MSBD Independent Project Report – Route planning for multiple logistic robots

Cheung Ka Sing 20824804 kscheungaq@connect.ust.hk

This is the final report on the development and implementation of a route planner in the provided dorabot-minions logistic robot simulator using reinforcement learning. Both the project and the report are practical-oriented, focusing on design decisions and technical details.

Project context

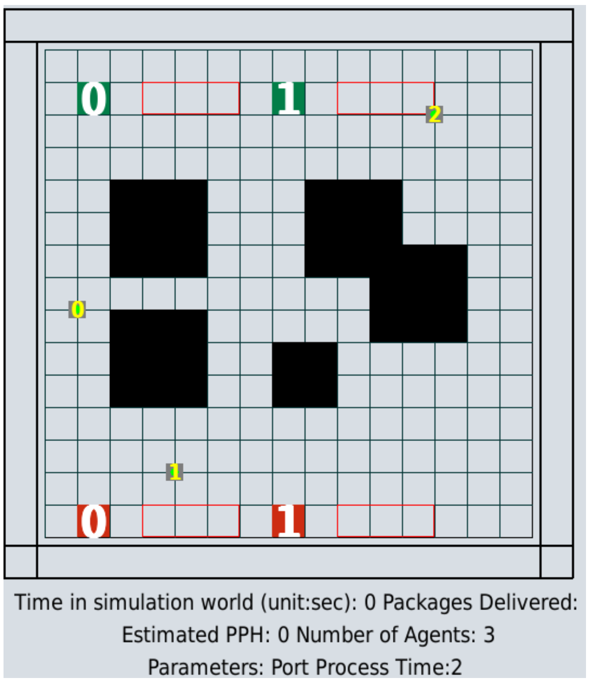
Robot logistic simulator

Dorabot-minions is a warehouse logistic simulation where robots are guided to transfer items from loading ports (top, green) to unloading ports (bottom, red) while avoiding obstacles (black). The simulator supports both continuous and discrete space, with a grid size of 1. In this project, continuous space is used. In addition, each port has an accompanying queuing zone (rectangular area with red outline). The sequence of port operations is summarized as follows:

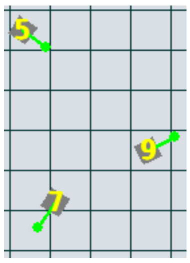
- When a robot is close enough to the port, it is instructed to go to the end of queueing zone.
- When a robot is close enough to the end of queuing zone, it is instructed to go to the next available position in the queuing zone. The queue position could change in the meantime.
- When a robot arrives at the queuing position, it stops. If it arrives at the head of the queue, it starts loading or unloading. Otherwise, this step is repeated when the next position in front is available.
- Afterwards, the robot moves to the other side of the map for the next task. The closest loading port by straight line distance is chosen for the loading task. The unloading port is pre-determined by the item.

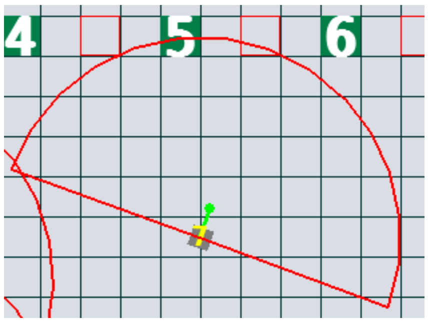
While the majority of travelling is done between loading ports and unloading ports, the movement in the queueing area can be considered a qualitatively different movement task.

Each robot is represented as a square of length 0.5; the shape is likely chosen for faster geometry calculation. Each robot has all-direction drive, where it can select the desired velocity in any direction with a speed limit of 1.5. Each robot also has a 180^{o} field-of-view, which is realized as ray casts outwards from the robot with a max range of 4 (emulating LIDAR). The viewing field aligns with its current velocity.



Time in simulation world (unit:sec): 0 Packages Delivered:
Estimated PPH: 0 Number of Agents: 3
Parameters: Port Process Time:2





The movement task in the simulator is structured as a two-level hierarchical problem: waypoint planning and point-to-point (P2P) control.

- Waypoint planning involves the computation of suitable waypoints between the robot's current position and destination. This is performed by the global planners and is only evoked when there is a change in destination for the robot.
  While there is a framework for a 'multi-agent' global planner, where the waypoints for all robots are collaboratively planned, the heuristic for collision has no knowledge of P2P behaviour. As such, the collaboration at the level of waypoint planning is conditional on near-optimal P2P control.
- P2P control involves the computation of the instantaneous desired velocity (action) to reach the next waypoint. This is performed at each step while the robot is travelling, and the robot has access to its current position, velocity, and recent LIDAR data (3 frames by default). There is some flexibility between consecutive P2P tasks, where the controller can decide if the current waypoint has already been reached or not.

Nominally, the simulator runs at 60 Hz (simulation time), and P2P control can be performed at every step. This version of P2P control might be more challenging than P2P control utilizing frame skip. Frame skip [1] is a rather common technique in control problems in a simulated environment. First, learning every frame is slow relative to simulation time, and to explore the problem sufficiently, an extreme amount of training is the minimal requirement. Furthermore, learning is prone to focus on the current situation that can exhibit distribution drift over the entire training time scale. Retaining learning progress throughout training requires special care. Thus, in this project, the P2P control task is defined as operating at 5 Hz (also more realistic for common robots).

Methodology

The project revolves around the P2P control task. Lightweight tuning of the provided waypoint planners based on the learned P2P planner is attempted as time and resources permit. This hierarchical organization is commonly adopted for robot route planning to reduce the complexity of the problem. Still, there is no limit to how contrived the P2P control problem can be in continuous state space, so various reinforcement learning techniques are often considered to handle the problem better.

With on-demand access to simulation in this project, it is a prime opportunity to consider a recent proposal [2] that the P2P control RL problem can also be optimized for (in the black-box sense).

- Parametrizing the RL problem provides extra degrees of freedom in problem formulation, improving the chance of successful learning. Reinforcement learning is sensitive to a whole slew of factors. For instance, as a reward signal 'arrival at the next waypoint' is sparse and not productive, a parametrized reward function can be constructed to overcome sparsity and improve learning. It is also possible to consider network architecture as part of the problem parametrization since the optimal architecture depends most strongly on the environment.
- The extra layer of optimization provides a view of the balance between different conditions and constraints. There are insights to be had on what and how trade-offs are made.

The objective of this project is to optimize over the P2P control problem space and locate a specific problem formulation that produces the best learning result for the task – with consistent training and evaluation.

Reinforcement Learning

<u>RL problem definition</u>

The following table shows definitions relevant to the reinforcement learning problem. These definitions are analogous to those in the original proposal, and subcomponents are simple concatenated. The problem is only partially observable; this is tacitly handled using the 3 frames of LIDAR data available to the agent.

State $S_{robot}^{(t)}$	Relative goal pose	$\left(x_{goal}, y_{goal}\right)$
	Velocity	$\left(v_x, v_y\right)$
	Length of 64 LIDAR rays (3 frames) Minmax normalized from $[0, 4]$ to $[-1, 1]$	$\left[l_0, l_2, ..., l_{63}\right] \times 3$ Frames
Action $A_{robot}^{(t)}$	New velocity Bounded to $[-1.5, 1.5]$	$\left(a_x, a_y\right)$
Reward components $\left(\theta_1, ..., \theta_6\right)$	Goal distance	d_{goal}
	Arrival at goal Boolean	$d_{goal} < 0.5$
	Angle difference of current velocity and previous action Bounded to $[0, \pi]$	α
	Clearance Bounded to $[0, 4]$	$d_{clearance}$
	Cumulative number of collisions	$N_{collision}$
	Constant term	1
Reward $R_{robot}^{(t)}$	A weighted combination of reward components	$\sum_{i=1}^{6} w_i \theta_i$
True objective Obj	Arrival at goal discounted by the cumulative number of collisions	$\frac{\left(d_{goal} < 1\right)}{0.5 N_{collision} + 1}$

State variables and reward parameters are already available in the simulator or transformed from existing variables. Clearance is approximated by extra LIDAR rays. Collision is determined by the underlying pybox2D simulator.

A parametrized reward function, based on an array of relevant components, is used in training in place of the true objective. The function 1) provides a dense signal throughout the episode and 2) is tailor-made to introduce desired agent behaviour (see below).

With reward signals now available at every step, there is no need to guarantee that the robot reaches the goal at the end of the episode. Therefore, fixed episode length is preferred for a more straightforward pipeline.

Project adjustments to the RL problem

Simultaneous simulation of multiple robots

The first major change in setup is the simulation of multiple robots in the simulator. While the original proposal has one single robot running in the simulator, we have multiple arguments supporting the simulation of multiple robots.

- Simulation is the rate-limiting part of the whole training pipeline. This is because the computation loads of one simulation step are heavier than that of one optimization step logically; furthermore, in between running one robot each in multiple simulators and running multiple robots in one simulation, the latter should represent less overhead cost.
- The presence of other robots as moving obstacles closely resembles the actual use case. In terms of frequency, a small but significant portion of steps involes moving obstacles. Therefore, the task itself might not be comprehensively harder to learn, although the agent would only learn how to handle moving obstacles in a statistical average sense.
- Relatedly, multi-agent interactions are taken into account naturally and coherently in simulation. The interaction dynamic does not need to be derived from single-agent behaviour obtained prior.
- There is perhaps some added resistance to divergence.

It should be emphasized that we still have a single agent RL problem by design. This is because no features providing extra information (frame-by-frame global state of all robots, specifically) is added to the simulation in this project.

Emphasis on collision

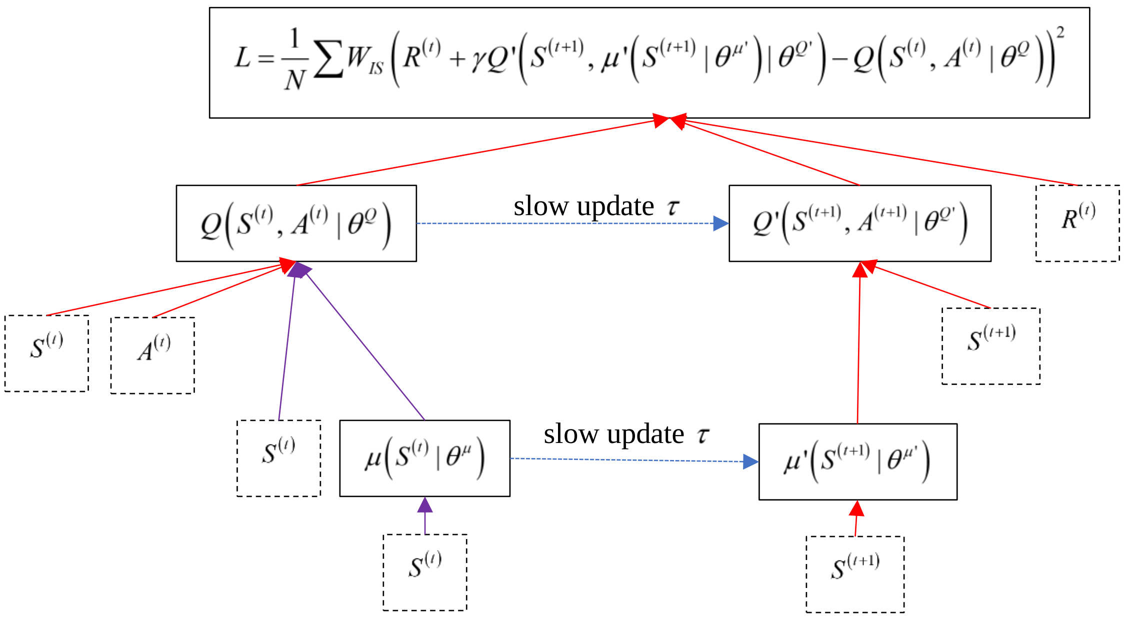
The second change is the strong emphasis on collisions. The original approach only considers collision as a binary variable in the reward function, with no direct impact on the true objective. Here collision is considered in both reward function and true objective as a numerical variable of the cumulative number of collisions.

RL agent definition

For practical reasons such as accountability and predictability, a deterministic policy is preferred over a stochastic policy for robots moving around autonomously.

The original approach has used an implementation of Deep Deterministic Policy Gradient (DDPG) [3] out-of-the-box from TF-Agent and demonstrated its feasibility. This has left plenty of room for future modifications, and since the simulation of multiple robots has necessitated sweeping changes already there is no reason to also perform some preliminary tuning along the way. The final design is best explained by building up the pipeline from the core DDPG algorithm.

A high-level computation graph of DDPG is shown below.


$L = \frac{1}{N}\sum W_{IS}\left(R^{(t)} + \gamma Q'\left(S^{(t+1)}, \mu'\left(S^{(t+1)} | \theta^{\mu'}\right) | \theta^{Q'}\right) - Q\left(S^{(t)}, A^{(t)} | \theta^{Q}\right)\right)^2$
$Q\left(S^{(t)}, A^{(t)} | \theta^{Q}\right)$
slow update τ
$Q'\left(S^{(t+1)}, A^{(t+1)} | \theta^{Q'}\right)$
$R^{(t)}$
$S^{(t)}$
$A^{(t)}$
$S^{(t+1)}$
$S^{(t)}$
$\mu\left(S^{(t)} | \theta^{\mu}\right)$
slow update τ
$\mu'\left(S^{(t+1)} | \theta^{\mu'}\right)$
$S^{(t)}$
$S^{(t+1)}$

Given a (batch of) sampled one-step trajectory of format $\left(S^{(t)}, A^{(t)}, R^{(t)}, S^{(t+1)}, A^{(t+1)}, R^{(t+1)}\right)$,

- The critic network Q is first updated by gradient descent using $S^{(t)}$ and $A^{(t)}$ as input. The update target is bootstrapped using the target network Q' and μ'; the subsequent action $A^{(t+1)}$ is not used and the update is off policy.
  There is no importance sampling correction for using one-step state-action value bootstrap nominally. Instead, the importance sampling weights W_{IS} comes from prioritized experience replay (PER, see below).
  This part corresponds to the red pipeline
- The actor network μ is then updated by gradient descent. The gradient of the value with respect to action $\nabla_a Q$ is computed, then further backpropagated to parameter θ^{μ}: $\nabla_{A=\mu\left(S^{(t)}|\theta^{\mu}\right)} Q\left(S^{(t)}, A | \theta^{Q}\right) \nabla_{\theta^{\mu}} \mu\left(S^{(t)} | \theta^{\mu}\right)$.
  The gradient of a batch is then a simple average. Policy gradient implementations typically ignore state-visitation distribution when computing update given states already, and it is unclear if there is a need to correct for sampling.
  This part corresponds to the purple pipeline.
- The target network Q' and μ' parameters are slow updated.
  This part corresponds to the blue pipeline.

Project adjustments for the RL agent

Exploration

Since DDPG is off-policy and the actor network directly output an action, exploration can be achieved by simply adding a randomization component to the action. The Ornstein–Uhlenbeck process is used in DDPG, and the parameters are tested to make sure the added exploration is reasonably scaled relative to the action range.

DDPG hyperparameters

The initial choice of network architecture was less significant in the original approach when it is also optimized for after the best reward function is found. Since it is known from the beginning that there would likely not be enough time and resources for a second search, a network architecture slightly larger than the reported optimal size is used throughout the project. The update rate is then chosen by finding the largest value that does not lead to divergence over a few different trials.

Prioritized experience replay

Experience replay is used in many deep off-policy reinforcement learning methods. Gradient descent with mini batch on neural networks usually assumes that the training batches are i.i.d, which is not the case for consecutive trajectories. To generate training batches that are closer to being i.i.d, trajectories are inserted into a replay buffer and training batches are sampled from the replay buffer. Data efficiency is also improved by reusing data.

However, this basic buffer does present other practical concerns. As learning progresses, the agent and the trajectory distribution shift towards convergence while generating more trajectories. It is not possible to keep every trajectory indefinitely and evicting trajectories FIFO can cause catastrophic forgetting due to distribution drift, so larger buffer size is frequently used. The larger the buffer, the more diluted newly added trajectories are, which dampens learning.

The core approach here is Prioritized Experience Replay (PER) [4], and the rank-priority variant has been applied to DDPG before [5]. Each trajectory is now associated with a priority p and the probability of sampling is proportional to a power of the priority p^{α}. The priority here is obtained by bounding the absolute TD-error (derived from the first pipeline) to a predefined range; each new trajectory is assigned the highest priority value currently in the buffer instead. The upper bound on priority is necessary to prevent divergence given that TD-error is unbounded.

Disrupting the sampling probabilities can still lead to divergence for value function approximations, which is why importance sampling is involved. The importance sampling weight is inversely proportional to the priority of the trajectory. To further stabilize training, the importance sampling weight is scaled down globally so that the least likely sampled trajectory has an importance sampling weight of 1. Trajectories with high priorities (i.e. sample with large gradient from high TD-error) are selected more frequently, and the stabilization provides valuable control on learning rate. Still, this importance sampling weight can be somewhat relaxed early on to speed up training. The importance sampling weight is then expressed as $\left(\frac{p_{min}}{p}\right)^{\alpha\beta_t}$, where β_t increases from an initial value β_0 to 1 as the buffer fills up. The lower bound on priority is then here to guarantee the importance sampling weights would not cause learning to grind to a halt.

Ratio of environment step to training step

In each training iteration, multiple training batches are sampled and passed to DDPG for each simulation step. The amount of training done overall has been increased; this is achieved by making sure the final ratio leads to higher CPU utilization but not slow down of simulation speed.

The final extended algorithm is shown below

```
DDPG with TD-error-based Prioritized Experience Replay

Initialization

1. Initialize network weights $\theta^{Q}$ and $\theta^{\mu}$ of the critic network  and actor
   network $\mu\left(S^{(t)} | \theta^{\mu}\right)$
2. Replicate $\theta^{Q} \to \theta^{Q'}$ and $\theta^{\mu} \to \theta^{\mu'}$ into the target $Q\left(S^{(t)}, A^{(t)} | \theta^{Q}\right)$critic network
   $Q'\left(S^{(t+1)}, A^{(t+1)} | \theta^{Q'}\right)$ and target actor network $\mu'\left(S^{(t+1)} | \theta^{\mu'}\right)$
3. Initialize current maximum priority $p_{max}$ and minimum priority $p_{min}$
4. Initialize prioritized replay buffer $B$ with one episode of trajectory
   $\left(S^{(t)}, A^{(t)}, R^{(t)}, S^{(t+1)}, A^{(t+1)}, R^{(t+1)}\right)$ driven by bounded uniform random policy

Iteration

1. For step $t$ from 0 to $N_{iteration}-1$
2.     If $start\_episode$
5.         Initialize exploration noise $\varepsilon_r^{(t)}$ for all robot $r$.
3.         Observe initial state $S_r^{(t)}$ for all robot $r$.
4.     Select action $A_r^{(t)} = \mu\left(S_r^{(t)} | \theta^{\mu}\right) + \varepsilon_r^{(t)}$ for all robot $r$.
5.     Send action $A_r^{(t)}$ to simulator for all robot $r$.
6.     Receive reward $R_r^{(t)}$ and next observation $S_r^{(t+1)}$ for all robot $r$.
7.     Pack $S_r^{(t)}$, $A_r^{(t)}$ and $R_r^{(t)}$ into $\left(S_r^{(t)}, A_r^{(t)}, R_r^{(t)}\right)$ for all robot $r$.
8.     For all robot $r$
9.         Insert $\left(S_r^{(t)}, A_r^{(t)}, R_r^{(t)}\right)$ into buffer writer memory $B_{mem}$
10.        If $\left|B_{mem}\right| = N_{robot}+1$
11.            Build $\left(S_r^{(t)}, A_r^{(t)}, R_r^{(t)}, S_r^{(t+1)}, A_r^{(t+1)}, R_r^{(t+1)}\right)$ from $\left(B_{mem}\right)_0$ and $\left(B_{mem}\right)_{N_{robot}}$.
12.            Insert $\left(S_r^{(t)}, A_r^{(t)}, R_r^{(t)}, S_r^{(t+1)}, A_r^{(t+1)}, R_r^{(t+1)}\right)$ into $B$ with priority $p_{max}$
13.            Remove $\left(B_{mem}\right)_0$
14.    For each training step
15.        Sample $K$ transitions $\left(S^{(l)}, A^{(l)}, R^{(l)}, S^{(l+1)}, A^{(l+1)}, R^{(l+1)}\right)_k$ at $P = \frac{p_k^{\alpha}}{\sum p^{\alpha}}$
16.        For trajectories $k$ from 0 to $K-1$
17.            Compute bootstrap target $y_k = R_k^{(l)} + \gamma Q'\left(R_k^{(l)}, \mu'\left(S_k^{(l+1)} | \theta^{\mu'}\right) | \theta^{Q'}\right)$
18.            Compute bounded absolute TD-error $\delta_k = \left|y_k - Q\left(S_k^{(l)}, A_k^{(l)} | \theta^{Q}\right)\right|$
19.            Update priority $p_k$ of $\left(S^{(l)}, A^{(l)}, R^{(l)}, S^{(l+1)}, A^{(l+1)}, R^{(l+1)}\right)_k$ in $B$
20.            Update maximum priority $p_{max}$ and minimum priority $p_{min}$
21.        Minimize loss $L = \frac{1}{K}\sum_k W_k \delta_k^2$ wrt $\theta^{Q}$ with weight $W_k = \left(\frac{p_{min}}{p_k}\right)^{\alpha\beta_t}$
22.        Maximize $Q\left(S_k^{(l)}, \mu\left(S_k^{(l)} | \theta^{\mu}\right) | \theta^{Q}\right)$ wrt $\theta^{\mu}$;
```

Training trial

It is now possible to summarize the entire training trial for one instance of the reinforcement learning problem (parameters listed in Appendix). In initialization, the replay buffer is filled with 1 episode worth of trajectories driven by a random policy since we have a larger than 1 sample to collect ratio for trajectories. For each iteration in the training trial, the environment is stepped forward 1 step using the policy with exploration added. The batched trajectory of all robots is then separated into individual trajectories and inserted into the replay buffer. The training batch is then sampled for DDPG training, during which priority of the training batch is updated.

Since different weights are introduced, the loss function naturally fluctuates and so a validation session is conducted every 10% of training trial completed. The policy without exploration is used to run the environment for a total of 10 episodes, and the average return is logged.

At the end of the training trial, the trained policy is evaluated. To accurately assess the agent, the evaluation is done in a different map using the true objective as reward. The evaluation includes multiple episodes where each episode is given longer time to amplify the signal of success. The evaluation result is then associated with the reward parameters of the trial.

Selection of the best RL problem formulation (reward function)

The key challenge in adding extra layer of optimization on top of RL – to find the reward function that would lead to the most performant agent – is the linear scaling of computation cost. An efficient, adaptive search strategy is critical for the feasibility of the whole approach. The project will use CMA-ES [6], which was used in [2] in the form of a black-box optimization service on Google Cloud's AI Platform.

At a very high level, CMA-ES is a search strategy that evolves depending on previous results – CMA-ES samples a population of reward function parameters, and the evaluation result of the trained policy is returned to CMA-ES. CMA-ES attempt to maximizing the likelihood of finding successful candidates in offspring populations and has step-size control to adjust the directions and magnitude of the adaptation.

The original approach has 100 instances of RL problem in training at the same time, adding up to 1000 trials. In the project, the default population size for a 6-dimensional search space, which is 9, is used. The total number of trials depends on how fast computational resources run out.

As a final note, CMA-ES works best when the scale of each dimension of search space is comparable. Given that each of the components in the reward function has a different range, there is a predefined transformation for each reward parameter, so the final reward function is not dominated by some of the components.

Rapidly-exploring Random Tree

The waypoint planner to be used alongside the P2P policy is the rapidly-exploring random tree algorithm (RRT). It is possible to control how close waypoints are to each other when the random tree grows (by sampling the map). Furthermore, in the RRT* variant [7], there is an additional parameter controller the range of rewind, where the newly added node is added to the tree minimizing the path length from root instead of the node sampling is based on. Both parameters affect the final separation of waypoints, and they are adjusted manually based on observed performance of the policy.

Implementation Adaptations

Dorabot-minions

Since robots in the simulator are defined by a state-machine involving other tasks, it is necessary to modify the simulation step logic. Other tasks like loading are not involved in the P2P task, so the simulation logic is condensed to the following loop

- Place each robot in a new randomized location, chosen with simulation functions.
- Generate P2P tasks for each robot based on their current location.
  The destination is chosen to be a free spot of 3 – 5 unit distance away.
- In each step, the simulator gathers LIDAR data for each agent:
  - At first step of every twelve steps, the simulator will send the robot states to the agent, which will return actions for each robot.
  - For subsequent steps, the simulator will repeat the action received.
- After fixed number of steps, the simulator reset the agent, including the local planner, before starting a new loop. The reset is necessary, so the simulator and the agent are not out-of-sync (i.e. the relative order of sending/ receiving observations/ actions).

Moreover, the original simulation step logic iteratively considers each robot. The first episode is processed in the order of $\left[S_{agent_1}^{(0)} A_{agent_1}^{(0)} S_{agent_2}^{(0)} A_{agent_2}^{(0)} S_{agent_3}^{(0)} A_{agent_3}^{(0)} S_{agent_4}^{(0)} A_{agent_4}^{(0)} \right]$. This deviates from the usual batch processing order $\left[S_{agent_1}^{(0)} S_{agent_2}^{(0)} S_{agent_3}^{(0)} S_{agent_4}^{(0)} A_{agent_1}^{(0)} A_{agent_2}^{(0)} A_{agent_3}^{(0)} A_{agent_4}^{(0)} \right]$. The simulator is modified to accommodate the batch definition to keep the environment wrapper and replay buffer writer on the agent side simple.

There are two simulator processes (one being idle) – training and evaluation – because each simulator loads a predefined map.

TF-Agent and Reverb

In this project, TF-Agent [8] is used to the reinforcement learning pipeline. The main building blocks are the environment wrapper, driver and the DDPG agent. TF-Agent has a standardized format for storing trajectories as tensors and an environment wrapper is needed as the communication interface between the exploration policy and the simulator. These functions are called drivers, where it is possible to specific what happens when the state and action are sent back-and-forth. Finally, the DDPG agent handles the 3 pipelines within the algorithm.

Since the simulator is Python 2 based and TF-Agent is Python 3 based, they are run on separate processes and message passing connections at specific ports are used for duplex communication. Since there are two simulator processes, there are also two environment wrappers with matching reward definitions. Each wrapper is responsible for combining information of every robot into a batch.

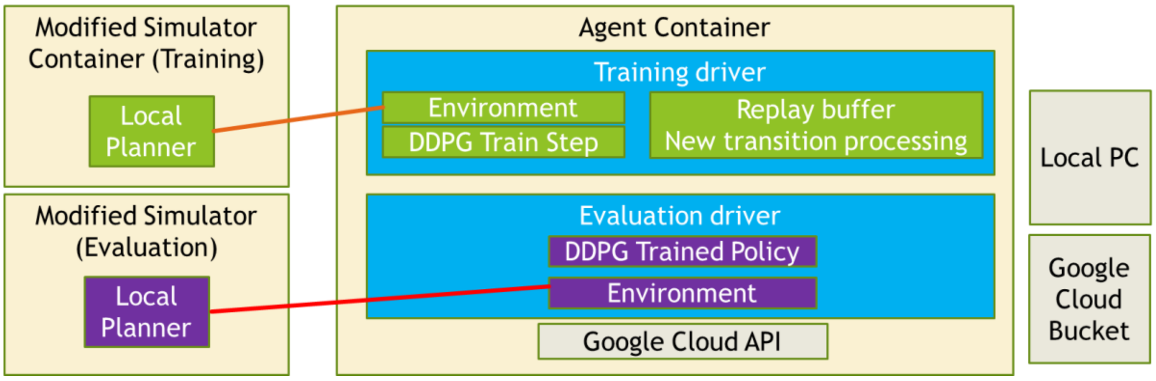
Reverb [9] is a fast, memory-based replay buffer that supports priority experience replay, and it is through the driver interface that the splitting of batched trajectory is executed.

The implementation of DDPG is missing certain features. Ad-hoc modifications are made on top of the agent framework conforming to TensorFlow requirements.

- Regularization
- Elementwise TD-error
- Independent weights for actor and critic network.

Docker

Deploying multiple instances of the RL pipeline for CMA-ES is expedited by running deploying 3 containers (one for DDPG and two for simulators) as a single application with Docker Compose, in which the Docker Network functionalities ensure the message passing connections are consistent no matter where the containers are deployed.



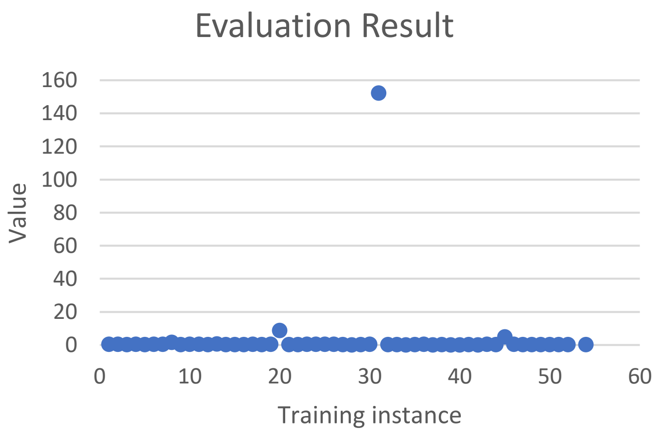
Google Cloud Platform

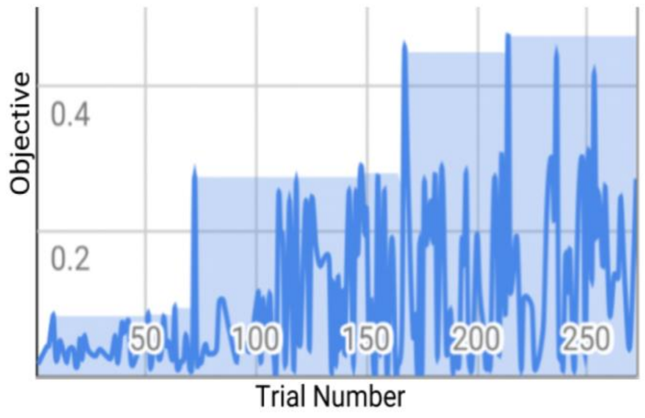
The containers are deployed on VMs on Google Cloud Platform (with free credits). The VM size is N2-standard-2 with extra memory (2 vCPU and 12 GB memory) running the Container-optimized OS version COS 93 LTS. This combination is picked to get the most performance per compute resources (Container-optimized OS has its own limitation). In total 9 VM instances were launched on GCP Compute Engine, and CPU utilization is consistently about 65% during training and 50% during evaluation. One trial takes roughly 23 hours if there is no interrupt.

In the TF-Agent container, there are extra boilerplate code for downloading reward parameters from GCP Cloud Bucket before starting the pipeline. DDPG agent checkpoint is also backup to Cloud Bucket after each validation section. Finally, evaluation result and the trained policy is also stored on Cloud Bucket. CMA-ES is run locally and regularly monitor the Cloud Bucket for evaluation results.

Result

54 instances (6 generations) in total were run. The maximum possible average return using the true objective is 500 (number of steps in one evaluation episode), and the best policy achieved 152 on training instance 31 (left plot).





The plot on the right is the corresponding plot from [2], and the leftmost part of the plot mostly agrees with the evaluation result here distribution-wise.

Deployment of the policy on a clean copy of dorabot-minions is referenced in subsequent discussion.

Discussion

Drawback of reshaping simulator logic

Multiple changes to different components were made in the simulator to accommodate the training of P2P control. After reviewing the deployment result and the poor performance in open space/ near ports, it was discovered that the brute-force modifications are too naïve.

- The P2P tasks defined in the simulator only represent a proper subset of all P2P task. P2P task in queuing zone is qualitatively different, and in general the map used for training and evaluation does not have large open spaces.
- A built-in simulator method is used to generate a list of valid reset position of agent, as sampling the map directly and checking for obstacles is very inefficient. However, the list of positions generated is later discovered to be not representative (e.g. excluding the top and bottom areas; only include grid point but not centre of grid).
- Oversight in not randomizing initial orientation and velocity.
- The built-in map generation produces map that have more uniform level of obstruction, and polices do not translate well to maps with very different degree of obstruction.

Definition of state

This is likely the worst mistake in the whole project. In the original approach, the LIDAR rays are all around the robot and the array of LIDAR ray length is arranged by absolute orientation, independent of robot velocity. This is clearly not the case here, and there are at least two repercussions:

- Orientation information is only present in the velocity input, and it will take more layers in the MLP for orientation information to be incorporate with LIDAR ray lengths.
- The old LIDAR ray lengths has no reference orientation at all and contains very little information.

There were plans initially to compute the LIDAR contact points instead of just ray lengths, which would have addressed this problem.

Collision

The cumulative number of collisions is considered to produce a heavier penalty when two robots collide. However, with the high frame skip (60 Hz/ 5 Hz = 12) accumulating collision count at every frame could produce very high cumulative number of collisions, which was not properly scaled for.

PER

An approximation to the minimum and maximum priority is used, since at the time of implementation no method of the replay buffer for querying the maximum and minimum priority is found in documentation or source code. On the other hand, the maximum and minimum priority might actually be a quantitative tool for comparing trials with different reward parameters.

Training trial

It is clear from fluctuating validation result in training instance logs that training has not completed. 4 robots running 1 million environmental steps is likely not comparable to 1 robot running 5 million environmental steps, as in the original approach.

Evaluation quality

Evaluation inherits problems regarding generation of P2P task, map features, etc. Also, the true objective has used a larger goal size than the reward function, and the best policy selected would in fact stop some distance away from the next waypoint. This is handled when the policy is deployed by changing waypoint earlier. Given how much variance is in the evaluation result, perhaps more evaluation episode would improve the performance of CMA-ES.

CMA-ES

It might be more productive to start optimizing for network architecture once a set of reward parameters leading to significantly better policy is found.

Deployment

When the final policy is deployed, its performance is improved by also considering the reflection of the state along the robot's orientation (velocity). Effectively, the relative goal position is reflected across the velocity vector, and the LIDAR length array is reversed. The action output by the policy on the 'reflected' stated (after being reflected again) can be combined with the actual action to filter out some divergence behaviour.

Conclusion

This project demonstrates, among other things, the importance of thorough definition and attention to the precise behaviour of the actual environment when reinforcement learning is applied to realistic scenarios. The constraints has overall pushed for a much deeper understanding of policy gradient methods and practical pitfalls. It is a great experience setting up a model with more complex pipeline and having at least one training instances producing a half-decent policy is a surprise.

Reference

[1] Bellemare, M. G., Naddaf, Y., Veness, J. and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47 (2013), 253-279.
[2] Chiang, H.-T. L., Faust, A., Fiser, M. and Francis, A. Learning navigation behaviors end-to-end with autorl. *IEEE Robotics and Automation Letters*, 4, 2 (2019), 2007-2014.
[3] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).
[4] Schaul, T., Quan, J., Antonoglou, I. and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).
[5] Hou, Y., Liu, L., Wei, Q., Xu, X. and Chen, C. *A novel DDPG method with prioritized experience replay*. City, 2017.
[6] Hansen, N. and Ostermeier, A. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9, 2 (2001), 159-195.
[7] Karaman, S. and Frazzoli, E. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30, 7 (2011), 846-894.
[8] *TF-Agent: A library for Reinforcement Learning in TensorFlow* City.
[9] Cassirer, A., Barth-Maron, G., Brevdo, E., Ramos, S., Boyd, T., Sottiaux, T. and Kroiss, M. Reverb: A framework for experience replay. *arXiv preprint arXiv:2102.04736* (2021).

Appendix

DDPG

Shape of critic network Q observation layers	$(200, 50)$
Shape of critic network Q joint layers	$(50, 50)$
Shape of actor network μ	$(200, 200, 50)$
L2 regularization	10^{-3}
Optimizer	Adam (learning rate 10^{-3})
Discount factor	0.99
Target update τ	0.01
Exploration (Ornstein-Uhlenbeck process)	$\theta = 0.2$ (friction/ dampening) $\sigma = 0.15$ (deviation scale)

PER

Buffer size	10^6
Priority bound	$[10^{-6}, 1]$
Prioritization power α	0.6
Initial importance sampling strength β_0	0.5

Training trial

Number of iteration	1000000
Number of environment step per iteration	1
Number of training step per iteration	4
Training batch size	8
Number of validation session per trial	10
Number of environment step per validation session	2500
Number of environment step in evaluation	50000

Note: Episode length during training: 250 steps; Episode length during evaluation: 500 steps

RRT*

Radius for sampling	1.6
Radius for rewind	2