Indian Institute of Science, Bengaluru.

Department of Computer Science and Automation

Robert Bosch Centre for Cyber Physical Systems (RBCCPS)

Obstacle avoidance and target acquisition for robot navigation using deep deterministic policy gradient(DDPG) algorithm

CP 312 - Foundations of Robotics

Project Report

by

Ashish Kumar (ashishkumar4@iisc.ac.in)
Devaraju Vinoda (devarajuv@iisc.ac.in)
Gaurish Gangwar (gaurishg@iisc.ac.in)
Udit Chaurasia (cudit@iisc.ac.in)

Instructor: Shishir N. Y. Kolathaya

Contents

List of Figures

List of Tables

1 Introduction

Path finding plays a major role in the operation of mobile robots. Conventional path planning algorithms are know to be computationally expensive and does not scale well. Moreover, conventional algorithm requires full knowledge of the environment. In many cases, this is not possible. In this project, we explore the usage of DDPG algorithm in path finding algorithm. Formally, problem statement of the project is to maneuver the robot from initial position to target position while avoiding collision with obstacles.

We have divided the report into six sections. In first section we discuss about different conventional path planning algorithms and their drawback. We have used a two wheeled robot called e-puck for this experiment. In second section, we discuss the kinematics of its motion. In third and fourth section, we discuss simulation software webots and RL interface library deepbots. In fifth section, we formulate the RL algorithm. In sixth section, we discuss the programming setup and experimental results. We end the report with conclusion.

2 Conventional path planning algorithms

- **Dijkstra's Algorithm (DA):** Dijkstra's algorithm is probably the most famous algorithm used in path finding. In this algorithm, all the free space between the obstacles is represented as node. Each node is then connected to its nearest nodes along with the cost to reach that node. Initial and final node is added and Dijkstra's algorithm is ran on the obtained graph to calculate the lowest cost path. This algorithm is not used due to its high computational requirements and poor performance when distance between robot and target is large.
- **Probabilistic Road Map (PRM):** In this algorithm, random points are selected from the environment. The point is then checked if it is in free space (point is not on some obstacle). The points are then connected with each other using some algorithm like k-nearest points to create feasible paths between them. All the paths that passes through some obstacle are discarded. Finally, shortest path between initial point and destination point is found out using Dijkstra's algorithm.
- **Rapidly-Exploring Random Tree (RRT):** This algorithm is similar to PRM. In this algorithm, we first start from initial position which is considered as root node. At each step, a node nearest to the current node is selected. This step is repeated till we reach target position or we hit a dead end, in which case we start again from root node.
- **Potential Field (PF):** Potential field is based on two opposite forces, attraction and repulsion. At each point in the environment, attraction force by the target node and repulsion

force by the obstacles and boundary of the environment is calculated. Both these forces are summed up to assign a artificial potential field at each point. The goal of the algorithm then becomes to maneuver the robot from point of high potential to point of low potential. The main drawback of this algorithm is the robot getting stuck in local minima.

- **Genetic Algorithm (GA):** The fundamental principle of this algorithm is the survival of fittest. In each iteration, multiple instances of robot are allowed to explore the environment. For each instance, a fitness value is calculated after the iteration is over. The *chromosome* of instance having highest fitness value is copied and ran again in next iteration. Algorithm is terminated when some instance is able reach to target location.

3 Differential Wheeled Robots

3.1 Differential Drive

Differential drive is a two wheeled mobile robot having independent actuators on both wheels. It may have non-driven wheel to provide stability to the system. The term 'differential' means that robot turning speed is determined by the speed difference between both wheels, each on either side of the robot.

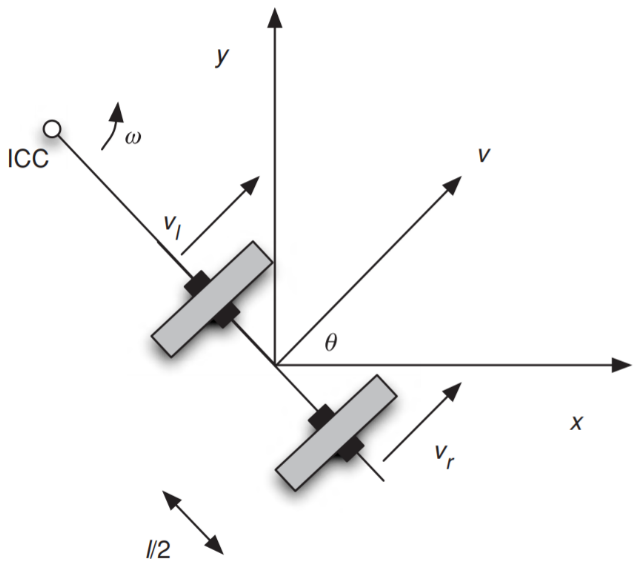


Figure 1: Differential Drive Mechanism

As the speed of the wheels varies, robot rotates about a point along the axis connecting the two wheels. This point is known as Instantaneous Center of Curvature (ICC) *fig 1*[1]. Let the center of the line connecting the two wheels be center of the mobile robot and R be the signed distance from ICC to center of robot. Since both wheels will be rotating with same angular speed ω, following equation could be written:

$$\omega(R + l/2) = v_r \tag{1}$$

$$\omega(R - l/2) = v_l \tag{2}$$

where l = Distance between the wheels,
R = Signed distance from ICC,
v_l = Left wheel velocity along ground,
v_r = Right wheel velocity along ground,

Solving equations (1) and (2), we get instantaneous radius of rotation R and angular speed ω:

$$R = \frac{l}{2}\frac{(v_l + v_r)}{(v_r - v_l)}; \omega = \frac{v_r - v_l}{l} \tag{3}$$

3.2 Three Interesting cases

Based on the equations (3), there are three interesting cases with differential drives that are generally used during path finding:

- CASE 1: If $V_l = V_r$, then $R = \infty$; The robot will move forward in straight line.
- CASE 2: If $V_l = -V_r$, then $R = 0$; The robot rotates about midpoint of wheel axis.
- CASE 3: If $V_l = 0$, then $R = \frac{l}{2}$; The robot rotates about left wheel. Similarly, for $V_r = 0$, $R = -\frac{l}{2}$; The robot rotates about right wheel.

3.3 Forward Kinematics

Assuming (x, y) is the position of mobile robot (center of wheel axle), moving in direction making angle θ with X-axis. Thus, pose of the robot could be describes as (x, y, θ). By manipulating left wheel velocity v_l and right wheel velocity v_r, we can move robot in desired position and orientation.

Position of ICC at time t could be given by:

$$ICC = (x - R\,sin(\theta), y + R\,cos(\theta)) \tag{4}$$

Thus, position of robot time instance $t + \delta t$ could be given by:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\omega\delta t) & -sin(\omega\delta t) \\ sin(\omega\delta t) & cos(\omega\delta t) \end{bmatrix} \begin{bmatrix} R\,sin(\theta) \\ -R\,cos(\theta) \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \end{bmatrix}$$

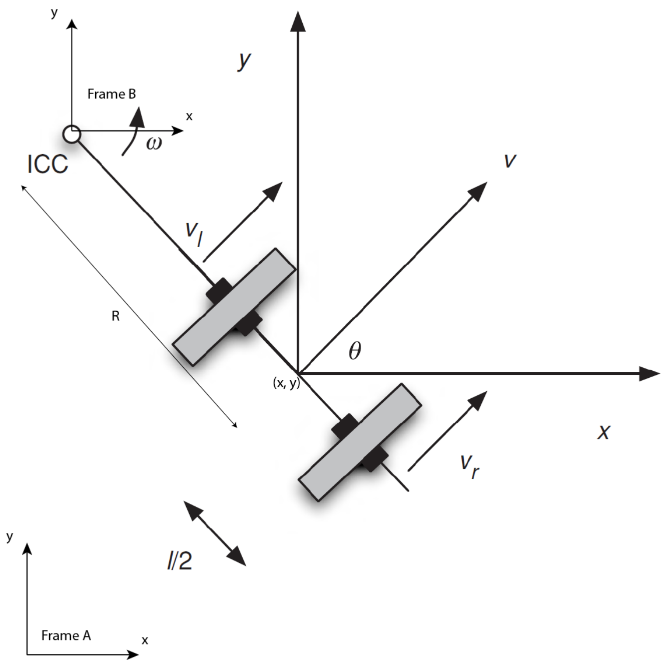


Figure 2: Differential Drive Kinematics

Similarly, orientation of robot at the same time $t + \delta t$ could be given as:

$$\theta' = \theta + \omega \delta t$$

Combining the above two equation to get robot's final pose at $t + \delta t$,

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} cos(\omega\delta t) & -sin(\omega\delta t) & 0 \\ sin(\omega\delta t) & cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{bmatrix} \tag{5}$$

3.4 Reverse Kinematics

A robot capable of moving in particular direction $\theta(t)$ at a given velocity $V(t)$ is given by:

$$x(t) = \int_o^t V(t)cos(\theta(t))dt \tag{6}$$

$$y(t) = \int_o^t V(t)sin(\theta(t))dt \tag{7}$$

$$\theta(t) = \int_o^t \omega(t)dt \tag{8}$$

In our case, since the center of robot is assumed to be center of wheel axle, thus velocity of robot could be written as average velocity of two wheels.

$$V(t) = \frac{v_r(t) + v_l(t)}{2}$$

Substituting the velocity in general equations (6), (7) and (8), we get

$$x(t) = \frac{1}{2}\int_o^t (v_r(t) + v_l(t))cos(\theta(t))dt \tag{9}$$

$$y(t) = \frac{1}{2}\int_o^t (v_r(t) + v_l(t))sin(\theta(t))dt \tag{10}$$

$$\theta(t) = \frac{1}{l}\int_o^t (v_r(t) - v_l(t))dt \tag{11}$$

Unfortunately, the above equations creates non-holonomic constraints on establishing its position. We could not an arbitrary pose of the robot and generate velocities that will get us there. However, the solutions are straightforward for limited cases.

Case 1: Constant velocity $(v_l(t) = v_l, v_r(t) = v_r, v_l \neq v_r)$

$$x(t) = \frac{l}{2}\frac{(v_l + v_r)}{(v_r - v_l)}cos\left(\frac{t}{l}(v_r - v_l)\right) \tag{12}$$

$$y(t) = -\frac{l}{2}\frac{(v_l + v_r)}{(v_r - v_l)}sin\left(\frac{t}{l}(v_r - v_l)\right) \tag{13}$$

$$\theta(t) = \frac{t}{l}(v_r - v_l) \tag{14}$$

where $(x, y, \theta)_{t=0} = (0, 0, 0)$

Case 2: $v_l = v_r = v$

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + v\,cos(\theta)\delta t \\ y + v\,sin(\theta)\delta t \\ \theta \end{bmatrix} \tag{15}$$

Case 3: $-v_l = v_r = v$

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + 2v\,\delta t/l \end{bmatrix} \tag{16}$$

Mobile robots using following general strategy used to reach desired pose:

- Turn in place to face the goal location
- Drive to the goal location
- Turn in place to the goal orientation

4 Webots

Webots[2] is a open-source software to simulate robots. It provides complete development environment to model, program and simulate robots. It contains large asset library which includes robots, sensors, actuators, objects and materials. It has GUI interface to edit simulation. The robots programmable in C, C++, Python, Java, MATLAB or ROS. Simulation can even be streamed to any web browser using webgl and websockets.

4.1 Components of webots

- Webots world file: It defines one or several robots and their environment. The definition includes shape, position, orientation, texture, etc. that are enough to describe a robot in 3D environment. It may sometimes depend on external texture and PROTO file which contains use defined nodes.

  World file is designed as scene tree. The tree contains several kind of nodes: Solid, Geometry, Robot, etc.

- Controllers programs for robot (in C/C++/Java/Python/MATLAB). Each control is attached with one or several robots. For each robot, control is started as a separate process. There are special type of controller called as supervisor controller. These controller can execute operations that are normally only be carried out by a human operator.

- Optional physics plugin to modify webots general physics behaviour.

4.2 Working of webots

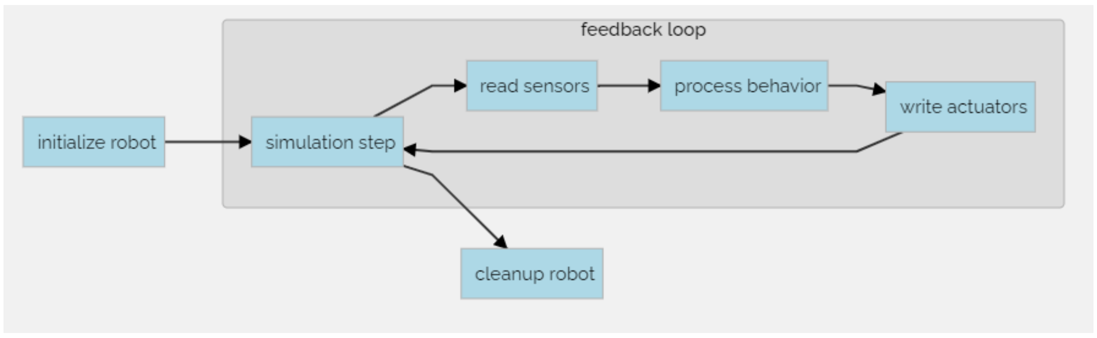


Figure 3: Webots Controller Feedback loop

First the sensors and actuators of robot are initialized. Then, in each simulation step:

1. Sensor data is read.

2. Controller processes the sensor data and generates control signal accordingly.
3. The control signal in sent to actuators for actual motion of robot.

After the simulation is over, the resources taken by the robot is cleaned up.

4.2.1 E-puck robot

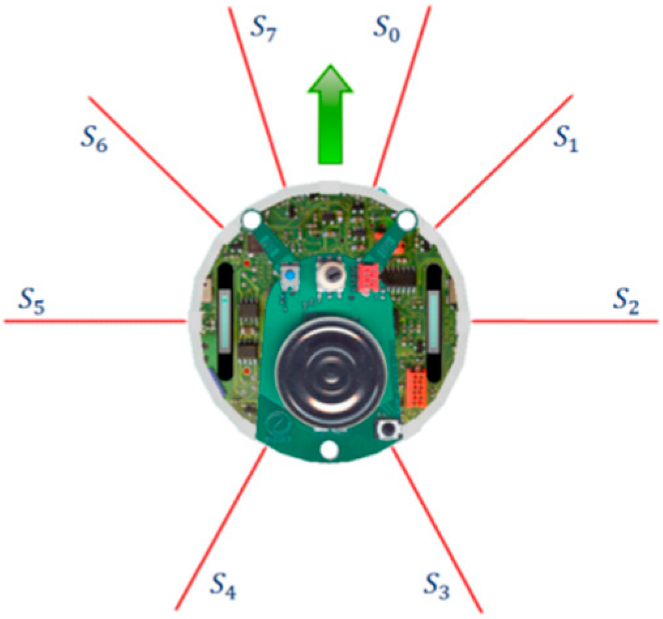


Figure 4: E-puck robot

E-puck[3] is small two-wheeled robot. It has 8 IR distance sensors all around which is given as input to DDPG algorithm. The sensors are more concentrated towards front. In addition it has accelerometer, gyroscope and camera. It also has bluetooth to communicate with other robots or central server.

5 Deepbots

Robotics community has shown great interest in the use of reinforcement learning algorithms instead of traditional control methods. Much of this has become possible due to existence of free and easily available simulators using RL environments such as OpenAIGym [4]. Gym has a very convenient interface for beginners and a great number of examples are available to get started. Due to these reasons many robotics students start their journey of using RL in robotics by using gym interface. But gym has a few limitations such as its pre-existing examples are very simplistic and it uses a proprietary simulator 'MuJoCo' which prohibits its use by students and hobbyists.

To overcome these limitations, students use various simulators and RL environments such as Bullet, Gazebo, Webots etc. There are free and opensourse simulators and webots specifically provides highly realistic simulations, that's why we have also used webots for our project. But webots can not be used directly for RL training purpose. We have to make our RL environment from scratch which can interact with the simulation environment. Having to do this everytime you want to test a RL model becomes tiring and instead of working on RL students' effort is wasted on bookkeeping stuff.

Deepbots [5] library has been designed to tackle exacly this problem. It lets us focus on implementing RL methods and does all the boilerplate implementation of RL environment and its interaction with webots. It uses the same structure as used by gym's interface which makes learning it very comfortable.

5.1 Classes

The library contains 6 major classes which are used for creating RL interface.

- SupervisorEnv
- SupervisorEmitterReceiver
- SupervisorEmitterReceiverCSV
- RobotEmitterReceiver
- RobotEmitterReceiverCSV
- RobotSupervisor

A brief description of these classes is as follows:

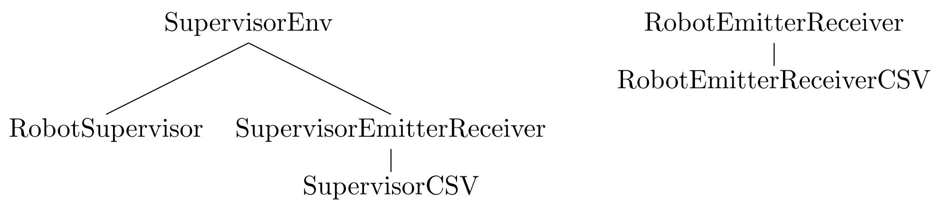


Figure 5: Class Hierarchy in deepbots library

5.1.1 SupervisorEnv

This is an abstract class which contains methods for training reinforcement learning algorithms. It follows gym's interface which is shown in figure 6. It provides following methods (standardized

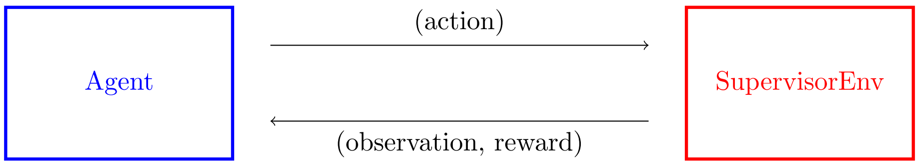


Figure 6: OpenAIGym RL interface basics

by openai gym interface).

get_observations It returns data (observations) captured by robot's sensors.

step It moves the simulation one step by choosing an action provided by the algorithm and as it is done in gym, it also returns a tuple containing four values i.e. observations from the environment, reward gained by robot for previous action, is_done parameter which indicated if simulation needs to go further or get terminated, debugging information if any.

get_reward It takes an action and returns suitable award.

is_done It returns true if simulation needs to be stopped otherwise returns false.

reset It resets the simulation to its initial state and also resets all RL parameters.

5.1.2 SupervisorEmitterReceiver

This class builds upon SupervisorEnv class. It provides following methods.

step It was an abstract method in base class, here it is provided an implementation.

handle_emitter It sends action to robot.

handle_receiver It receives a packet of message from the message queue.

5.1.3 SupervisorCSV

This class builds upon SupervisorEmitterReceiver class. It just provides implementation for two abstract methods handle_emitter and handle_receiver which send and receive observations and action in the form of comma seperated values.

5.1.4 RobotEmitterReceiver

This is the most basic abstract class which must be inherited by all robot controllers created by user. It provides following methods.

initialize_comms It is an abstract method, it should initialize the emitter and receiver nodes of simuation environment and return them as a tuple.

handle_emitter It is an abstract method, it should send the observations received from robot's sensors to supervisor.

handle_receiver It is an abstract method, it should receive the message transmitted by supervisor and parse it to get action to be taken.

run It updates the robots by one step, and at every step it sends its observations to supervisor while taking action from it. A basic implementation has been provided by the library.

5.1.5 RobotEmitterReceiverCSV

This class builds upon RobotEmitterReceiver by providing implementation for emitter and receiver to send data as comma seperated values. All the methods of this are as follow.

initialize_comms Initializes emitter and receiver nodes and returns them as tuple.

handle_emitter Implementation is provided to send data captured by robots sensors in the form of comma seperated values.

handle_receiver Implementation is provided to receive the message send by supervisor as comma seperated values.

create_message It is an abstract methods which should create a comma seperated list from observations captured by robot's sensors and return it.

use_message_data It is also an abstract method which should use the message received from supervisor (which will be as another comma seperated value list) and use to controller various actuators of the robot.

5.1.6 RobotSupervisor

This class implements robot controller and supervisor, both as one. It is useful in cases when we need to send large amounts of data between robot and supervisor e.g. if the robot has hi-def cameras attached to it then it will be send high resolution images in emitter and receiving the same in receiver, which will make the code slow and take very large amount of memory. To work around that limitation, we combine both functionalities of robot controller as well as supervisor into one so that we don't have to send and receive any data. It contains following methods.

step An implementation is provided here for this method. It applies the action received and returns observations, reward, is_done and debugging information.

apply_action This method is similar to use_message_data of RobotEmitterReceiverCSV class.

6 RL Formulation

Under this section we provide the analysis of the various requirements for the algorithm implemented and theoretical solution for each one of them.

6.1 Model Free Approach and High Dimensional Action Spaces

A model-based approach is one where the agent(robot) has complete access of the state of environment, i.e. it knows completely or partially about the environment, which makes task of computing action-value or state-value functions much simpler and typically can be solved using System of Equations but one of our main requirements is that the agent should be able to do well in any terrain or environment. It is not the case that agent will always be provided with the prior knowledge of the environment, so it is required to have a Model Free approach. [6, 7]

Many popular algorithms namely - DQN works well with low-dimensional action spaces but for high-dimensional action spaces it does not work well even if we try to discretize the action spaces then due to high dimensionality the number of actions increases exponentially with the degree of freedom. Therefore we require an algorithm that work well for high-dimensional action spaces.

6.2 Off-Policy

There has always been a dilemma for an agent as when to explore and when to exploit its knowledge about the environment. Off-policy is one of the ways which helps us in dealing with the exploration problem. Off-policy provides us a new way for learning value functions. Off-Policy agent learns about a policy, by generating data by following a different policy, i.e. the policy which agent is learning is different from that of the one used for action selection. [8] In Off-Policy approach agent learns the target policy(usually the optimal policy) using behavior policy(usually random policy), which is responsible for selecting actions. In order to perform exploration we separate the Target and Behavior policy. In order to make our agent explore we have added noise by following some distribution to the actor policy. A rough Mathematical equation is as follows -

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma max_a(\mathscr{N}(Q(S',A))) - Q(S,A)] \tag{17}$$

In the above equation the Target Policy that the agent is trying to learn is the Optimal one that's why it follows the greedy approach and applies max_a function in order to pick the best action for the state given. Here Behavior Policy is the Random one and this is achieved by adding the $\mathscr{N}$ to all the possible actions from the state, hence maintaining the probability to explore to explore.

6.3 Deep Neural Networks

This section provides the mathematical and theoretical analysis on Deep Neural Network used to estimate the action-value function. Below figure gives an overview of a simple Deep Neural Network with 4 layers. Input is fed at the Layer 0, which subsequently propagates to further layers in order to get the estimated output. In order to estimate the output we define two parameters W and b. In Forward Propagation our main goal is to predict the output of the action-value, while in Backward Propagation our goal is to train our model by calculating the parameter values as close as possible to its true value, for this purpose we use Gradient Descent Method, we also use mini-batch gradient method(used in DDPG). Mathematical analysis for Forward and Backward Propagation is given in subsequent sub-section.

6.3.1 Forward Propagation

In forward propagation we compute the input in order to predict the output as close as possible to that of the original value. For example in Binary Classification we input an image and predict whether the image matches with that of the desired one or not.

Parameters (each layer has its separate parameters) $W^{[l]}$ and $b^{[l]}$ (superscript represents the layer number to which the parameters belong to) are initialized to predict the output by following the general equations given below.

Layer 1 equations: $Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$; $A^{[1]} = g^{[1]}(z^{[1]})$
Layer 2 equations: $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$; $A^{[2]} = g^{[2]}(z^{[2]})$
Layer 3 equations: $Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$; $A^{[3]} = g^{[3]}(z^{[3]})$
Layer 4 equations: $Z^{[4]} = W^{[4]}A^{[3]} + b^{[4]}$; $A^{[4]} = g^{[4]}(z^{[4]})$

In the above equation input X can be represented as $A^{[0]}$, since input has been assigned as Layer 0. $W^{[1]}, b^{[1]}$, $W^{[2]}, b^{[2]}$, $W^{[3]}, b^{[3]}$ and $W^{[4]}, b^{[4]}$ represent the parameters for the respective layers 1,2,3,4.

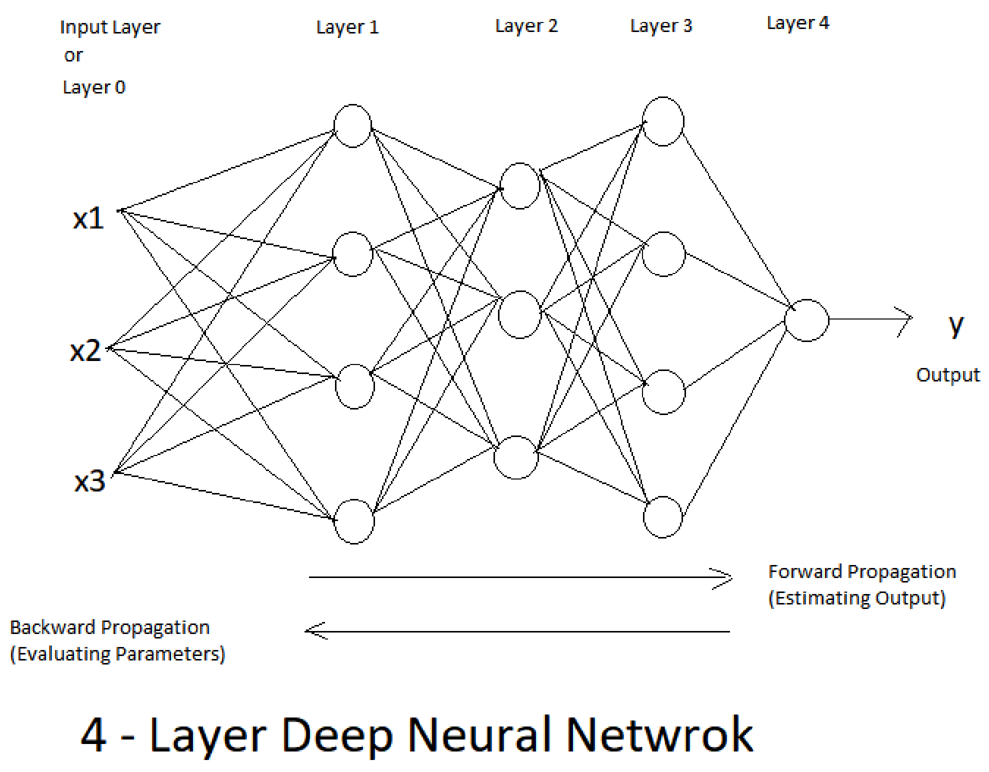


Figure 7: 4-Layer Deep Neural Network

6.3.2 Backward Propagation

In Backward Propagation we tune the parameters based on the Loss Function obtained. For tuning we apply Gradient Descent (Mini-Batch Gradient is used in the algorithm implemented) and compute the derivatives of Loss Function with respect to the parameters involved at each stage.

Loss Function, $\mathscr{L} = \frac{1}{N}(\hat{Y} - Y)^2$

Here $\hat{Y}$ represents predicted output vector and Y represents the exact output and $(\hat{Y} - Y)^2$ represents the squared error of all the outputs. Equations involved in Backward Propagation is given as -

$dZ^{[l]} = W^{[l+1]T} dZ^{[l+1]} * g^{[l]\prime}(Z^{[l]})$

$dW^{[l]} = \frac{1}{N} dZ^{[l]} * X^T$

where $dz = \frac{\partial \mathscr{L}}{\partial z}$ Above equations are for N training example, for $N-$training examples we use

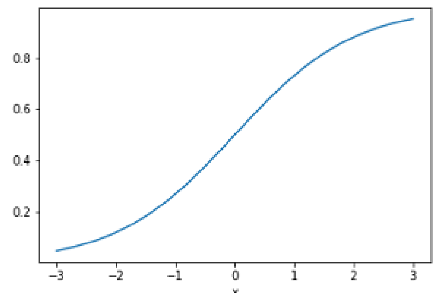
Gradient Descent by converting the inputs and parameters in an appropriate matrix of valid dimensions, the updated value of parameter W is given as -

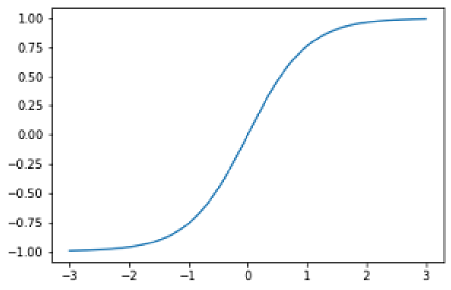
$$W^{[l]} = W^{[l]} - \alpha(dW^{[l]})$$

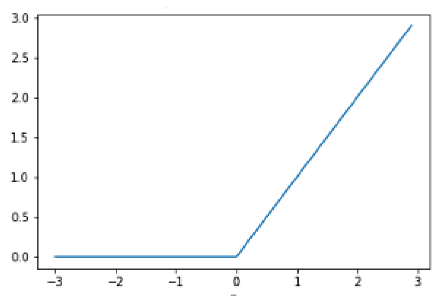
The function $g^{[l]}$ in above set of equations represent the activation functions that is discussed in the new sub-section.

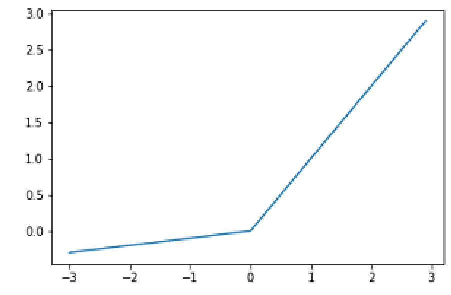
6.3.3 Choice of Activation Functions

The output we compute using Neural Networks may not be the linear function of the input features, so to introduce non-linearity we introduce Activation Functions. Most of the times the output is a non-linear function of the input features. [9] One main requirement of the Activation Functions is that they should be differentiable in order to compute the Gradient or Mini-Batch Gradient Descent. Some Activation Functions are given below, most of them we used in our training model -


Sigmoid Function


tanh Function


ReLU Function


Leaky ReLU Function

First one is the *Sigmoid* function, not much in use but can be used for Binary Classification. Equation for Sigmoid function is given as -

$$s_{\theta}(x) = \frac{1}{1 + e^{-x}}$$

Tanh function used is an improvement over *Sigmoid*, it is a shifted version of *Sigmoid* with mean 0, advantage of *Tanh* is that it is much steeper than *Sigmoid* so it converges faster than *Sigmoid* and sometimes when we require to centre the data and to have mean 0 we use it. We have also used it in implementation as one of the activation functions in one of the layers of Neural Network. Equation is given as -

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The disadvantage of both the functions is that when x becomes very small or very large, the derivative becomes very small, converges slower and slow down the gradient descent.

To overcome above issues we use another Activation Function *ReLU* (Rectified Linear Unit) this function has derivative $+1$ when $x > 0$ and 0 when $x < 0$, hence eliminating the possibility of low function value. Since derivative is not very small at each point so it converges faster. *ReLU* is used as an activation function in *DDPG*. Below is the equation of *ReLU*

$$relu(x) = max(0, x)$$

Though in most of the cases the input x is positive but sometimes it may turn out that it becomes negative so for that it always compute derivative as 0, Derivative is 0 when $x < 0$ is the only disadvantage with *ReLU*. To overcome this we apply the rectified version of *ReLU* as *Leaky ReLU*.

Leaky ReLU does not have derivative 0 when $x < 0$ rather it has derivative given as $a * x$, usually a is of the order 10^{-2} to 10^{-3}. The shares the same advantage as that of *ReLU*, also it has an additional advantage over *ReLU* is that it can even work well for $x < 0$. *Leaky ReLU* has been used extensively in the implementation part. Mathematically *Leaky ReLU* equation is given as -

$$Lrelu(x) = max(a * x, x)$$

If we do not introduce non-linearity, then there is no meaning for adding a neural network, since output will result as a linear function of input. Linearity implies that $A^{[l]} = Z^{[l]}$

$$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

Substituting $A^{[l]} = Z^{[l]}$ in above equation and replacing it with $Z^{[1]}$, we get,

$$Z^{[2]} = W^{[2]}(W^{[1]}A^{[0]} + b^{[1]}) + b^{[2]}$$

$$Z^{[2]} = (W^{[2]}W^{[1]})A^{[0]} + W^{[2]}b^{[1]} + b^{[2]}$$

Here parameters are initialized randomly and changed linearly without involving any non-linear function.

6.3.4 Hyper-parameters

Hyper-parameters are the one which controls the learning of our model. While parameters($W^{[l]}$ and $b[l]$) are tuned during training process, hyper-parameters will not update during training, they need to be tuned manually. Major hyper-parameters which is used in implementation are - Learning Rate (α), Number of Hidden Layers, Number of Hidden Units, Choice of Activation Functions, Iteration, Mini-Batch size.
Some of the tuned value of hyper-parameters are given at the end of the report.

Random Initialization - The parameters $W^{[l]}$ and $b^{[l]}$ have to be initialized randomly, they should not be initialized to zero, if we initialize to zero then output of each layer is same as that of the previous layer, so introducing the Neural Networks will not be of much help.

7 Reinforcement Learning Framework for Obstacle Avoidance and Target Acquisition

7.1 Problem Definition

In our work we consider a standard reinforcement learning setting consisting of an agent(robot) interacting with the environment E for discrete timestamps *fig 8*. At each timestamp t the agent receives an observation x_t, performs an action a_t, then the agent receives a scalar reward r_t.

We express the problem of obstacle avoiding and target acquisition as a Markov decision process (MDP). An MDP is represented by a tuple $\{\mathcal{S},\ \mathcal{A},\ P,\ R,\ \gamma\}$. Here $\mathcal{S} \in \mathbb{R}^n$ is set of robot states called state space, action space $\mathcal{A} \in \mathbb{R}^m$ is the set of feasible actions the robot can perform, transition dynamics $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function that models the state at timestamp *t+1* based on the state and action at timestamp t *ie.* $p(s_{t+1}|s_t, a_t)$, we also consider an initial state distribution $p(s_1)$, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward obtained for the given state and action $r(s_t, a_t)$ and $\gamma \in [0, 1]$ is defined as the discount factor.

The goal of the mobile robot is to find the target and simultaneously avoid collision with the obstacles present in the environment.

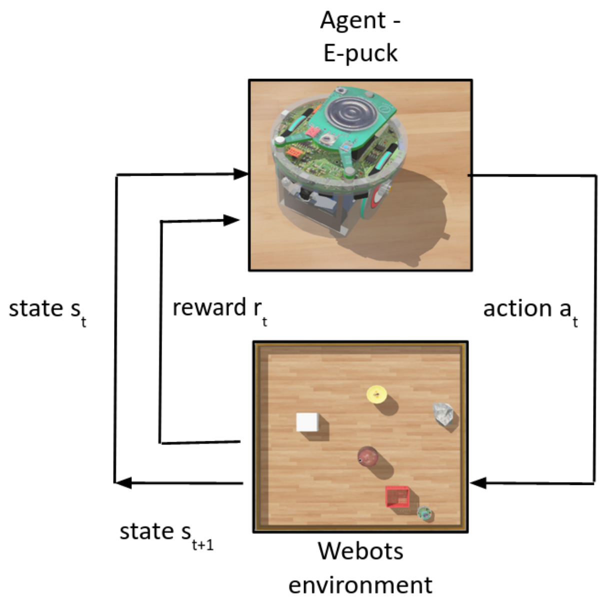


Figure 8: Agent-Environment interaction in RL

The rule used by the agent to decide what action to take next is called policy π. The policy π which defines the agent's behavior maps states to probability distribution over actions $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$. The goal of Reinforcement Learning is to learn a policy π which maximizes the expected cumulative reward from the start distribution $J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1]$. Here, the return from a state is defined as the sum of discounted future reward $R_t = \sum_{i=t}^{T} \gamma^{(i-1)} r(s_i, a_i)$. The return R_t depends on the actions and therefore it in-turn depends on the policy π.

The action-value function describes the expected return after taking an action a_t in the state s_t and thereafter following the policy π.

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i>t} \sim E, a_{i>t} \sim \pi}[R_t | s_t, a_t]$$

The off-policy algorithms like Q-learning use the greedy policy $\mu(s) = argmax_a Q(s, a)$. We consider function approximators with parameter θ^Q, which is optimized by minimizing the loss function:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^{\beta}, a_t \sim \beta, r_i \sim E}[(Q(s_t, a_t | \theta^Q) - y_t)^2]$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}) | \theta^Q)$$

7.2 Learning Algorithm

As the states and actions are continuous, we use the algorithm that enable us to construct optimal policies that yield action values in the continuous space. The Deep Deterministic Policy Gradient(DDPG)[10] algorithm has proved to be effective in dealing with problems involving continuous action space. We use DDPG to tackle obstacle avoidance and target acquisition task.

DDPG is an off-policy, model free, actor-critic algorithm based on deterministic policy gradient(DPG)[11]. The deterministic policy gradient algorithm consists of a parameterized actor function $\mu(s|\theta^{\mu})$ which specifies the policy at the current timestamp by deterministically mapping states to a specific action. The critic *Q(s, a)* is learned using the Bellman equation the same way as in Q-learning. The actor is updated by applying the chain rule to the expected return from the start distribution J with respect to the actor parameters. The policy gradient which gives the performance of the policy is given by,

$$\nabla_{\theta^{\mu}} J \approx \mathbb{E}_{s_t \sim \rho^{\beta}}[\nabla_a Q(s, a|\theta^{Q})|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|s = s_t]$$

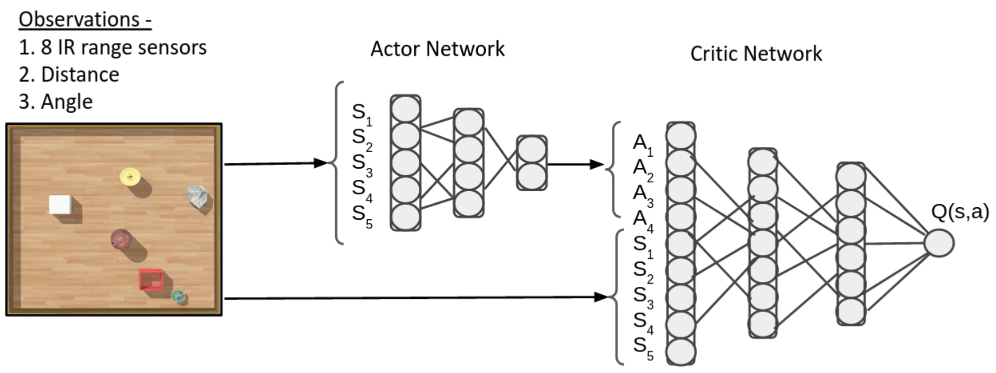


Figure 9: DDPG actor-critic network model

DDPG combines the useful ideas from DQN[12] and DPG[11] algorithms to make it more robust and efficient in learning. DDPG concurrently learns a Q-function and a policy. It uses off policy data and Bellman equation to learn the Q-function, then it uses the Q-function to learn the policy. The samples obtained from exploring sequentially in an environment are not independently and identically distributed so to address this issue DDPG uses the idea from Deep Q-Networks(DQN) called replay buffer. Replay buffer is finite sized cache, transitions were sampled from the environment and a tuple (s_t, a_t, r_t, s_{t+1}) is stored in the replay buffer. A mini-batch is sampled from the reply buffer at each timestamp to update actor and critic.

Since the network $Q(s, a|\theta^{Q})$ being updated is also used in calculating the target value, the Q update is prone to divergence. So, DDPG uses the concept of soft target updates rather than directly copying the weights to the target network. This is achieved by creating a copy of actor and critic networks, $Q'(s,a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$. The weights of these target networks are then updated by having them slowly track the learned networks:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta', \tau \ll 1$$

This means that the target values are constrained to change slowly, greatly improving the stability of learning.

To account for different physical units from different components of the observation a technique called batch normalization is used. Batch normalization technique normalizes each dimension across the samples in a mini-batch to have unit mean and variance.

A major hurdle of learning in continuous action space is exploration. To make DDPG policies explore better, we add noise to the actor policies at training time. Noise is sampled from a noise process $\mathcal{N}$,

$$\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$$

We have used an Ornstein-Uhlenbeck process with μ as 0.0, θ as 0.2 and σ as 0.15 to generate temporally correlated exploration.

```
Algorithm 1 DDPG algorithm
  Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
  Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
  Initialize replay buffer $R$
  for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
      Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
      Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
      Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
      Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
      Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
      Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
      Update the actor policy using the sampled policy gradient:
          $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
      Update the target networks:
          $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
          $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end for
  end for
```

Figure 10: DDPG algorithm

7.3 State and Action Space

The E-puck robot uses 8 infrared ray proximity distance sensors and it has two motors as shown in *fig 4*. It uses differential drive mechanism for moving. The agent, in addition to the distance sensor values, also receives the Euclidean distance and angle from the target. Therefore the agent gets an one-dimensional vector with 10 values as observations. Since the actuators are motors, which means that the outputs of the agent are two values controlling the forward/backward movement and left/right turning respectively. So, there are two actions that can be performed by the agent, moving forward/backward and turning which can be referred to as gas and the wheel.

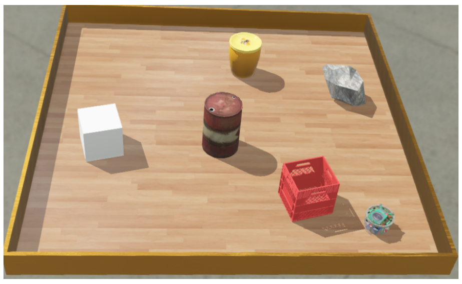
7.4 Reward Function

$$r(s_t, a_t) = \begin{cases} -10 & collision \\ +500 & d_{target} < th_{distance} \\ -d_{target} & otherwise \end{cases}$$

where $th_{distance}$ is the threshold distance - it is the minimum distance between robot and target which is considered as reaching the target, d_{target} is the Euclidean distance between robot and the target. If the robot collides with the obstacles then it will be rewarded -10. If the robot reaches the target within the threshold distance then it will rewarded +500. Otherwise the reward is awarded bases on d_{target}.

7.5 Training Results

We used Webots an open source simulator that is capable of recreating realistic environments for robot based applications, which relies on Open Dynamics Engine(ODE) for physics simulation. In the DDPG[13] algorithm, the actor network in the learning algorithm consists of 3 hidden layers with 400 units, 300 units and 200 units respectively. The critic network also consists of hidden layers with 400 units and 300 units; after the 2^{nd} hidden layer, the output of actor network is included followed by the fully connected layers with 200 units. The activation units used in the actor and critic network are *LeakyRelu* $\rightarrow$ *LeakyRelu* $\rightarrow$ *LeakyRelu* $\rightarrow$ *sigmoid* and *LeakyRelu* $\rightarrow$ *LeakyRelu* $\rightarrow$ *LeakyRelu* $\rightarrow$ *Relu* respectively. Table 1 contains the hyper-parameter values used for training the networks.


(a) configuration 1

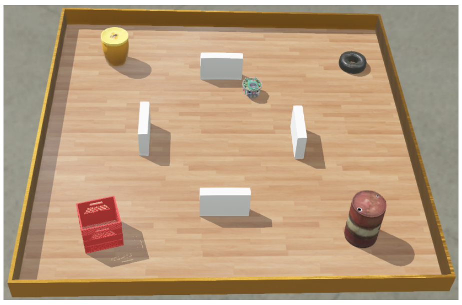

(b) configuration 2

Figure 11: Different environment configurations used for training the agent.

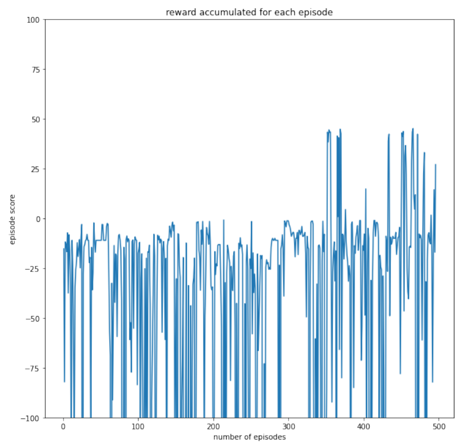
Parameter	**Value**
Discount factor (γ)	0.99
Learning rate - actor	0.00025
Learning rate - critic	0.00025
Mini Batch size	64
Replay buffer size	10^6
Soft target update parameter(τ)	0.001

Table 1: Hyper-parameter values of DDPG networks

We trained the agent on different environment configurations. Some of the sample configurations are shown in *fig 11* and the *Table 2* contains the outcome of training the agent in different configurations. The video presenting the demo of trained agent is available at https://youtu.be/CD4EVCAvcBQ and https://youtu.be/i1c6DUlwLcA. The reward obtained while training the agent is shown in *fig 12*.

Environment configuration type	**Outcome**
No obstacles, position of target and initial position & orientation of robot is fixed.	✓
No obstacles, position of target and initial position & orientation of robot is randomized.	✓
With obstacles, position of target, obstacles and initial position & orientation of robot is fixed.	✓
With obstacles, position of target and initial position & orientation of robot is fixed but position of obstacles is randomized.	$\times$
With obstacles, position of target, obstacles and initial position & orientation of robot is randomized.	$\times$

Table 2: Different environment configurations in Webots simulator for which the agent was trained. ✓:agent was able to learn. $\times$:agent was unable to learn.



(a) When the position of target,obstacles and initial position & orientation of robot is fixed.

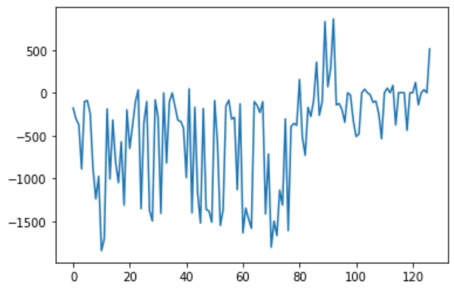

(b) When the position of target and initial position & orientation of robot is randomized.

Figure 12: The reward accumulated over each episode.

8 Conclusion

In this project we used deep deterministic policy gradient(DDPG) algorithm for obstacle avoidance and target acquisition task. We learnt about simulation software Webots and RL interface library Deepbots. We understood the mathematical concepts behind differential drive and DDPG algorithm. We created Webots world file which includes the environment of the robot. We implemented the controller of robot and supervisor and finally interfaced the supervisor with DDPG algorithm using Deepbots library. We also ran the experiment with multiple setup which includes fixed initial robot and target position and random robot and target position and their combinations, along with or without obstacles. We found out that DDPG algorithm works well when there are no obstacles, even if initial robot and target position is randomized. With obstacles, performance of DDPG algorithm was great when the position of robot, target and obstacles were fixed. However, DDPG algorithm did not give positive result when the position of the obstacles were randomized.

9 Acknowledgement

We would like to thank Prof. Shishir N. Y. Kolathaya for providing an opportunity to work on this project and for timely guidance and support.

References

[1] Dudek and Jenkin. *Computational Principles of Mobile Robotics.*

[2] Webots. *http://www.cyberbotics.com*. Ed. by Cyberbotics Ltd. Open-source Mobile Robot Simulation Software. URL: http://www.cyberbotics.com.

[3] *e-puck education robot*. E-puck.org, 2018. URL: http://www.e-puck.org/ (visited on 02/02/2021).

[4] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016). URL: https://gym.openai.com/.

[5] M. Kirtas et al. "Deepbots: A Webots-Based Deep Reinforcement Learning Framework for Robotics". In: *Artificial Intelligence Applications and Innovations*. Ed. by Ilias Maglogiannis, Lazaros Iliadis, and Elias Pimenidis. Cham: Springer International Publishing, 2020, pp. 64–75. ISBN: 978-3-030-49186-4.

[6] Sebastian Dettert. *Introduction to Model-Based Reinforcement Learning*. URL: https://medium.com/analytics-vidhya/introduction-to-model-based-reinforcement-learning-6db0573160da.

[7] Ryan Wong. *Model-Free Prediction: Reinforcement Learning*. URL: https://towardsdatascience.com/model-free-prediction-reinforcement-learning-507297e8e2ad.

[8] Ram Sagar. *On-Policy VS Off-Policy Reinforcement Learning*. URL: https://analyticsindiamag.com/reinforcement-learning-policy/.

[9] Keon Yong Lee. *[Personal Notes] Deep Learning by Andrew Ng — Course 1: Neural Networks and Deep Learning*. URL: https://medium.com/@keonyonglee/bread-and-butter-from-deep-learning-by-andrew-ng-course-1-neural-networks-and-deep-learning-41563b8fc5d8.

[10] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning." In: *ICLR*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

[11] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, June 2014, pp. 387–395. URL: http://proceedings.mlr.press/v32/silver14.html.

[12] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[13] Phil Tabor. *Deep Reinforcement Learning in Python Tutorial – A Course on How to Implement Deep Learning Papers*. URL: https://morioh.com/p/529d13cde6ff.