Multi-agent Reinforcement Learning for Collective Transport

Marcel Torné Villasevil
marcel_tornevillasevil@g.harvard.edu
Harvard University

Abstract

We apply the Deep Deterministic Policy Gradient (DDPG) Reinforcement Learning algorithm to a challenging task in robotics which is the one of Collective Transport. More concretely, the task consists in having a robot swarm move a box to a target point. We develop two RL models, the first one for a single agent and the second one for a robot swarm, which was tested with up to 3 robots. Both models successfully solve the task. Furthermore, we design a benchmark algorithm based on the occluded area algorithm by Chen et al. Finally, the models are trained and ran using the ARGoS simulator.

1 Introduction

We have dreamed about having robots complete tasks that are impossible to humans or to enhance human performance. There are currently two tendencies towards achieving this, the first one is to build very complex robots that have super-human capabilities. The second one, and the one we believe will exist earlier is to have swarms of simple robots which together can achieve super-human capabilities. Ant-colonies are clear examples that this is possible. We believe collective transport is a good starting point for observing such desired swarm behavior and this is what we will be focusing on in this paper. When these multi-robot systems are capable of successfully performing collective transport they will help humans in many tasks. From automation in construction to rescue missions.

As robots and environments become more complex we will arrive at a point where human-designed algorithms will not be possible to design or it will take us too long to find a good one. We believe the solution will be to use Machine Learning and more concretely Reinforcement Learning. In this paper, we are setting a baseline to build upon so that in the future such a technique can be generalized to more complex tasks, environments, and robots. To the best of my knowledge, the approaches that have been taken previously consisted in building deterministic algorithms designed by human researchers, and we will be the first to train an end-to-end system using Reinforcement Learning. We train an RL model that will control in a decentralized manner a swarm of robots (of up to 3 robots) to collectively transport an object to a target point. [1]

2 Related Work

During the last years, the field of Machine Learning and more concretely Deep Learning has been expanding extremely fast and has obtained some incredible results. More concretely many of these advances have happened around the field of Reinforcement Learning (RL). Researchers at Deepmind have pushed this field to the other level by first beating the best human players in the game of Go [7], continuing by beating the best humans at Starcraft [9] and now being closed to solving the big challenge of protein folding, all of this using Deep Reinforcement Learning. This is why we believe that using Reinforcement Learning we should be able to solve many other challenges and in particular the problem of collective transport in robotics. To the best of our knowledge, we will be the first to solve this problem uniquely using RL from end to end.

In the past, some papers have already tried to achieve collective transport by robots. Chen et al. designed and proved the validity

[1] The code from this paper can be found in the following GitHub repo:
https://github.com/MarcelTorne/RLCollectiveTransport

of an algorithm that achieved collective transport in a fully decentralized way with minimal communication among agents. Bloom et al. [2] developed an algorithm for collective transport where each agent would have different predefined roles to play a game (as a pusher, defender ...) and these roles were chosen using RL. To solve the game they designed, the robots had to collectively transport and object to a given goal. This research has been very useful for us since the experiments are conducted using the ARGoS simulator [6]. Hence it was a base to build upon our experiments without having to reinvent how to introduce the RL models into this experimental design. Our work will differ from this last paper in that our RL algorithm will completely control the robots from the receival of the inputs to the actions that these will take.

3 Methods

In this section, we will give a detailed explanation of how the project was designed. We will not go deep into coding technicalities since we do not believe it gives much value to the paper. Nevertheless, the code written for this paper can be found on the GitHub repository for which the link is given on the first page of this paper.

3.1 Project Design

The project design is depicted in Figure 1. We have two main components: the environment's simulation and the RL model. These two main components will be completely separated into two processes. On one side we will have the environment's simulation developed using the ARGoS simulator where all of the code is written in C++. On the other side, we will have a second process running the RL model, coded in Python and using the Keras library [3]. These two processes will have to communicate between them. On one side, the simulation process will send information to the RL model's process about the state of the environment and the corresponding reward for a given action. On the other side, the RL model's process will communicate which actions should be taken given a state. This inter-process communication is performed using the ZeroMQ library [2], which is an open-source universal messaging library.

3.2 Simulation

In this section, we will proceed by explaining the implementation of the simulation using the ARGoS simulator.

3.2.1 Implementation details

In Figure 2, we can see a simplified overview of the file structure. This is what has been used in the project and it is specific to the ARGoS simulator, however, we believe it is very easily generalizable to other simulators as Webots, for example.

The *"main.cpp"* file is the main file and is the one that is executed when we want to start the simulation for training the model. This file mainly consists of an instantiation of a CRL object, defined in the *"rl.cpp"* file and it will launch sequentially as many simulations as specified in the *un_generations* parameter.

The *"rl_loop_function.cpp"* file corresponds to the definition of a single episode. This file defines the behavior for each step of the episode, the initial setup of the environment and the cleanup to be performed when the episode ends and has to be reset. Furthermore, from this class, we have a global overview of the whole arena and the position of the different entities (robots, box, target light). Hence, it is the place where we will compute the rewards for each robot and where this information will be sent to the parallel process running the RL model. This information will be sent once at the end of each step through a single message. This message will have the form of an array with the form of [robot_id: associated reward].

The *"rl.argos"* file is where the configuration of the arena and the simulation is specified. In this file, we set the dimensions of the arena, the different entities that will be instantiated (robots, light, and box), the sensors used by the robot. This is the file that is specifically used for training and hence, we disabled the visualization since this makes the simulation run more slowly and would be an overhead for training. However, since we want to see how the model is performing, we have an

[2] https://zeromq.org

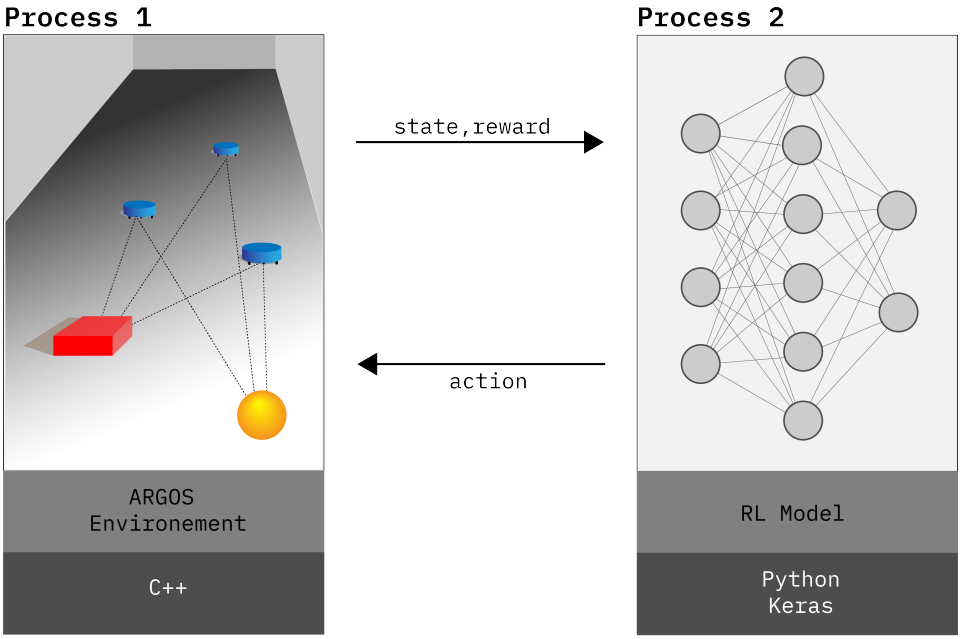

Figure 1: The project code is designed in two different processes. The one on the left represents the process running the simulation, using ARGoS and written in C++. The one on the right represents the process running the RL model written in Python and using the Keras library.

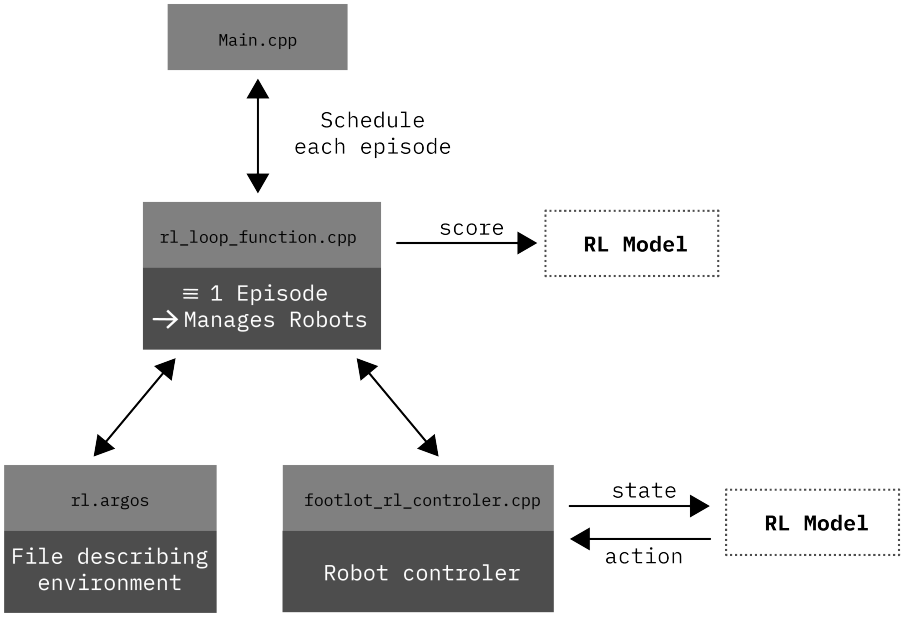

Figure 2: File organization for the simulation in ARGoS. The main file schedules each episode for a given number of generations. The loop function is equivalent to one episode, it has a global view of the arena and its entities, and it sends the reward to the RL model. The *"rl.argos"* is a file specific to ARGoS that handles the definition of the environment. The controller controls the robot, sends the observed state to the RL model, and receives action to execute from it.

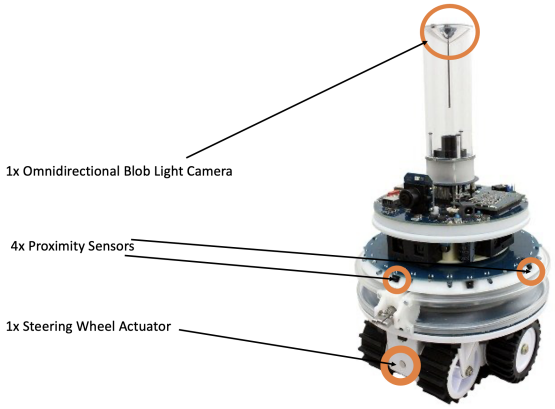

Figure 3: The foot-bot is the robot used in this project. The sensors used are an omnidirectional blob light camera and 4 proximity sensors. The steering wheel actuator is the only actuator used.

additional file *"rl-trial.argos"* where the configuration is the same as for the *"rl.argos"* file with the sole difference that the visualization is enabled and hence we will be able to use it to observe the learned model.

Finally, the *"footbot_rl_controller.cpp"* file defines the class that will be run to control each robot. The scope of the class is restricted to each robot and it is the place where we read the sensors to get the current state of the robot. Once we have the state, this is sent to the RL model process and we wait to receive a reply that will come with the action. This action is then executed by the robot.

3.2.2 Robot

The robot we are using is defined as the foot-bot in the ARGoS simulator and is based on the MarXbot [1]. More importantly, is the sensors that we are using, which are visible in Figure 3. Since, as we mentioned before, we are basing this algorithm using the occluded area idea, we are mainly going to work with light sensors. The light sensor we are using is an omnidirectional blob light camera. This is a light sensor with the additional qualities that it can differentiate between different colors for the lights and it has also 360-degree coverage. Secondly, we use four proximity sensors, placed 90 degrees from each other around the robot. These were added thinking about collision avoidance by these robots. Finally, we are using a single actuator on the robot and this is the steering wheel actuator, which is used to set the speeds of the left and right wheels of the robot.

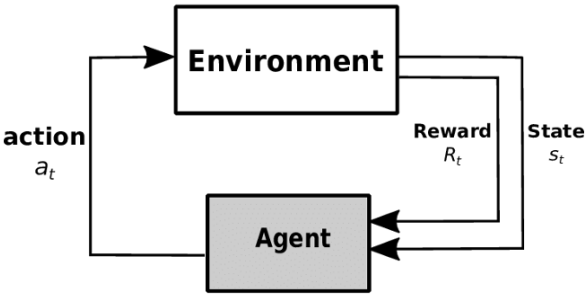

Figure 4: Overview of the components needed for training RL models. And environment informs about the state and rewards given an action and the agent learns to predict the best action from a given state.

3.3 Reinforcement Learning

Recall that the goal of this project is to build a Reinforcement Learning model that will control the robots in a decentralized manner to perform a collective transport task. Hence, in this section, we will start with a summary of the Reinforcement Learning framework. We will continue by going deeper into the important sections of the RL framework, starting with exploring the design of the environment, continuing with the controller of the robots, the RL algorithm we chose to use, and finishing by the approach we took to make the learning of a policy possible within the resources available.

A Reinforcement Learning model consists in learning a policy that will return the best action to take given a state, it can be seen as a Markov Decision Process. For learning such policies we need a simulated environment where the model, more concretely the Actor model, will observe different states, then try an action and observe the reward given by the environment. In Figure 4, we can see this interaction between the Actor model (Agent) and the environment.

The reward is a very important part of training RL models, this one is defined given a state and an action and the goal of the agent will be to maximize it. Designing good reward functions is something very important and is necessary for learning proper policies.

Furthermore, these RL models contain a second important component, besides the Actor: the Critic. This model predicts how well

each action is given a state and an action.

Finally, we want to emphasize two technicalities of training these RL models. First, is the usage of two pairs of neural networks for the Actor and the Critic: the Target-Actor/Actor and the Target-Critic/Critic. This is used for training stability, the reason being that we will update much more quickly Actor and Critic networks and then we update more slowly the Target-Actor and Target-Critic. In this way, the training is more robust to the agent exploring many bad actions in a short amount of time such that the target network does not get completely biased. The second technicality is the usage of an Experience Replay Buffer, the tuples of actions, states, and rewards are kept in a buffer so that the gradient descent step over the networks can be performed using many previous states and not just the last one. Again, this makes it more robust and reduces variance if a bad action/state is explored in the last step for example.

3.3.1 Environment

The environment is one of the key components in the RL framework. The environment we designed consists of a 30 by 30 unit sized arena, with walls as boundaries. Moreover, it contains three different types of entities. The first is the target light, this represents the point where the robots have to move the box to. It emits a yellow light and is centered at the point (5,5) and 0 in height. The second component is the box, which has dimensions 1 by 1 unit, height 0.3 units, and starts centered at (6,3). This box is surrounded by 12 lights, 3 lights placed at each vertical face of the box in positions [-0.4,0,0.4] units from the center of the phase and height 0. These lights are used by the robot to detect where the box is. Recall that we are not using a camera but a blob light camera that only detects light signals. Furthermore, we colored these lights in blue so that we could differentiate that this light comes from the box and not from the target light later in the signal processing at the robot controller. And all the lights are placed at height 0 because otherwise, the blob light camera will not detect them. Finally, the last entity corresponds to the robot, and as we mentioned earlier, we used the foot-bot equipped with the mentioned sensors.

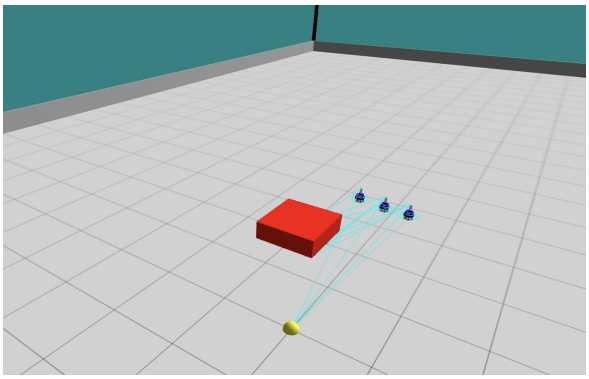

Figure 5: Screenshot of the environment used for the simulations. It consists of a target light (yellow), a box to be moved (red), robots to perform the task (blue).

Furthermore, when designing an RL environment we have to properly define the input and action space. In our case, the input space corresponds to the readings of 4 proximity sensors, with values between 0 and 1, a 2D vector to the target light, and a 2D vector to the closest light from the box. These 2D vectors are represented in the form of an angle with respect to the robot and a distance. The positive aspect of using such a representation instead of absolute Cartesian coordinates is that we do not need to know the absolute position of the robot in the arena but we can simply work with the observed angle. This implies a simplification in the learning of the policy since more information is given and this unnecessary learning is avoided. The action space, corresponds to the left and right wheel velocities of the robot. A screenshot showing the environment simulated in ARGoS can be seen in Figure 5.

3.3.2 Algorithm: DDPG

The RL algorithm used in this paper is the DDPG algorithm by Lillicrap et al. [4]. The DDPG algorithm is shown in Algorithm 1. It consists in learning optimal critic and actor networks. The reasons why we used this specific algorithm are that it has already been tested in many scenarios and it managed to obtain good results. Furthermore, this algorithm to the difference of other algorithms as DQN [5] has the property that it works for environments with continuous action and input space, which is exactly our case.

For Reinforcement Learning models to learn

Algorithm 1 DDPG: Deep Deterministic Policy Gradient

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process N for action exploration
  Receive initial observation state s_1
  for t=1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu\prime})|\theta^{Q'})$

    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q\left(s, a \mid \theta^Q\right)|_{s=s_i, a=\mu(s_i)}$$
$$\nabla_{\theta\mu} \mu\left(s \mid \theta^\mu\right)|_{s_s}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

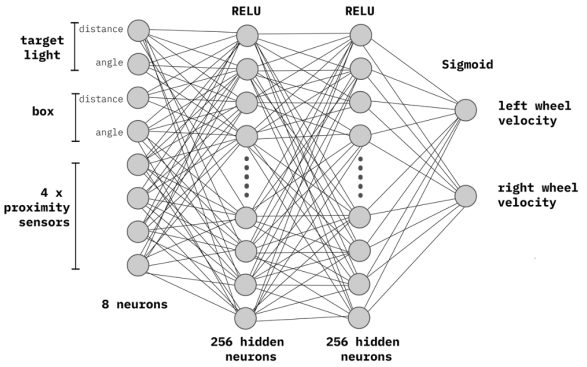

Figure 6: Neural Network architecture used for the Actor model.

the best policy, there has to be a trade-off between exploration and exploitation. For this reason, the action exploration noise is a very important factor in these algorithms. We decided to use the Ornstein–Uhlenbeck process, the most common exploration noise in these settings.

Two important components of the DDPG algorithm are the network used for the critic and the one for the actor. Since it can be very tedious to construct and fine-tune such networks we decided to reuse some of the most common settings that have already been proven to work in other environments. More concretely we used the one used in the DDPG paper by Lillicrap et al. The critic network remained very similar, only the input layer was modified to have as many nodes as actions and state signals (10). However, we had to tweak the actor-network more. First, both the input and output layers had to be modified to have the number of input signals for the input layer (8) and the number of actions for the output layer (2). Furthermore, we changed the activation function in the output layer to a sigmoid function. The reason is that we will interpret the output as the velocity of the wheels and we want this to be between [0, MAX_VELOCITY]. Hence, if we use the sigmoid function we will get an output between [0, 1] and then we just have to multiply by MAX_VELOCITY to have it in the desired scale. The resulting network is depicted in Figure 6.

3.3.3 Improving training

Training Reinforcement Learning models is not a trivial task and always comes with some tricks that will make the learning of the policy much easier. After trying different ideas we observed that some methods would help us achieve the desired result. First, we did not train the robot swarm collaboratively from the very beginning but instead, we trained a model for a single agent, that could achieve the desired task. Once we had such a model, we fine-tuned it by training it in a robot swarm. Furthermore, when training in a robot swarm, one single instance of the model will be running and providing the actions to each robot given their local state. This will also speed up training and make the model more robust since it should generalize to all robots, no matter where they are positioned.

3.3.4 Computing Resources

We had a limited amount of computing resources to run the experiments. Hence, training the models and running the simulation was completely done on our local computers. This is clearly a limitation for the complexity of the model we are going to be able to train. With more computing resources we would have been able to run the simulations for longer and hence we would hopefully see even better behaviors emerge from this training. Furthermore, we really felt that the limited computing resources were a bottleneck with respect to the size of the robot swarm that we could train. We had to limit ourselves to running it with 3 robots but it would be very interesting to see the emerging behaviors with much bigger swarms. Despite that, we still got some very interesting results that are presented next.

4 Results

In this section, we are going to present the results obtained. The first result will present will be our benchmark algorithm which was developed to prove that a policy existed to perform such a task with the given sensors. We will continue by presenting three results using the RL models. The first model of these three controls the robot to move to the light. The second controls a single robot to move the box to the light. Finally, the third one controls the robots in a swarm to move the box to the light.

4.1 Benchmark

Algorithm 2 Collective Transport Controller Benchmark

```
while episode not ended do
  if target light is seen then
    SetRandomWalk();
    ObstacleAvoidance();
  else if box is seen then
    MoveTowardsBox();
  else
    FullRotation();
  end if
end while
return ;
```

We designed a benchmark algorithm to show that it was possible to have a policy working with the given sensors and the designed environment. Our algorithm was inspired by the Chen et al. algorithm mentioned before. The idea here is also to have the robots push towards the box when they find themselves in the occluded area. A representation of this occluded area can be seen in Figure 8. One first problem that we had to solve was how do we find this occluded area? Our decision was to make the robots perform a random walk until they find themselves in this desired occluded area. How to perform the random walk was a second problem to solve. The way we approached it was to make the robot move in a straight line and after some random amount of time, make it turn in a uniformly randomly chosen angle between -90 and 90 degrees. Furthermore, we saw that when performing this random walk, the robots could actually push the box away from the target point, hence, what we did to solve this was to introduce collision avoidance when the robot was not in the occluded area. Finally, we had to add one last edge case, which consisted in performing a full rotation when the robot does not see the light and does not see the box. This could happen when other robots are between the current robot and both the box and the target light. In this case, we decided to stop the robot by doing a full rotation until it recovers sight of the targets.

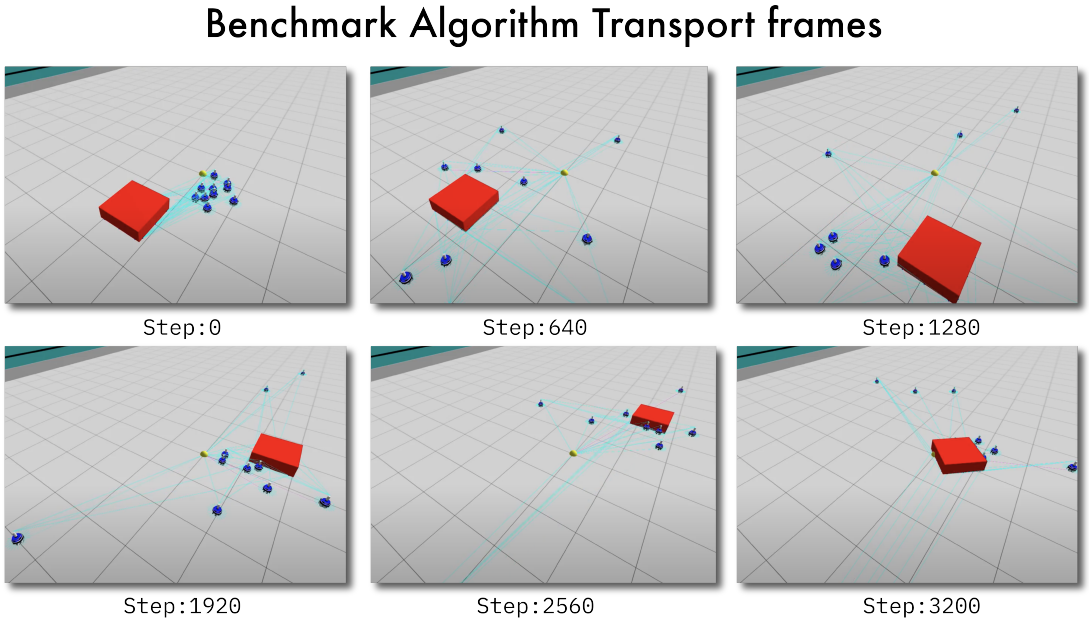

Figure 7: Frames of steps extracted from a successful collective transport episode using the Benchmark algorithm.

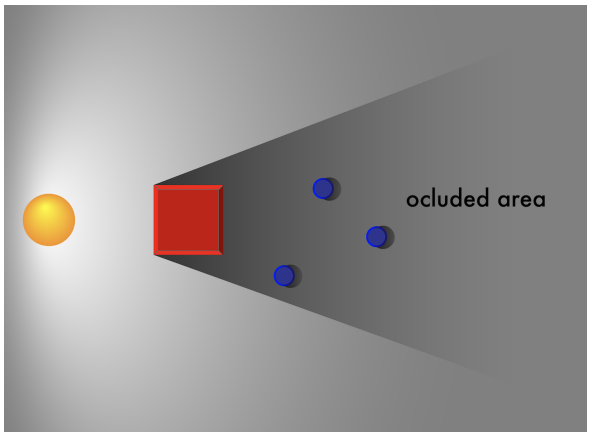

Figure 8: Representation of the occluded area.

The pseudocode for the benchmark algorithm can be seen in Algorithm 2.

We tried to make the environment as hard as possible, to make sure that such a policy was possible. We believed that the way to make it as hard as possible but still stay within our setting was to put the robots between the box and the target light. Since our algorithm was designed such that the robots had to get to the occluded area then this seemed the hardest setting.

We observe in the video[3] and in Figure 7 that even though it takes many steps (3200), the robots accomplish the task.

4.2 Single robot to light

Designing and training RL models is a tedious task. To get a working model you first must be sure that the environment is properly designed and that there are no bugs. Bugs can appear in the in the signal processing of the sensors among many others. Furthermore, RL models contain many parameters hence hyper-parameter tuning is necessary, for learning rates of the neural networks, replay buffer sizes, gamma for reward discount to name a few. For this reason, we first tried to train a model for a simpler task. This simpler task consisted in having the robot move to the target light, no box was involved here. The result can be seen in the video[4] and in Figure 9

The working version was achieved with the following hyper-parameters: critic_lr = 0.002, actor_lr = 0.001, gamma = 0.9999 (discount factor), buffer_size = 50000, which were reused for the rest of the tasks. The reward function used is presented in (1). Moreover, in Figure 10, we observe how the reward per episode increases with the number of episodes.

[3] https://youtu.be/o6TBpNbLeJw

[4] https://youtu.be/i5mBqr2pW6c

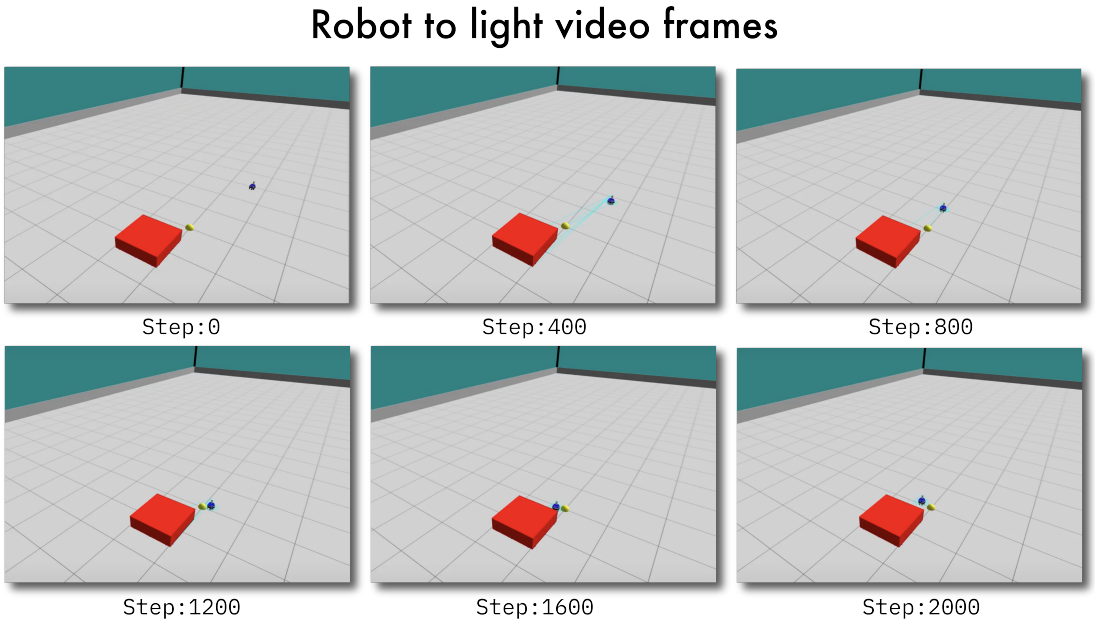

Figure 9: Frames of steps extracted from a successful episode of an agent moving to the light using the RL model.

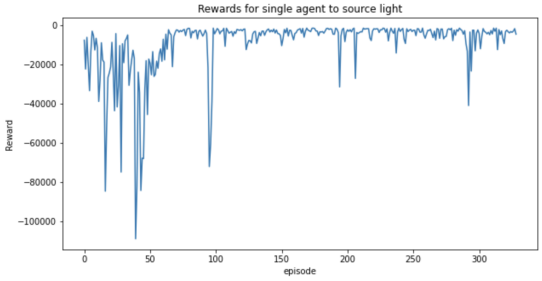

Figure 10: Reward learning curve of the single RL agent for going to the light.

We observe that this increasing trend seems to achieve a horizontal asymptote and hence its maximal value around episode 80. For this reason, we could say that convergence has been achieved.

$$Reward = -distance\_robot\_light^2 \quad (1)$$

4.3 Single robot transport

As was mentioned before, the approach we took in this paper was to first train an RL model to control a single robot to move the box to the target source of light, and then we will fine-tune it for a swarm of robots. In this section, we trained the model for the single robot to move the box.

$$Ideal \quad Reward = -(distance\_box\_light^2) \quad (2)$$

The ideal reward would be (2). The reason is that in this reward we are clearly specifying what's the goal of this task, which is to move the box to the source of light and we are not enforcing any other constraints. Furthermore, since this is a step reward, meaning that at each step this reward will be recomputed. Hence, when maximizing the reward function of the model will also force the robot to do it in the less time possible since these rewards accumulate throughout the episode.

$$Actual \quad Reward = -(distance\_robot\_box^2 + distance\_box\_light^2) \quad (3)$$

Unfortunately, when using this reward, the model did not manage to learn any working policy with the limited computing resources we had available. Another reason is that given the high dimensionality of the possible number of paths that did not lead to the box the robot was not even able to arrive to move the box. Hence, we had to do some reward shaping and the final reward function was (3).

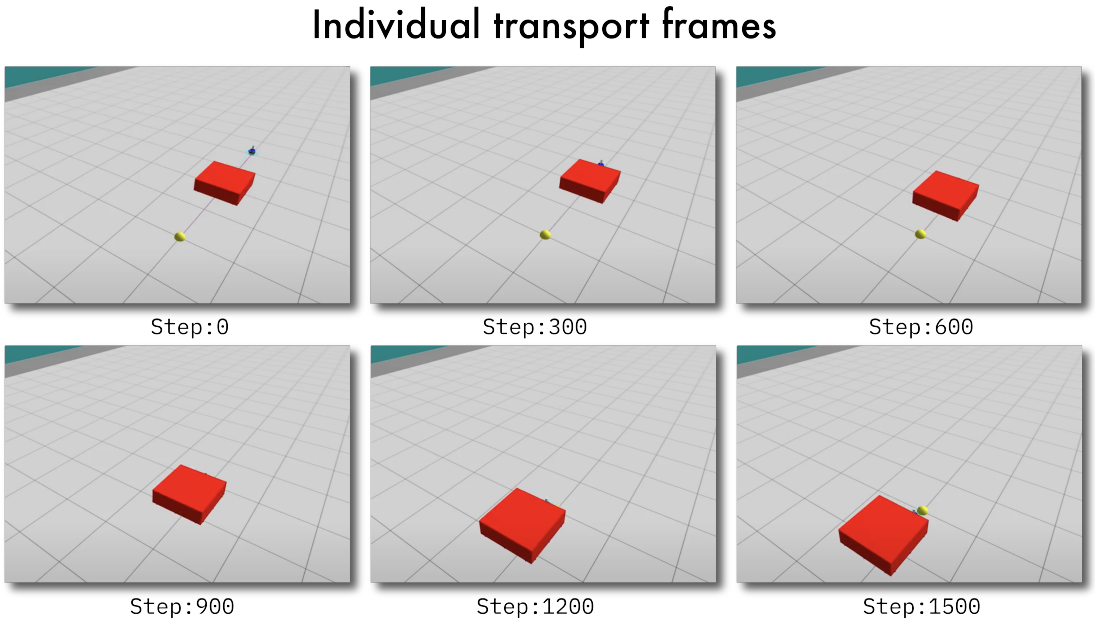

Figure 11: Frames of steps extracted from a successful individual transport episode using the RL model.

Figure 12: Reward learning curve of the single RL agent for individual transport.

Using this reward function we did manage to learn a good policy. We observe in Figure 12, the reward keeps increasing and stabilizes around episode 90. Furthermore, around episode 80 we can observe a curious phenomenon. Before this episode, the robot manages to learn a policy to move the box to the light, but at some point, it runs over the target light and continues to push the box further away from it without ever stopping. In the following episodes, the model does learn to stop pushing the box once the light is observed and this is the behavior seen in the video[5] and Figure 11.

4.4 Collective transport

Finally, the last experiment consisted in training a model that allowed a robot swarm to perform collective transport. As we explained earlier, the way we approached it was that we first trained an RL model that would solve the task of collective transport, and then we fine-tuned it for performing collective transport in a robot swarm. The result can be seen in the video[6] and in Figure 13.

That is exactly what we did, we reused the model learned from the section before and retrained it in a robot swarm made of 3 robots. The way we trained this model was by having a common model that would learn from the actions taken by all of the robots but that would make its decisions based solely on the observed state by each robot independently. Having such a model will allow having a robot swarm with a decentralized model, meaning that no central coordinator will be needed and all robots will be able to behave independently.

We reused the same reward function as in the section before (3), where the total episodic reward corresponds to the sum over all of the local rewards from each robot (4).

[5] https://youtu.be/3wF8eM2zTHY
[6] https://youtu.be/2RE˙2JDM-kY

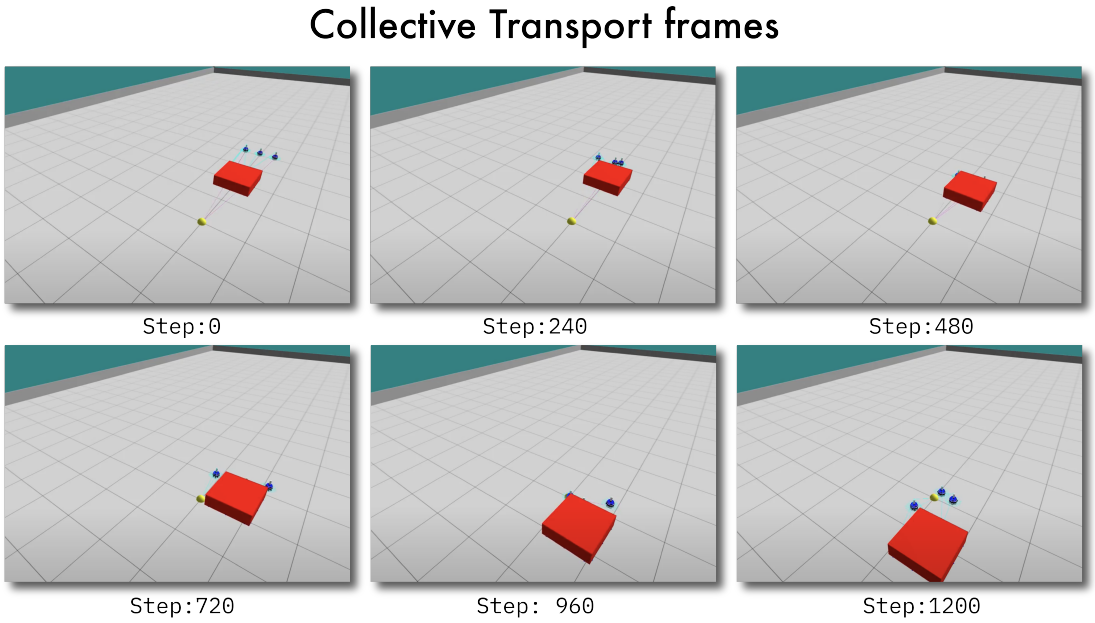

Figure 13: Frames of steps extracted from a successful individual transport episode using the RL model.

Figure 14: Reward learning curve of the multiple RL agent for collective transport.

$$General \quad Reward = -\sum_{i} distance\_robot\_box_i^2 + distance\_box\_light_i^2 \tag{4}$$

The curve of the reward while training this model can be seen in Figure 14. We have multiple observations. First, there is a big drop in performance around episode 60. However, this drop in performance is recovered later. Furthermore, we observe that the reward obtained by the agents at the beginning and at the end is very similar, hence we could say that unfortunately there was not much learning. However, we believe that there was indeed some training since the variance in the reward over the episodes is reduced with more episodes which is a good sign of the model learning. Moreover, despite this issue, we are optimistic about what this model can achieve since collective transport is successful as seen in Figure 13.

Furthermore, it is very hard to learn a simple Reinforcement Learning model in a multi-agent setting. The reason being that at each episode step, all robots will act, hence, the reward will probably be affected by the action of other robots. This means that there will be some stochasticity in the reward since now the reward function seen by each robot is not deterministic anymore, but it also depends on the actions of the rest of the robots. Moreover, the DDPG algorithm does not handle properly this stochasticity in the reward function. For this reason, we should, as future steps, implement a proper Multi-Agent Reinforcement Learning algorithm as the ones presented by Zhang et al [10].

Finally, during some of the episodes, we observed collisions between robots, which is not desirable if we want to deploy this model in the real world with real hardware. One of the solutions we could deploy to solve this is to add a factor in the reward that penalizes being too close to other robots, similarly as what is done for flocking algorithms [8]. We could also hard code it outside of the model, meaning that af-

ter getting the action from the model, we will evaluate if this will lead to a collision between robots and if it is the case we will instead execute an action that prevents this to happen.

5 Conclusion

In this paper, we accomplished our goal, we have a Reinforcement Learning model that controls a robot swarm in a decentralized manner to perform collective transport. The way we achieved this, was by first training an RL model that would control a single robot to perform collective transport, and then we fine-tuned it to generalize for a robot swarm.

Moreover, watching many experiments of these robots we observed that the optimal learned policy seems to be also based on the occluded area paradigm. The reason we believe this is because when looking at the swarm experiments we observe that when one of the robots is over the light (which then becomes occluded for the rest of the robots), the rest of the robots continue pushing the box, and they just stop once the first robot stops occluding the light.

This does not mean that the occluded paradigm is the best algorithm for collective transport, however, it probably means it is the best algorithm for collective transport given the environment we designed and the sensors used by the robots. Hence, to find even better policies we will have to equip robots with different types of sensors and have more diversity of environments to train them on.

6 Future Work

There are some next steps that would be great to have first. It would be interesting to train models that generalize for different shapes of the object, for example, cylindrical shape. The way we believe this can be achieved is by incorporating some randomness in the shapes of the object during the training process. Furthermore, we would like that our models accomplish harder tasks for which human-designed algorithms do not work well. For example, we were thinking about adding obstacles to make the environment harder and other-robot awareness to enhance robot collaboration.

Continuing by some longer-term goals, we consider that this project was successful since we managed to build an algorithm that presents a collaborative behavior among the agents and successfully transports the object. Despite this, the observed behavior is still far from the one observed in the ant colonies. One of the reasons is the capabilities of these robots. These are very limited and in this case, the robots are just pushing the object. We believe a future step should be to empower these robots with more capabilities. A great improvement would be to have the robots pull up the object. We foresee that this capability would allow collaboration, where some robots would pull up the object and some other robots, would go below this one. This would improve the performance in this task in multiple ways, first, more robots would be able to move the object at the same time which allows a bigger object to be moved. Second, if the object is pulled up, then this is not touching the ground hence removing the friction factor which will also allow moving the objects in different types of terrain and not just the ideal case scenario with very little friction. Moreover, if the robots could elevate the object, it would mean that they would be able to interact and transport the object in a much complex topology of the terrain, for example, we can imagine that these robots would be able to move an object up the stairs. It is impressive that, with such a simple addition to the robots, so many new behaviors could arise and how much this would improve the overall performance of the swarm. For this reason, we are very excited since if these future steps are attained, it will offer a new path to having robot swarms executing transportation and performing many different tasks in complex environments. Finally, we will be really able to say that our goal was fully achieved when we manage to train a model for a task for which there is no human-designed algorithm that manages to complete it successfully.

References

[1] 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. In *2010 IEEE/RSJ International Conference on Intel-*

ligent Robots and Systems, pages 4187–4193, 2010.

[2] D. S. Catherman, C. Neville, J. Bloom, and S. S. White. Reinforcement learning adversarial swarm dynamics. In *2020 SoutheastCon*, pages 1–6, 2020.

[3] F. Chollet et al. Keras, 2015.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2019.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[6] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. D. Caro, F. Ducatelle, T. Stirling, A. Gutiérrez, L. M. Gambardella, and M. Dorigo. ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, pages 5027–5034. IEEE Computer Society Press, Los Alamitos, CA, September 2011.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[8] H. Tanner, A. Jadbabaie, and G. Pappas. Stable flocking of mobile agents, part ii: Dynamic topology. *Departmental Papers (ESE)*, 2, 05 2003.

[9] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[10] K. Zhang, Z. Yang, and T. Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms, 2021.