Avoiding Obstacles and Navigating to a Goal in Unity Using Deep RL

Cong Yuan
Department of Computing Science
Simon Fraser University
cong_yuan@sfu.ca

Luciano Oliveira
Department of Computing Science
Simon Fraser University
loliveir@sfu.ca

Oleksandr Volkanov
Department of Computing Science
Simon Fraser University
avolkano@sfu.ca

Yu Guo
Department of Computing Science
Simon Fraser University
yu_guo_4@sfu.ca

***Abstract*—Safe navigation of a robot to a designated position through a complex environment with buildings and walking humans is useful to many real life situations. A robot equipped with our model is shown to be able to successfully navigate to any assigned position without hitting randomly placed stationary obstacles and moving human agents. As a proof of concept, we design a virtual environment using Unity game engine, and train a deep reinforcement learning model using the ML-Agents framework.**

I. Introduction

There has been a high interest for robotic autonomy in the recent years. People are always fascinated to see machines perform tasks that were believed to be only accomplishable by intelligent biological agents. A while ago, autonomous driving technology seemed liked a distant dream. The automation agents have also been extremely useful in assisting us in numerous daily activities and jobs, such as high-precision surgery, laboratory research, manufacturing, space exploration [1] etc. With the recent development of drones and self-driving cars [2], the motivation for designing an autonomous agent that can navigate in an unknown environment has only increased.

In this paper, we aim to contribute by training a robot agent, inside a Unity game engine simulation, to reach a specific target location while avoiding stationary obstacles and moving human agents. The learning could not be done via supervised learning because it requires pairs of input and output. In this case, there are many sets of valid solutions to a specific input scenario rather than a single optimal ground truth. Therefore, a network designed with a reinforcement learning model is the best way to handle this problem. With this approach, the agent can learn by itself what to do in any scenario with a different number of obstacles at unknown starting positions. Thus, we firstly care about the robot reaching a specific target location while avoiding any kind of collisions. The path and time it takes is not the main concern, so the problem is better stated by considering the final state and collisions for our loss instead of any optimal route.

The problem has been handled in a way that could be easily exported to a real world environment. First, we use a Robot Operating System (ROS) to control the robot's movement, so that it could be easily imported to a real robot and integrated with its other tasks. Moreover, the robot agent does not have access to the world space coordinates of any obstacles, and instead relies on its observation modules to navigate the space. For example, we use a set of rays shooting out of the robot in different directions. If these rays intersect with anything before its range limit, then we get the distance to such intersections. This mechanism mimics the behaviour of Light Detection and Ranging (LIDAR), that shoots pulsed lasers in a similar manner. If these lasers reach any object, then the light is reflected back to the receiver, and based on the time it takes to travel to an object and go back, it is possible to measure the distance to the object. Finally, we use the relative position of the target location to the robot. Thus, all robot agent observations could be directly applied to a real world environment.

II. Related Work

There are mainly two approaches for deep reinforcement learning problems, the methods based on value functions like DQN [3], and the methods based on policies, such as PPO and DDPG. Deep Q-learning has many successful examples to the robot control problems, but the problem is that Q-learning is designed to solve the discrete-time decision problem. On the other hand, Deep Deterministic Policy Gradient (DDPG) [4] has also became increasingly popular in the robotic control problems because their good performance. However, the DDPG performed not as good as Proximal Policy Optimization (PPO) [5] in the real world robot tasks.

III. Experiment Setup

For the purpose of this project, we have created a virtual environment using Unity game engine. We use a TurtleBot3 robot agent prefab that can be found in UnityRos2 GitHub repository [6]. First, we create a square playground with wall boundaries. In the beginning of each experiment, a preset number of stationary obstacles (e.g. cubes, cylinders) and moving human agents are generated at random positions in

the scene. The TurtleBot3 will start at the bottom middle of the scene and navigate to a randomly placed final goal. If the TurtleBot3 reaches the goal without hitting walls, human agents or stationary obstacles, a new goal will be generated at a random position as the next target for robot to continue to navigate. However, if the TurtleBot3 hits anything before reaching the goal, the entire experiment will be restarted. The detailed setup of each component of our experiment can be found below.

A. Robot Agent

The TurtleBot3 is a small mobile ROS standard platform robot equipped with distance sensor. The height of the TurtleBot3 is 192mm, the width is 178mm, and the weight is 1kg. In total, it has three wheels. Two large wheels located on the side are equipped with DYNAMIXEL, and the other small wheel is located at the middle front. In our experiment, we use ROS interface to send the robot commands that adjust its linear and angular velocities.

B. World Environment

At first, we have experimented using a large 20x20 unit environment with many stationary obstacles and moving human agents. However, the training was considerably slow, especially at the beginning. As we start training, the TurtleBot3 moves in random directions, and we reset the episode every time a collision occurs. For that reason, it was taking too much time for the robot to occasionally arrive at the goal location, and, consequently, learn how to reach it. Therefore, in order to simplify and speed up the training, we decided to use a smaller 10x10 unit environment.

Our final environment consists of a small squared room of size 10x10 bounded by walls. For the stationary obstacles, we have two kinds of cubes that are generated at random. One of the cubes is relatively large, with a size of 1x1. And the other cube has size 0.5x0.5. During the training, we generate a total of 8 stationary obstacles with different combinations of large and small cubes, and a total of 2 autonomous human agents. Moreover, we have assigned different tags inside Unity for the stationary obstacles, walking human agents, and walls. This way, when the LIDAR detects an object in the scene, it also gets the tag of the object. Differentiating between distinct types of objects is beneficial for the network to learn what to do based on what obstacle lies ahead. For instance, it can learn to turn around when it sees a wall, while also learning how to avoid other obstacles in a specific manner.

C. Human Agents

When designing the human agents for our scene, the main idea was to simulate the dynamic behaviour of a human crowd, where each person moves independently to an unknown target. Since the location of the static obstacles is random during each training episode, we could not use predefined checkpoints for the human agent navigation. As a result, we had to derive a routine that can control each agent independently and make them avoid the randomly generated obstacles. For this purpose, we propose the $HumanAgentRoutine(H)$ that implements the world environment navigation logic of each human agent H.

Algorithm 1 $HumanAgentRoutine(H)$

Initialize the agent's target goal position $H_{target} \leftarrow GetNextTargetPosition(H)$.
Initialize the agent's boolean lock variable $H_{lock} \leftarrow$ **false**.
while true do
  $MoveToTarget(H)$
end while

Algorithm 2 $GetNextTargetPosition(H)$

Require: Initialize ray cast hit distance threshold $D_{th} > 0$.
Require: Initialize maximum ray cast hit distance $D_{max} = 0$.
Shoot N rays 360 degrees around the agent H, with ray cast hit distance D and ray cast hit target T.
for all D, T **do**
  if $D > D_{max}$ **then**
    $D_{max} \leftarrow D$, $T_{best} \leftarrow T$
    if $D > D_{th}$ **then**
      return T_{best}
    end if
  end if
end for
return T_{best}

Algorithm 3 $MoveToTarget(H)$

Require: Initialize ray cast hit distance threshold $D_{th} > 0$.
Require: The agent's boolean lock variable H_{lock}.
Require: The agent's current position $H_{current}$.
Require: The agent's target goal position H_{target}.
if $distance(H_{current}, H_{target}) < D_{th}$ **then**
  $H_{target} \leftarrow GetNextTargetPosition(H)$
end if
Shoot 1 ray in front of the agent, with ray cast hit distance D and ray cast hit target T.
if (**not** H_{lock}) **and** $D < D_{th}$ **then**
  $H_{target} \leftarrow GetNextTargetPosition(H)$
  $H_{lock} \leftarrow$ **true**
else
  $H_{lock} \leftarrow$ **false**
end if
Adjust the trajectory of the agent towards H_{target}.

IV. Reinforcement Learning Setup

For the reinforcement learning training setup, we chose to use ML-Agents Unity package. The training process is applying the Proximal Policy Optimization (PPO). In real world problems, convergence is always a big problem because the natural policy gradient has a second-order derivative matrix, which makes it hard for large scale problems, and usually ends

in a computationally expensive hell. However, PPO is using a different approach. Instead of having a hard coded constraint, PPO makes it as a penalty in the objective function. Therefore, we can use gradient descent method to optimize the objective. In that case, even we break the constraint sometimes, the damage done is less computationally expensive. As a result, the PPO algorithm is trying to compute an update for each step that minimizes the cost function and make sure that the deviation from the previous is also relatively small.

$$L_{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_1, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_1)]$$

- θ is the policy parameter.
- $\hat{E}_t$ denotes the empirical expectation over time steps.
- r_t is the ratio of the probability under the new and old policies, respectively.
- $\hat{A}_1$ is the estimated advantage at time t.
- ε is a hyper-parameter, usually 0.1 or 0.2.

Algorithm 4 *Proximal Policy Optimization(PPO)*

for episode = 0, 1...T **do**
  for iteration = 0, 1..., N **do**
    Run policy π_{old} in environment for T time steps;
    Compute advantage estimates $\hat{A}_1$,...$\hat{A}_T$;
  end for
  Optimize $L_{CLIP}(\theta)$ with respect to θ, with K epochs and mini-batch size $M < NT$;
  Update $\theta_{old} \leftarrow \theta$
end for

Our network is shown in Fig. 1 with the following parameters:

- 3 dense layers - the number of hidden layers in the neural network. Corresponds to how many hidden layers are present after the observation input, or after the CNN encoding of the visual observation.
- 512 hidden units - the number of units in the hidden layers of the neural network. Corresponds to how many units are in each fully connected layer of the neural network.
- A normalization is applied to the vector observation inputs. This normalization is based on the running average and variance of the vector observation.
- 3 input stacks x_1, x_2, x_3, x_1 refers to the actions of the robot (linear and angular velocity), x_2 is the relative position of the robot to the final goal, and x_3 is the ray cast information.
- 1 output stack y_1, y_1 refers to the actions of the robot (linear and angular velocity).

A. Observations

The robot AI agent has the following observations during the model training and inference processes:

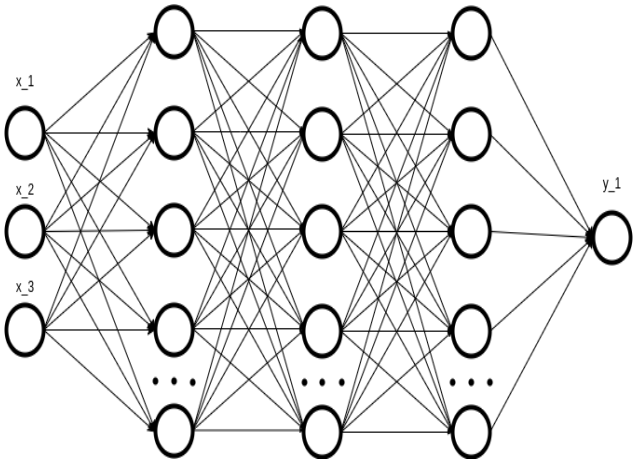

Fig. 1. Network Architecture

- A Vector3 object referencing the current relative position of the final goal inside the robot agent's coordinate system.
- A float value referencing the current distance between the robot agent and the final goal.
- A float value referencing the current angular velocity of the robot agent.
- A float value referencing the current linear velocity of the robot agent.
- A RayPerceptionSensor object representing the LIDAR scan of the environment, with a field of view of 120 degrees and the maximum range of 5 units (in a 10x10 environment). This module can differentiate between the types of obstacles and the final goal.

We have trained two model variants (simple and enhanced) to examine the impact of having a history of observations on the robot agent's performance. Both model variants utilize the last 10 vector space observations to represent a limited memory of the robot agent's actions and the progress to the final goal. For the LIDAR module of the robot agent, the simple model is using 10 rays with no ray-cast stacking, while the enhanced model is using 20 rays and a ray-cast stack of size 10 to represent a limited memory of the environment.

B. Actions

The robot agent is allowed to perform the following actions during the model training and inference processes:

- Change its angular velocity that controls the robot agent's rotation around the global Y-axis. in the [-1, 1] range.
- Change its linear velocity that controls the robot agent's forwards and backwards movement. in the [0.5, 1] range.

The angular velocity values are restricted to the [-1, 1] range, giving the robot agent a full control of turning left or right. The linear velocity values are restricted to the [0.5, 1] range, meaning the robot agent is always moving forward and has to mainly rely on turning to avoid the obstacles and reach the final goal.

C. Rewards & Penalties

The robot agent has the following positive reward signals:

- Reaching the final goal (1.0 reward value).
- Being closer to the final goal compared to the previous observation frame (0.01 reward value).

The robot agent has the following negative reward signals:

- Hitting the environment boundary (e.g. a wall) (-1.0 reward value).
- Hitting a stationary obstacle (e.g. a randomly placed cube or cylinder) (-1.0 reward value).
- Hitting a moving obstacle (e.g. a person) (-1.0 reward value).
- Being further away from the final goal compared to the previous observation frame (-0.02 reward value).

All reward values are accumulative. Hitting a boundary or an obstacle stops the training episode and resets the accumulated rewards, whereas reaching the goal resets the environment (i.e. the stationary obstacles and the final goal positions) while keeping the accumulated reward. This setup allows the robot to reach multiple consecutive goals for a bigger reward, while the ongoing final goal distance difference reward function encourages the robot agent to move towards the current final goal.

V. Results

We have trained both the simple and the enhanced models using an Nvidia GTX 1080 GPU over the span of 4 days (approximately 100 hours) in total. In Fig. 2, 3, 4, and 5 are the cumulative reward, episode length, policy loss, and value graphs, respectively, over 10 million training steps each. In addition, we have tested both models after the training process over 100 episodes each as shown in Table I, to see how well each model policy can avoid randomly placed obstacles and final goals. As previously mentioned, the number of stationary obstacles in each scene was set to 8, and the number of human agents was set to 2.

Looking at the cumulative reward values, we can can observe that the simple model is less consistent but performs better overall than the enhanced model in terms of collecting consecutive positive rewards. This observation is further supported by the overall longer average episode length of the simple model design, meaning that the robot is able to survive for longer periods of time and collect more rewards. Moreover, the empirical results in the inference results in Table I clearly show that the simple model reaches the final goal more times than the enhanced design, while avoiding the obstacles more efficiently. Interestingly, despite the simple model design performing better than the enhanced one, the value loss of the enhanced model appears to be lower during the whole training process, while the policy loss values appear to converge to the same value.

VI. Discussion

Compared to the simple model, the enhanced model features 20 ray casts (vs. 10 for the simple model) and a stack of 10

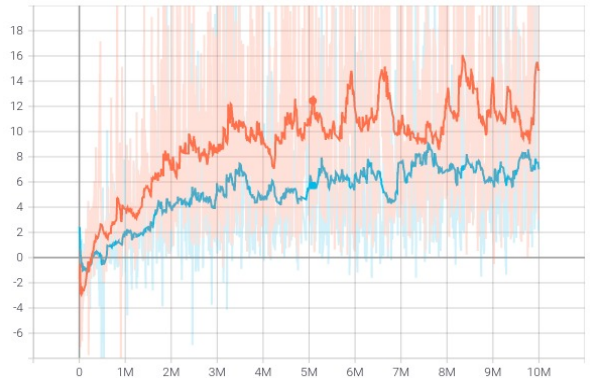

Fig. 2. Cumulative Reward vs. Epoch of Simple Model (Orange) and Enhanced Model (Blue)

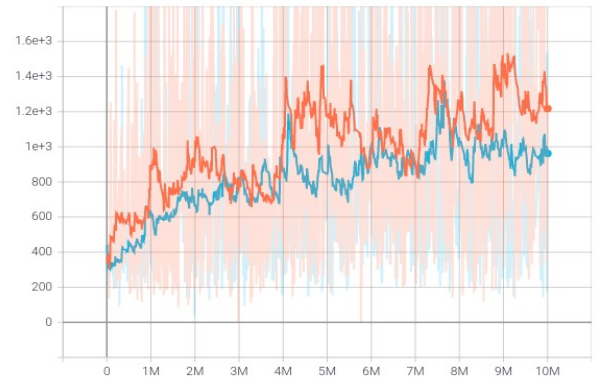

Fig. 3. Episode Length vs. Epoch of Simple Model (Orange) and Enhanced Model (Blue)

Fig. 4. Policy Loss vs. Epoch of Simple Model (Orange) and Enhanced Model (Blue)

TABLE I
Test Results of Simple Model and Enhanced Model after 100 Inference Episodes

Model	Reached Goal	Hit Boundary	Hit Stationary Obstacle	Hit Human Agent
Simple	83	1	6	10
Enhanced	61	1	25	13

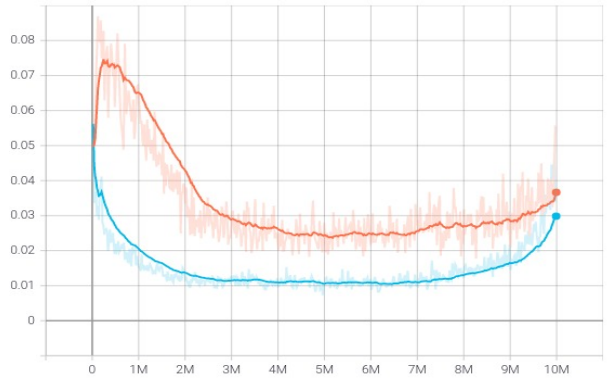

Fig. 5. Value Loss vs. Epoch of Simple Model (Orange) and Enhanced Model (Blue)

previous ray cast observations (vs. 1 for the simple model). The main idea of having more ray casts was to get a more fine-grained perception field, as well as introducing a primitive notion of spatial observation memory using the ray cast stacks without adding complexity of a recurrent neural network.

Despite having more spatial observation data as its input, the enhanced model has performed overall worse than the simple model design for our tasks. It appears that simply increasing the input parameters without adjusting the neural network architecture to handle additional input data had a generally negative effect. Corollary, the network architecture design requires careful scaling when increasing the perception field of the robot agent, which can explored as a follow-up research project. Nevertheless, we have shown that the simple model design can still achieve a reasonable performance in a procedurally generated scene with moving human agents.

It should be pointed out that the simple model appears to be better at avoiding the stationary obstacles (0.6 obstacle to human agent hit ratio), while the enhanced model is relatively better at avoiding the human agents (1.92 obstacle to human agent hit ratio). This difference can be attributed to the ray cast stack that allows the second model to be able to estimate the human agent's speed and direction from the previous human agent's positions. As such, we can attempt to find a middle ground between both model variants in the future work.

VII. Conclusion

Our results demonstrate that the task of navigating and avoiding obstacles in a complex scene can be achieved successfully with reinforcement learning. We have introduced two model designs with the simple variant outperforming the enhanced one. Even though the simple model has room for improvement, since TurtleBot3 can still occasionally hit human agents or stationary obstacles while navigating, we have demonstrated that our reinforcement learning setup was successful for the autonomous navigation task in a virtual environment. Both model designs could be further improved by continuing the training process, or by adjusting the model architecture and the reinforcement learning setup parameters.

VIII. Future Work

In our project, the robot agent can mainly observe its environment using its LIDAR sensor. As a follow-up, we believe that using a light-weight convolutional neural network, such as YOLOv3 [7], that processes the robot's front and side view camera outputs to detect the potential hazards (both stationary and moving objects) can be beneficial to the system. Furthermore, we can use another neural network to process the detected human video frames to estimate the human pose, velocity and direction, thus not having to rely solely on the LIDAR scan data.

IX. Group Member Contributions

The following tasks and responsibilities were split equally among all group members:

- Preliminary deep RL, Unity, ROS 2 research.
- Unity project setup and development.
- Unity deep RL setup and training.
- Poster design and editing.
- Report write-up and editing.

References

[1] Goel, A. (2019, March 27). 10 impacts of robots in everyday life. Retrieved April 23, 2021, from https://engineering.eckovation.com/10-impacts-robots-everyday-life.

[2] Drones, driverless cars, and A.I. (2017, September 06). Retrieved April 23, 2021, from https://permissionlessinnovation.org/drones-driverless-cars.

[3] Watkins, Christopher J., and Peter Dayan. "Q-learning." Machine Learning 8.3-4 (1992): 279-92. Print.

[4] Lillicrap, Timothy P., Hunt, Jonathan J., Pritzel, Alexander, Heess, Nicolas, Erez, Tom, Tassa, Yuval, Silver, David and Wierstra, Daan. "Continuous control with deep reinforcement learning.." Paper presented at the meeting of the ICLR, 2016.

[5] Schulman, John, Wolski, Filip, Dhariwal, Prafulla, Radford, Alec and Klimov, Oleg. "Proximal Policy Optimization Algorithms.." CoRR abs/1707.06347 (2017): .

[6] GitHub. 2021. DynoRobotics/UnityRos2. [online] Available at: ¡https://github.com/DynoRobotics/UnityRos2¿ [Accessed 22 April 2021].

[7] Redmon J, and Farhadi A. "Yolov3: An incremental improvement." arXiv:1804.02767v1, 2018.