Vision Based Obstacle Avoidance Through Deep Reinforcement Learning

I. Definition

Project Overview and Statement

Mobile robots have been widely used across different industries and sectors. It is crucial that the robot can navigate freely in an unknown environment, reaching its goal position while avoiding static and dynamic obstacles across its path. The fundamental problem of avoiding obstacles accurately and planning a path efficiently has been a hot topic for robotic research communities for decades. Many efficient algorithms have been proposed and implemented. However, obstacle avoidance is always closely linked to a path planning module, which normally decides the motion of the robot when knowing the position of the robot and obstacles. Obstacle detection is normally achieved by using range-sensing systems such as sonar, radar or time of flight sensors. However, systems like these require path planning and a sensing module to work together, which requires tuning of a large amount of parameters. Furthermore, range sensors can be heavy, bulky, power consuming and expensive to deploy. In addition, these types of systems are constrained within the environment they operate in, not benefiting from large datasets or continuous use. Compared to range-sensing systems, cameras encapsulate rich information of the environment and are low cost and low weight, suitable for a wide range of applications. Therefore, in this research proposal, I would like to explore vision based obstacle avoidance through deep reinforcement learning. The research aims to enable the robot to use camera images with an relative goal position to learn and understand its environment and quickly navigate to its goal position while avoiding obstacles.

Literature and Related Work

The standard framework for obstacle avoidance is generally decoupled into two steps. Firstly, the sensing system detects obstacles locations relative to the robot. Secondly, the path planning module works out the appropriate path to take. New obstacles are registered when traversing in the planned path and another route is planned. There have been various algorithms proposed over the decades [1][2] and each brings benefit and has its shortcomings such as run time, accuracy, path smoothness, efficiency. For example, algorithms such as the bubble rebound [3]could be applied on a low power low cost robot, but it is far from optimal, leading to unsmooth motions and a high failure rate. Overall, conventional model based methods require a certain amount of effort to fine-tune the parameters in order to reach a smooth level of operation. And yet these systems are constrained to the environment they operate in and are rarely able to adapt to new settings.

With the development of small and low cost cameras, image based approaches have been replacing conventional range sensing systems to an extent. Through camera images, obstacles and traversable areas are identified and path planner calculates accessible paths. Methods such as image segmentation, optical flow[4][5] or finding vanish points[6] are used to estimate the depth and the angle of obstacles. Nevertheless, despite the shortcomings of lots of parameter tuning and the lack of adaptability to new environments, image based methods are not as accurate or reliable as range sensors in avoiding obstacles.

Deep learning has proven to have great results in robotics and computer vision. Supervised learning has been used to achieve obstacle avoidance in several research projects. For example, one project uses neural networks to predict the depth from camera images and directly output control strategies without a path planner [7]. Another project forms end to end control policies just from raw images[8], eliminating the conventional two step process. However, the nature of supervised learning still means a huge amount of labelled data is required and scenes outside the training sets are mostly not recognised.

Reinforcement learning obtains the optimal policies by exploring an interactive environment through trial and error. Policies are evaluated based on rewards that the environment returns. One research project has proposed model based learning by training the collision prediction model to calculate the collision probability with uncertainty [9]. With the very recent advancements in deep reinforcement learning (DRL), an artificial intelligence agent could beat human experts in various games such as Go and Atari 2600 games. Building on this, DRL has been explored for robotic obstacle avoidance. There has been some promising research showing its potential. With robot positions tracked by adaptive Monte Carlo localization and obstacle positions obtained through LASER, one research group uses DRL to output control policies based on these two inputs to reach a goal [10]. This system shows good results when navigating in simple settings such as a small office environment. In another research project [11], predicted depth from the mono camera is used as input and the trained policy outputs one of ten actions, enabling the robot to run around in small loops while avoiding obstacles. However, it is repeating similar paths without a goal position to reach, and it was only trained in simulation then transferred to a real robot. In a different set-up another group uses a RGB-D camera to train control polices consisting of five moving commands [12],. Similar to the previously mentioned research, no goal position is given hence the robot only randomly explores the simulated environment and survives as many steps as possible.

Metrics

There are various metrics used to assess the performance of the agent and the algorithms. They are plotted into graphs using "Tensor Board" during the project. These metrics can be divided into three different categories, which are monitored during training and produced for all settings. Each metric is obtained over 1000 steps.

- Environment: cumulative reward, episode length

The mean cumulative episode reward indicates how well the agent is learning and the mean length of each episode indicates how the learning is progressing. During a successful training session, we should observe that the reward graph grows up steadily and the episode length graph reduces gradually.

- Learning loss function: policy loss and value loss

Policy loss is the mean magnitude of policy loss function, and it is correlated to how much the policy is changing. It is expected to decrease thought out the training session. The value function update shows how well the model is able to predict the value of each state. Therefore its loss function would increase while the agent is learning, decrease once the learning is coming to an end and the reward stabilizes.

- Policy: Entropy, learning rate, and value estimate

The entropy shows how random the decisions of the model are. It should decreases slowly during a successful training session. Learning rate shows the size of the step the algorithm takes as it searches for the optimal policy. It should also decrease over time. The last one is the mean value estimation for all states visited by the agent. It should increase for a success training process.

To have a direct comparison on agent's performance, the average rewards and standard deviation of rewards are calculated every 1000 steps. During one training session, we should find a steady increase of the average reward and a reduction on the standard deviation, ideally converging towards 0 near the end of the training. Also this is a good indication when we should stop the training

II. Analysis

Data Exploration and Visualization

The goal is to construct a reinforcement learning based model to allow an agent navigate using camera to reach a goal while avoiding difficult obstacles. We construct a simple environment in Unity that the agent can collect observations and make decisions. The environment is customizable and configurable during run time, so we could reset the environment upon finishing an episode.

Input

Vector Observation: the input of the vector space could be three dimensional vectors such as position in x , y, z coordinate or relative position in space.

Visual Observation: the camera could be mounted on the agent and offers a 640x480 pixel rendering an image of the environment.

Environment

The whole scene is set on a plane with sunlight casting above so there is light shadow for objects. The floor is a 3D plane object in Unity. The centre of the plane is the origin of the scene and the plane is sitting in the x and z direction with other objects changing their heights in the y direction. The floor size is 10x10 and objects can fall off the floor from the four edges. We deliberately did not construct the walls on each side to allow a more complex and difficult environment for the agent to learn.

Agent

Unity offers 3D game objects that have physics properties. Each game object can have the collider and the rigidbody component. Collider can detect any collision and simulate real world object interactions while ridgebody gives the flexibility to change its mass, drag, angular drag and constraints on moving position and rotation. In this project, we started with a roller ball, a sphere, as the agent. Later in the project, the agent was replaced by a block cube that can translate in x and z, and rotate in y direction. The agent will reset to a random position on the plane if it falls off the plane or reaches to the target.

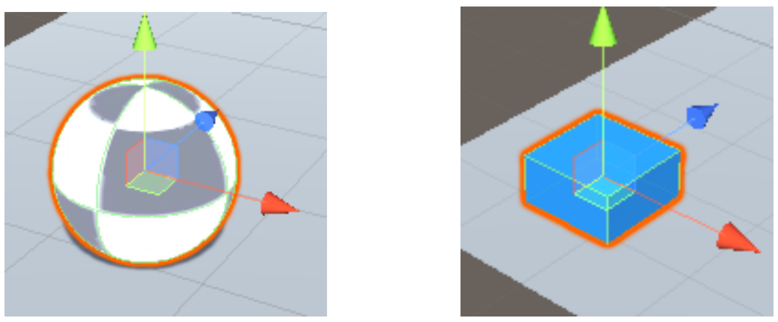


Figure 1 Agent

Agent Camera

The agent has vision simulated by a camera fixed on top of the block. The relative height and the rotation of the camera with respect to the agent can be changed through the transform property. The field of view is set about 30 degrees. This image below if a vision the camera provides.

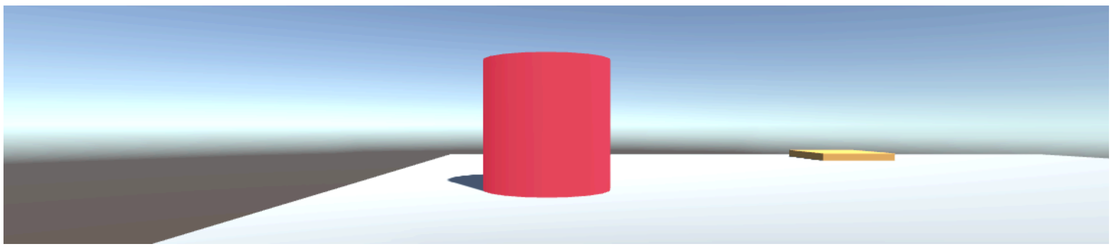


Figure 2 Image from the Agent Camera

Target

A yellow flat cube is used as the target. It is fixed in place in each episode and its position is reset to random on the plane if the agent falls off the plane or reaches the target.

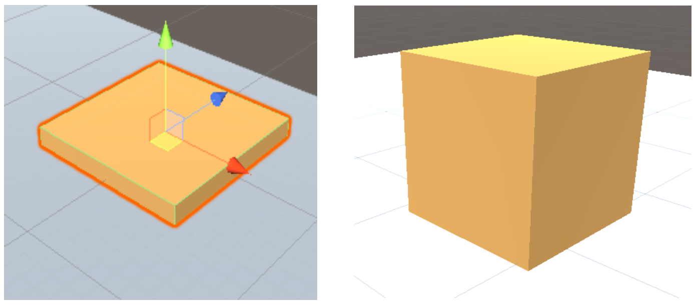


Figure 3 Target

Obstacle

There are a few options used here throughout this project. First we used a red cylinder shaped obstacle that has a similar height as the agent. Then we used the "ObiRope" to simulate a small and deformable rope in the real world as the figure showing below.

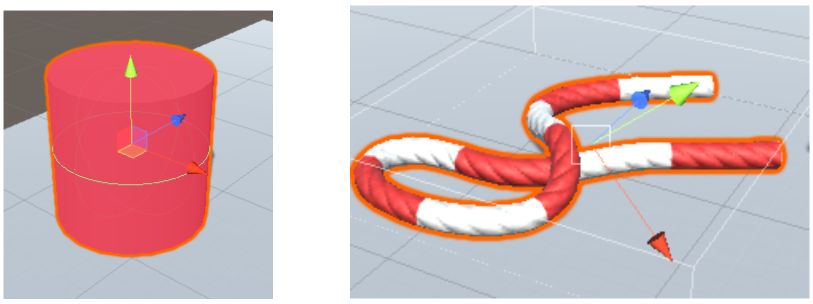


Figure 4 Obstacles

Training Area

In order to speed up the training and gather many experience in parallel, multiple training areas have been set up. For example in the figure below, there are 15 identical training areas in this one scene. All the agents in the scene are sharing the same brain. In this way, the training process would update the policy base on multiple experience and observation at one single time, leading to a faster training process. Although this requires some changes in the Unity settings and the scripts, which will be explained in the implementation section.

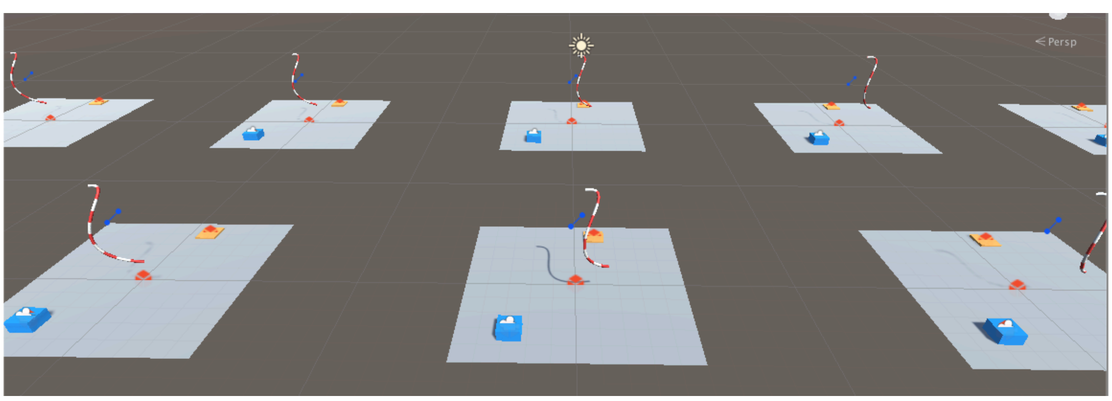


Figure 5 Multiple Training Areas

Algorithms and Techniques

In this project, we are using a reinforcement learning algorithm called Proximal Policy optimization (PPO)[13]. Given a state, it uses a neural network to map the observation the agents collect to the best action the agent could take.

PPO is a type of policy gradient descent policy, built on the foundation of Actor Critic with Advantage (AAC) algorithm. The main idea of PPO is avoiding taking too large of a step when doing gradient descent to update the policy. To achieve this, the loss equation below uses a ratio $r_t(\theta)$ that indicates the difference between the new and old policy, and it is clipped between $1-\varepsilon$ and $1+\varepsilon$. In addition, unlike the standard gradient method that does one gradient update per data sample, PPO introduces multiple epochs of mini batch updates. It has the stability benefit like trust region policy optimization but also much simpler to implement, with better data sampling.

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

- 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
- $\hat{A}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

$$\boldsymbol{r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}}$$

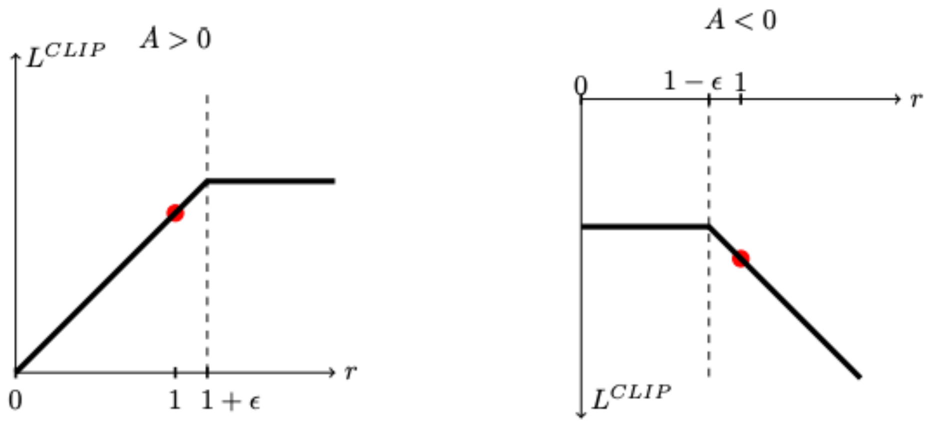


Figure 6

The left graph of figure 1 shows L_{clip} as a function of the probability ratio r for positive advantages and right graph shows the function with negative advantages. Overall, L_{clip} objective function ensures r_t stays inside $[1-\varepsilon, 1+\varepsilon]$. We can take the minimum of clipped and unclipped objective.

The PPO algorithm is implemented in the ML-Agent toolkit using TensorFlow and runs in a separate python process, communicating with Unity application over a socket.

Benchmark

Since the problem statement of vision based obstacle avoidance thought deep reinforcement learning is a relatively new topic, there is no appropriate direct comparison available as a benchmark, especially in the Unity environment. There are some classic planning algorithms mentioned in the related work section such as A* search algorithm, but they don't provide much meaningful comparison with reinforcement learning based approach. In addition, they are not based on vision and require both target and agent position. Hence we decide to establish a baseline version in Unity using PPO. We construct the standard environment with the floor, the agent and the target, which is common for all later experiments. The benchmark version provides a baseline that all later experiment should aim to achieve, even with gradually added complexity.

Vector Observation Only Model

In the benchmark model, camera vision is not involved but the velocity and the position of the agent is given as observations, as well as the position of the target. It is the simplest scenario where no obstacle presents. This is important as a first step is to verify the environment setup as well as providing benchmark performance evaluation. The environment is set up as the figure shows below.

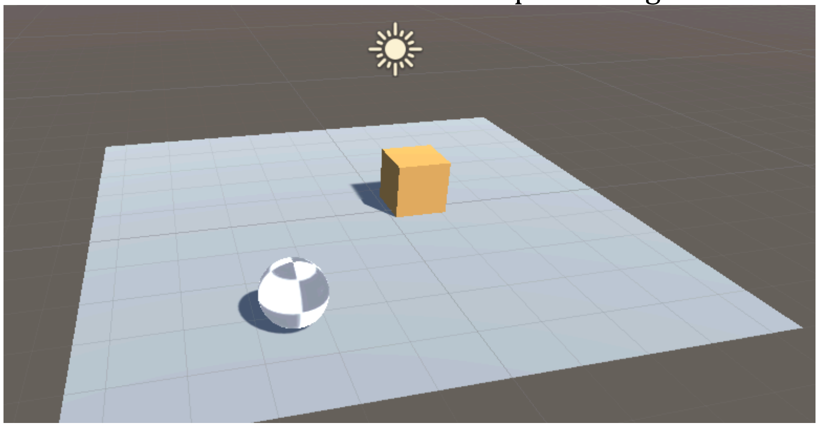


Figure 7 Benchmark Environment

Action space:
There are 6 continuous actions. Move in x and z, rotate in y.

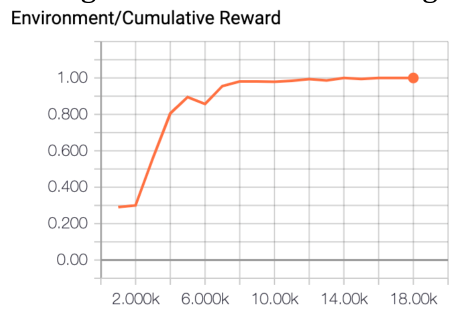
In order to verify that the agent has the correct actions and can successfully interacts with the environment before starting the training, we set up a RollerBallPlayer brain where 6 keys are mapped for 6 actions, moving up and down in the x and z axis, and rotating clockwise and anticlockwise in the y axis. In this way, we can start Unity simulation and test out agent's movement and interaction before conducting trainings.

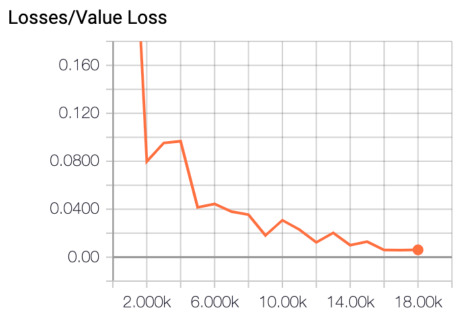
Configuration:

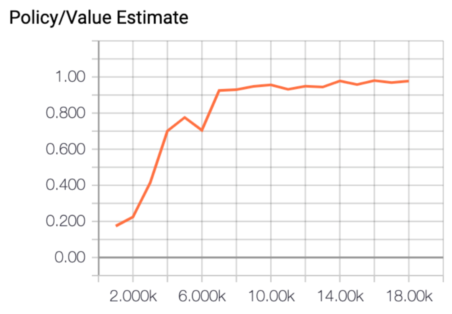
batch_size: 40, beta: 5.0e-3, buffer_size: 200, epsilon: 0.2, gamma: 0.99, hidden_units: 128, lambd: 0.95, learning_rate: 3.0e-4, max_steps: 5.0e4, normalize: false, num_epoch: 3, num_layers: 2

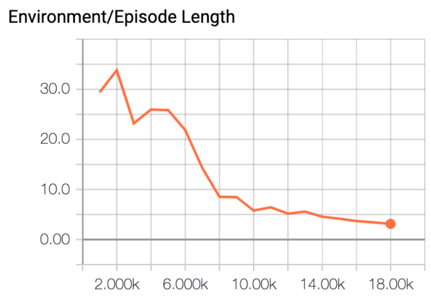
Reward:
If the agent falls off the edge, it received -1 rewards and environment resets. If the agent reaches the target, it receives +1 rewards and the environment resets.

After 10,000 steps, the average reward is 1 with standard deviation near 0, which means the agent can find the target every time and never falls off the edge. The graphs below shows the successful training session that also provides a useful guidance for later trainings.









III. Methodology and Results

The project aims to gradually build on top of the benchmark model, increasing the complexity of the environment such as using camera observation as input instead of vector observations (locations of the target and agent), simulating a deformable rope instead of a simple obstacle with a fixed position. Throughout the project, we are constantly adjusting and improving the model, such as its parameters, reward function, and physical interactions until the agent learns how to achieve these tasks.

Roller Agent with Vison no Obstacle

Setup: We use a sephere as the roller ball agent and yellow block as the target. Most of the configuraton parameters are the same as before, but the batch size is 500 and buffer size is 5000.
Observation: relative position, agent speed in x and z, and camera image 80x80 pixels

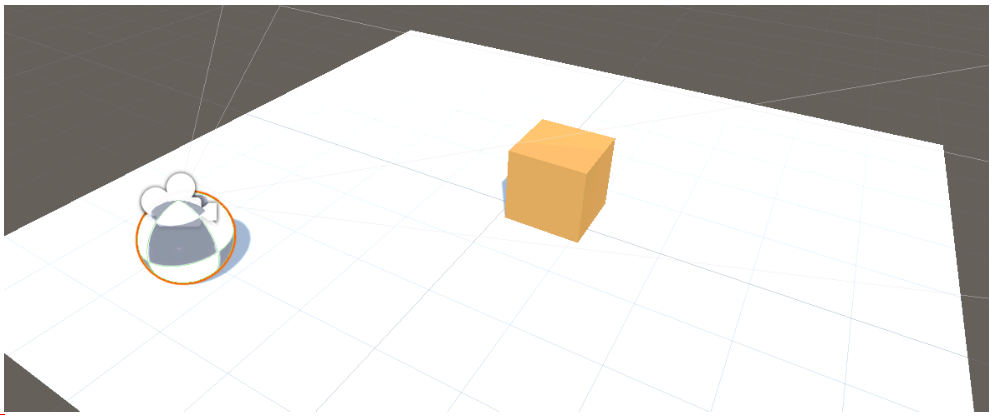


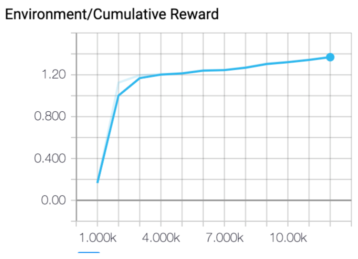
Figure 8 Roller Agent with Target

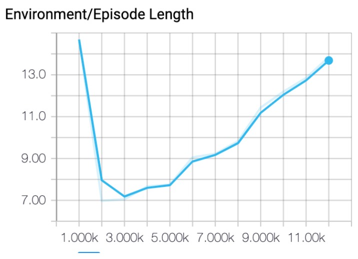
First we tried the same algorithm as before, such as only 2 conditions for the reward function, +1 for reaching the target and -1 for falling off . But the agent learns very slowly, leading to a slow and sometimes never converging policy. Therefore, some modification was implemented to help the agent learn faster. The dist_delta is current distance minus previous distance. If the agent gets closer to the goal, it gains reward of 0.005, if it moved futher away, lose reward of 0.005.

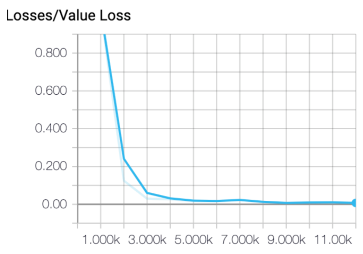
Reward function : dist_delta < 0, add 0.005, else -0.005
Reach target add reward +1, then reset env
Fall off set reward -1, reset target

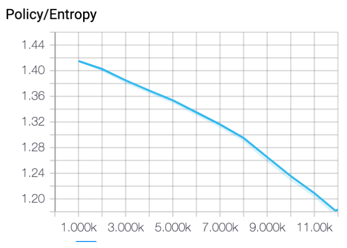
Multiple training areas
To speed up the traing process, we increased the training area from one to 19 areas. However, the interface between Unity and ML-Agent runs on a single thread, so too many training area will slow down each episode too much so 19 areas seems to be a good balance. In the Unity enviornment, the original training area needs to be converted into a prefab, then mutiple identical areas can be created with this prefab. All the training area need to be spaced out in the enviornment. In the scripts, all the position references in one training area has to be implemented as a localpostion to that each training area is operating on its own and never interfere with each other.

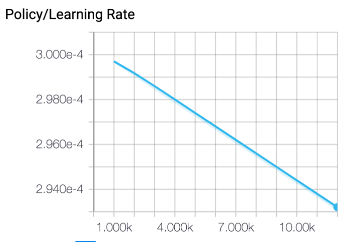
The policy can converge and stablises after 10,000 steps wih an average reward of 1.37 and the standard deviation of 0.24. The non zero standard diviation is due to the number step it takes for each differenent espisode. Because it is continuous action so the accumulation of 0.005 over 100 step in one episode would change the final reward by addding 0.5 for that episode. And the distance between the agent and obstacle is random for each episode.

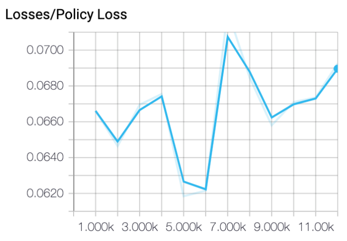












Block Agent with Vision and Obstacle:

Initial Setup

We use a block as the agent and a cylinder as the obstacle, obstructing the path to the target. The setup is show in the figure below.

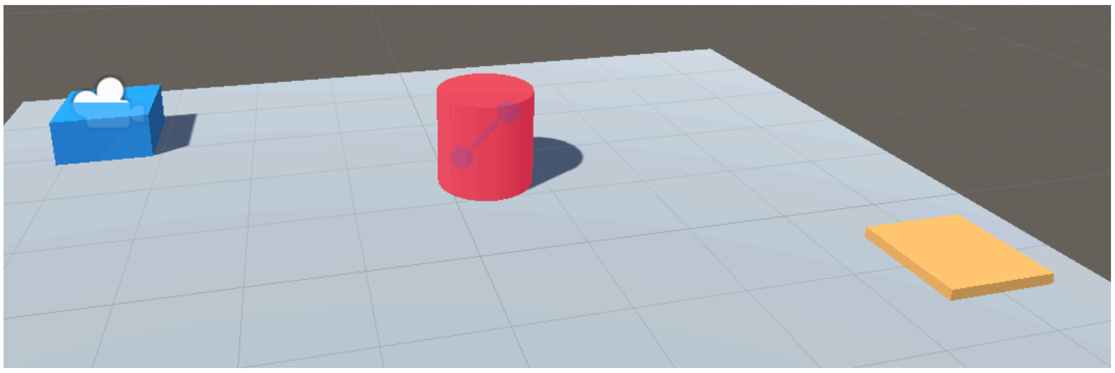


Figure 9 Block Agent Enviornment Setup

Observation: relative position and camera image
Reward function : dist_delta < 0, add 0.005, else -0.005
Every step -0.0005
Add reward when reach target add 1, then reset
Fall off set reward -1
Colide with obstabe -0.5

From reading researches and the trials done here, it comes to an conclusion that all rewards should be around -1 to 1, otherwise it is difficult to judge the final result, such as if agent has actually reached target. Values outside this range could lead to unstable training.

However the above reward function is not able to get a converged policy. In this case, we shouldn't penalise every step, as sometimes the agent has to go around the obstacle to reach the goal. Early penalty of every step would limit the agent to not explore all the areas and states. Same reason applies to changing rewards based on change in the distance to the goal. In order to reach the goal and avoid the obstacle which is right in between agent and target, the agent has to learn to choose a circled path which could be increase the distance to target first before even getting close. Sometime the obstacle target and agent are all in a line next to each other (such as scenario below), agent has to learn to get out of the crowded area first then explore to discover the target again. Less complex reward function leads to better training, which is not exactly according to our initial understanding.

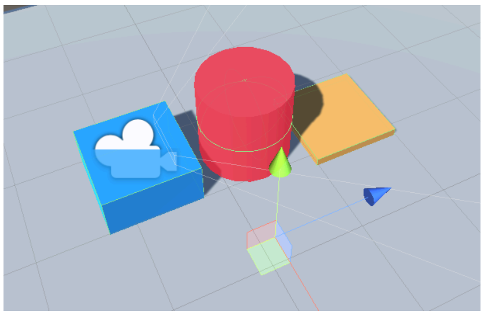


Figure 10 Speical Scenario

So the final working reward function is:

```
// Reached target
          if distanceToTarget < threshold
              SetReward(5.0f);
              Done();
          // Fell off platform
          if agent.localPosition.y < 0
              SetReward(-1.0f);
              Done();
          //Collide with the obstacle
         if distanceToObstacle < threshold
              AddReward(-0.05f);
```

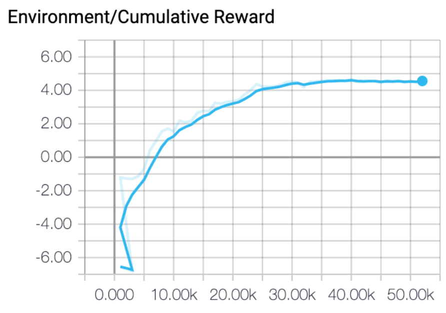
Change from continuous action to discrete action
So far, we have used continuous action for the agent, which are moving in x z direction for translation and y direction for turning. The values are between -1 and +1. However, since this problem setup is quite complex and 3 actions space covers three dimensions, continuous action space doesn't lead to converged policy estimation. Hence our agent was not able to learn and complete the task. During the training, it has been observed that the best outcome is that the agent can always avoid the obstacle and not falling down the edge, but not able to reach the goal. The average reward is 0.

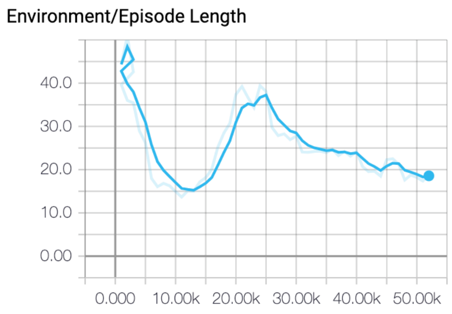
In the discrete action space, the space action size is set to 7, which are moving forward, backwards, left, right, rotate clockwise and anti-clockwise and do nothing. Also the agent is replaced as a block instead of a rolling ball.

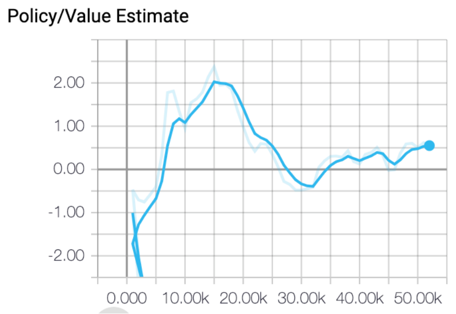
With the above changes, the PPO finally is able to learn steadily. The training process took about one day to finish. The results are shown below with average rewards very close to 5 which is the reward when reaching the goal.

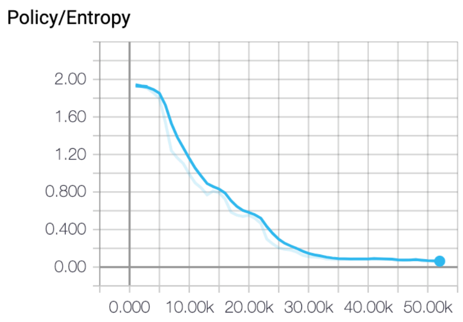
Configuration and parameters:

batch_size: 128, beta: 5.0e-3, buffer_size: 2560, epsilon: 0.2, gamma: 0.99, hidden_units: 128, lambd: 0.95, learning_rate: 3.0e-4, max_steps: 5.0e5, normalize: false, num_epoch: 3, num_layers: 2, time_horizon: 64, summary_freq: 1000, use_recurrent: true, sequence_length: 64, memory_size: 256

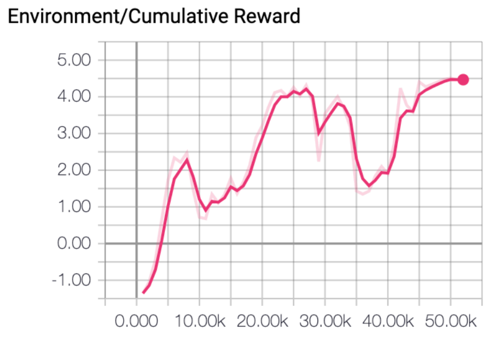


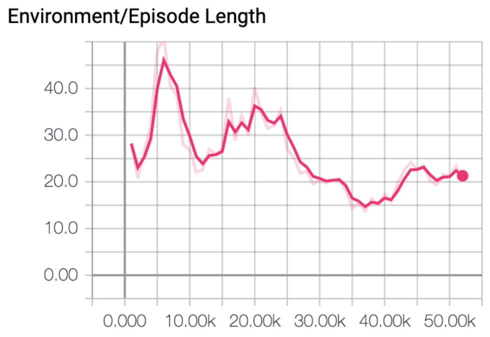


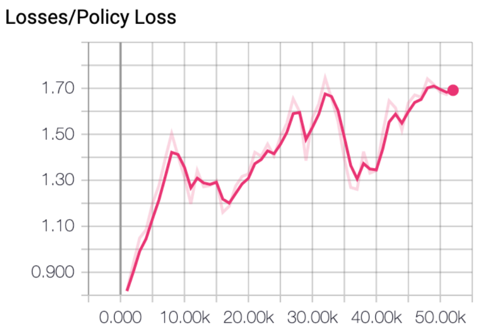


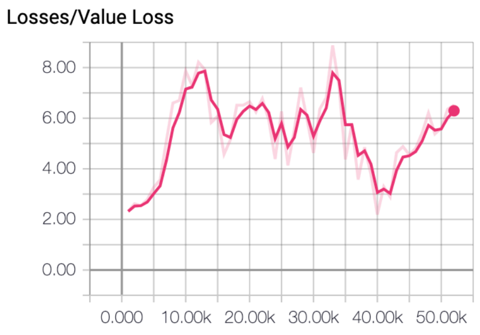


Experimenting with smaller batch size for faster training
As an attempt to tune the parameters to allow faster training, the configuration batch size is halved to 64 and buffer size is halved to 1280, the result shows a less stable learning process. One explanation could be the batch size is too small when updating the policy so the gradient decent leads to the wrong direction. This setting requires a minimum batch size of 128 and a buffer size of 2560 to allow the agent to learn the optimal policy, and any larger values would lead to more stable learning but longer training process.









Block Agent with Vision and Deformable Rope

To construct an environment that better represents the real world problem, the obstacle is replaced with a physically simulated rope. The best simulation is called ObiRope, which is on the Unity asset store. The picture below shows one of the scenarios where the rope lying on the floor between the agent and the target.

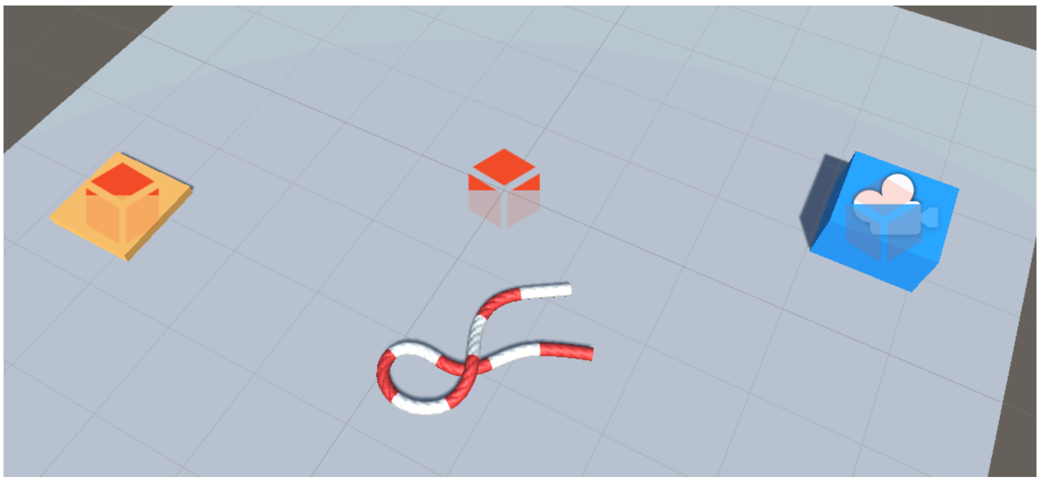


Figure 11 Block Agent with Deformable Rope

The rope has many properties and can be adjusted in length, collision model, bend, mesh generation etc. The rope can be initiated and reset mid-simulation. ObiRope is made of particles and each particle's property can be accessed and modified at run time. So the agent can detect collision by calculating particles' position in regards to the agent's position.

Reset rope position

As the example code shown below, the program iterates through all particles in the rope and check if any of them has y-axis value below the threshold. In this example, we set the values as the floor height which means the yThreshold is 0, with 0.1 as a buffer region.

```
Bool CheckIfFallingOff(yThreshold):
        loop through all particles:
                if particle.y < yThreshold
                        fallOffPlane = true
                else
                        fallOffPlane = false
End
```

After every action decision, the CheckIfFallingOff method is called. And if it does fall off the plane, the rope position is reset. Resetting is done by dropping the rope from mid air. At the very first initial iteration, the position of all the particles are saved so the relative shape of the rope is maintained but it can be dropped anywhere randomly on the plane. To ensure the agent can avoid the rope anywhere in any shape, the reset position for target and agent are random, and the rope is always obstructing the path to the target.

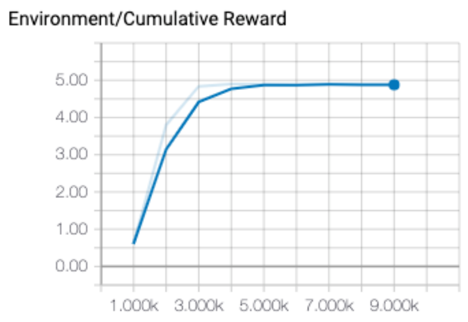
Calculation collision

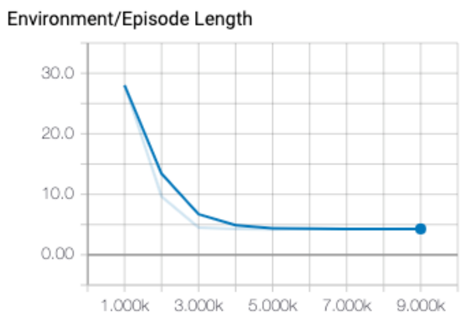
Different from the previous collision detection method with a block or cylinder, where we can define the collision when the distance between two objects is less than a threshold. This method easily applies since each object has a regular shape and is non deformable. In the case of using a rope, its shape and location is constantly changing during a collision hence the challenges it brings. The solution is found based on the design that the rope is made of particles and we can calculate the distance between each particle and the center of the agent. Looping through the rope particles, collision happens when any of the particles is less than a predefined threshold. Some pseudo code is outlined below.

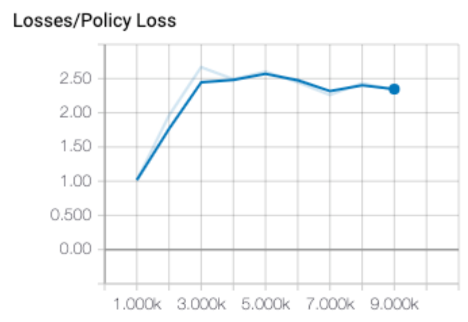
```
Bool ColiderDector(agentLocation)
      loop through all particles:
            Disatance = agentLocation to particleLocation
            If Distance < threshold
                  Collide = true
            Else
                  Not collide
End
```

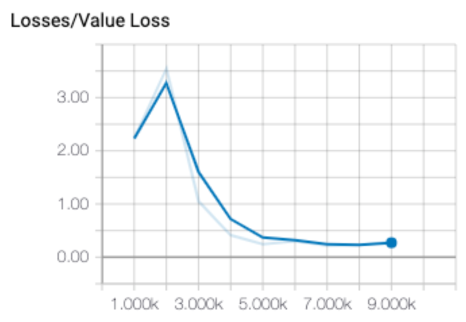
We started with the same configurations as the previous experiment, even though it is a more complex setting here. This would help to keep a similar training time, if the policy optimization doesn't converge to a desired value, we can adjust the parameters again.

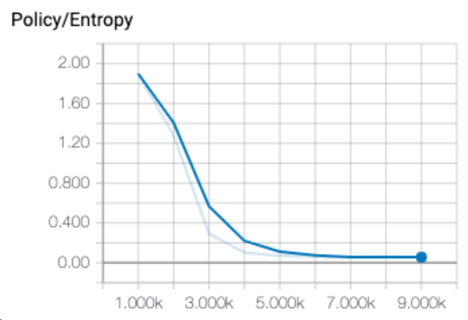
After about a day of training with multiple training areas, the agent is able to learn to avoid the rope and reach target. The average reward is very close to 5 and standard deviation is about 0.3. This means the agent will collide with the rope in rare occasions but quickly move away from it. In some settings, when the rope is reset and dropped from the sky, part of the robe could collide with the agent even before the episode begins, which would cause the loss of reward. But overall, it was a very successful learning session. All the metrics graphs are shown below and they represent a smooth and steady learning process.

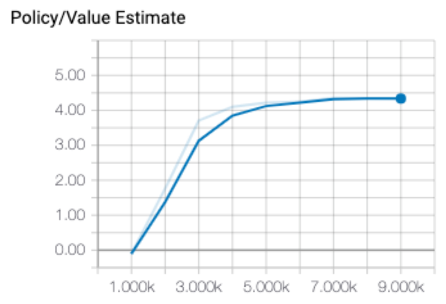












V. Conclusion

With the increasing desire for robots to free up peoples' time for more creative tasks, we rely on innovation from academics and industry. A key component for robotic navigation is that robots need to be able to avoid obstacles and reach their goal efficiently. In this project, we draw upon results from existing research to suggest using the latest PPO algorithm to improve the performance of vision based obstacle avoidance. The camera based obstacle avoidance through deep reinforcement learning could be a stepping-stone in the next era of research in robotic navigation. An initial attempt in the simulation environment has been explored, which contains complex obstacles such as a deformable rope. The continued success in this type of project would enable robots to navigate and reach their goal with camera vision only, without parameter tuning even in new settings. In a more practical sense, small and deformable object such as a wire or a rope imposes a huge challenge for many sensing systems, but camera based reinforcement learning would bring a new light on detection and avoidance.

Reflection

This project has been very rewarding. I have learned a whole new simulation environment, Unity, understood how to construct scene and make real world simulations. It has been interactive and fun. In addition, I was able to use ML-Agent tool kit, in which I learned new algorithm PPO and studied through many examples provided by ML-agent to gain insights. On a more challenging note, I used the last remaining time to implement ObiRope to simulate real physics of a rope in the desired format, dropping it from mid sky, colliding with the floor and fold onto itself. In terms of reinforcement learning, I gained practical experience to construct reward function and tune various parameters in order to train the agent to complete different tasks. I have invested many hours and days in training and gone through many challenges, eventually gained a good sense of how to construct a successful reinforcement learning model.

Improvement

Although we have made considerable progress since the beginning, there are still many areas for improvement, ideas worth exploring. Here are some of them.

1. Construct a more complex house environment with several rooms, so the agent needs to explore more before seeing the target.
2. Mixed different floor types to emulate real world home settings.
3. Place multiple ropes and rigid obstacles on the floor so the path to the target is more challenging.
4. Enable multiple agents to work in the same environment but all sharing the same brain
5. Add noise to the camera model and image to simulate real sensors

Works Cited

[1] K. F. Tang SH and K. W. Zulkifli N, ‘A Review on Motion Planning and Obstacle Avoidance Approaches in Dynamic Environments’, *Adv. Robot. Autom.*, vol. 04, no. 02, 2015.

[2] V. Sezer and M. Gokasan, ‘A novel obstacle avoidance algorithm: “Follow the Gap Method”’, *Robot. Auton. Syst.*, vol. 60, no. 9, pp. 1123–1134, Sep. 2012.

[3] I. Susnea, A. Filipescu, G. Vasiliu, G. Coman, and A. Radaschin, ‘The bubble rebound obstacle avoidance algorithm for mobile robots’, in *IEEE ICCA 2010*, 2010, pp. 540–545.

[4] Z. Gosiewski, J. Ciesluk, and L. Ambroziak, ‘Vision-based obstacle avoidance for unmanned aerial vehicles’, in *2011 4th International Congress on Image and Signal Processing*, 2011, vol. 4, pp. 2020–2025.

[5] K. Souhila and A. Karim, ‘Optical Flow Based Robot Obstacle Avoidance’, *Int. J. Adv. Robot. Syst.*, vol. 4, no. 1, p. 2, Mar. 2007.

[6] Y. Kim and S. Kwon, ‘A heuristic obstacle avoidance algorithm using vanishing point and obstacle angle’, *Intell. Serv. Robot.*, vol. 8, Apr. 2015.

[7] S. Yang, S. Konam, C. Ma, S. Rosenthal, M. Veloso, and S. Scherer, ‘Obstacle Avoidance through Deep Networks based Intermediate Perception’, *ArXiv170408759 Cs*, Apr. 2017.

[8] A. Giusti *et al.*, ‘A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots’, *IEEE Robot. Autom. Lett.*, vol. 1, no. 2, pp. 661–667, Jul. 2016.

[9] G. Kahn, A. Villaflor, V. Pong, P. Abbeel, and S. Levine, ‘Uncertainty-Aware Reinforcement Learning for Collision Avoidance’, *ArXiv170201182 Cs*, Feb. 2017.

[10] L. Tai, G. Paolo, and M. Liu, ‘Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation’, *ArXiv170300420 Cs*, Mar. 2017.

[11] L. Xie, S. Wang, A. Markham, and N. Trigoni, ‘Towards Monocular Vision based Obstacle Avoidance through Deep Reinforcement Learning’, *ArXiv170609829 Cs*, Jun. 2017.

[12] L. Tai and M. Liu, ‘Towards Cognitive Exploration through Deep Reinforcement Learning for Mobile Robots’, *ArXiv161001733 Cs*, Oct. 2016.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, ‘Proximal Policy Optimization Algorithms’, *ArXiv170706347 Cs*, Jul. 2017.