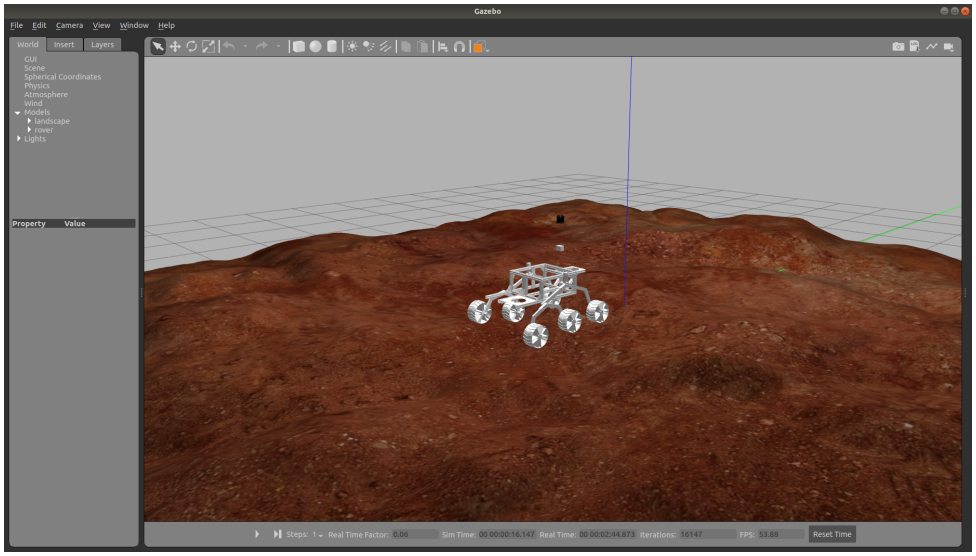

Figure 2: Example of the gazebo simulation running

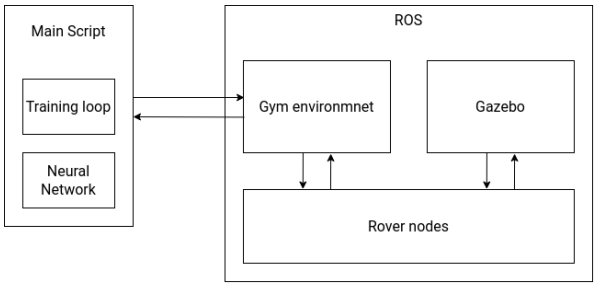

Figure 3: Diagram of the code architecture

to the training loop and is where the neural network is defined and trained. Its only point of communication with the simulation is through the gym environment. As explained above in Section II-B, the script is able to ask the environment to take a step by giving it an action, and the environment returns state and reward which are used to compute the loss and update the network's weights.

The gym environment is itself a ROS node, and it can subscribe and publish on topics related to the rover. For example, it needs data from the IMU, from the joint angles of the rover or the output of the path planner, in addition to having to send a movement command or setting a goal point. Inside the environment, the state is saved at each step, which will be used as input to the neural network (Section IV-C).

Gazebo corresponds to a node as well, and similarly to the gym environment, it communicates with many other nodes responsible for the rover simulation. The gazebo node is in charge of the physics of the simulation and it's from here that all the data is generated, such as the rover position, the joint positions or the elevation data of the map. All this information is sent to different topics which are then accessed by other nodes.

Other than the main script, everything is implemented with the ROS framework. Fig. 4 gives a more in depth view of the entire graph containing all the nodes (ovals) and topics (rectangles). Highlighted in red are the gazebo and gym nodes, in light blue the part related to the rover state (position, joints, IMU data), in orange the part for the elevation and map building, in green the path planner and in blue the nodes responsible for the movement of the rover.

Some parts of the simulation were already existing when the project was started, namely the gazebo simulator and parts of the rover nodes. What was added is the entirety of the gym environment, the main script for the network training and the nodes and topics used by the gym environment to receive and send the needed information from and to the simulation.

III. Neural Network

A. *The inspiration*

As mentioned previously (Section I) the network architecture is inspired by the one used by Lee et al. [8]. In their paper they are training a

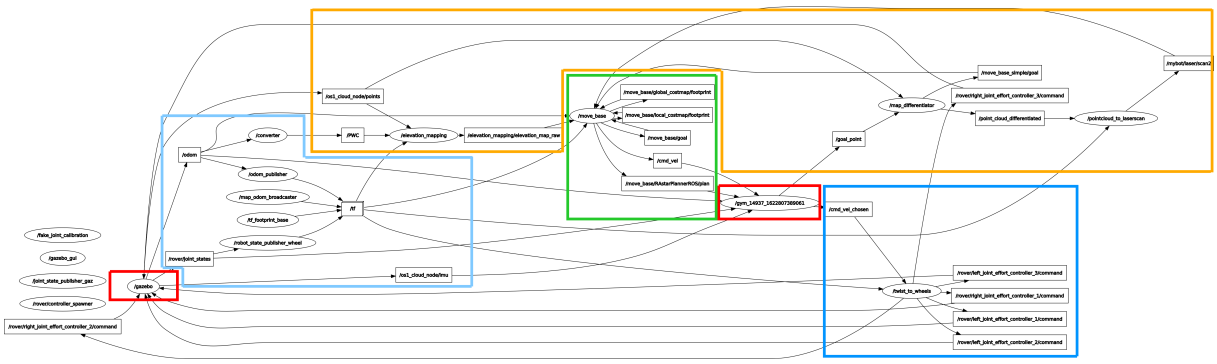

Figure 4: Graph of the ROS nodes

quadruped robot to follow a trajectory and react to obstacles that come up in the way. To do this, they have a network with a teacher-student training, where in simulation, the teacher learns with access to a lot of information which will actually not be available on the real robot, for example the terrain profile, contact forces with the ground or friction coefficients. Then, part of the network is transferred to a student, which learns without all the privileged information that the teacher had and tries to replicate its actions.

The architectures of both network are specified in Fig. 5, where *tanh(x)* means a fully connected layer with x outputs and a *tanh* activation function, *1D conv* is a convolution in one dimension, with a kernel of size 5 and dilation and stride of 1 if not otherwise specified and followed by a *ReLU* activation function. The id column signifies that each corresponding layer is duplicated with the same weights four times, one for each leg. Therefore at layer 3 for the teacher and layer 8 for the student, the concatenation happens between the four outputs of the previous part of the network.

The particularity of this network is that it uses a TCN or Temporal Convolutional Network [9]. A TCN takes as input a series of data with N elements and C channels where N is the number of time steps to take into consideration and C is the number of parameters. In the implemented network, the channel size is kept identical throughout all the convolutional layers. Therefore for each parameter there are N elements and the convolution in one dimension is performed along the time dimension, reducing it, while keeping the channel size the same. This means that if the input array is of size NxC, after a convolution of size k, the output will be an $(N-k+1)$xC array. This TCN approach is very useful when the decisions are to be taken based on what happened in the last few time steps, since the convolutions allow to extract the useful patterns and information from each parameter and the fully connected layers combine it in order to output a decision. This is why it is very well suited for both the Anymal's quadruped robot and the rover, since both have to use the information from various internal sensors in order to understand what is happening and react in some way.

Layer	Teacher		TCN-N Student	
input	o_t	x_t	o_t	h (60×N)
1	id	tanh(72)	id	1D conv dilation 1
2	id	tanh(64)	id	1D conv stride 2
3	concatenate		id	1D conv dilation 2
4	tanh(256)*		id	1D conv stride 2
5	tanh(128)*		id	1D conv dilation 4
6	tanh(64)*		id	1D conv stride 2
7	Output*		id	tanh(64)
8	-		concatenate	
9	-		tanh(256)*	
10	-		tanh(128)*	
11	-		tanh(64)*	
12	-		Output*	

Figure 5: Architecture of the neural network used by Lee et al.

B. Temporal Convolutional Network

The implemented TCN doesn't use a teacher-student policy, but instead only a standard policy without additional training, as can be seen from its architecture in Fig. 6. The input size is taken to be N_params x N_steps, where $N\_params = 39$ is the number of parameters used (see Section IV-C for more details) and $N\_steps = 30$ is the number of time steps taken for each parameter.
The first part of the network is composed of 6 temporal convolutional layers with a kernel of size 5. With this layout, the output size of the convolutional layers is $(N\_steps - 24)$ x N_params which in this specific case corresponds to a final time dimension of size 6[1]. Because of the already small output size, no dilation or stride were introduced, unlike in the example of Lee et al. The code however is very modular and this can easily be changed if that were needed.
After the convolutions, the output is concatenated in a one dimensional vector of size $N\_fc\_input = (N\_steps - 24) \cdot N\_params$ and after 4 fully connected layers, the size is reduced to $N\_actions$ which corresponds to the number of possible actions that the rover can take.
The activation functions vary depending on the layer: for the convolutions, a *ReLU* activation is used, while for the fully connected ones, the activation is a *tanh* function. For the final layer, a *softmax* is added, so that the output is an array of probabilities for each action. In addition for future training, the possibility to add batch normalization or dropout after each layer is already implemented.
Regarding the output actions, two possibilities exist: either having a predefined set of actions from which to choose from, or giving directly the action, for example in the form of a linear and an angular velocity. The first option was chosen in the presented networks as a starting point, but given time the second one could also be implemented and a comparison of the results could be done in order to find the optimal choice.

C. Addition of local path follower

Unfortunately, due to some misinformation among the team members, the local path planner that was implemented the semester before was not working properly, and therefore the previously presented neural network wasn't able to learn anything (more details in Section VI). Following that, a local path follower was added directly inside the network. This was done as shown in Fig. 7, where after the convolutional layers, the output was directly concatenated with another vector. This added part contains the information about the rover position and orientation, and the coordinates of points along a generated path. The idea is to give all the information necessary in order to be able to follow a path and in addition complete the original task of reacting to unexpected obstacles in the way, given an appropriate reward function (Section IV-D). In order to test if a path follower could be achieved using a fully connected network, a simpler version of the network was created, presented in the next section.

[1]The reduction on the time dimension is $N\_steps - 6 \cdot (k-1) = 6$

D. Network specifically for local path follower

Fig. 8 shows the architecture of the network used for training the path follower. Since for this task there is no need for the data from internal sensors, only the input from the rover position/orientation and the path are kept. The network then becomes a simple fully connected network with 6 layers and an output of the same dimension as previously, which gives a probability for each of the possible actions.

IV. Gym Environment

The gym environment contains all the information needed to generate and control the environment. It is here that the actions, the step function and the reset function are defined.

A. Step function

First let's look at the `step` function, which corresponds to the basic algorithm that controls the simulation:

- First it unpauses the simulation
- Then it performs the given action by sending the command to a ROS topic, to which another node responsible for giving this command to the motors, is subscribed to
- Wait for a predefined amount of time, corresponding to a time step
- Pauses back the simulation

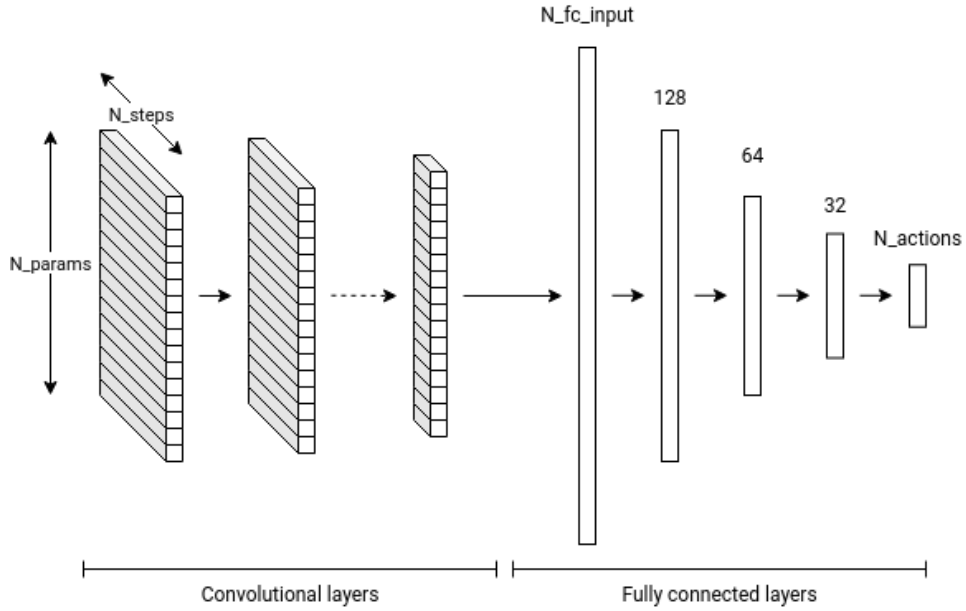

Figure 6: Architecture of the neural network used for obstacle detection and avoidance

- Makes an observation and gets the state of the system
- Determines if the system reached an end state
- Computes the reward based on the previous and current state
- Returns the state information, the reward and a `done` boolean to indicate if the episode has ended

B. Actions

The possible actions that the rover can take are very basic in order to keep the learning that the network has to do the simplest as possible. It can go forward, go backwards, turn left or turn right in place, all at a constant speed. This can however be changed by just adding more predefined commands in the `action_commands` array.

C. State

The definition of the state is a bit more complex. It is divided into two parts, which correspond to the two different inputs to the neural network in Fig. 7. The first part is the *sensor state* and it is composed of:

- The IMU data which includes 10 values: the orientation of the IMU given as a quaternion, the linear acceleration in the x,y,z axis and the angular acceleration on the same axis
- The joint state which includes 29 values:
  - The wheel data. For each of the six wheels there is the position, the velocity and the effort of the motor ($3 \cdot 6 = 18$ values). The position and the velocity will be available from the encoders on the motors and the effort from the torque.
  - The joint data. For the four joints on the rocker-bogie mechanism (two per side) there are the position and the velocity ($4 \cdot 2 = 8$ values). On the real rover, there will be potentiometers at each joint, which will measure the joints angles. In addition because of the way the simulation is built and because of some constraints of Gazebo, the effort is only available for one of those joints (1 value). Finally in Gazebo there also exists a *base joint* which corresponds to the chassis, for which position and velocity are available (2 values).

For each of those values in the *sensor state*, the last N_{steps} are kept in memory, as the Temporal Convolutional Network performs convolutions along the time dimension (see Section III).
The second part of the state is the *path state* and it contains the information needed for the local path follower:

- The position and orientation of the rover: the x,y coordinates and the yaw orientation of

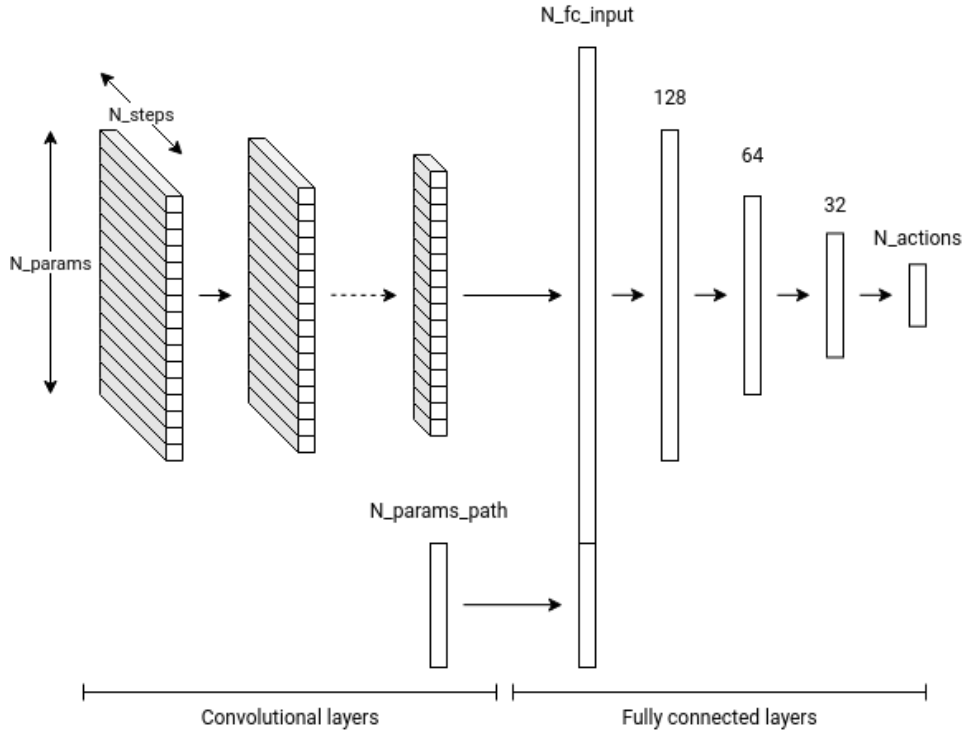

Figure 7: Architecture of the neural network used for obstacle detection and avoidance and local path follower

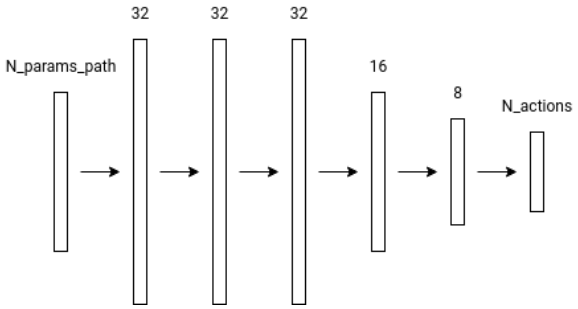

Figure 8: Architecture of the neural network used for the local path follower

the rover (3 values)

- The path data, which contains 20 values: the x,y coordinates for the first 10 points along the path are used. Since the path is recomputed at regular intervals starting from the current rover position, only the first few points along the path are needed for a path follower algorithm.

Normally everything used in the state is also available on the real rover, but the implementation is very modular, and could be easily modified if something were to change.

D. Reward function

The reward function is also defined in the gym environment. The one used for the rover is inspired again from the paper of Lee et al. [8]. Their reward function is as follows:

- reward the velocity following the path (by taking the projection)
- penalize the orthogonal velocity to the path,
- reward the turning velocity in order to turn faster
- penalize roll and pitch of the body
- penalize collisions
- reward foot clearance
- reward trajectory smoothness
- penalize joint torques

Some of those are specific to their case, such as foot clearance or roll/pitch of the body, but most of them are found in the literature [10, 11, 12, 13]. The final choice of reward function for the path follower network is:

$$R = k_{vp} \cdot r_{vp} + k_{vo} \cdot r_{vo} + k_r \cdot r_r$$

where:

- r_{vp} is velocity parallel to the path, computed using the projection of the rover movement on the closest points on the path
- r_{vo} is the velocity orthogonal to the path, computed similarly to r_{vp}, but taking the orthogonal component instead
- r_r is the yaw rotation to align itself with the path. This value will be positive when rotating in order to align itself with the path, and negative when rotating in the other direction

The values for the gains are $k_{vp} = 1$, $k_{vo} = -1$ and $k_r = 5$. The yaw rotation weight was found to be optimal for the learning process, and it makes sense to have a bigger impact for the rotation as the rover should first align itself with the path, and only then start going forward.

When implementing the network for obstacle avoidance, the reward function should change a bit. An idea would be to introduce the penalization for roll/pitch of the body or a penalization when the movement of the rover doesn't match the given command. This would detect cases when for example the rover encounters a rock that it cannot overcome, or when a wheel falls in a hole, and it would ideally learn to react as best as possible to those situations.

V. Training loop

The last step missing for the implementation is the training loop. The algorithm used is based on the *reinforce* method also known as *Monte Carlo policy gradient*. It is a reinforcement learning technique where the basic idea is to generate a trajectory corresponding to a sequence of actions, compute the rewards for that trajectory and learn from it. It belongs to the policy-based methods, and in particular it is a policy-gradient method, meaning that it tries to optimize directly the policy using gradient ascent. The steps of the algorithm are:

1) Initialize the policy (in this case the neural network weights)
2) Generate a trajectory
   a) Choose an action, based on the output of the neural network
   b) Perform the action using the `step` function of the environment
   c) Repeat until the episode ends
3) Compute the reward at each step of the trajectory
4) Compute the loss
5) Update the policy (backward propagation in the neural network)
6) Repeat from 2)

For the computation of the rewards, the standard approach is to compute the expected reward for an action, by adding up all the subsequent rewards obtained in the trajectory. In order to not gain infinite rewards and to account for short term gain, after each step the rewards are multiplied by a factor γ (usually $\gamma = [0.9; 0.99]$, therefore the reward at step k is

$$R_k = r_k + \gamma \cdot r_{k+1} + \gamma^2 \cdot r_{k+2} + ...$$

In the implementation though, after some testing, it was observed that actually another simpler approach works better: by using only the reward for the current step the algorithm converges to a good solution faster, which corresponds to setting $\gamma = 0$. The reward at step k therefore becomes $R_k = r_k$.

The first implemented loss function is the gradient of the logarithm of the probabilities for the action chosen, multiplied by the reward and summed over the entire trajectory:

$$L_1 = \sum_{k=0}^{N} \nabla_\theta log\left(\pi_\theta\left(a_k|s_k\right)\right) \cdot G_k$$

where $\pi_\theta\left(a_k|s_k\right)$ is the policy for action a_k in state s_k, and G_k is the normalized expected reward at step k. With this loss, the optimization happens on the log of the probabilities, and the reasoning behind is that it is generally more stable and scaled correctly, since the range of the probabilities is only $[0; 1]$, while for the log of the probabilities it's $]-\infty; 0]$.

A slightly more advanced loss function was implemented in order to reward more diverse actions and it includes a term for an entropy loss:

$$L_2 = \sum_{k=0}^{N} k_{policy} \cdot \nabla_\theta log\left(\pi_\theta\left(a_k|s_k\right)\right) \cdot G_k + k_{entropy} \cdot \pi_\theta\left(a_k|s_k\right) \cdot log\left(\pi_\theta\left(a_k|s_k\right)\right)$$

The second term corresponds to the entropy loss and it favors more exploration by rewarding probabilities which are more spread out (i.e. that have a higher entropy), and therefore penalizes a very high probability (close to 1) for one of the actions, and nearly 0 for all the others. k_{policy}

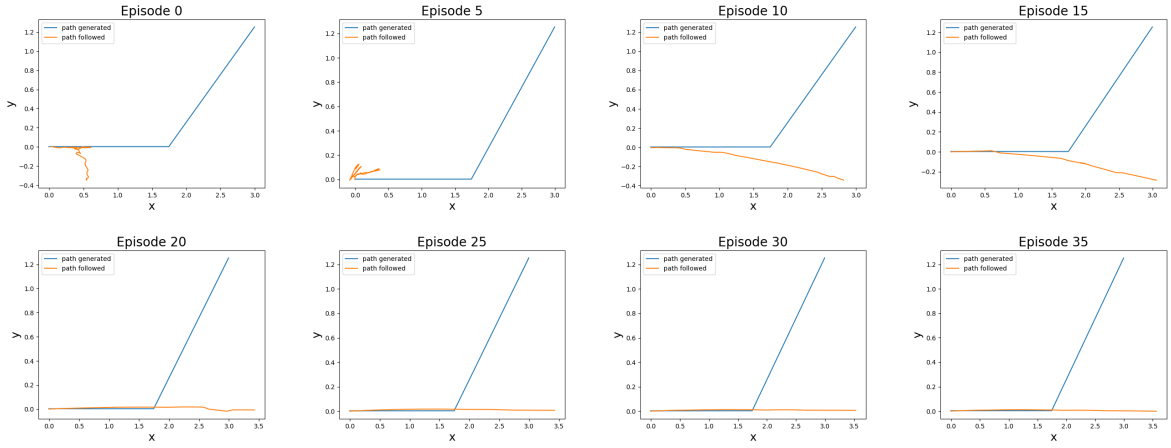

Figure 9: Trajectories of the episodes during the first version training. In blue the path generated by the global path planner and in orange the path followed by the rover. The rover always starts each episode at $x = 0$, $y = 0$ and orientation towards the positive x direction.

and $k_{entropy}$ determine the impact that both the policy and entropy loss have on the total loss. In the final version, the values $k_{policy} = 1$ and $k_{entropy} = 0.1 \cdot 0.9^n$ were found to work well for the presented results (Section VI), where n is the episode number and therefore the entropy has a diminishing impact in later stages of the training.

VI. Procedure and Results

In the previous sections, the technical details and methods used are presented. This section now covers the work procedure that has been followed and the motivation behind the choices made, in addition to showing the results obtained.

The first version of the network is shown in Fig. 6, and presented in details in Section III-B. This architecture was tested on simple tasks, such as learning to go forward or turning in place, by creating an appropriate reward function, which simply gives a reward for a specific action. The network unsurprisingly was able to learn those tasks quickly, after 2-3 episodes.

After the above mentioned network was implemented, unfortunately a problem appeared: the local path follower was not working properly. At the start of this project, the other team members had stated that the complete path planner had been implemented correctly the prior semester and was now working well, but due to a misunderstanding that wasn't the case. This posed a problem for this project, since the output of the network is a set of predefined actions, where one of them is to follow what the path planner gives as a command. The idea behind this would be that under normal conditions, the network would learn to use the path planner command to reach the goal, but when it encountered an obstacle, it would instead give other commands.

As the global path planner is working correctly (a correct path is generated) and the issue comes exclusively from the path follower, the solution to train a path follower directly within the network was tested. For this task, the network shown in Fig. 7 was created, by giving as input to the fully connected layers all the information needed for following the path (i.e. the path itself and the rover position and orientation). In order to assess in the first place, if a fully connected network would be able to follow the path, a simpler version of the network, which removed all the unnecessary parameters, was implemented (Section III-D). To note that the following presented results are obtained by giving a constant goal during a training session, therefore the network was learning to follow a specific path each time. Also as the rover in simulation has sometimes difficulties to overcome hills and holes due to slipping which doesn't happen in real life, a flat world was used as a starting point.

After a first tuning of the network parameters, the rover was able to follow the first segment of the path consistently, but would get stuck in selecting a single action (Fig. 9). For example if the path started straight, the rover would learn to go straight,

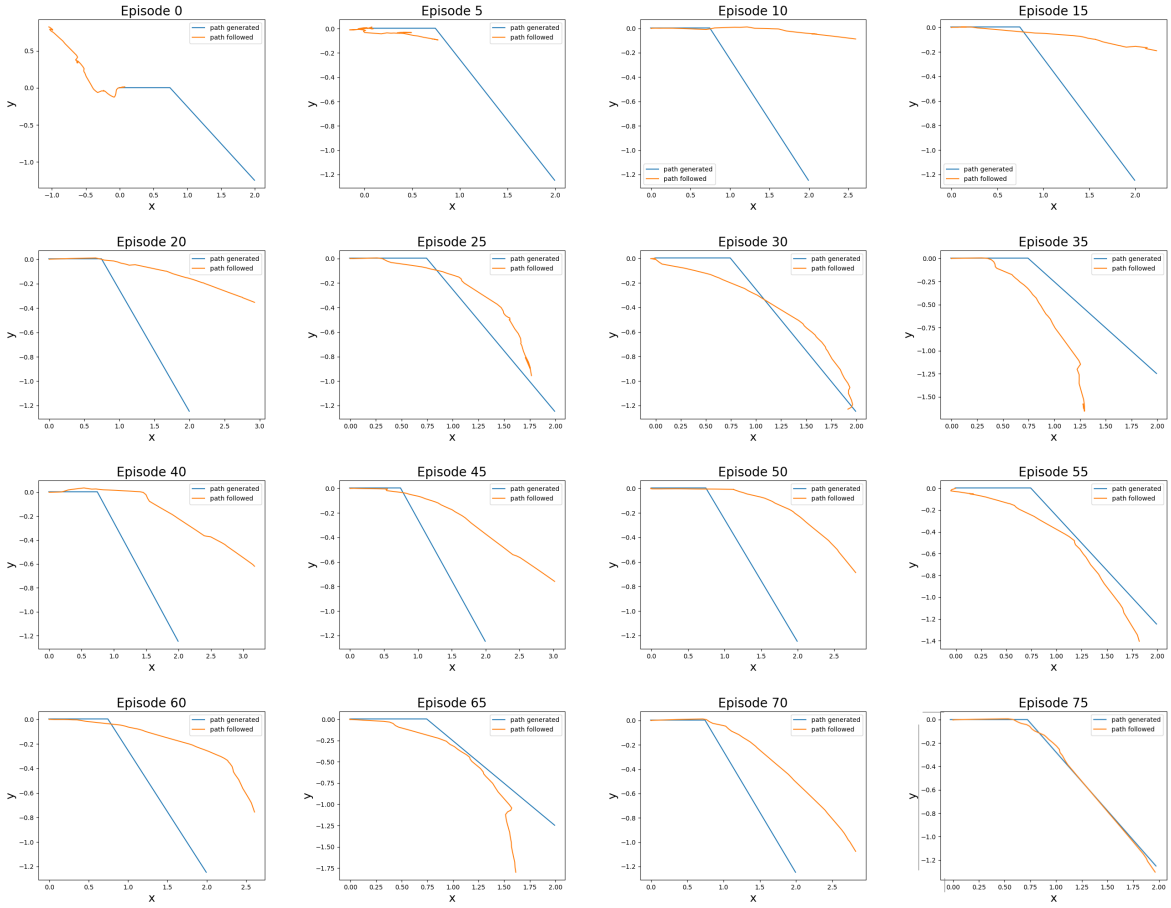

Figure 10: Trajectories of the episodes during the final version training.

but keep going forever, or if the path started by a turn, it would just learn to turn in place.

After this first test, a new loss including an entropy component was introduced (see Section V). This allowed for more focus on exploration of different actions and as can be seen in Fig. 10, it improved the path following.

Even though the rover is able to reach the goal consistently in the shown result after episode 75, the training sometimes fails. Indeed the network does not always converge to a good solution and can still reproduce the same cases as before (Fig. 9) where only one action ends up being executed. This is likely due to the tuning of the parameters for the weights on the loss (Section V), but the ones presented were able to achieve the most consistency. Due to the limited power of the computer used for the training (a laptop without a graphic card), only small trajectories were tested, where the path had only 1 or 2 turns in it. Although a longer path would have been interesting to test on, this should not pose a problem, as this can be solved in a different way. Indeed since the rover travels along the path, the goal point can be constantly changed to have roughly the same distance from the rover, and therefore simulate a shorter "moving" path at each time step.

VII. Discussion and Future Work

In this project a lot of time was spent on the creation of the framework using OpenAI Gym, in order to train a reinforcement learning algorithm. Some results were also achieved, where a path follower was implemented successfully for simple cases and the rover was able to learn to follow a fixed path in simulation by training a neural network. In addition, a Temporal Convolutional Network was fully implemented for the task of learning to react to unexpected obstacles.

As this project is not finished, it will be continued the following semester by another student in order to achieve what is missing. Regarding the path

follower, the next step would be to test the fully connected network (Section III-D) on a general path, by giving a random goal at each episode. In order to do that however, access to a more powerful machine is needed. Even though that was the case for this project, due to an unresolved issue, the simulation wasn't able to run on the more powerful computer provided by the Vita laboratory, which was equipped with a graphic card.
As the results achieved so far with the presented path follower neural network were not able to be reproduced consistently (Section VI), another possibility would be to not use a deep learning approach for that task, but instead go back to implementing a path follower using a ROS package. There are existing packages for path planning and following available on internet, but building one from scratch that suits this project better could also be considered.
After that, the main network (Section III-B) should be tested on a map first without obstacle to check that it learns to follow the path, and then on a map with obstacles, to learn to react to unexpected problems.
Here are also some other interesting modifications that could be explored in order to achieve better performance with the presented networks:

- Add more choices of action, i.e. turning while moving forward, as currently the rover can only turn in place.
- Instead of having a predefined set of actions, another possibility would be for the network to output two values corresponding to a linear and an angular velocity.
- A better training algorithm could be considered. The Monte Carlo Policy Gradient method used (Section V) has the advantage of being easier to implement, but it sometimes performs poorly in longer episodes or more complicated cases. Therefore other methods might be better suited for this project.

VIII. Supplementary Material

The entirety of the code can be found on Github:

- Gym repository: contains the main script and all the environment files. A quick guide on how the code works can be found in the repository description. The most important files are:
  - `/examples/rover/rover_main.py`: main script containing the training loop
  - `/examples/rover/dnn.py`: file containing the definition of the neural networks
  - `/gym-gazebo/envs/rover/rover_gazebo_env.py`: file containing the environment definition
- Catkin workspace: this is the workspace with the catkin packages for running the rover simulation. Note that this contains a copy of the repository NAV workspace, which is the complete version of the simulation, but without all the packages that are not needed for this project.

REFERENCES

[1] Dhananjay Chinchkar et al. 'Design of Rocker Bogie Mechanism'. In: *International Advanced Research Journal in Science, Engineering and Technology* 4.1 (2017), pp. 46–50.

[2] Alexandra Wander and Roger Förstner. *Innovative fault detection, isolation and recovery strategies on-board spacecraft: state of the art and research challenges*. Deutsche Gesellschaft für Luft-und Raumfahrt-Lilienthal-Oberth eV, 2013.

[3] Mario Valenti et al. 'Indoor Multi-Vehicle Flight Testbed for Fault Detection, Isolation, and Recovery'. In: *AIAA Guidance, Navigation, and Control Conference and Exhibit*. DOI: 10.2514/6.2006-6200. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2006-6200. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2006-6200.

[4] Murray L Ireland et al. 'Inverse Simulation as a Tool for Fault Detection and Isolation in Planetary Rovers'. In: (2017).

[5] Deborah Braid, Alberto Broggi and Gary Schmiedel. 'The TerraMax autonomous vehicle'. In: *Journal of Field Robotics* 23.9 (2006), pp. 693–708. DOI: https://doi.org/10.1002/rob.20140. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.20140. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.20140.

[6] T. Estier et al. 'Shrimp, a Rover Architecture for Long Range Martian Mission'. In: (May 2021).

[7] Fumito Uwano et al. 'Recovery System Based on Exploration-Biased Genetic Algorithm for Stuck Rover in Planetary Exploration'. In: *Journal of Robotics and Mechatronics* 29 (Oct. 2017), pp. 877–886. DOI: 10.20965/jrm.2017.p0877.

[8] Joonho Lee et al. 'Learning quadrupedal locomotion over challenging terrain'. In: *Science Robotics* 5.47 (Oct. 2020), eabc5986. ISSN: 2470-9476. DOI: 10.1126/scirobotics.abc5986. URL: http://dx.doi.org/10.1126/scirobotics.abc5986.

[9] Shaojie Bai, J. Zico Kolter and Vladlen Koltun. *An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling*. 2018. arXiv: 1803.01271 [cs.LG].

[10] Peng Wang et al. 'Research on Dynamic Path Planning of Wheeled Robot Based on Deep Reinforcement Learning on the Slope Ground'. In: *J. Robot.* 2020 (Jan. 2020). ISSN: 1687-9600. DOI: 10.1155/2020/7167243. URL: https://doi.org/10.1155/2020/7167243.

[11] Xi Chen et al. *Deep Reinforcement Learning to Acquire Navigation Skills for Wheel-Legged Robots in Complex Environments*. 2018. arXiv: 1804.10500 [cs.RO].

[12] Tamir Blum and Kazuya Yoshida. *PPMC RL Training Algorithm: Rough Terrain Intelligent Robots through Reinforcement Learning*. 2020. arXiv: 2003.02655 [cs.RO].

[13] Kaichena Zhang et al. 'Robot Navigation of Environments with Unknown Rough Terrain Using deep Reinforcement Learning'. In: *2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. 2018, pp. 1–7. DOI: 10.1109/SSRR.2018.8468643.