Learning Relative Interactions through Imitation

Università della Svizzera Italiana
Faculty of Informatics
Lugano, Switzerland
Project for the Robotics course 2019–2020

Giorgia Adorni
(giorgia.adorni@usi.ch)

Elia Cereda
(elia.cereda@usi.ch)

***Abstract*—In this project we trained a neural network to perform specific interactions between a robot and objects in the environment, through imitation learning. In particular, we tackle the task of moving the robot to a fixed pose with respect to a certain object and later extend our method to handle any arbitrary pose around this object.**

We show that a simple network, with relatively little training data, is able to reach very good performance on the fixed-pose task, while more work is needed to perform the arbitrary-pose task satisfactorily. We also explore the effect of ambiguities in the sensor readings, in particular caused by symmetries in the target object, on the behaviour of the learned controller.

***External Resources*—source code [1], pitch presentation [2] and final presentation [3].**

I. Introduction

In robotics, some tasks are relatively easy to perform with complete knowledge of the environment, but become more challenging when the environment is only partially observable using a robot's sensors. Imitation learning deals with this problem by recording the trajectories of an omniscient controller performing the desired task, then training a machine learning model to replicate them using just the data from the sensors.

As such, the machine learning model must learn how to extract the relevant information from the data it receives, sidestepping the difficulty of implementing the perception part manually.

The target platform we choose for our project is the marXbot [4], a research robot originally designed to study collective and swarm robotics. The main characteristic making the marXbot interesting for this project is its rotating laser scanner, which perceives distances and colours of the objects surrounding the robot.

The experiments are run in Enki [5] [6] a high-performance open-source simulator for planar robots, which provides collision detection and limited physics support for robots evolving on a flat surface. Moreover, it can simulate groups of robots hundreds of times faster than real-time.

We explore the symbolic task of moving the robot to a specific pose with respect a certain object. While easy to accomplish when the exact poses of the robot and the goal object are known, it becomes non-trivial when only sensor data is available. In particular, the model must learn to extract characteristic features of the object surface that allow it to determine its pose. Furthermore, in our case the task is complicated by inherent ambiguities due to the symmetry of the chosen object. Sections II and III explore these aspects, how we dealt with them and the results we obtained from the experiments.

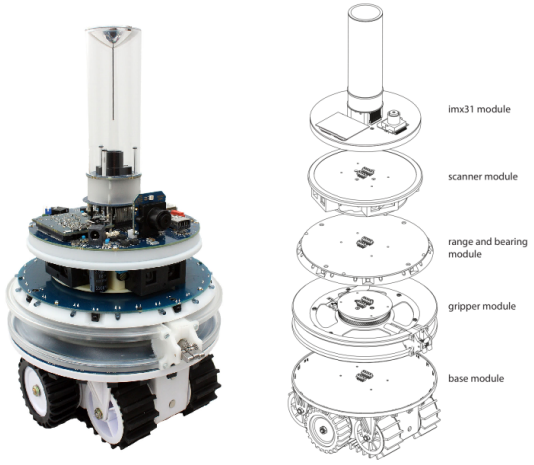

Fig. 1. Actual image and exploded CAD view of a marXbot.

In Section IV we instead attempt to generalise our model, so that the goal pose that it should reach can be arbitrary and provided as input.

II. Controllers

In an imitation learning setting, there are two controllers involved: an *omniscient* controller, which performs the desired task with perfect knowledge of the environment, and a *learned* controller, which is trained to imitate the behaviour of the omniscient controller.

A. Omniscient controller

We implemented a controller from the literature that simultaneously controls position and orientation of the robot toward a certain goal pose [7].

We chose this particular controller because it promised smooth and intuitive trajectories, which globally converge to

arbitrary goal poses without singularities, from any initial pose. Furthermore, this specific formulation makes it easy to impose limits on the velocity, acceleration and jerk of the resulting paths, ensuring that they would be physically realizable if executed on a real robot.

The control law is described in an egocentric polar coordinate system, relative to the current pose of the robot. The control variables are the linear v and angular ω velocities. Given a target pose T, the state of the robot is expressed as the triple (r, θ, δ), where r is the Euclidean distance from the target position; $\theta \in (-\pi, \pi]$ the target orientation with respect to the line of sight from the robot to the target position; $\delta \in (-\pi, \pi]$ the vehicle orientation with respect to the line of sight, as shown in Figure 2.

It can be seen that (r, θ) completely identify the position of the robot, while δ identifies its orientation. In this formulation, moving the robot to the target pose corresponds to bringing the state to the origin, $(r, \theta, \delta) = (0, 0, 0)$.

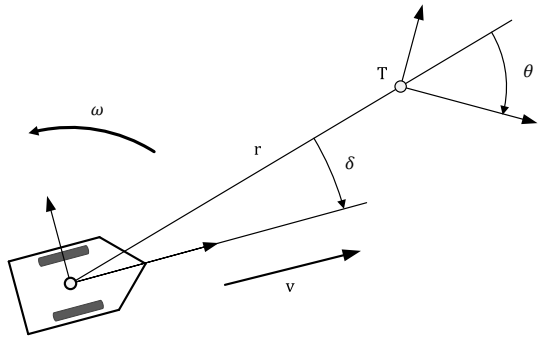

Fig. 2. Egocentric polar coordinate system, relative to the current pose of the robot.

Assuming initially the linear velocity v is nonzero positive and given (although not constant), the authors show that the angular velocity ω only influences δ directly, which in turn influences (r, θ). As such, the control problem is decomposed in a slow and a fast subsystem.

The slow subsystem first computes a reference orientation $\hat{\delta}$ to steer the robot toward the origin:

$$\hat{\delta} = \arctan(-k_1 \theta) \tag{1}$$

The fast subsystem then controls the angular velocity ω to bring the current orientation δ toward the reference orientation $\hat{\delta}$ computed by the slow subsystem (somewhat confusingly, the original paper uses δ for both the current and reference orientations):

$$\omega = -\frac{v}{r}[k_2(\delta - \underbrace{\arctan(-k_1\theta)}_{\hat{\delta}}) + (1 + \underbrace{\frac{k_1}{1 + (k_1\theta)^2}}_{-\dot{\hat{\delta}}})\sin\theta] \tag{2}$$

It can be seen from (2) that there is a linear relation between v and ω. In particular, $\omega = \kappa(r, \theta, \delta)\, v$ where κ is the curvature of the resulting path. It is possible to rewrite (2), such that

$$\kappa = -\frac{1}{r}[k_2(\delta - \hat{\delta}) + (1 - \dot{\hat{\delta}})\sin\theta] \tag{3}$$

which implies that the shape of the path does not depend on the choice of v. To ensure a smooth and comfortable trajectory, the authors suggest to choose v so that $v \to 0$ as $\kappa \to \pm\infty$ and $v \to v_{\max}$ as $\kappa \to 0$:

$$v = \frac{v_{\max}}{1 + \beta|\kappa(r, \theta, \delta)|^{\lambda}} \tag{4}$$

As written, the control law has a singularity as $r \to 0$, in other words when the robot approaches the target. We address this problem as suggested in the original paper, by setting $v = k_3 r$ in the neighbourhood of $r = 0$:

$$v' = \min(v, k_3 r) \tag{5}$$

In the equations above $k_1 > 0$, $k_2 > 0$, $k_3 > 0$, $\beta > 0$ and $\lambda > 1$ are design parameters. Figure 3 shows some trajectories generated by this controller, obtained with parameters $k_1 = 1$, $k_2 = 3$, $k_3 = 2$, $\beta = 0.4$ and $\lambda = 2$.

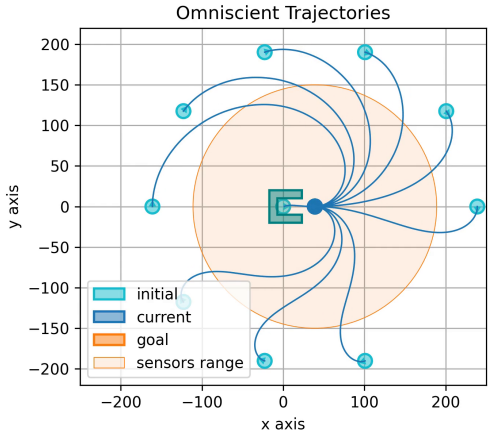

Fig. 3. Trajectories of the omniscient controller from 9 initial poses.

Finally, we implemented support for reverse gear as suggested by the authors: by simultaneously flipping the orientation of both the robot and the target when v is negative.

We use the reverse gear to teach the learned controller how to behave when it overshoots the target. This normally never happened with the omniscient controller, so the neural network wouldn't know what to do if it didn't stop precisely over the goal. We used an augmentation technique to ensure that this situation would be present in the training set: we made the robot move in reverse if its initial position was inside a region in front of the goal (i.e. between the arms of the object in Figure 3).

B. Learned controller

Our learned controller is implemented as an end-to-end neural network, which receives the sensor inputs and produces commands for the motors. Section III-B will describe the specific architectures that we used.

III. Task 1

A. Data Generation

Relying on Enki and on the omniscient controller, a dataset of 2000 simulation runs is generated. Each run sets up a world containing:

- a *horseshoe-shaped object*, that represents a hypothetical docking station, always at pose ($x = 0$, $y = 0$, $\theta = 0$);
- a *fixed goal pose*, in front of the two arms of the object;
- a *marXbot*, at a random uniform position and orientation around the goal, up to a maximum distance of 2 m — slightly more that the range of the distance sensors.

A simulation run is stopped either 1 s (10 time steps) after the robot reaches the target pose, or after 20 s (200 time steps). This ensures enough timesteps in which the robot is standing still at the goal, improving the performance of the network. The marXbot is considered at target if the Euclidean distance from the goal is less than 1 mm and the robot orientation is less than 0.5 deg from the goal orientation.

The resulting distribution of positions at the beginning and end of each run are shown in Figure 4.

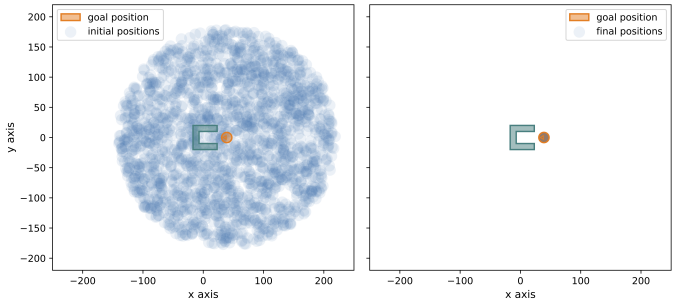

Fig. 4. Initial and final positions.

The density of samples in each position can be seen in Figure 5 while the histogram of the time needed to reach the goal is shown in Figure 6. In this case, all the runs terminate at the goal.

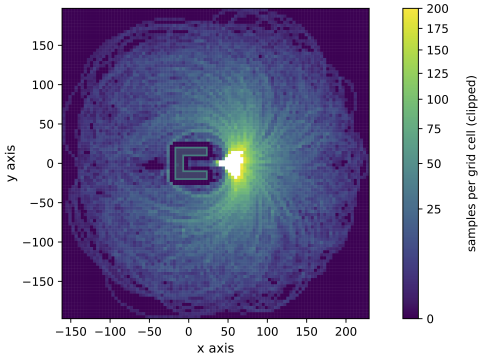

Fig. 5. Positions heat map.

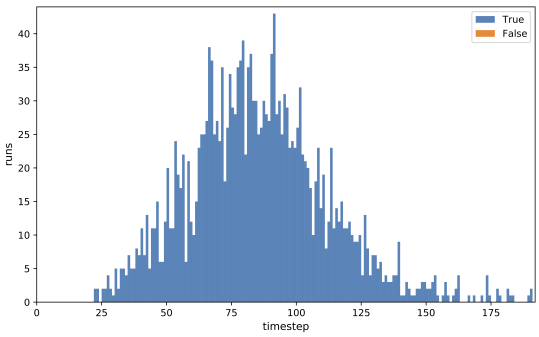

Fig. 6. Time to reach the goal.

Plotting the position and orientation errors over time shows the convergence of the omniscient controller. In Figure 7 are shown the Euclidean distance and the angular difference between the pose of the robot and the goal pose over time. Figure 8 shows instead the position over time.

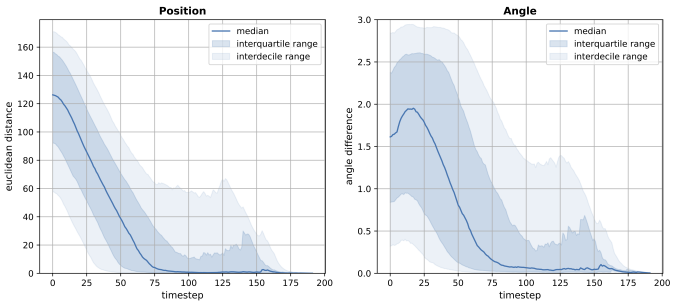

Fig. 7. Distance from goal over time.

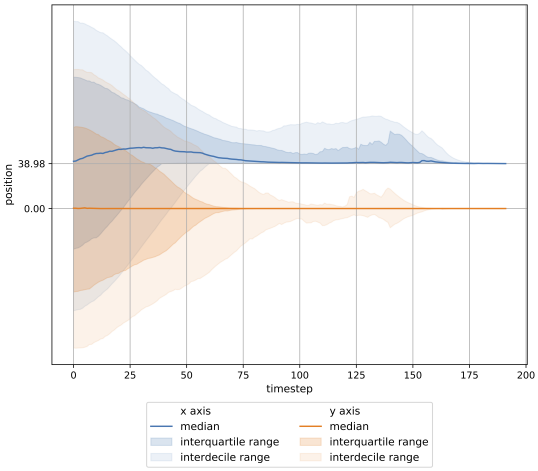

Fig. 8. Position in world coordinates over time.

Finally, the generated dataset is shuffled and split, assigning each run into either the train (70%, 187'000 samples), validation (15%) or test (15%) sets.

B. Proposed Model

We designed a *Convolutional Neural Network (CNN)* that takes as inputs the sensor distances and colour data obtained

from the laser scanner (size: 180×4) and produces as output the left and the right wheel target speeds.

One peculiarity is that we used convolutional layers with circular padding, since the laser scanner returns a 360° view of the world around the robot.The *Rectified Linear Unit* (ReLU) activation function is applied after every layer, except for the last one.

The input data are normalised by subtracting and dividing the channel-wise mean and standard deviation over the training set. Furthermore, channel-wise multiplicative α and additive β parameters are learned during training to rescale the input to the most convenient range for the network:

$$y = (x - \mu)/\sigma \tag{6}$$

$$z = \alpha y + \beta \tag{7}$$

This is implemented in the code with a `BatchNorm1d` layer.

The training set data are shuffled at the beginning of each epoch, so the mini-batches (that are of size 2^{14}) are generated independently between epochs.

The model is trained with the *Adam* optimiser and learning rate 0.001, while the other parameters have their default values. The training is interrupted using *early stopping*, if the validation loss doesn't improve for 20 epochs, or after 500 epochs.

During the various experiments, four different architectures are evaluated:

- *Baseline network*: 3 convolutional and 3 fully-connected layers (Table I)
- Baseline network plus one max pooling layer (Table II)
- Baseline network plus dropout (Table I + Table III)
- Baseline network plus one max pooling layer and dropout (Table II + Table III)

TABLE I
ARCHITECTURE OF THE BASELINE NETWORK

Layer	Channels	Kernel size	Stride	Padding
conv1	$4 \rightarrow 16$	5	2	2, circular
conv2	$16 \rightarrow 32$	5	2	2, circular
conv3	$32 \rightarrow 32$	5	1	2, circular
fc1	$45 \times 32 \rightarrow 128$			
fc2	$128 \rightarrow 128$			
fc3	$128 \rightarrow 2$			

TABLE II
ARCHITECTURE OF THE NETWORK WITH MAX POOLING

Layer	Channels	Kernel size	Stride	Padding
conv1	$4 \rightarrow$ **32**	5	2	2, circular
conv2	**32** $\rightarrow$ **96**	5	2	2, circular
mpool1		**3**	**3**	**1, circular**
conv3	**96** $\rightarrow$ **96**	5	1	2, circular
fc1	$15 \times 96 \rightarrow 128$			
fc2	$128 \rightarrow 128$			
fc3	$128 \rightarrow 2$			

TABLE III
ARCHITECTURE OF THE NETWORK WITH DROPOUT

Layer	Channels	Kernel size	Stride	Padding
...				
fc1	$1440 \rightarrow 128$			
drop1	**dropout with p = 0.5**			
fc2	$128 \rightarrow 128$			
drop2	**dropout with p = 0.5**			
fc3	$128 \rightarrow 2$			

C. Experiment 1

The first experiment performed compares the different architectures using the *Mean Squared Error (MSE)* loss function. The baseline model is able to reach the goal, but with little precision and often colliding with the object, as shown in Figure 9.

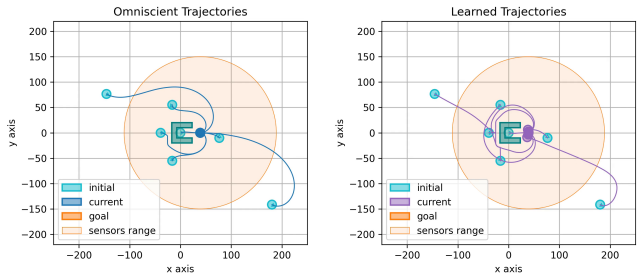

Fig. 9. Trajectories of the controller learned from the baseline network.

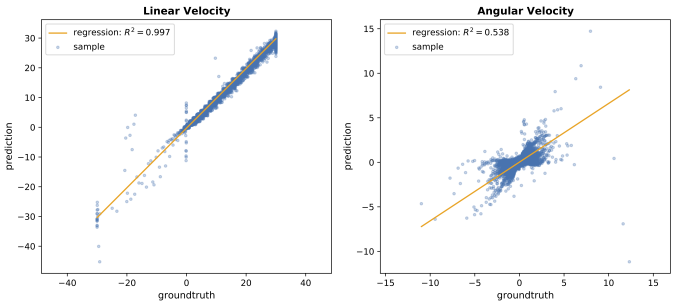

Fig. 10. R^2 regressor on the validation set of the baseline network.

Adding either max pooling or dropout alone does not solve the problem, but combining them results in a visible improvement: the robot reaches the goal position more precisely even if oscillating a bit.

The regression coefficient of the angular velocity, displayed in Figure 12, increases from 0.54 to 0.64 compared to Figure 10, confirming the improvement of the second model. As shown in Figure 13, it shows also a lower validation loss (in red) and does not overfit toward the end of training, like the baseline model (in orange).

Finally, the end positions are more tightly clustered over the goal. Both heat maps in Figure 14 show a tendency to rotate around the object, which are caused by its symmetry. We will explore this issue in Section III-E.

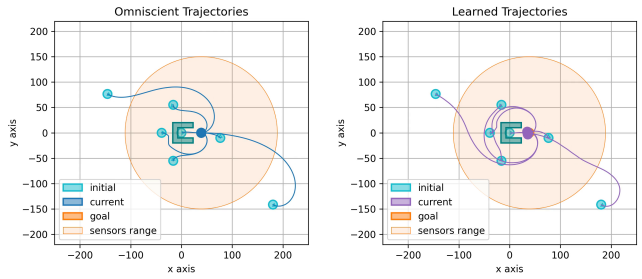

Fig. 11. Trajectories of the controller learned from the max pooling + dropout network.

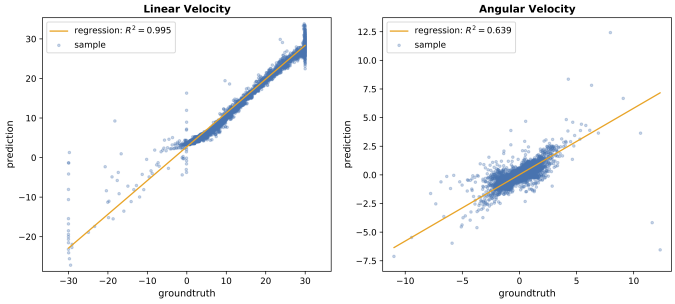

Fig. 12. R^2 regressor on the validation set of the max pooling + dropout network.

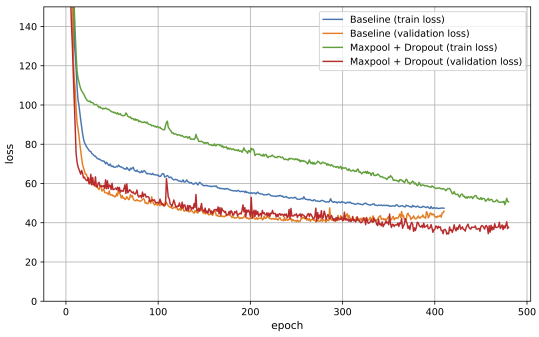

Fig. 13. Comparison of the losses among train and validation sets.

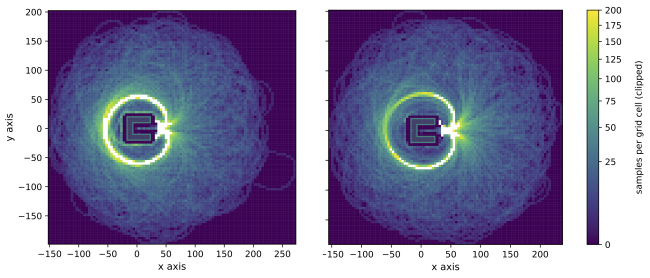

Fig. 14. Positions heat maps.

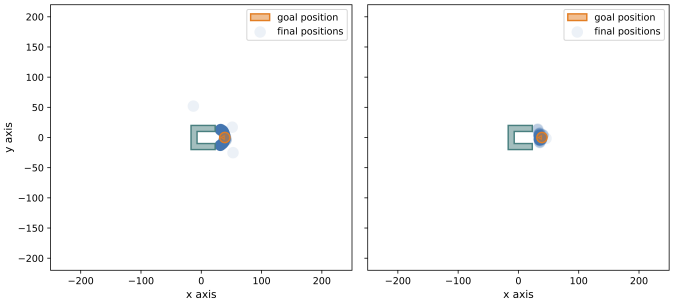

Fig. 15. Final positions.

Even though the overall behaviour of this model is good, the main drawback is a slight systematic error in the final orientation that can be seen in Figure 16.

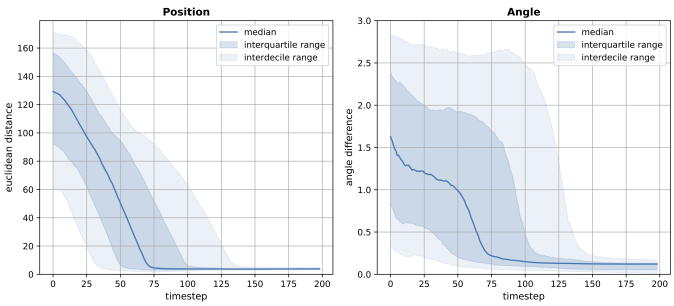

Fig. 16. Distance from goal over time.

D. Experiment 2

The second experiment evaluates the performance of the same max pooling + dropout model, but trained with a different loss function, *Smooth L1* [8], which is less sensitive to outliers than *MSE* and has been shown to prevent exploding gradients in some cases. It is computed as

$$\mathrm{L}(x, y) = \frac{1}{n} \sum_i z_i \tag{8}$$

where z_i is given by

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases} \tag{9}$$

Although it results in less precise final positions, it solves the oscillation issue, as shown in Figure 17.

The regression coefficient of the angular velocity, shown in Figure 18, decreases from 0.64 to 0.58, confirming the superiority of the previous model.

E. Experiment 3

The monochromatic goal object shown so far has symmetries that make the trajectory to follow ambiguous, causing the robots converge to the goal in sub-optimal paths. One such path is visualised in Figure 19, in other cases we saw

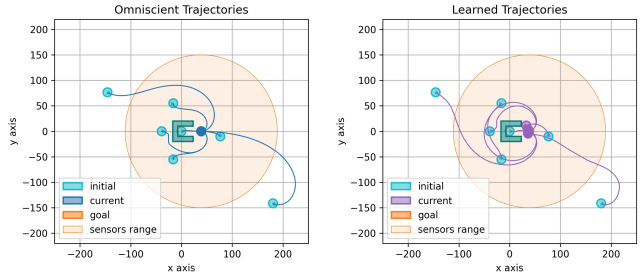

Fig. 17. Trajectories.

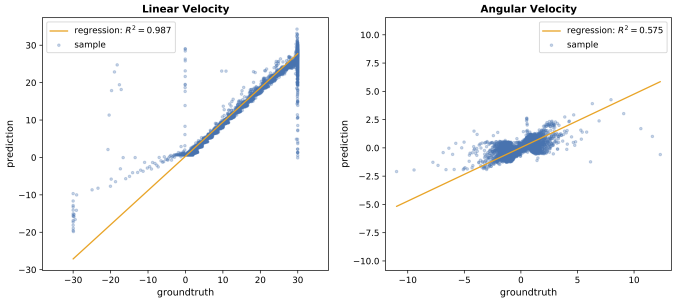

Fig. 18. R^2 regressor on the validation set.

the learned controller always moving counter-clockwise or alternating depending on the initial pose.

In either case, the controller learns a good behaviour for the available data, ultimately reaching the goal pose, albeit following a different path. This is particularly interesting, considered that we only train the network to imitate trajectories, with no indication of the goal.

The symmetries are addressed in this final experiment, by using a polychromatic goal object that has a different colour for each of its faces. This removes any localisation ambiguity in the sensor readings.

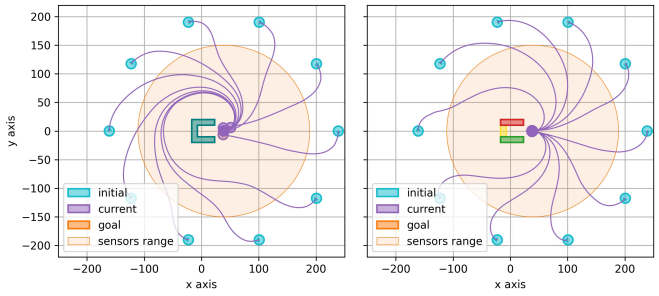

Fig. 19. Comparison of the trajectories of the monochromatic and polychromatic goal object.

The same network architecture and loss function of the first experiment are used to train the the model with a polychromatic object, and result in a significant improvement both in regression coefficient of the angular velocities, shown in Figure 20, and in training and validation losses (in green and red), in Figure 21.

Finally, Figure 22 shows how the end positions are more

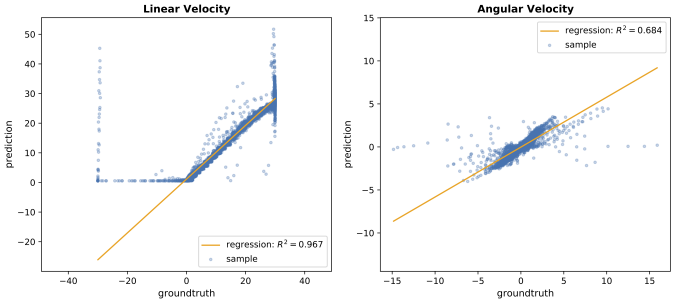

Fig. 20. R^2 regressor on the validation set.

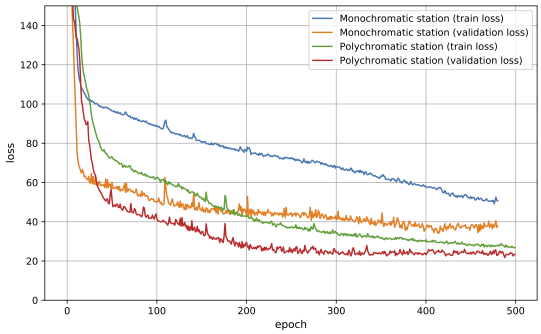

Fig. 21. Comparison of the losses among train and validation sets.

tightly clustered over the goal than before. Moreover, the model is able to follow the optimal trajectories without rotating around the object. Also in terms of convergence, the robot reaches the goal more precisely, as shown in Figure 23, and sometimes even faster than the omniscient controller.

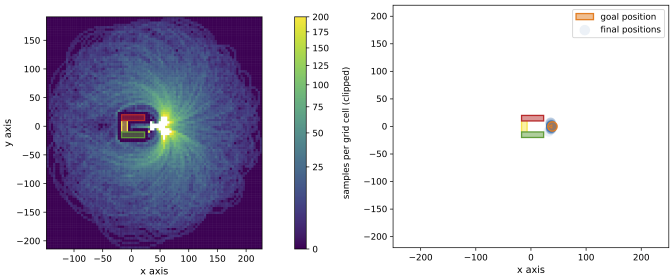

Fig. 22. Positions heatmap and final positions.

IV. Task 2

As an additional task, we experimented with arbitrary goal poses relative to a certain object.

To this end, we first generated a dataset with the same omniscient controller as before, but with random goal poses located in a ring around the object, as shown in Figure 24.

Then, we implemented a neural network which receives the desired goal pose as input to the first fully-connected layer. The architecture used is shown in Table IV.

From the plots shown in Figures 25 and 26 we can see that the unchanged omniscient controller is not perfectly suited

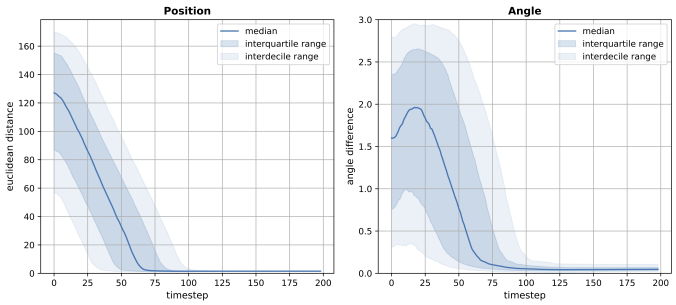

Fig. 23. Distance from goal over time.

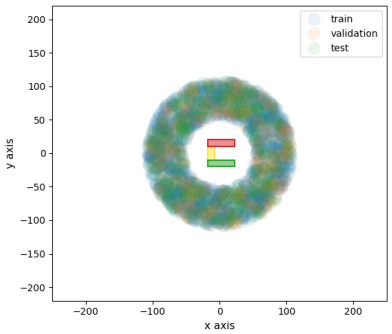

Fig. 24. Goal positions.

for this task: while it was possible to skip collision avoidance before, here it results in many runs getting stuck on the object.

The distances from goal, in Figure 27, confirm this issue, but in general the performance is good when there are no collisions.

This results in trajectories like those shown in Figure 28.

V. Future works

In task 2, it appears that the performance of the neural network is quite good at moving in the general direction of the goal, less so at arriving with the correct orientation or stopping at the right time.

The first issue should be mitigated by a bigger architecture and more training data, since at the moment we are using only 2000 runs as before, which might not be enough given the higher complexity of the task.

TABLE IV
Architecture of the Network for Task 2

Layer	Channels	Kernel size	Stride	Padding
conv1	4 → 32	5	2	2, circular
conv2	32 → 96	5	2	2, circular
mpool1		3	3	1, circular
conv3	96 → 96	5	1	2, circular
fc1	1440 **+ 3** → 128			
drop1	dropout with p = 0.5			
fc2	128 → 128			
drop2	dropout with p = 0.5			
fc3	128 → 2			

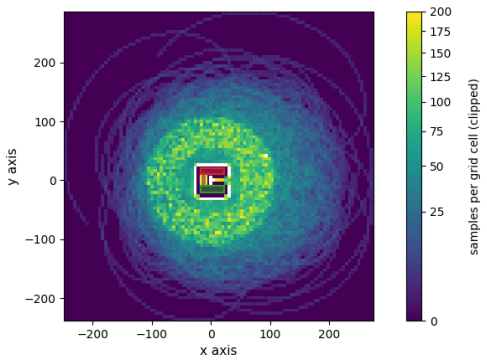

Fig. 25. Positions heatmap.

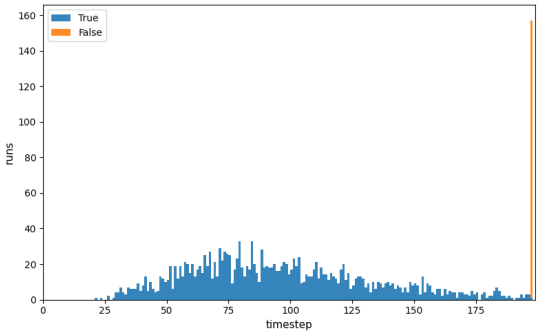

Fig. 26. Time to reach the goal.

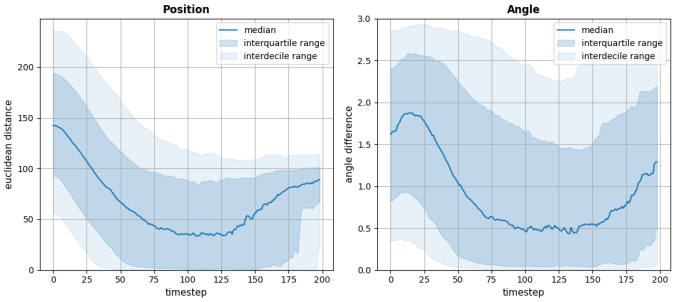

Fig. 27. Distance from goal.

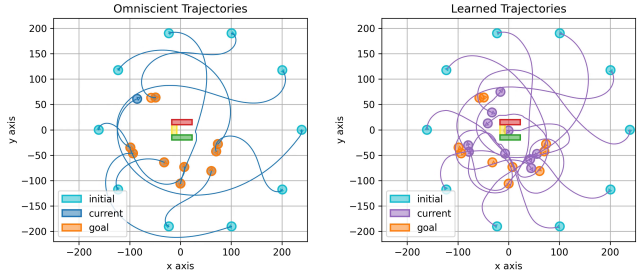

Fig. 28. Trajectories of the controller learned.

The reason for the second problem is that the omniscient controller never overshoots the goal, so the network does not see this situation in training. In Task 1, we solved the same issue with a special case: we had the omniscient controller move in reverse when it spawns inside the arms of the object. A similar solution could be applied here, extending this trick to arbitrary goal poses.

As a future work, we could try to see what happens to the current model when slightly changing the shape of the object. Moreover, we could also generalise the network by creating multiple polychromatic object with the faces coloured randomly.

Another slightly more complex task could include adding obstacles that the robot should avoid, for example by orbiting around them.

In another spin-off, we can consider to add a second marXbot and learn to control the two robots with respect to each other (distributed control).

REFERENCES

[1] G. Adorni and E. Cereda. (2020) learning-relative-interactions-through-imitation. [Online]. Available: https://github.com/GiorgiaAuroraAdorni/learning-relative-interactions-through-imitation

[2] ——. (2020) Learning relative interactions through imitation – Pitch Presentation. [Online]. Available: https://youtu.be/3HVONBpMMms

[3] ——. (2020) Learning relative interactions through imitation – Final Presentation. [Online]. Available: https://youtu.be/odCmFFsUs7A

[4] M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada, "The marXbot, a miniature mobile robot opening new perspectives for the collective-robotic research," IEEE, pp. 4187–4193, 2010. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5649153

[5] M. Stéphane *et al.* (1999) Enki: A fast 2D robot simulator. [Online]. Available: https://github.com/enki-community/enki

[6] G. Jérôme. (2020) marXbot support for Enki simulator. [Online]. Available: https://jeguzzi.github.io/enki/intro.html

[7] J. J. Park and B. Kuipers, "A smooth control law for graceful motion of differential wheeled mobile robots in 2D environment," IEEE, pp. 4896–4902, 2011. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5980167

[8] G. S. C. Adam, Paszke; Sam and C. Gregory. Pytorch: Smooth l1. [Online]. Available: https://pytorch.org/docs/master/generated/torch.nn.SmoothL1Loss.html#smoothl1loss