Faculty of Engineering and Materials Science
German University in Cairo

Reinforcement-Learning-based navigation for autonomous vehicles

Bachelor Thesis

Author: Abdalla Mohamed
Supervisor: Prof. Dr.Ayman ElBadawy
Submission Date: 1 August, 2021

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowledgment has been made in the text to all other material used

---

Abdalla Mohamed
1 August, 2021

Acknowledgments

I would like to acknowledge the Humongous effort done by my supervisor Prof. Ayman A.El-Badawy throughout the bachelor for his continuous mental and technical support, and for his guidance. Being one of his students has been a great honor for me, his enthusiasm is one of the main reasons that kept me going through the struggles that I encountered through the bachelor. I would like to thank my family for their great support since day one. I would like to thank Eng. Micheal Magdy for the technical support and the tremendous help in writing my thesis and for providing the residence at his own place after long working hours.

Abstract

Learning to navigate in an unknown environment is a substantial capability of mobile robot. Conventional methods for robot navigation consists of three steps, involving localization, map building and path planning. However, such methods assume complete knowledge of the environment dynamics, but in real-life applications the environment parameters are usually stochastic and difficult to deduce so it becomes impractical to rely on an explicit map of the environment. To adapt to map-less environments, learning ability is a must to ensure obstacle avoidance and path planning flexibility. Recently, Reinforcement learning techniques were applied widely in the adaptive path planning for the autonomous mobile robots. In this thesis, we propose different variations of Q-learning to obtain an optimal navigation trajectory considering goal-oriented tasks of a skid-steering mobile robot. Furthermore, we examine the performance of integrating neural network approximators with Q-learning by applying deep Q Learning and comparing its results with the standard Q-learning and dynamic Q-learning in maze-like environments. To clarify, we simulate the paths taken by the skid-steering mobile robot in a grid world environment containing static obstacles and a single static target. The goal of the robot is to find the shortest path to the target while avoiding the obstacles without having any access to the map. Deep Q learning, traditional Q-learning and Dyna-Q learning algorithms are implemented and the robot simulation parameters are captured and discussed in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Robots are playing an increasingly important role in our daily life, like cleaning, rescuing, mining, unmanned driving and so on. Its a fundamental ability for robots to navigate autonomously in an unknown environment. Benefiting of the development of artificial intelligence and computer vision, great progress has been made on intelligent robot technologies. However, enabling robots to navigate in a real world autonomously is still a challenging task. Traditional navigation method consists of localization, map building, and path planning . Localization is the process of determining the position and orientation of the robot with respect to its surrounding. The robot needs to recognize the objects around it. It needs to recognize each object as a target or as an obstacle. Many techniques deal with this localization problem using laser range finders, sonar range finders, ultrasonic sensors, infrared sensors, vision sensors and GPS that have been developed on-board or off-board. When a larger view of the environment is necessary, a network of cameras has been used. The other problem is the path planning in which the robot needs to find a collision free path from its starting point to its end point. In order to be able to find that path, the robot needs to run a suitable path planning algorithm, to compute the path between any two points . Recently, due to the invasion of artificial intelligence and machine learning approaches , new path-planning trajectories and hypotheses are being developed which are mainly based on reinforcement learning techniques . Where the navigation problem is being formulated as a reinforcement-Learning problem . Reinforcement Learning approach contains many subsets such as model-based and model-free Reinforcement learning and each subset covers a variety of methodologies and algorithms .Recently, the majority of algorithms depend on neural networks and deep learning techniques . However, we propose in this paper a model-based version of Q -learning which we call Dynamic Q-learning that involves planning while learning basing its planning steps on simulations from an updatable environment model in order to obtain the fastest trajectory towards reaching the target goal while avoiding the obstacles .

1.2 Reinforcement Learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning. One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best [14] .

1.2.1 RL Algorithm

1. First, the agent interacts with the environment by performing an action
2. The agent performs an action and moves from one state to another
3. And then the agent will receive a reward based on the action it performed
4. Based on the reward, the agent will understand whether the action was good or bad
5. If the action was good, that is, if the agent received a positive reward, then the agent will prefer performing that action or else the agent will try performing an other action which results in a positive reward. So it is basically a trial and error learning process [12] .

1.2.2 Elements of RL problem

- Agent: It is the program or the mind of our robot which is able to intelligently plan and take actions and receive rewards .
- Model: in model-based Algorithms , the model is the representation of the environment from the point of view of the agent , in our case the environment is a set of states (positions) representing the obstacle-map in which our agent (SSMR robot) interacts with , while navigating to reach a specific target

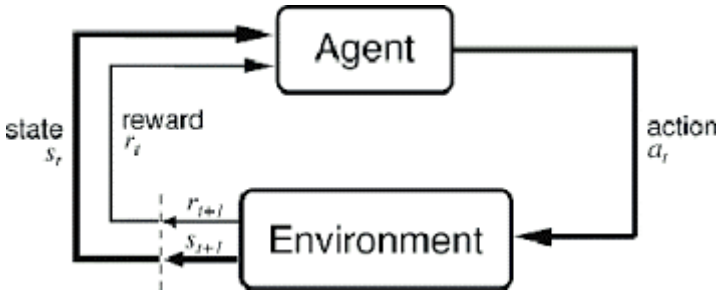

Figure 1.1: RL paradigm [14]

- Policy: A policy defines the agent's behavior in an environment. The way in which the agent decides which action to perform depends on the policy. In our case the policy is the planning algorithm of choosing the direction of motion (action) for our SSMR robot to take .
- Reward signal: It is a scalar value received by the agent due to interaction with the environment, the reward value states how good or bad was the action taken , so it generally represents the feedback of doing an action a in step time t , the goal of the agent is to maximize the total reward it receives in the long run till finishing the task . There are various systems of assigning reward values representing a process called Reward Shaping.
- Value function: A value function denotes how good it is for an agent to be in a particular state. It is dependent on the policy and is often denoted by v(s). It is equal to the total expected reward received by the agent starting from the initial state.
- State-Action value function : A state-action value function is also called the Q function. It specifies how good it is for an agent to perform a particular action in a state with a policy . The Q function is denoted by Q(s). It denotes the value of taking an action in a state following a policy .

1.2.3 Markov Decision Process

The Markov Decision Process (MDP) provides a mathematical framework for solving the reinforcement learning (RL) problem. Almost all RL problems can be modeled as MDP.MDP is widely used for solving various optimization problems. The Markov property states that the future depends only on the present and not on the past. The Markov chain is a probabilistic model that solely depends on the current state to predict the next state and not the previous states, that is, the future is conditionally independent of the past. The Markov chain strictly follows the Markov property. In a finite MDP, the sets of states, actions, and rewards (S, A, and R) all have a finite number of elements. In this case, the random variables Rt and St have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these

random variables, s0 ∈ S and r ∈ R, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r|s, a) \doteq Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{1.1}$$

In a Markov decision process, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible value for St and Rt depends only on the immediately preceding state and action, S_{t-1} , A_{t-1}, and given them, not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the state. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the Markov property [14].

The agent is devoted to maximize the sum of the rewards taken in the long run from the environment, so we denote the expected cumulative sum of reward by Gt (the return value) that is represented by the following equation .

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_t \tag{1.2}$$

Tasks in MDPs can be divided into two categories , episodic and non-episodic tasks ,episodic tasks are the tasks which terminate by reaching a final state and the agent is reset to start all over again, but in non-episodic task the environment lacks the terminal state and the task is continuous . However, in both cases a reward discounting factor is needed to discount the future rewards based on the policy followed by the agent which states the importance of future rewards in respect to immediate ones so the equation 1.3 is updated by adding the discounting factor γ .

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \doteq \sum_{k=0}^{k=\infty} \gamma^k R_{t+1+k} \tag{1.3}$$

1.2.4 Model-based Reinforcement Learning

By a model of the environment we mean anything that an agent to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward as shown in figure 2 . If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call distribution models. Other models produce just one of the possibilities, sampled according to the probabilities; these we call sample models. Model based reinforcement learning extends the traditional Q-learning by an online model and solve the problem using Dynamic Programming techniques . We usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, S, A, and R, are finite, and that its dynamics are given by a set of probabilities p(s0,r—s,

a), for all s ∈ S, a ∈ A(s), r ∈ R, and s0 ∈ S+ (S+ is S plus a terminal state if the problem is episodic). Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods can use.

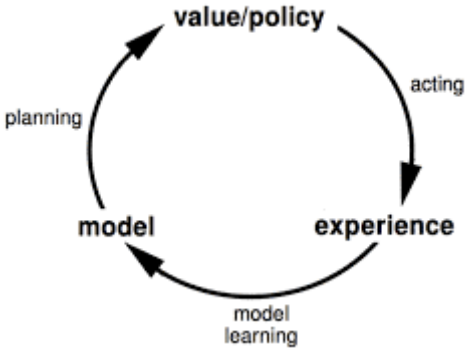

Figure 1.2: model-based RL paradigm [14]

1.2.5 Model-free Reinforcement Learning

Many reinforcement learning algorithms consider the environment as unknown, which makes the interaction between the agent and the environment based on trial and error. These algorithms are called model-free which assumes no information about the future after taking some actions. Model-free algorithms prove that this an effective way to interact with the environment for learning a specific task. These algorithms depend only the reward value at the end of each episode for the policy improvement by maximizing the expected total reward.

1.3 Aim of the project

In recent years, neural networks and deep learning approaches have been used in the majority of algorithms. However, in this study, we offer Dynamic Q-learning, a model-based form of Q-learning that integrates planning while learning. Its planning steps are based on simulations from an updatable environment model created by the agent previous experiences. Also, A comparison in terms of efficiency, convergence speed and success rates between deep Q-learning , standard Q-Learning and dynamic Q-learning is widely discussed. The aim of this thesis is to deeply explore and answer the following objectives:

- How to overcome the slow convergence of the traditional Q-learning using an environment model ?

- What is the effect of extending the Q-learning with neural networks on the performance of the robot ?
- Which variation wins the comparison in terms of efficiency ?
- How to model the motion of the mobile robot to integrate it with the proposed algorithms ?

This report shows the research work aimed to achieve these research objectives and is organized as follows:

- **Chapter 2** summarizes the three algorithms used in this literature: the standard Qlearning, dyna Q-learning, and deep Q-learning. The kinematics of the four-wheels mobile robot is also introduced in this chapter.
- **Chapter 3** introduces python implementation of the environment, the three agents, and the robot kinematics model. Then, the experiment properties are proposed.
- **Chapter 4** presents the outcomes of the three algorithms' simulations and compares them to see which one is best for the navigation challenge. The outcomes of applying these algorithms to the robot kinematic model are then shown.
- **Chapter 5** gives the conclusion and the possible future work.

Chapter 2

Background

2.1 Literature Review

In this literature, we will reveal the different navigation styles constructed using reinforcement learning while comparing their results and eventually configuring our motivation based on the literature.

In [4] an approach was proposed to redefine the states based on human reasoning . In other words, the agent is considered to be the center of the environment and will represent its states depending on its relative location from its nearest obstacle and it target goal such mimicking the human navigation behavior as When a human tries to walk safely in a dynamic environment to reach a specific goal, he/she will not care about his/her specific location or the specific distance to the target or to the obstacles. He/she will care for the relative or approximate distances and directions between him and the target, and between him and the closest obstacle; since he/she will try to deal with the obstacles one at a time. The robot divides its surrounding into 4 orthogonal regions R1,R2,R3,R4 alongside with an orientation angle as shown in figure 2.2. The state of the environment at each instant of time is determined by the region containing the target and the region containing the closest obstacle to the robot. For example if at time T, the target is in R2 and the closest obstacle is in R4,then the state at this time instant is S(R2,R4,θ). To avoid the infinite number of states due to the continuous values of θ which is in the range [0,2π] , it has been discretized into 8 angular regions and the final state representation is shown in figure 2.1. A greedy policy was followed to choose the maximum Q value at each state after the Q training is done. This new definition of state space was proven to have better convergence rate than the traditional Q-learning approach.

$$S_t = (R_g, R_o, G_n) n \in [1, 8] \tag{2.1}$$

A 3 level neural network was introduced in [3] to bypass the states discretization step . Instead, the neural network takes sensory data as an input from 8 ultrasonic sensors and outputs the 5 action Q-values . The weight W1 is used to connect the input layer and the

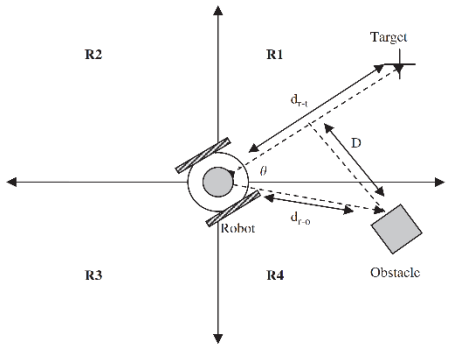

Figure 2.1: Regions and orientation of the robot

hidden layer, and similarly, the weight W2 links the hidden layer and the output layer. For the hidden units, the sigmoid function is used. The Feed forward neural network structures is used to obtain approximate Q values and the back propagation network is used to train the weights by updating its values in the training process, then after the weights are sufficiently trained, a greedy policy is to be followed to choose the max Q value in each step. In [5] a similar neural network is used but was combined with a fuzzy logic implementation.

In [2] a navigation approach for mobile robot was proposed , in which the prior knowledge is used within Q-learning. Individual behavior design was addressed using fuzzy logic. The strategy of behaviors-based navigation reduces the complexity of the navigation problem by dividing them in small actions easier for design and implementation. The Q-Learning algorithm is applied to coordinate between these behaviors, which make a great reduction in learning convergence times. The basic idea in the fuzzy logic is to imitate the capacity of reasoning and decision making from the human been, using uncertainties and/or unclear information. In fuzzy logic the modeling of a system is ensured by linguistic rules (or fuzzy rules) between the input and output variables.Each behavior has its own input and output parameters independently .The navigation behavior is subdivided into goal reaching, Obstacle avoidance, wall following, and emergency situations. States are represented by a combination of parameters.The proposed method enables the mobile robot to navigate through complex environment where a local minimum occurs and to adapt to new environments. Since the rule bases are constructed in learning-by-observation manner, the two behaviors can be autonomously realized.

In [9] , Learning was based on A clustered discrete coding of the input state space that was developed using Kohonen neural network structure as opposed to continuous internal state representation. This enabled individual state-action credit assignment, giving a more precise evaluation computation and a faster The learning algorithm was verified in simulation environment where the mobile robot performed obstacle avoidance capability which was developed by using initially unknown control strategy. In perspective, optimality of the solution obtained should be taken into account as well as an adaptive neural network structure which could reduce the size of relevant convergence rate. As shown in figure 2.3, the network structure is close to the approach used in [5] as both

require training of the weight functions in the back propagating neural network in order to obtain accurate action value functions, then a greedy policy is followed to establish the choice of each state-action pair.

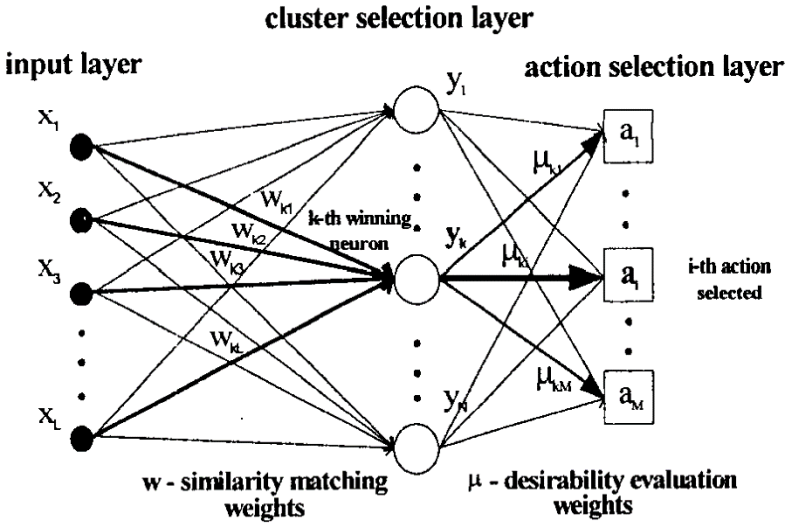

Figure 2.2: clustering neural network structure

Based on dueling network architectures for deep reinforcement learning (Dueling DQN) and deep reinforcement learning with double q learning (Double DQN), a dueling architecture based double deep q network (D3QN) is adapted in [13] . Through D3QN algorithm, mobile robot can learn the environment knowledge gradually through its wonder and learn to navigate to the target destination autonomous with an RGB-D camera only. The experiment results show that mobile robot can reach to the desired targets without colliding with any obstacles.

In [16], the paper focused on the application of the Q-learning in mobile robot navigation. In terms of the developed mobile robot CASIA-I and its working environment, an approach is proposed, used to determine the reward-penalty function of Q-learning. The experimental result showed that, under the design of the sparse reward-penalty function, the action policy obtained through Q-learning can make the mobile robot reach the destination without obstacle collision. Continuous multi-network neural Q-learning was discussed in this paper but it was not implemented.

A different variation of Q-learning was stated in [11]. A two dimensional (2D) set-up where a robot tries to learn its path through its environment by avoiding any obstacles that may be encountered on its way from its home to a final destination (a target state). During the navigation, trajectory of all the state-action pairs is stored and is replayed in a backward direction to propagate the refined Q values from any state to a target state. This effort greatly reduces the convergence rate for the Q-table as the results obtained

from the simulations indicate an excellent level of performance once compared with the traditional Q-Learning.

In [10], An investigation of whether discrete state space algorithms are a viable solution to continuous alternatives for mapless navigation was discussed . It presented an approach based on Double Deep Q-Network and employed parallel asynchronous training and a multi-batch Priority Experience Replay to reduce the training time. Results showed that this method trains faster and outperforms both the continuous Deep Deterministic Policy Gradient and Proximal Policy Optimization algorithms. Moreover, The model was trained in a custom environment built on the recent Unity learning toolkit and show that they can be exported on the TurtleBot3 simulator and to the real robot without further training. Overall the optimized method is 40 % faster in comparison to the original discrete algorithm. This setting significantly reduces the training times with respect to the continuous algorithms, maintaining a similar level of success rate hence being a viable alternative for mapless navigation.

2.2 Background on Q-learning

Learning is a model-free property of the agent, as the agent interacts with the environment by performing an action At without having any previous knowledge on the expected next state nor the reward, Thus it is a trail and error system where the agents builds a tabular function mapping each state-action pair into a next state and a reward and obtaining a database of state-action values . Eventually, after a considerable number of trials the agent constructs a policy based on the learnt Q-values and that what we call Q-learning. the learned action-value function, Q, directly approximates q*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated[1][4]. Q-Learning is an off-policy algorithm that learns about the greedy policy while using a different behaviour policy for acting in the environment/collecting data. This behaviour policy is usually an ϵ-greedy policy that selects the greedy action with probability ϵ and a random action with probability 1-ϵ to ensure good coverage of the state-action space.

$$Q^{\pi}(s,a) = E^{\pi}((\sum_{k=0}^{k=\infty})\gamma^{k} R_{t+1+k} | S_t = s, A_t = t) \tag{2.2}$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 2.3: Q-learning algorithm [14]

At first the state space is initialized with arbitrary Q values and then through sweeping along the state-action space the Q values are approximated using the previous equation (2.1). An update rule is essential to update the Q values in order to converge to the optimal Q value which is the most realistic evaluation of taking an action A in respect to a state S. A step-size parameter or a learning rate α is used to emphasize the effect of the newly updated value of each on the new estimate for the Q value. Also, a discount rate γ is adjusted to add emphasis on the future rewards,as when its value approaches unity, the agent will focus only on enhancing the future rewards and when it approaches zero the agent will just ignore the futuristic consequences on its current actions. The actions are selected greedily based on the maximum Q value till an optimal policy is generated.

2.2.1 Deep Q learning

The DQN (Deep Q-Network) algorithm was created by DeepMind in 2015. By mixing reinforcement learning and deep neural networks at scale, it was able to tackle a wide range of Atari games (some to superhuman levels). The algorithm was created by combining deep neural networks and experience replay to enhance and evolve the Q-learning mechanism as seen in figure 2.4 . Deep-Q-learning uses the neural networks as function approximators instead of going through a table of Q-values. These function approximators predict the Q-values based on the network weights then the predicted value is compared to the target value provided by the bellman equation and the loss function is calculated. Afterwards, the networks weights are updated using gradient descent and back propagation and the data fitting process takes place. To avoid high correlation, a replay memory is initialized with a preset length saving each transition taken by the agent and for each training, a batch of randomly sampled transitions are taken from the replay memory to train on.

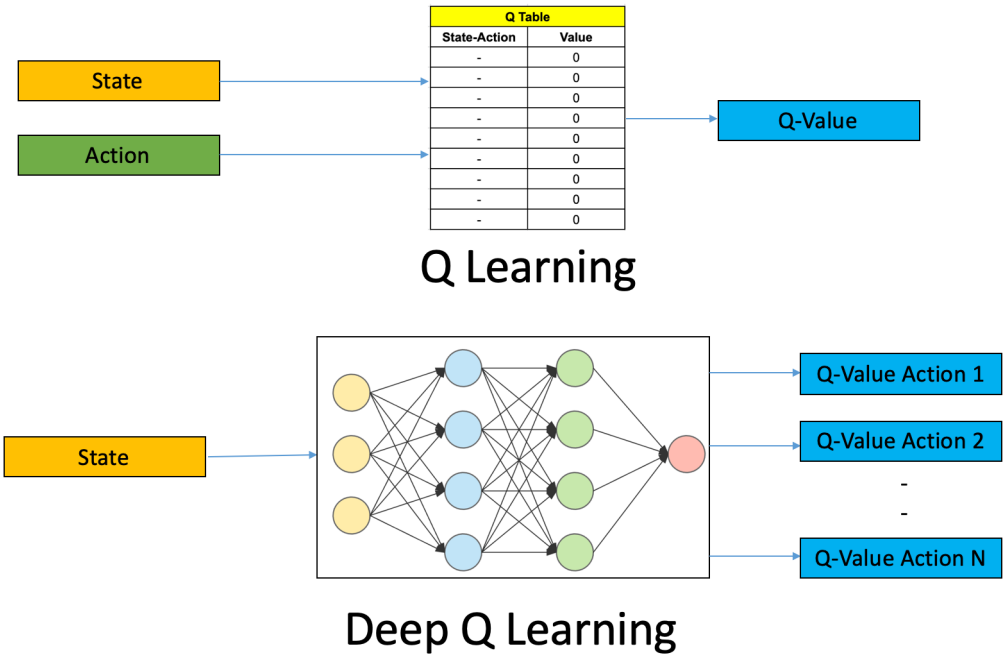

Figure 2.4: Deep Q Learning versus Q learning

Deep Q Networks

Suppose we have some arbitrary deep neural network that accepts states from a given environment as input. For each given state input, the network outputs estimated Q-values for each action that can be taken from that state. The objective of this network is to approximate the optimal Q-function. The optimal function is calculated by the Bellman equation :

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \alpha \underbrace{[R(s,a)}_{\text{Reward}} + \gamma \overbrace{\max Q'(s',a')}^{\substack{\text{Maximum predicted reward, given} \\ \text{new state and all possible actions}}} - Q(s,a)]$$

Learning Rate

Discount rate

With this in mind, the loss from the network is calculated by comparing the outputted Q-values to the target Q-values from the right hand side of the Bellman equation, and as with any network, the objective here is to minimize this loss. After the loss is calculated, the weights within the network are updated via Stochastic Gradient descent and back-propagation, again, just like with any other typical network. This process is done over and over again for each state in the environment until we sufficiently minimize the loss and get an approximate optimal Q-function.

The Network Layers

The first layer of the network is called the input layer. The input layer, consists of a number of nodes. Each of the nodes in this layer represents an individual feature from a given sample in our data set.The state of the environment is passed to the first layer as an input of the neural network. The input layer nodes usually represents the number of dimensions of the system state. Each connection between two nodes has an associated weight, which is just a number that represents the strength of the connection between the two nodes. When the network receives an input at a given node in the input layer, this input is passed to the next node via a connection, and the input will be multiplied by the weight assigned to that connection.For each node in the second layer, a weighted sum is then computed with each of the incoming connections. This sum is then passed to an activation function, which performs some type of transformation on the given sum. For example, an activation function may transform the sum to be a number between zero and one. The actual transformation will vary depending on which activation function is used. Once we obtain the output for a given node, the obtained output is the value that is passed as input to the nodes in the next layer and This process continues until the output layer is reached. The number of nodes in the output layer depends on the number of possible actions [7].

Network Training

We are simply trying to solve an optimization issue when we train a model. We're attempting to optimise the model's weights. The goal is to identify the weights that translate our input data to the proper output class the most precisely. This is the mapping that the network needs to learn.The weights are optimized via a process known as optimization. The optimization process is determined by the optimization algorithm used. The selected algorithm is sometimes referred to as an optimizer. Stochastic gradient descent, or SGD for short, is the most well-known optimizer. The loss is the error or difference between what the network is predicting versus the ideal value, and The SGD will to try to minimize this error to make our model as accurate as possible in its predictions. To sum up, we feed the data repeatedly to the network and the networks starts to learn in the long run by fitting the predictions to the optimal values through optimizing the weights.

Experience Replay And Replay Memory

The agent experience at each time step is stored in a data set called the replay memory. The agent experience e_t consists of the agent's transition data and it is defined as this tuple :

$$e_t = (s_t, a_t, r_t, s_{t+1}) \tag{2.3}$$

This tuple contains the state of the environment s_t, the action a_t taken from state s_t , the reward r_t given to the agent at time t as a result of the previous state-action pair , and the

next state of the environment s_{t+1}. This tuple indeed gives us a summary of the agent's experience at time . All of the agent's experiences at each time step over all episodes played by the agent are stored in the replay memory. The memory is maintained at a a finite size of N so that it will store the last N experiences. After each episode a number of random samples is taken from the memory to train the network. The replay memory is sampled randomly to avoid high correlation between data samples as the batch size is selected randomly and the data can be relatively independent on each other. A summary of replay memory usage with deep Q networks can be written as follows :

1. Initialize replay memory capacity.
2. the network with random weights.
3. For each episode:
   (a) Initialize the starting state.
   (b) For each time step:
      i. Select an action.
      ii. Execute selected action in an emulator.
      iii. Observe reward and next state.
      iv. Store experience in replay memory.
      v. Sample random batch from replay memory.
      vi. Preprocess states from batch.
      vii. Pass batch of preprocessed states to policy network.
      viii. Calculate loss between output Q-values and target Q-values. Requires a second pass to the network for the next state.
      ix. Gradient descent updates weights in the policy network to minimize loss.

Extending Deep Q Network with a Target network

When updating weights , our outputted Q-values will update, but so will our target Q-values since the targets are calculated using the same weights. So, our Q-values will be updated with each iteration to move closer to the target Q-values, but the target Q-values will also be moving in the same direction, thus instability occurs as the network will be chasing its tail as the targeted Q-value is not fixed. The target network is a clone of the policy network. Its weights are frozen with the original policy network's weights, and we update the weights in the target network to the policy network's new weights every certain amount of time steps. As it turns out, this removes much of the instability introduced by using only one network to calculate both the Q-values, as well as the target Q-values. The algorithm only differs in the last couple of steps, but it differs greatly in terms of efficiency and stability. The updated version of the algorithm can be summarised as follows :

1. Initialize replay memory capacity.
2. the network with random weights.
3. For each episode:
   (a) Initialize the starting state.
   (b) For each time step:
      i. Select an action.
      ii. Execute selected action in an emulator.
      iii. Observe reward and next state.
      iv. Store experience in replay memory.
      v. Sample random batch from replay memory.
      vi. Preprocess states from batch.
      vii. Pass batch of preprocessed states to policy network.
      viii. Calculate loss between output Q-values and target Q-values. Requires a second pass to the Target network for the next state.
      ix. Gradient descent updates weights in the policy network to minimize loss.After time steps, weights in the target network are updated to the weights in the policy network.

2.2.2 Kinematic model of a four wheeled mobile robot

In this section a mathematical description of a four wheeled robot moving on a planar surface is formulated.To consider the kinematic model of a robot , it is assumed that the robot is placed on a plane surface with the inertial orthonormal basis (Xg, Yg, Zg) . A local coordinate frame denoted by (xl , yl, zl) is assigned to the robot at its center of mass (COM). According to 2.5, the coordinates of COM in the inertial frame can be written as COM = (X, Y, Z).

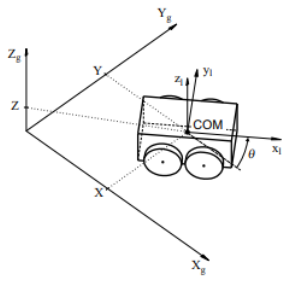

Figure 2.5: Robot in the inertial frame [6]

Since in this paper the plane motion is considered only, the Z-coordinate of COM is constant.Suppose that the robot moves on a plane with linear velocity expressed in the

local frame as v = $[vx\ vy\ 0]^T$ and rotates with an angular velocity vector $\omega = [0\ 0\ \omega]T$. If q = $[X\ Y\ \theta]^T$ is the state vector describing generalized coordinates of the robot (i.e., the COM position, X and Y , and the orientation θ of the local coordinate frame with respect to the inertial frame), then $\dot{q} = [\dot{X}\ \dot{Y}\ \dot{\theta}]^T$ denotes the vector of generalized velocities. From 2.6 it can be noted that the variables $\dot{X}$ and $\dot{Y}$ are related to the coordinates of the local velocity vector and it can be shown that $\omega = \dot{\theta}$.

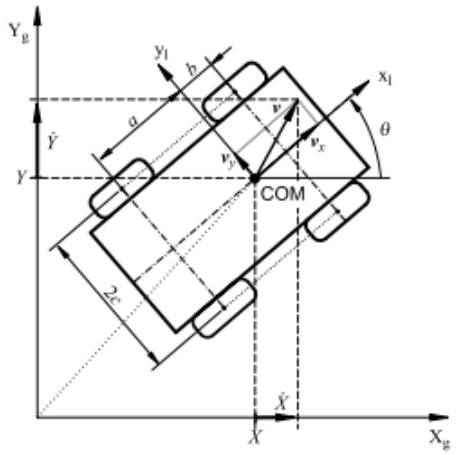

Figure 2.6: Free body diagram [6]

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}. \tag{2.4}$$

It is obvious that equation 2.11 does not impose any restrictions on the robot plane movement, since it describes free-body kinematics only. Therefore it is necessary to analyze the relationship between wheel velocities and local velocities.Suppose that the i-th wheel rotates with an angular velocity ω_i(t), where i = 1,2,3,4, which can be seen as a control input. For simplicity, the thickness of the wheel is neglected and is assumed to be in contact with the plane at point P_i as illustrated in Fig 2.7. In contrast to most wheeled vehicles, the lateral velocity of the robot, v_{iy}, is generally nonzero. This property comes from the mechanical structure of the robot that makes lateral skidding necessary if the vehicle changes its orientation. Therefore the wheels are tangent to the path only if $\omega = 0$, i.e., when the robot moves along a straight line.

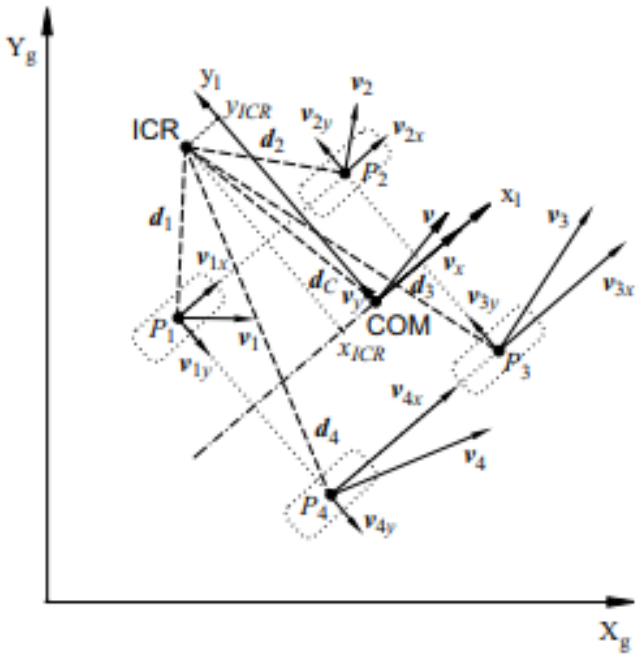

Figure 2.7: wheel velocities[6]

To develop a kinematic model, the robot body rotation with respect to an instantaneous centre of rotation point ICR is defined through the radius vector $d_i = [d_{ix}\ d_i]^T and\ d_C = [d_{Cxd}\ d_{Cy}]^T$. Consequently, based on the geometry of figure 2.7 , it holds that:

$$\frac{v_x}{Y_{ICR}} = \frac{v_y}{X_{ICR}} = \omega \tag{2.5}$$

and From figure 2.7, the coordinates of vectors d must satisfy the following relationships:

$$d_{1y} = d_{2y} = d_{CY} + c \tag{2.6}$$

$$d_{3y} = d_{4y} = d_{CY} - c \tag{2.7}$$

$$d_{1x} = d_{4x} = d_{CY} - a \tag{2.8}$$

$$d_{2x} = d_{3x} = d_{CY} + b \tag{2.9}$$

Eventually by combining the past equation and assuming a constant effective wheel radius r for all wheels, we can conclude equation

$$\begin{bmatrix} V_L \\ V_R \\ V_F \\ V_B \end{bmatrix} = \begin{bmatrix} 1 & -c \\ 1 & c \\ 0 & -X_{ICR} + b \\ 0 & -X_{ICR} - a \end{bmatrix} \begin{bmatrix} v_x \\ \omega \end{bmatrix}. \tag{2.10}$$

$$\begin{bmatrix} v_x \\ \omega \end{bmatrix} = r \begin{bmatrix} \frac{\omega_L + \omega_R}{2} \\ \frac{-\omega_L + \omega_R}{2c} \end{bmatrix} \tag{2.11}$$

The accuracy of the previous equation depends on the longitudinal slip and can be valid if this longitudinal slip is not dominant. Finally, to complete the kinematic model, the lateral velocity vy is constrained by the rotation of the robot and can be expressed by the following equation:

$$v_y + X_{ICR}\dot{\theta} = 0 \quad X_{ICR} \in [-a, b] \tag{2.12}$$

To be able to use the previous mathematical model we must assume the following conditions :

1. The robot centre of mass coincides with its body geometric center.
2. The contact between the ground and the wheels is reduced to a point contact.
3. The rolling resistance force of the wheels are negligible.
4. Each side's two wheels has the same rotating speed.
5. The normal forces applied to the robot from the ground are equally distributed among the four wheels.
6. The robot motion is on a flat surfaces and the four wheels are always in contact with the ground surface.

Under these assumptions, the relation between the wheels velocity vector $[\omega_L\ \omega_R]^T$ and the generalized velocity vector $[\dot{X}\ \dot{Y}\ \dot{\theta}]^T$ can be derived as shown in :

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r/2 & r/2 \\ \frac{X_{ICR}r}{2c} & \frac{-X_{ICR}r}{2c} \\ \frac{-r}{2c} & \frac{r}{2c} \end{bmatrix} \begin{bmatrix} \omega_L \\ \omega_R \end{bmatrix} \tag{2.13}$$

Chapter 3

Methodology

3.1 Python implementation

3.1.1 Software Specifications

The software requirements used in the implementation :

- Python 3.9.6
- Keras 2.5
- Tensorflow 2.5
- numpy 1.9.1
- scipy 1.7
- Gym 2.0
- Pyglet 1.5

3.1.2 Simulation Environment

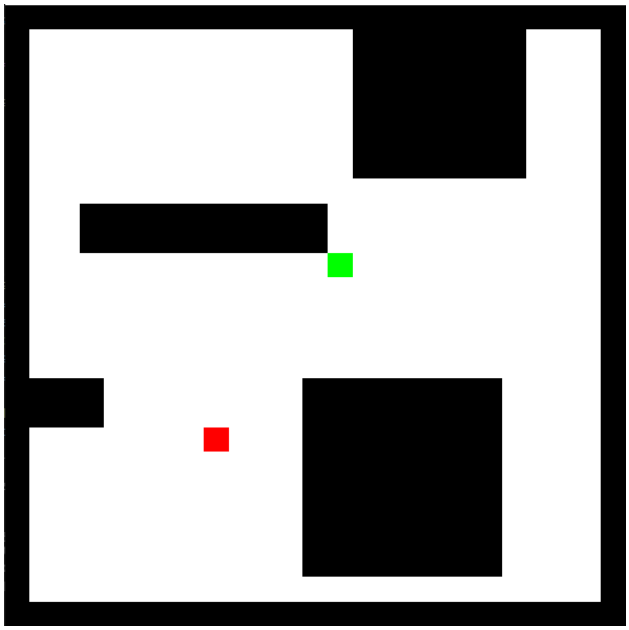

Figure 3.1: Obstacle Maze (25x25) [15]

In order to simulate different obstacles configurations, the library gym-path-finding developed by [15] is used. The library enables creating discrete navigation environments with obstacles at different configurations by manipulating the environment class and choosing a different number for seed creation.Figure 4.2 shows an example of an obstacle grid-world like environment (25x25) showing the goal as a red state and the state occupied by the agent is showed as a green square.The environment chosen is a square grid of 25 rows and 25 columns, the states are simply the row and column numbers (r,c) of each square on the grid so that the environment has 625 different states and 4 available actions(up,down,left and right) per each state. The environment is fully deterministic, meaning that each action maps a state to a distinct next state.As any other OpenAi gym environment, our environment has the following features :

- action_space : it is an array representing the set of discrete actions. In this grid, there are 4 possible actions up,down,left,right and they can be represented by an array of 4 items [0, 1, 2, 3].
- Observation space : set of values reflective of the environment state that the agent has access to. In other words, they are pieces of information that represent each state of the environment, so in a maze like environment the observation space is represented as a two dimensional array (25 by 25) filled with number representing the environment specs. The free cells are represented by a 0, obstacles represented by 1, the agent is represented by a 2 and finally the target is represented by 3. For example the agent is at (10,23) in 3.2 and the target is at (17,4).

```
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 2 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 3 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]]
```

Figure 3.2: Environment State

- Step function : the step function is the core of the environment class, it is responsible for moving the agent step and updating the environment state as well as returning the observation resulting from taking a certain action. Also, it includes the reward function which is vital in evaluating the agent's actions thus representing the environment reaction towards different actions taken by the agent. Reward shaping is the most significant part of formulating and solving the RL problem. Various techniques were used in the literature such as sparse and binary reward functions but they resulted in corrupted and similar Q values for the deep-Q agent so a potential reward shaping was added to put more emphasis on the agent actions and to prevent the existence of congruent Q-values in both deep and standard Q-learning algorithms. The reward function can be described by the following set of equations :

$$R_t = \begin{cases} \Phi(s) = -0.1(e), \quad e = \sqrt{(X_{target} - X_{agent})^2 + (Y_{target} - Y_{agent})^2} \\ -10 \quad hitting\ an\ Obstacle \\ 10 \quad reaching\ the\ Target \end{cases}$$

- seed(spawn_seed) : this method takes an integer parameter and it is basically used to switch between different configurations of the environment based on the input entered by the user.

3.1.3 The Agent

The agent is the program or the mind of our robot which is able to intelligently plan and take actions and receive rewards. Four agents are implemented, three for testing

and comparing the Q-learning, Dyna Q-learning and Deep Q-learning algorithms and the final agent is the winner algorithm integrated with the kinematic model of the robot.

Standard Q-learning and Dyna-Q Agent Implementation

Both the standard Q-learning and the Dyna-Q agents are implemented in the same class as they both have the same attributes and build up, despite the fact that the Dyna-Q agent contains a planning algorithm which is discussed below. has many attributes initialized with specifically chosen values that play a substantial role in the algorithm efficiency.The values and the nature of these parameter are shown in table 3.1. First of all, the exploration vs exploitation trade-off can be handled by using a variable exploration Rate ϵ which starts by a relatively large value (0.7) to encourage exploration in the first episodes where the agent should focus on discovering the environment states and attaining primary value functions for them to build its own model of environment.However, by the end of each episode it is discounted by the decay factor until it converges to its minimum limit such getting an efficient Exploration-exploitation strategy in the long run. Moreover, both the learning rate α and the discount factor γ values are assigned to a constant value (0.9) and the choice of the values is based on a trial and error study in order to get the most appropriate combination of values.Training sessions trials are limited to 1000 episodes and each episode own trials is initially set to 500 steps per episode and reassigned to the length of the shortest discovered trajectory to the target.

Parameter	Value	Behaviour
Exploration rate (ϵ)	$70\% \rightarrow 1\%$	decays episodically
Learning rate (α)	0.8	constant
Discount factor (γ)	0.9	constant
Planning steps (n)	100	constant
Epsilon decay factor	0.9	constant
Maximum trials per episode	500→ shortest path	assigned to shortest discovered path continuously
Maximum episodes count	1000	constant

Table 3.1: Experimental parameters

Planning-steps parameter n is the computational intensive part of the algorithm as it represents the number of simulated steps planned by the agent per each real action taken.The standard Q-learning algorithm has zero planning steps and our Dyna-agent has been trained among a variety of of planning steps to deduce the optimum value of n

which was found to be 100. A simple representation of different n values is shown against cumulative rewards and episodic steps in figures 3.2 and 3.3.

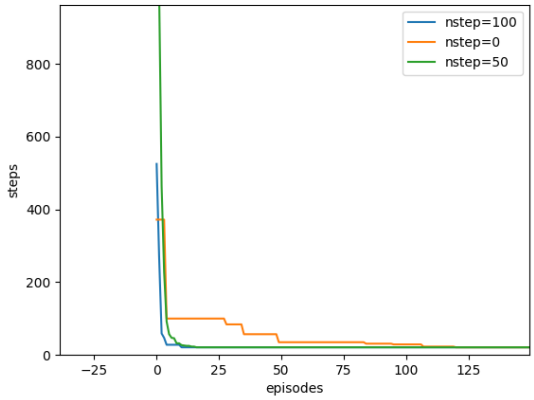

Figure 3.3: steps comparison

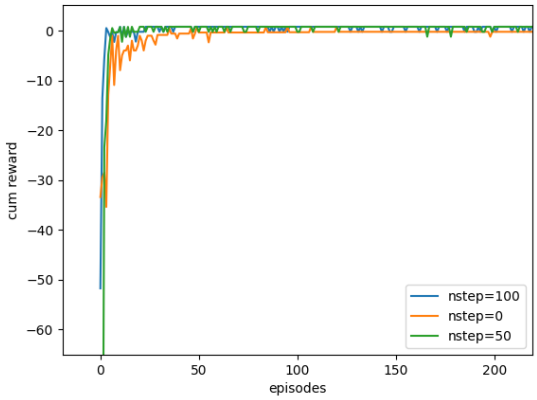

Figure 3.4: reward comparison

After selecting the parameters, methods play their role in simulating the Dyna-Q agent.The core methods of the class can be listed as follows:

- ChooseAction(self) : this method takes no external parameter rather than the agent itself, it chooses the action based on the ϵ-greedy policy and returns the appropiate action in each step.
- Randomstart() : this method was implemented to randomize the starting point of the agent to test the algorithm robustness.
- Reset(self) : this method is used to end the episode, resetting all the conditions and starting a new one by placing the agent in the starting state or in a random one if the Randomstart() method is enabled.
- play(self) : this method is the core of the training process where all learning and planning takes place .The method loops over the number of permitted episodes and inside each episode it internally loops till the episode terminates by the agent exceeding the maximum number of steps or by reaching the goal.For each step and action taken by the agent representing the real experience, n planning steps are executed by looping n times updating the Q values randomly of the agent's model of previously visited states . so both Real-learning and model-learning occur inside this method.

DQN Agent Implementation

The deep Q-learning agent uses the neural network architecture to approximate state-action pairs values knows as Q-values, the agent simply inputs its current state into the

input layer of neural network and the network outputs the Q-values of the every possible action. At first, the approximations are random and miss accuracy as the weights of the network are randomly initialized, but the chosen optimizer (Adam) calculates the loss function using mean squared error difference between the approximations and the real Q-values given by the bellman equation shown below. Eventually, after many training episodes the weight values are updated using back propagation and gradient descent and eventually the predicted values converge to the targeted values.

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount rate}} \overbrace{\max Q'(s',a')}^{\substack{\text{Maximum predicted reward, given} \\ \text{new state and all possible actions}}} - Q(s,a)]$$

[7]

DQN Architecture

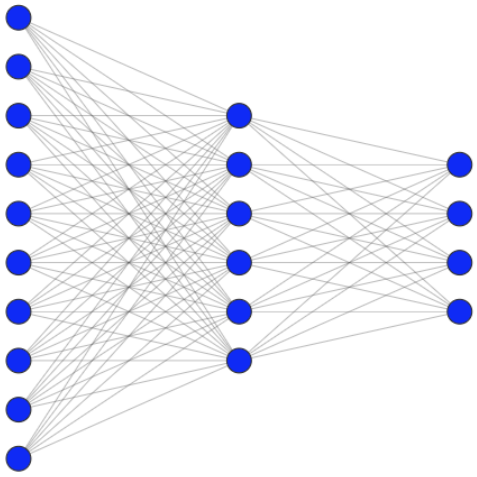

Figure 3.5: neural network structure

The main neural network was built using Keras package in build_model method , the architecture of the main network is shown in 3.5. The network below consists on an input layer, single hidden layer and an output layer. It was found by trial and error that the network layers should have the following specs.

- Input layer dimension should have the same size as the input state array (the environment state dimension), In a 25 by 25 environment the input layer shall be 625 .

- Output layer dimension should have the same size as the action space of our environment which is 4 .
- The hidden layer size should be a number in between input and output sizes and it was chosen to be 32 .
- activation function "tanh" was used for the input and hidden layer and a linear activation function is used for the output layer.
- Adam optimizer is used with a learning rate of 0.001 and the loss function was chosen to be "MSE" (mean squared error).

A copy of the network called the target network is used to stabilize the learning process. In deep Q-learning, the agent updates the value of executing an action in the current state, using the values of executing actions in a successive state. This procedure often results in an instability because the values change simultaneously on both sides of the update equation. A target network is a copy of the main network that is kept frozen and less frequently updated than the main network so that stability and convergence is granted instead of chasing a running target and looping forever. This was implemented by creating a counter to update the target network weights frequently[12].

DQN Agent Prameters

Hyper-parameters	Value	Behaviour
experience replay memory size	20000	constant
Exploration rate (ϵ)	$1 \rightarrow 0.01$	decays episodically
Epsilon decay factor	0.997	constant
network update counter (τ)	100	constant
batch size	64	constant
Learning rate (α)	0.001	constant
Discount factor (γ)	0.998	constant
Maximum trials per episode	2000	constant
Maximum episodes count	2500	constant

Table 3.2: Experimental parameters

The hyper-parameters values of the DQN-agent shown in 3.2 are settled after tremendous amount of trials to obtain the best converging performance. The replay buffer size is set to

store the last 20000 experience and the out dated ones are eliminated. A batch size of 64 random experiences is sampled from the memory after each episode. A huge exploration rate and a slow decay rate is a must at the first episodes so that the agent can visit all the states and do most of the possible action pairs. The target network is updated every 100 episodes to provide more robustness to the algorithm. The agent steps per episode were limited to 2000 steps only to execute the risk of the infinite loops.

DQN agent methods

- build_model(self): this function uses keras library to build the main network of the architecture stated previously. An input layer of 625 neurons with "Tanh" activation function and a middle hidden layer of 32 with "Tanh" function activation and finally an output layer of 4 neurons one with linear activation function . Adam optimizer is called with a learning rate of 0.001 and a Mean squared error loss function. This method is used to build a copy of the main network that is called a target network .

- remember(state,action,reward,done,next_state): This method stores the experience of the agent step by step and append it to the replay memory of the agent.

- Replay(self,batch-size): This is the method in where the training takes place. It takes a random batch sample from the replay memory and trains the network weights to fit the neural network predictions to converge to the targeted Q-values of the bellman equation . The prediction and fitting processes are done by the built-in functions of the keras package model.predict() and model.fit().

- Update_target_network(self) : This method updates the frozen target networks weights by overwriting the old weights by the new adjusted weights of the main network when number of the frozen episodes reaches the update counter.

- act(state) : It chooses the action of the current state using epsilon-greedy technique, either to choose a random action or the predicted one based on the epsilon value.

Kinematic Robot Agent Implementation

A class has been built to simulate the motion of a four wheeled mobile robot with its geometric parameters and the kinematics presented in section 2.1 . The robot model is trained in a simulation environment similar to the real-life environment. The agent is simulated in our grid environment and uses its step function. To deal with the kinematic equations in this class, we have to deal with continuous state values so a virtual environment is created with width of 4 meters and a length of 4 and the agent state will be the agent coordinates and orientation (x, y, θ).The agent coordinates are then translated and mapped to the grid-world environment using an approach similar to tile-coding as well as the agent orientation will be discretized to range from (-10 as -180 to 10 as 180)[1].

For example if our agent location is x = 2.5m, y = 3.5m, and θ is 90 degree, the discretized state should be (15, 21, 5).The following functions are built to help creating the environment and controlling the kinematic class :

- deltax(): It updates the x position of the robot using the kinematic equations . In this method, ω is calculated then vx , vy and ˙x are calculated too and finally update the x position.
- deltay(): It updates the y position of the robot using the kinematic equations. In this method, ω is calculated then vx , vy and ˙y are calculated too and finally update the y position.
- deltaTheta(): It updates the orientation θ of the robot using the kinematic equations. In this method, ω is calculated then θ is updated
- get discretized state(): This function takes nothing and returns x discrete , y discrete and θ discrete. x and y are discretized to range between 0 and the real environment width and length respectively. θ is discretized to be an integer ranges between -10 and 10 (-10 as -180 and 10 as 180).
- take step(): It sets ω_r and ω_l based on the action taken in this step then it calls the function step().The list of possible actions can be listed as :
  1. Forward : setting $\omega_r = \omega_l$, $\omega_r > 0$, $\omega_l > 0$
  2. Backward : setting $\omega_r = \omega_l$, $\omega_r < 0$, $\omega_l < 0$
  3. Left : setting $\omega_r =$ -ω_l, $\omega_r > 0$, $\omega_l < 0$
  4. Right : setting $\omega_r = \omega_l$, $\omega_r < 0$, $\omega_l > 0$
- Step() : this function takes nothing and returns nothing. It calls 3 other methods which responsible for updating the location of the robot each step. These 3 methods are deltax(), deltay() and deltaTheta().

Dyna-Q Robot Parameters

Parameter	Value	Behaviour
Exploration rate (ϵ)	70% $\rightarrow$ 0.001%	decays episodically
Learning rate (α)	0.8	constant
Discount factor (γ)	0.9	constant
Planning steps (n)	100	constant
Epsilon decay factor	0.99	constant
Maximum trials per episode	500$\rightarrow$ shortest path	assigned to the average of the shortest discovered path and current path
Maximum episodes count	4000	constant

Table 3.3: Experimental parameters

The Dyna-Q robot parameters are shown in 3.3. The robot starts with a relatively high exploration rate that decays by a decay factor of 0.99 episodically in order to increase the exploration phase in the early learning phases and to encourage exploitation in advanced episodes. Moreover, both the learning rate α and the discount factor γ values are assigned to a constant values 0.8, 0.9 and the choice of the values is based on a trial and error study in order to get the most appropriate combination of values.Training sessions trials are limited to 4000 episodes and each episode own trials is initially set to 500 steps per episode and reassigned to the average of the shortest discovered trajectory to the target and the latest path discovered .

Chapter 4

Results

In this chapter, the reinforcement algorithms presented in previous chapters are validated using a Python-based environment. Validating the designed algorithms using simulations is desirable before testing them on the real set-up since the real-time experiments are significantly long. In this chapter, the reinforcement learning algorithms are validated and compared using a Python-based environment explained in the previous chapter.

4.1 Comparing the Algorithms

Reinforcement learning algorithms are compared based on the following performance measures :

- The cumulative reward: It is defined by the sum of scalar rewards harvested over a training episode. It provides a quantitative measure to the quality of the robot trajectory all over the episode. If the movement trajectory avoids obstacles and reaches the goal in minimum steps, the effect of the positive reward received when reaching the target dominates the rewards sum. In contrast, if the major portions of the trajectory surround obstacles or slippery areas, the negative rewards received will dominate the sum. The ideal behaviour of the cumulative rewards during the training is explained by figure 4.2.

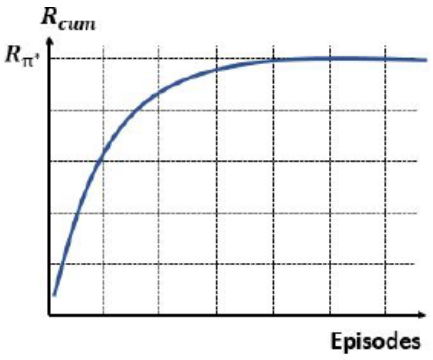

Figure 4.1: ideal cumulative reward behaviour [1]

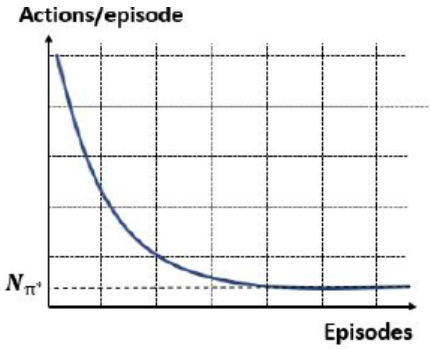

Figure 4.2: ideal number of steps behaviour [1]

- **The number of actions per episode**: The length of a trajectory is characterized by the number of actions executed between the initial and the target state. In initial training episodes, the exploration phase dominates. Therefore, the number of actions taken to reach the goal is large. Once the action-values start to converge, the exploitation phase follows the learned optimal policy and the number of steps taken to reach the goal decreases. The ideal behaviour of the episode length during the training is explained by figure 4.1.

- **The convergence speed**: It is characterized by the number of episodes required to converge to an approximate optimal policy. Thus, it is usually desirable to minimize such a number to achieve high efficiency.

- **success rate** : The training episode terminates if the goal is reached or if the number of actions exceeds a predefined trials number, success rate is percentage of episodes in which the agent reached the target over the total number of terminated episodes.

- **Training time** : It is the time required by the agent in order to converge to an optimal policy .

4.1.1 Standard Q-learning algorithm

The Q-learning algorithm has been simulated for 1000 episodes using the Dyna-Q agent class and setting the planning steps to zero. The environment in figure 3.1 is used.The performance factors are calculated and shown in table 4.1

Measure	Value
Maximum cumulative reward	0.799
Convergence speed	170 episodes
Success Rate	65.8 %
Number of actions of the optimal policy	21 actions
Training Time	20 minutes

Table 4.1: Q-learning training results

As shown in table 4.1 the Q-learning agent took 170 episodes to converge to its optimal policy which enables the agent to reach the goal in 21 steps only with a cumulative reward of 0.799 and a success rate of 65.8%. The Q-learning agent converges in a relatively slow manner with an undesirably small success rate. The convergence of both the cumulative reward and the agent steps_per_episode can be viewed in the plots 4.4 and 4.3. Also, the agent optimal path is shown in 4.8.

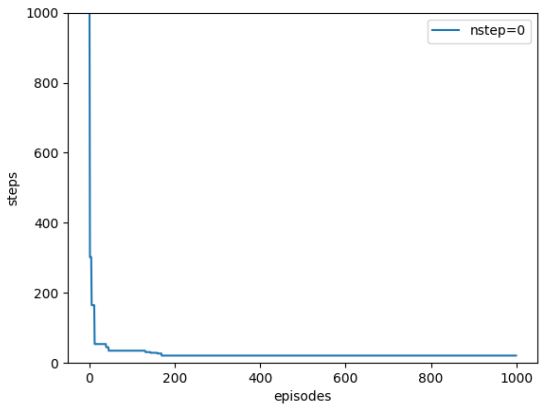

Figure 4.3: the number of executed actions per training episode of Q-learning agent.

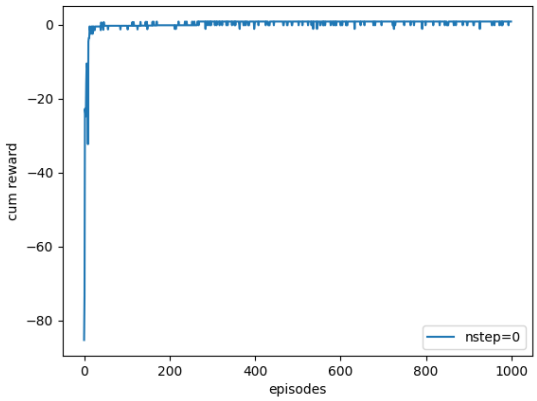

Figure 4.4: the maximum cumulative reward per training episode of Q-learning agent.

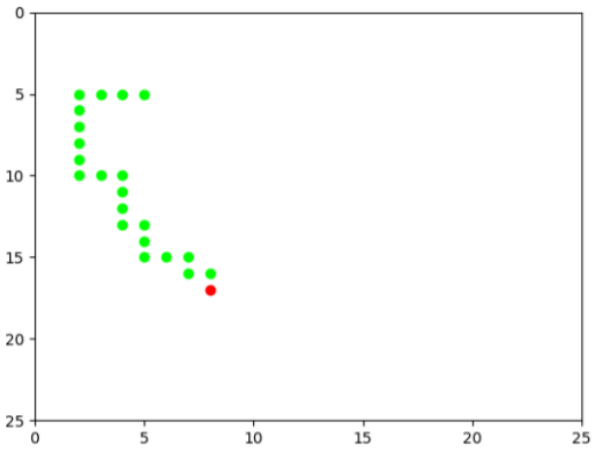

Figure 4.5: The optimal path taken by the agent

4.1.2 Dyna-Q algorithm

The Dyna-Q algorithm is simulated for 1000 episodes and performing 100 hypothetical planning steps(n=100) after each real interaction experience as explained in the previous chapter.Training results in 4.2 had shown that the agent converges in 24 episodes only which is much faster than its Q-learning peer.However, both agents reached the same optimal trajectory by executing 21 actions and having a maximum cumulative reward of 0.799 ,but the Dyna-agent owns a much higher success rate of 86.2%.

Measure	Value
Maximum cumulative reward	0.799
Convergence speed	24 episodes
Success Rate	86.4 %
Number of actions of the optimal policy	21 actions
Training Time	45 minutes

Table 4.2: Dyna-Q training results

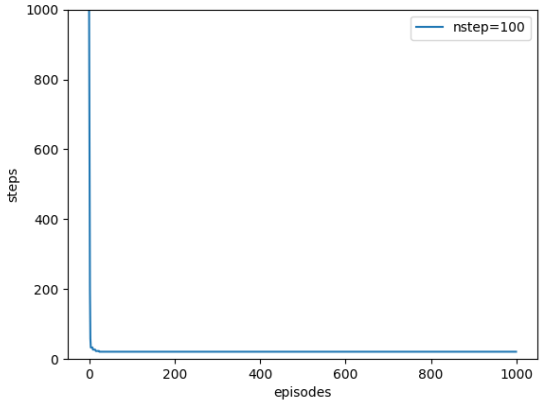

Figure 4.6: the number of executed actions per training episode of Dyna-Q agent.

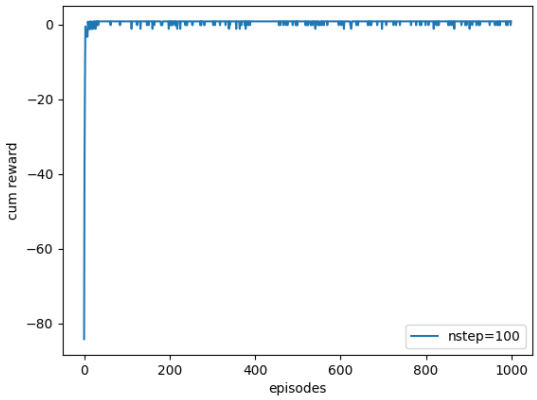

Figure 4.7: the maximum cumulative reward per training episode of Dyna-Q agent

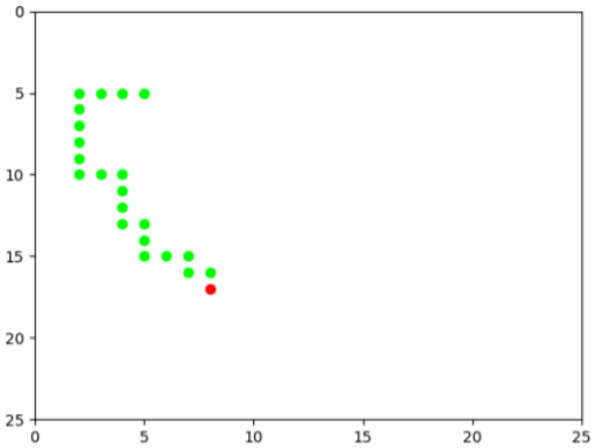

Figure 4.8: The optimal path taken by the agent

In conclusion, it is crystal clear to deduce from figures 4.6,4.7 that the Dyna-Q agent has a brighter performance than the traditional Q-learning agent in terms of the number of successful episodes and convergence speed.In conclusion, despite the Dyna-Q algorithm requires more computational power , it takes the upper hand in all the performance measures compared to the traditional Q-learning agent.The differences can be clearly shown in table 4.3 for both agents trained in the environment shown before in figure 3.1. The Dyna-Q agent path is shown in and it can be deduced that both algorithms reached the same optimal policy.

Measure	Q-learning agent	Dyna-Q agent
Maximum cumulative reward	0.799	0.799
Convergence speed	170 episodes	24 episodes
Success Rate	65.8 %	86.2 %
Number of actions of the optimal policy	21 actions	21 actions
Training Time	20 minutes	45 minutes

Table 4.3: Dyna-Q and Q-learning agents comparison

4.1.3 DQN Algorithm

DQN Algorithm is simulated for 2500 episodes in maze 3.1, each episode is limited to a maximum of 2000 steps to avoid infinite loops. The network is trained by sampling a batch from the replay buffer episodically. The agent performance was random in the first hundred episodes as the epsilon value was high in order to explore the whole state-action space. The agent performance slowly converged and by the end of 2500 episodes, the agent had learnt an optimal path and successfully learned to minimize the negative reward as shown in 4.11 and 4.9 .

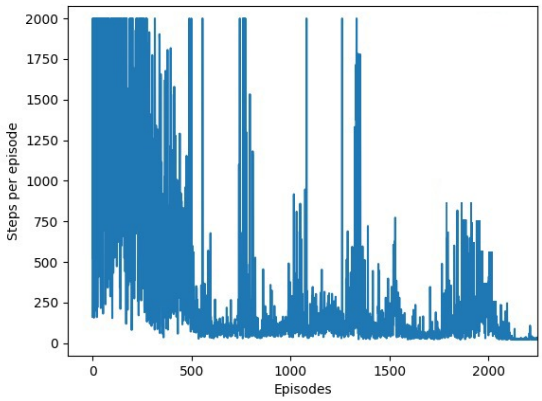

Figure 4.9: the number of executed actions per training episode of DQN agent.

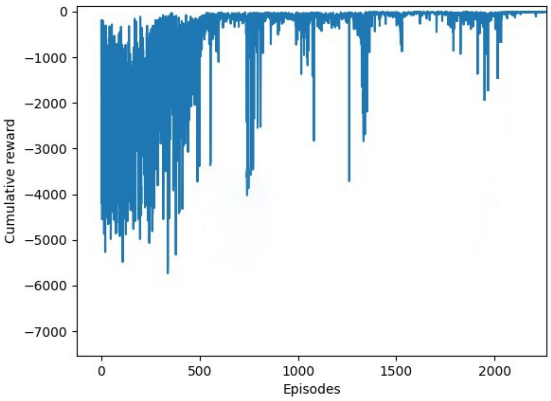

Figure 4.10: the maximum cumulative reward per episode of DQN agent

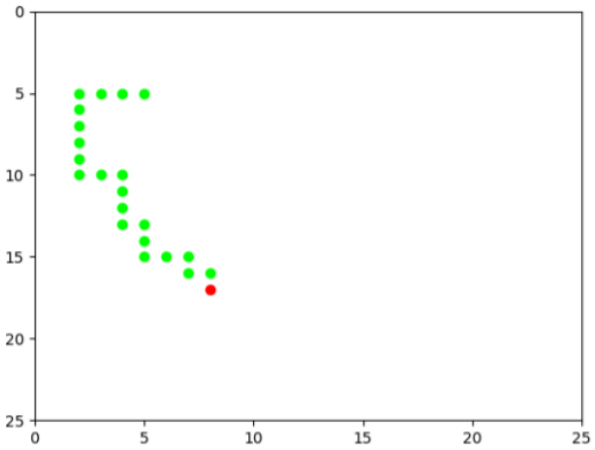

Figure 4.11: The optimal path of the agent

The DQN agent training is computationally intensive as the agent took nearly 6 hours to train for 2500 episodes. It took 600 episodes to converge to discover the optimal policy and 2200 episodes to reach a satisfactory success rate.

Measure	Value
Maximum cumulative reward	0.799
Convergence speed	600 episodes
Success Rate	78 %
Number of actions of the optimal policy	21 actions
Training Time	360 minutes

Table 4.4: Deep Q Learning agent

4.2 Robustness of The chosen algorithm

4.2.1 stochasticity of the starting point

The robustness of Dyna-Q algorithm can be tested by forcing the agent to start from a random position in each training episode, for this sake a method called Randomstart() is created to randomize the staring point in the beginning of each training episode. A dyna-Q agent with 100 planning steps (n=100) is simulated for 1000 episodes and allowed to have up to 500 trails per episode and its performance is plotted against the total number of episodes.

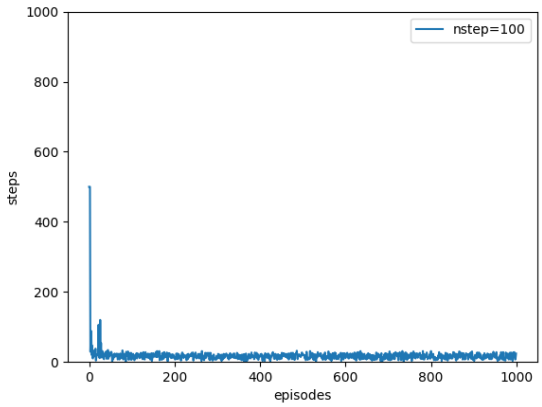

Figure 4.12: the number of executed actions per training episode of random Dyna-Q agent.

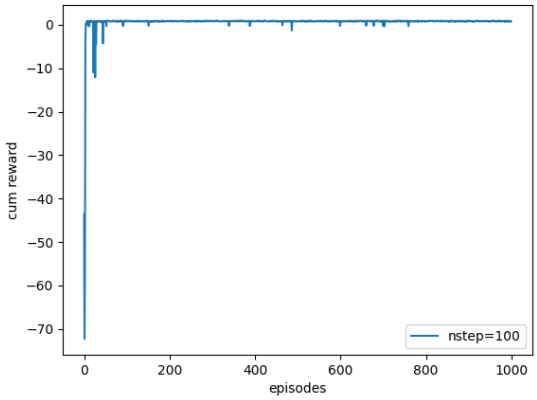

Figure 4.13: the maximum cumulative reward per training episode of random Dyna-Q agent

In figure 4.12 the agent succeeded in reaching the goal in almost all of the episodes with an excellent convergence speed as after 100 episodes the agent can easily figure out

short and effective trajectory towards the goal.Also, the agent learned how to get the maximum cumulative reward from random episodes as the plot 4.13 converges after 100 episodes. To sum up, the algorithm is robust to random starting points and the agent always learned to reach the goal after sufficient training episodes (100) in the shortest trajectory while attaining a maximum cumulative reward.

4.2.2 Invariance of the environment

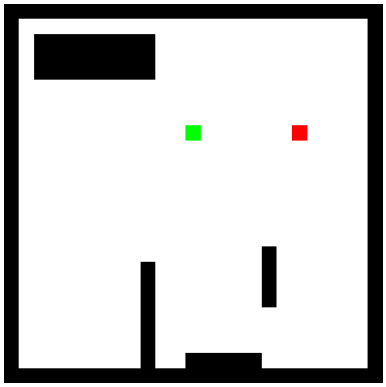

Figure 4.14: env (a)

Figure 4.15: env (b)

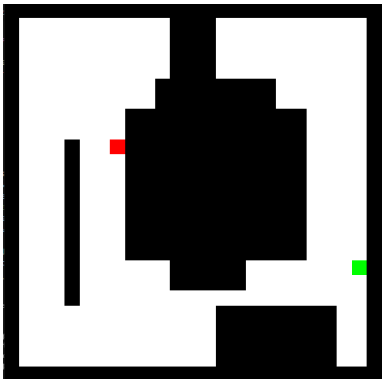

Figure 4.16: env (c)

Another way of testing the algorithm robustness is to train it in different environments to see if the algorithm is map-invariant or not.The 100-step Dyna-agent is trained in the environments shown in figures 4.14,4.15 and 4.16 for 1000 episodes and its performance is measured and plotted by the following tables and plots .

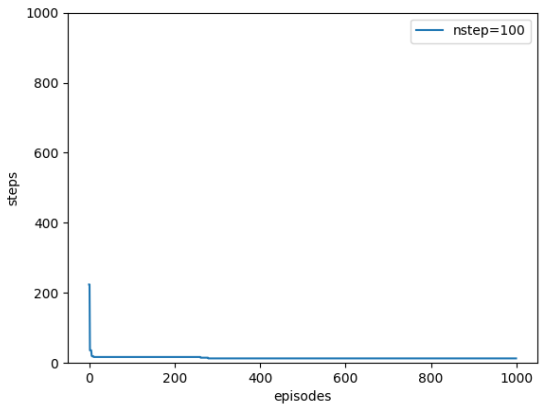

Figure 4.17: the number of executed actions per training episode of Dyna-Q agent in env (a) .

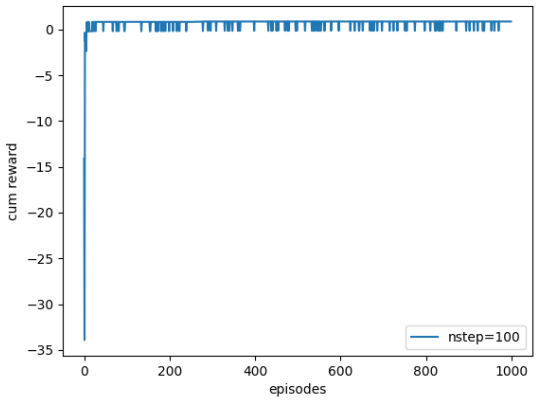

Figure 4.18: the maximum cumulative reward per training episode of Dyna-Q agent in env (a).

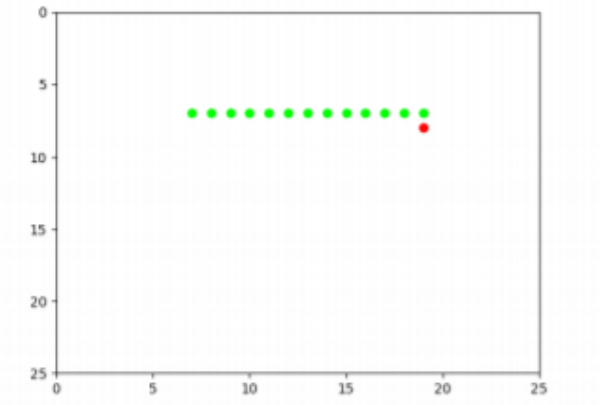

Figure 4.19: The optimal path of the agent in environment a

Measure	Value
Maximum cumulative reward	0.88
Convergence speed	29 episodes
Success Rate	89.2 %
Number of actions of the optimal policy	13 actions

Table 4.5: Dyna-Q training results in env (a).

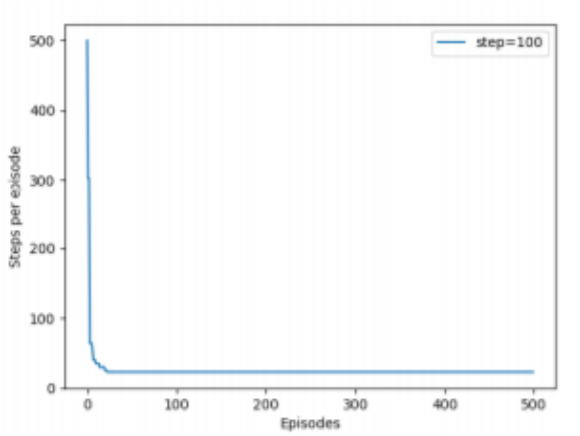

Figure 4.20: the number of executed actions per training episode of Dyna-Q agent in env (b) .

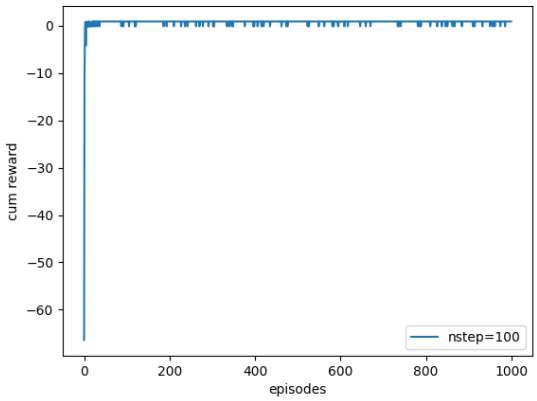

Figure 4.21: the maximum cumulative reward per training episode of Dyna-Q agent in env (b)

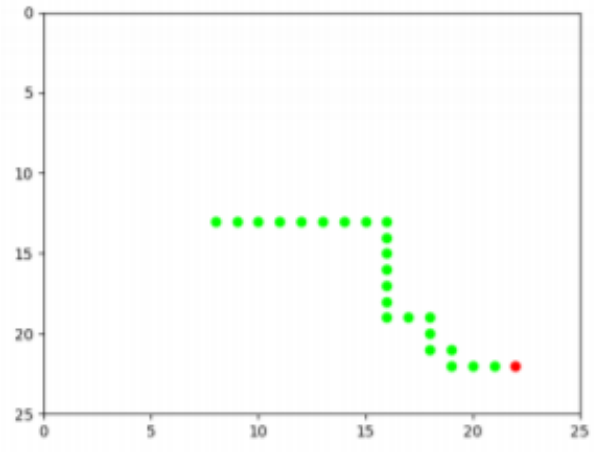

Figure 4.22: dyna agent path in env (b)

Measure	Value
Maximum cumulative reward	0.78
Convergence speed	23 episodes
Success Rate	91.3 %
Number of actions of the optimal policy	23 actions

Table 4.6: Dyna-Q training results in env (b).

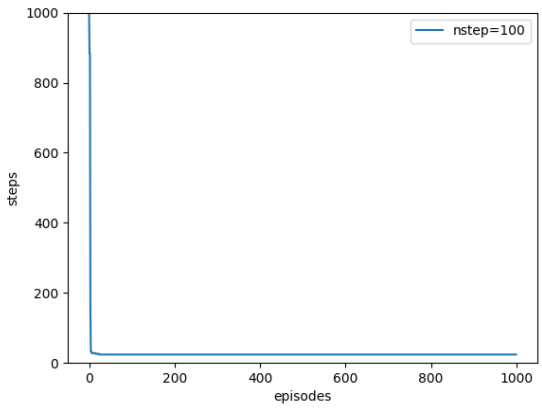

Figure 4.23: the number of executed actions per training episode of Dyna-Q agent in env (c).

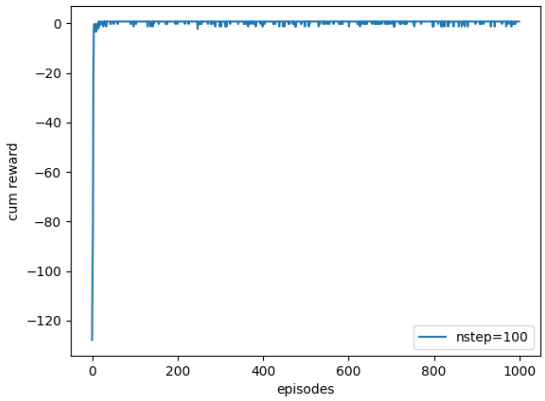

Figure 4.24: the maximum cumulative reward per training episode of Dyna-Q agent in env (c).

Measure	Value
Maximum cumulative reward	0.769
Convergence speed	24 episodes
Success Rate	83.1 %
Number of actions of the optimal policy	24 actions

Table 4.7: Dyna-Q training results in env (c).

In conclusion, all the results of the different training environments proved that the agent converged quickly to an optimal trajectory with a high success rate and large reward value in a small number of episodes. Finally, the 100-step dyna-Q agent is proven to be a map invariant algorithm.

4.2.3 Dyna-Q agent extended with a kinematic model

It was concluded from the previous results that Dyna-Q algorithm is the most successful and time-efficient algorithm in discrete environments. Consequently, we will extend our agent with a kinematic class of a four-wheeled-robot.

As shown in the figures below , the Dyna-Q algorithm is simulated for 2500 episodes in the main environment 3.1 by executing 100 hypothetical experiences after each real interaction experience. So, the Dyna-Q algorithm converges faster than the standard Q-learning algorithm in just 440 episodes. The minimum number of steps to reach the

goal is the same 16 steps and also the maximum reward for this optimal policy is 0.85. The success rate reaches 88 this value is higher than the standard Q-learning as the convergence is faster.

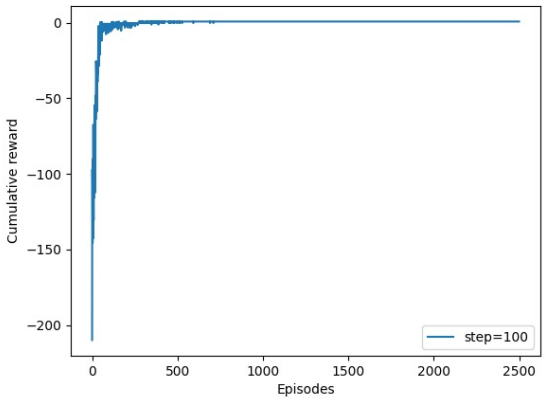

Figure 4.25: The maximum cumulative reward per training episode of the Dyna-Q robot

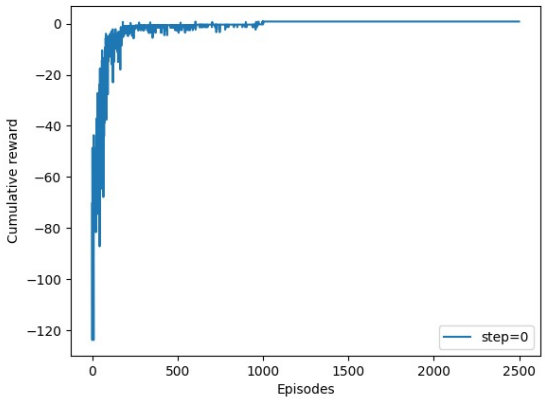

Figure 4.26: The maximum cumulative reward per training episode of the Q-learning robot

It can be seen from 4.25 and 4.31 that the Dyna-Q robot settles faster on the maximum reward policy in nearly half the number of the episodes of the Q-learning robot.

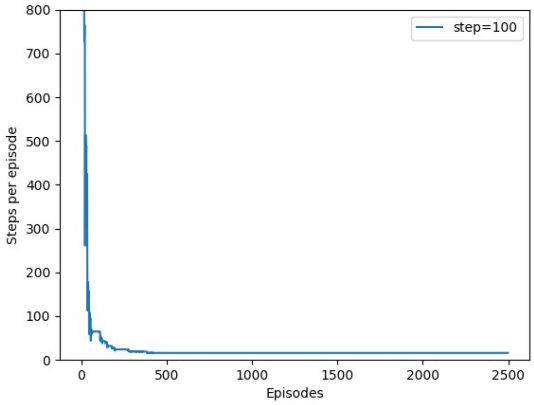

Figure 4.27: the number of executed actions per training episode of Dyna-Q robot.

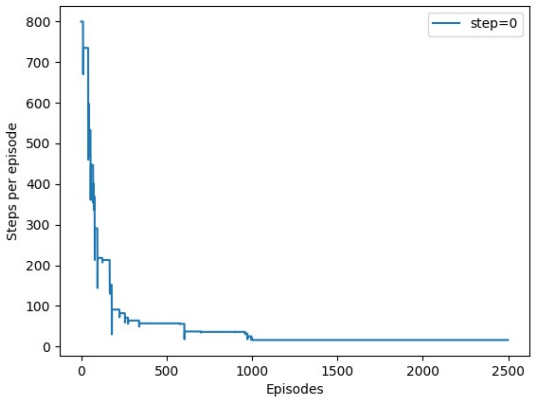

Figure 4.28: The number of executed actions per training episode of Q-learning robot.

The optimal path taken by the robot is shown in figure. The robot steps are actually taken in a continuous virtual environment of 4 by 4 size but then its location is mapped to the 25 by 25 grid world to be represented on the plot easily.

Measure	Value
Maximum cumulative reward	0.85
Convergence speed	440 episodes
Success Rate	88 %
Number of actions of the optimal policy	16 actions

Table 4.8: Dyna-Q robot training results

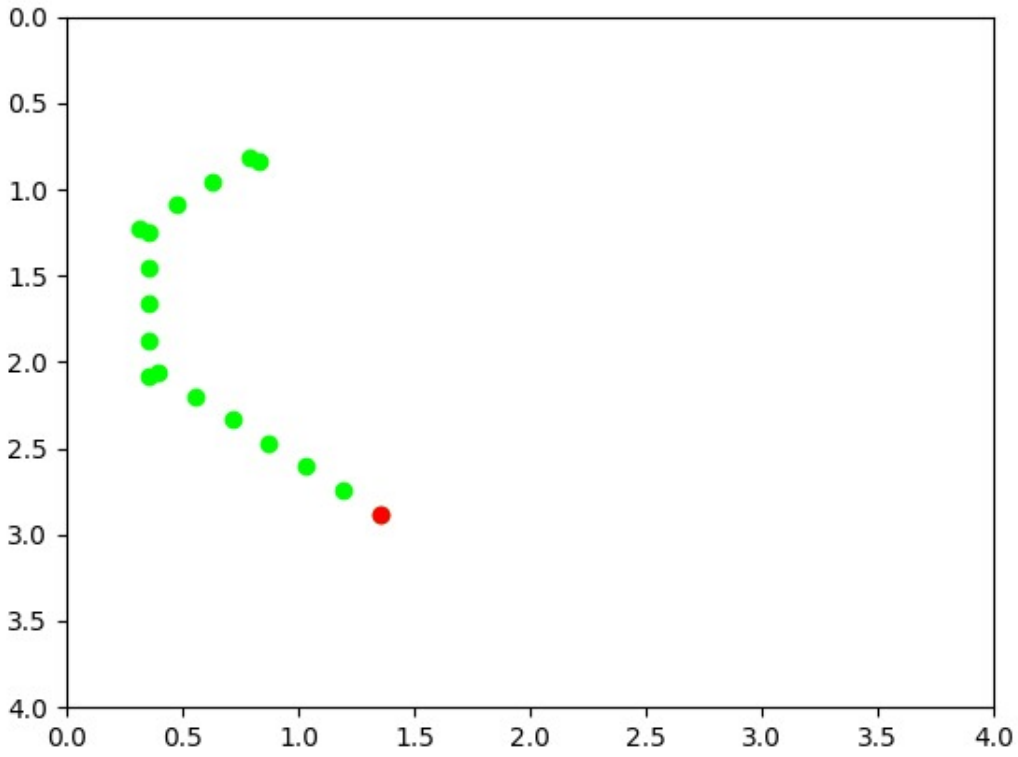

Figure 4.29: robot path

The Figure 4.29 represents the optimal path taken by the robot in environment 3.1.

Stochastisty Of The Starting Point

The robustness our algorithm can be tested by forcing it to start from a random position in each training episode, for this sake a method called Randomstart() is created to randomize

the staring point in the beginning of each training episode. A Dyna-Q robot with 100 planning steps (n=100) is simulated for 4000 episodes and allowed to have up to 800 trails per episode. The robot achieved an success of 80% and The results of the Dyna-Q algorithm with a random starting point show that the mobile robot can reach the target from any location in the environment in a very short time with high reward values.

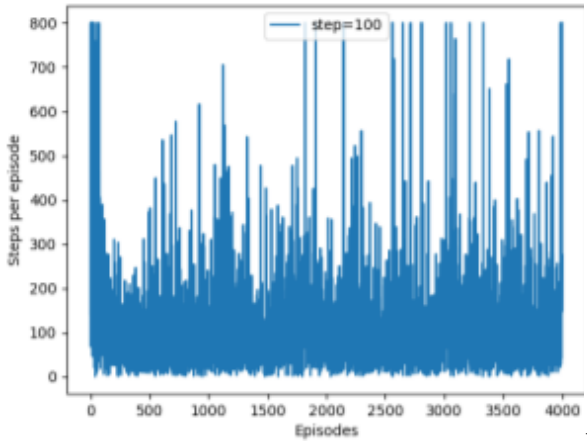

Figure 4.30: The number of executed actions for each training episode

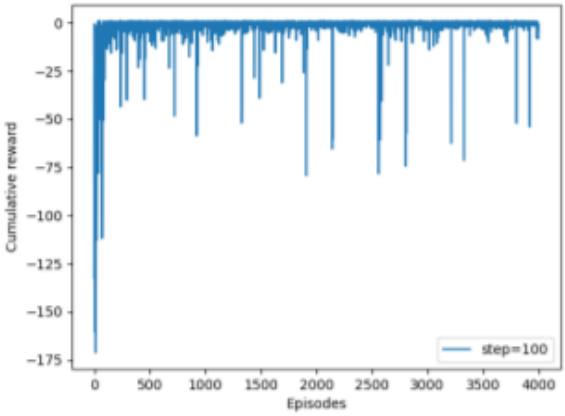

Figure 4.31: The maximum cumulative reward per training episode of the random robot

Invariance Of The Environment

Another way of testing the algorithm robustness is to train it in different environments to see if the algorithm is map-invariant or not. The robot will be trained in two randomly selected environments 4.32 and 4.33 .

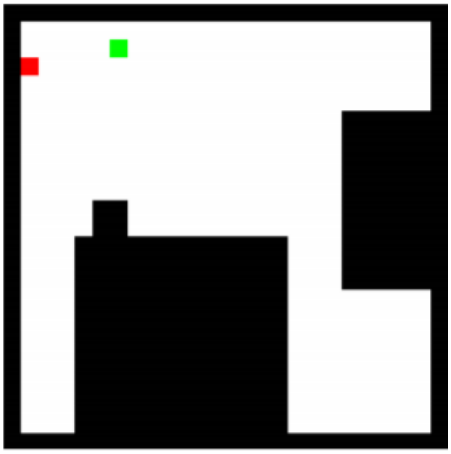

Figure 4.32: env (a)

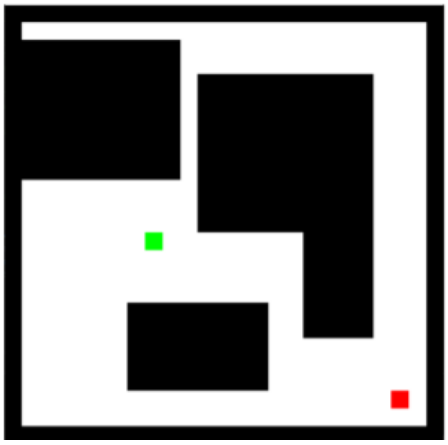

Figure 4.33: env (b)

The simulation results of the robot performance in both environments is shown below :

Measure	Value
Maximum cumulative reward	0.95
Convergence speed	251 episodes
Success Rate	93 %
Number of actions of the optimal policy	6 actions

Table 4.9: robot training results in environment (a)

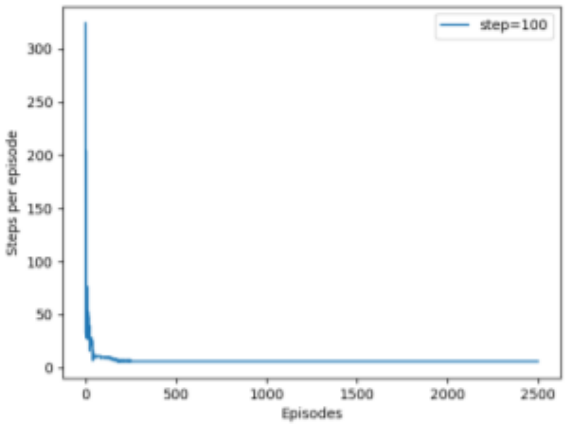

Figure 4.34: The number of executed actions per training episode of the robot in env(a).

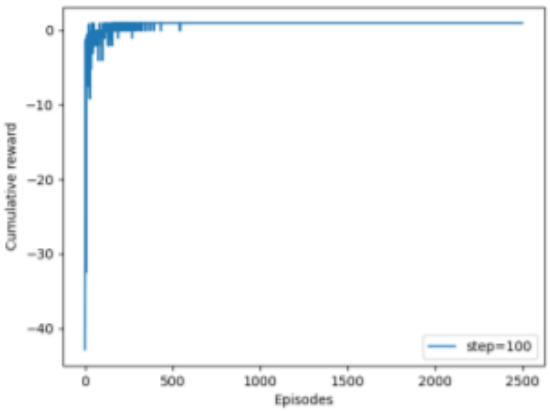

Figure 4.35: The maximum cumulative reward per training episode of the robot in env (a)

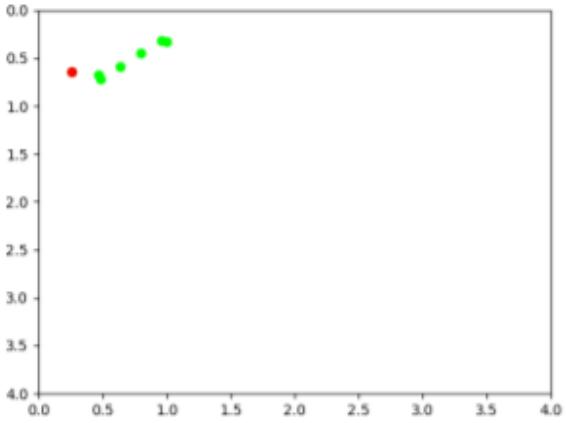

Figure 4.36: robot path in env (a)

The robot showed excellent performance in environment (a) as shown in the past figures and tables. The robot achieved a success rate of 93% and converged to an optimal policy of 6 actions in only 251 episodes. The results indicate the quick adaptation of the

robot in new environments and its ability to achieve convergence in relatively short time. The same procedure is done to environment (b) as shown below :

Measure	Value
Maximum cumulative reward	0.77
Convergence speed	553 episodes
Success Rate	81 %
Number of actions of the optimal policy	24 actions

Table 4.10: robot training results in environment (b)

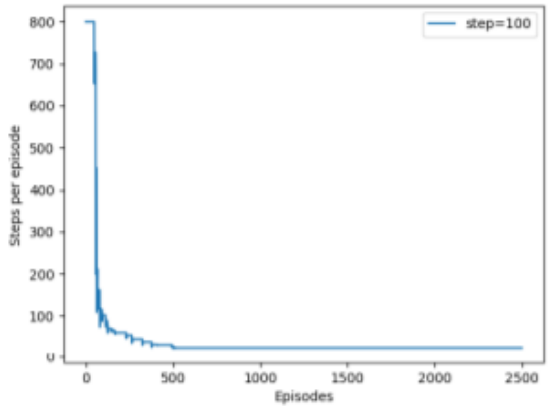

Figure 4.37: The number of executed actions per training episode of the robot in env(b).

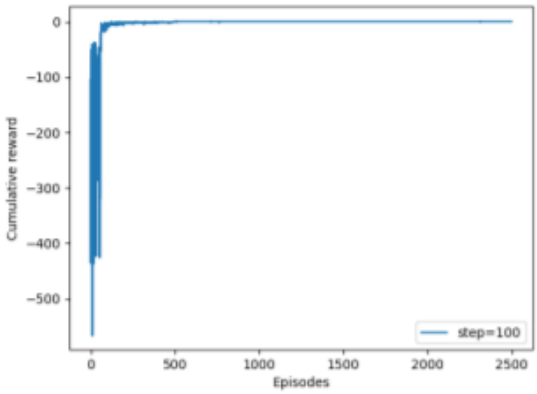

Figure 4.38: The maximum cumulative reward per training episode of the robot in env (b)

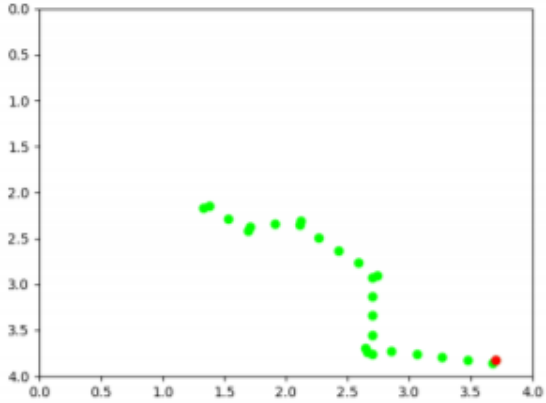

Figure 4.39: robot path in env (b)

In conclusion, all the results of the different training environments proved that the robot converged quickly to an optimal trajectory with a high success rate and large

reward value in a small number of episodes. Finally, the 100-step dyna-Q robot is proven to be map invariant algorithm that can successfully represent the robot kinematics and navigate through a set of unknown environment.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis aimed to find a solution to the autonomous navigation of mobile robots in using reinforcement learning. The solution was based on Dyna Q-learning and is realized by extending the standard model-free Q-learning with an inter-acting model. This model learns the state transition probabilities of the robot in a given environment with unknown static obstacles. Additional planning steps used simulated experience to rapid the convergence. To validate the algorithms used, simulations have been designed and implemented in Python-based environment. The designed algorithms were compared based on four optimality criteria: the convergence speed, the optimal trajectory length, the success rate and the cumulative reward. From the results in the previous section, the plots and numbers showed the supremacy of Dyna-Q learning over the Q-learning and the deep Q-learning algorithms regarding discrete environments. It has been experimentally proven that the model-based Dyna Q-learning are more efficient than the standard Q-learning and the deep Q-learning. Dyna-Q can converge 6 or more times faster than standard Q-learning and about 25 times faster than deep Q-learning. The success rate of the Dyna-Q is higher than the standard Q-learning and deep Q-learning. There is no significant difference between the lengths of the optimal trajectories and also the cumulative reward for all the designed algorithms. Moreover, we have successfully implemented a kinematic model of a simple four-wheeled robot that navigates through an unknown discrete environment using Dyna-Q algorithm as a path planner. In a nutshell, the Dyna-Q algorithm was proved to be a robust map-variant path-planning technique for a mobile robot when deployed in a discretized state-space.

5.2 Future Work

We are looking forward for a complete hardware implementation of the robot and its environment . The localization of the robot can be driven from a camera vision system

while the reinforcement learning agent can be implemented on a micro-processor on the robot. The communication between the camera system and the robot will guarantee an accurate localization while the path planning is done simultaneously using the real-time algorithm implemented in the robot brain (processor).

Appendix

Appendix A

Python Simulation code

A.1 Dyna-Q Agent Class

```
import gym
import gym_pathfinding
import numpy as np
import random
import matplotlib.pyplot as plt
from time import sleep

class DynaAgent:
    def __init__(self, exp_rate=0.7, lr=0.8,gamma=0.9 , n_steps=5, episodes=1):
        self.maze = gym.make('pathfinding-obstacle-25x25-v0')
        self.state = self.maze.game.player
        self.actions = [0,1,2,3]
        self.state_actions = []  # state & action track
        self.exp_rate = exp_rate
        self.lr = lr
        self.gamma=gamma
        self.steps = n_steps
        self.episodes = episodes  # number of episodes going to play
        self.steps_per_episode = []
        self.epsilon_discount= 0.9
        self.Q_values = {}
        self.cumreward_per_episode = []
        # model function
        self.model = {}
        for row in range(25):
```

```
            for col in range(25):
                self.Q_values[(row, col)] = {}
                for a in self.actions:
                    self.Q_values[(row, col)][a] = 0


    def chooseAction(self):
        # epsilon-greedy
        mx_nxt_reward = -999
        action = ""
        if np.random.uniform(0, 1) <= self.exp_rate:
            action = np.random.choice(self.actions)
        else:
            # greedy action
            current_position = self.state
            # if all actions have same value, then select randomly
            if len(set(self.Q_values[current_position].values())) == 1:
                action = np.random.choice(self.actions)
            else:
                for a in self.actions:
                    nxt_reward = self.Q_values[current_position][a]
                    if nxt_reward >= mx_nxt_reward:
                        action = a
                        mx_nxt_reward = nxt_reward
        return action

    def randomstart(self):
        while True :
            x = np.random.randint(0,self.maze.game.lines)
            print(x)

            y = np.random.randint(0,self.maze.game.columns)
            print(y)
            if( self.maze.game.grid[x,y]==0) :
                return (x,y)

    def reset(self):
        self.maze.reset()
        self.maze.seed(1)
        self.maze.game.player=self.randomstart()
        self.state=self.maze.game.player
```

```
67          self.state_actions = []
68
69      def play(self):
70          self.steps_per_episode = []
71          Successful_eps=0
72          self.reset()
73          self.reset()
74          self.reset()
75          optimal_policy_steps=1000
76          for ep in range(self.episodes):
77              cumreward = 0
78              stepscount = 0
79              # 500 in random mode but optimal policy in normal
80              while not self.maze.game.terminal and stepscount
    < 500   :
81                  #print(self.state)
82                  self.maze.render()
83                  sleep(0.005)
84
85                  #sleep(0.005)
86
87                  action = self.chooseAction()
88                  self.state_actions.append((self.state, action
    ))
89
90                  nxtState2D, reward, self.maze.game.terminal,
    _ = self.maze.step(action)
91                  stepscount+=1
92                 # print(stepscount)
93                  #print(self.maze.step(action))
94                  nxtState = self.maze.game.player
95                  cumreward+=reward
96                  #print(reward)
97                  #print(cumreward)
98                  # print(nxtState)
99                  # print(self.Q_values[nxtState])
100                 # print(list(self.Q_values[nxtState].values()
    ))
101                 # print(reward)
102                 # print(self.maze.game.terminal)
103
104                 # update Q-value
105                 self.Q_values[self.state][action] += self.lr
    *(reward + self.gamma*np.max(list(self.Q_values[nxtState].
```

```
values())) - self.Q_values[self.state][action])
106
107                 # update model
108                 if self.state not in self.model.keys():
109                     self.model[self.state] = {}
110                 self.model[self.state][action] = (reward,
nxtState)
111                 self.state = nxtState
112
113                 # loop n times to randomly update Q-value
114                 for _ in range(self.steps):
115                     # randomly choose an state
116                     rand_idx = np.random.choice(range(len(
self.model.keys())))
117                     #print(self.model.keys())
118                    # print( rand_idx)
119                     _state = list(self.model)[rand_idx]
120                     # randomly choose an action
121                     rand_idx = np.random.choice(range(len(
self.model[_state].keys())))
122                     _action = list(self.model[_state])[
rand_idx]
123
124                     _reward, _nxtState = self.model[_state][
_action]
125
126                     self.Q_values[_state][_action] += self.lr
*(_reward + self.gamma*np.max(list(self.Q_values[_nxtState
].values())) - self.Q_values[_state][_action])
127
128             if self.exp_rate>0.01:
129                  self.exp_rate=self.epsilon_discount*self.
exp_rate
130             # end of game
131
132             if ep % 10 == 0:
133                 print("episode", ep)
134             self.steps_per_episode.append(len(self.
state_actions))
135             self.cumreward_per_episode.append(cumreward)
136             max_cum_reward=0
137
138             if(len(self.state_actions)<optimal_policy_steps)
:
```

```
                convergence_episode = ep
            optimal_policy_steps = min(self.steps_per_episode
    )
            if(self.maze.game.terminal==True and stepscount<=
     optimal_policy_steps ):
                Successful_eps+=1

            if(ep == 999):
                max_cum_reward = max(self.
    cumreward_per_episode)
                optimal_policy_steps = min(self.
    steps_per_episode)
                print(max_cum_reward)
                print(optimal_policy_steps)
                success_rate = Successful_eps/1000 *100
                print(success_rate)
                print(convergence_episode)
            self.reset()

if __name__ == "__main__":
    N_EPISODES = 1000
  #agent1
    agent = DynaAgent(n_steps=100, episodes=N_EPISODES)
    agent.play()
    # agent.reset()
    # agent.reset()
    # agent.reset()
    # agent.maze.render()
   # sleep(10)
    cumulative_r_episode = agent.cumreward_per_episode
    steps_episode_50 = agent.steps_per_episode
    plt.figure(1)
    plt.ylim(0, 1000)
    plt.plot(range(N_EPISODES), steps_episode_50, label="
   nstep=100")

    plt.xlabel("episodes")
    plt.ylabel("steps")
    plt.legend()

    plt.figure(2)
    plt.plot(range(N_EPISODES),cumulative_r_episode, label="
   nstep=100")
    plt.legend()
```

```
177     plt.xlabel("episodes")
178     plt.ylabel("cum reward")
179     #agent 2
180 #     N_EPISODES = 1000
181 
182 #     agent = DynaAgent(n_steps=0, episodes=N_EPISODES)
183 #     agent.play()
184 #     cumulative_r_episode = agent.cumreward_per_episode
185 #     steps_episode_50 = agent.steps_per_episode
186 #     plt.figure(1)
187 #     plt.ylim(0, 1000)
188 #     plt.plot(range(N_EPISODES), steps_episode_50, label="
    nstep=0")
189 
190 #     plt.xlabel("episodes")
191 #     plt.ylabel("steps")
192 #     plt.legend()
193 #     plt.figure(2)
194 #     plt.plot(range(N_EPISODES),cumulative_r_episode, label
    ="nstep=0")
195 #     plt.legend()
196 #     plt.xlabel("episodes")
197 #     plt.ylabel("cum reward")
198 
199 #    # agent 3
200 #     agent = DynaAgent(n_steps=50, episodes=N_EPISODES)
201 #     agent.play()
202 #     cumulative_r_episode = agent.cumreward_per_episode
203 #     steps_episode_50 = agent.steps_per_episode
204 #     plt.figure(1)
205 #     plt.ylim(0, 1000)
206 #     plt.plot(range(N_EPISODES), steps_episode_50, label="
    nstep=50")
207 
208 #     plt.xlabel("episodes")
209 #     plt.ylabel("steps")
210 #     plt.legend()
211 #     plt.figure(2)
212 #     plt.plot(range(N_EPISODES),cumulative_r_episode, label
    ="nstep=50")
213 #     plt.legend()
214 #     plt.xlabel("episodes")
215 #     plt.ylabel("cum reward")
216     plt.show()
```

A.2 DQN Agent Class

```
# -*- coding: utf-8 -*-
import random
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense, TimeDistributed
from keras.layers import SimpleRNN
from keras.optimizers import Adam
from MCs.SSMR_class import SSMRobot
import gym
import gym_pathfinding
import matplotlib.pyplot as plt
from time import sleep
# from environment.environment import Environment
# from environment.environment_node_data import Mode
import action_mapper

EPISODES = 1500
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = list()
        self.max_mem = 2000
        self.gamma = 0.98   # discount rate
        self.epsilon = 1.0  # exploration rate
        # self.epsilon_min = 0.01
        # self.epsilon_decay = 0.995
        # self.learning_rate = 0.001
        self.epsilon_min = 0.001
        self.epsilon_decay = 0.9
        self.learning_rate = 0.001
        self.replay_cnt = 0
        self.actions = [0,1,2,3]
        self.target_model = self._build_model()
        self.model = self._build_model()
        self.update_model = 10
    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = Sequential()
        model.add(Dense(625, input_dim=625, activation='tanh'
  ))
```

```
42          model.add(Dense(32, activation='tanh'))
43          model.add(Dense(self.action_size, activation='linear'
    ))
44          model.compile(loss='mse',
45                        optimizer=Adam(lr=self.learning_rate))
46          return model
47      def remember(self, state, action, reward, next_state,
    done):
48          self.memory.append((state, action, reward, next_state
    , done))
49          if(len(self.memory)>self.max_mem) :
50              self.memory.pop(0)
51      def act(self, state):
52          if np.random.uniform(0,1) <= self.epsilon:
53              return random.choice(range(self.action_size) )
54          act_values = self.model.predict(state)[0]
55          #print(self.model.predict(state))
56          return np.argmax(act_values)  # returns action
57      def replay(self, batch_size):
58          self.replay_cnt+=1
59          minibatch = random.sample(self.memory, batch_size)
60          for state, action, reward, next_state, done in
    minibatch:
61              target = reward # q value of the state  if it is
    the final
62              if not done:
63                  target = (reward + self.gamma *
64                            np.max(self.target_model.predict(
    next_state)[0])) # q value of state is computed by bellman
65              target_f = self.target_model.predict(state) #
    value of the current state q_value
66              target_f[0][action] = target #
67              self.model.fit(state, target_f, verbose=0)
68          if self.epsilon > self.epsilon_min:
69              self.epsilon *= self.epsilon_decay
70          if (self.replay_cnt % agent.update_model ==0)  :
71                  agent.target_train()
72      def target_train(self):
73          weights = self.model.get_weights()
74          target_weights = self.target_model.get_weights()
75          for i in range(len(target_weights)):
76              target_weights[i] = weights[i]
77          self.target_model.set_weights(target_weights)
78  #     def train_model(self, episode, done):
```

```
79 #         '''
80 #         Trains all the models that are required for this Q-
   Learning implementation.
81 #         :param episode: Takes in the current episode of
   training.
82 #         '''
83 #         if len(self.memory) >= self.batch_size:
84 #             minibatch = random.sample(self.memory, self.
   batch_size)
85 #             minibatch = np.array(minibatch)
86 #             observations = np.vstack(minibatch[:, 0])
87 #             next_observations = np.vstack(minibatch[:,3])
88 #             target = np.copy(minibatch[:,2])
89 #             done_states = np.where(minibatch[:, 4] == False
   )
90 #             if len(done_states[0]) > 0:
91 #                 predictions = self.target_model.predict(
   next_observations)
92 #                 predictions = np.amax(predictions, axis=1)
93 #                 target[done_states] += np.multiply(self.
   gamma, predictions[done_states])
94 #             actions = np.array(minibatch[:, 1], dtype=int)
95 #             target_f = self.target_model.predict(
   observations)
96 #             target_f[range(self.batch_size), actions] =
   target
97 #             self.target_model.fit(observations, target_f,
   epochs=1, verbose=0)
98 # double deep q learning
99 #   if len(self.memory) >= self.batch_size:
100 #             minibatch = random.sample(self.memory, self.
   batch_size)
101 #             minibatch = np.array(minibatch)
102 #             observations = np.vstack(minibatch[:, 0])
103 #             next_observations = np.vstack(minibatch[:,3])
104 #             target = np.copy(minibatch[:,2])
105 #             done_states = np.where(minibatch[:, 4] == False
   )
106 #             if len(done_states[0]) > 0:
107 #                 q_values = self.q_model.predict(
   next_observations)
108 #                 best_actions = np.argmax(q_values, axis=1)
109 #                 q_targets = self.target_model.predict(
   next_observations)
```

```
110 #                 target[done_states] += np.multiply(self.
    gamma, q_targets[done_states, best_actions[done_states
    ]][0])
111 #             actions = np.array(minibatch[:, 1], dtype=int)
112 #             target_f = self.target_model.predict(
    observations)
113 #             target_f[range(self.batch_size), actions] =
    target
114 #             self.q_model.fit(observations, target_f, epochs
    =1, verbose=0)
115     def load(self, name):
116         self.model.load_weights(name)
117     def save(self, name):
118         self.model.save_weights(name)
119     def reset(self):
120         self.env.reset()
121         self.env.seed(2)
122         # self.env.game.player = self.randomStart()
123         self.state = self.env.getPlayer()
124         self.state_actions = []
125 if __name__ == "__main__":
126     env = gym.make('pathfinding-obstacle-25x25-v0')
127     steps_per_episode = []
128     cumulative_reward_per_episode = []
129     env.reset()
130     env.reset()
131     #env.set_mode(Mode.PAIR_ALL, terminate_at_end=True)
132     #env.set_mode(Mode.ALL_RANDOM, terminate_at_end=False)
133     # env.use_observation_rotation_size(True)
134     #env.set_cluster_size(10)
135     # env.set_observation_rotation_size(128)
136     state_size= 625
137     #print("state space" , state_size)
138     # action_size = action_mapper.ACTION_SIZE
139     action_size = env.action_space.n
140    # print("action space" , action_size)
141     agent = DQNAgent(state_size, action_size)
142     # agent.load("./save/cartpole-dqn.h5")
143     done = False
144     batch_size = 128
145     print("START DQN")
146     donecount=0
147     for e in range(EPISODES):
148         # visualize = (e % 5 == 0)
```

```
149         # print("state space" , state_size)
150          #print("action space" , action_size)
151          reward_sum = 0
152          state = env.reset()
153          env.seed(2)
154          state = [[0]*state_size]*1
155          state[0][env.getPlayer()[0]*25 + env.getPlayer()[1]]
    = 1
156          state = np.reshape(state, [1, state_size])
157          # print("state", state)
158          steps = 0
159          while steps < 2000:
160          #     env.render()
161              #sleep(0.005)
162              action = agent.act(state)
163              # linear, angular = action_mapper.map_action(
    action)
164              next_state, reward, done, _ = env.step(action)
165              #sprint(reward)
166              next_state = [[0]*state_size]*1
167              next_state[0][env.getPlayer()[0]*25 + env.
    getPlayer()[1]] = 1
168              #print(reward)
169              next_state = np.reshape(next_state, [1,
    state_size])
170              if(done == True):
171                  donecount+=1
172              reward_sum += reward
173                 # print("memory length", len(agent.memory))
174              agent.remember(state, action, reward, next_state,
     done)
175              state = next_state
176              # if visualize:
177              #     env.visualize()
178                  #time.sleep(1.0)
179              if done:
180                  break
181              steps += 1
182          print("episode: {}/{}, score: {}, e: {:.2} iteration
    :{}"
183                    .format(e, EPISODES, reward_sum, agent.
    epsilon, steps))
184          if len(agent.memory) > batch_size & donecount>0:
185                  agent.replay(batch_size)
```

```
186             #if e % 1000 == 0:
187             #     agent.save("./save/dqn" + str(e) + ".h5")
188             steps_per_episode.append(steps)
189             cumulative_reward_per_episode.append(reward_sum)
190     plt.figure(1)
191     plt.plot(range(EPISODES), steps_per_episode)
192     plt.xlabel("Episodes")
193     plt.ylabel("Steps per episode")
194     plt.figure(2)
195     plt.plot(range(EPISODES), cumulative_reward_per_episode)
196     plt.xlabel("Episodes")
197     plt.ylabel("Cumulative reward")
198     plt.show()
```

A.2.1 Kinematic class of the robot

```
class Robot(object) :
    """ Defines basic mobile robot properties """
    def __init__(self) :
        self.pos_x = 0.0
        self.pos_y = 0.0
        self.theta = 0.0
        self.plot = False
        self._delta = 0.1 #sample time
        self.env_width = 4
        self.env_length = 4

    # Movement
    def step (self):
        """ updates the x , y and angle """
        self.deltax()
        self.deltay()
        self.deltaTheta()

    def move(self , seconds):
        """ Moves the robot for an ' s ' amount of seconds
    """
        for i in range(int(seconds/self._delta)):
            self.step()
            if i%3 == 0 and self.plot: # plot path every 3
    steps
                self.plot_xya ( )

    #  P r i n t i n g andplotting  :
    def print_xya (self):
        """ prints the x , y position and angle """
        print("x = " + str(self.pos_x)+" "+"y = "+ str(self.
    pos_y))
        print("a = " + str(self.theta))

    def plot_robot (self):
        """ plots a representation of the robot """
        plt.arrow(self.pos_y ,self.pos_x , 0.001 * math.sin (
    self.theta ),0.001 * math.cos (self.theta ) ,
        head_width=self.length , head_length=self.length ,
        fc= 'k' , ec= 'k' )

    def plot_xya (self):
```

```
39          """ plots a dot in the position of the robot """
40          plt.scatter(self.pos_y ,self.pos_x,c= 'r',edgecolors=
    'r' )
41
42  class SSMRobot (Robot ) :
43      """ Defines a SSMR drive robot """
44      def __init__(self):
45          Robot.__init__(self)
46          # geometric parameters
47          self.r = 0.03
48          self.c = 0.065
49          self.a = 0.055
50          self.b = 0.055
51          self.Xicr = 0.055
52          self.length = 0.11
53
54          # states
55          self.omega = 0
56          self.vx = 0
57          self.vy = 0
58
59          self.omega_r = 0.0
60          self.omega_l = 0.0
61
62          self.orientation = 0
63
64      def deltax(self) :
65          # calculate vx and vy
66          self.omega = self.r * (self.omega_r - self.omega_l)
    /(2* self. c )
67          self.vx = self.r * (self.omega_r + self.omega_l)/2
68          self.vy = self.Xicr * self.omega
69          # calculate X_dot
70          X_dot = math.cos(self.theta)*self.vx - math.sin(self.
    theta)*self.vy
71          self. pos_x += self._delta * X_dot
72
73          if self.pos_x > self.env_width:
74              self.pos_x = self.env_width
75          elif self.pos_x < 0:
76              self.pos_x = 0
77
78      def deltay ( self) :
79          # calculate vx and vy
```

```
80          self.omega = self.r * (self.omega_r - self.omega_l)
    /(2 * self.c)
81          self.vx = self.r * (self.omega_r + self.omega_l)/2
82          self.vy = self.Xicr * self.omega
83          # calculate Y_dot
84          Y_dot = math.sin ( self. theta ) * self.vx + math.cos
     ( self. theta ) * self. vy
85          self. pos_y += self. _delta * Y_dot
86
87          if self. pos_y > self.env_length:
88              self. pos_y = self.env_length
89          elif self. pos_y < 0:
90              self. pos_y = 0
91
92      def deltaTheta ( self) :
93          # calculate omega
94          self.omega = self. r * ( self. omega_r-self. omega_l
    ) /(2* self. c )
95          self. theta += self. _delta * self.omega
96
97      def reset (self,player) :
98          # given 4mx4m arena discretized to 25x25
99          self.pos_x = player[0] *self.env_width/24 #discrete
    levels
100         self.pos_y = player[1] *self.env_length/24 #discrete
    levels
101         self.theta = 0 # np.random.uniform(-np.pi,np.pi)
102         # print(self.theta)
103
104     def optimal_action ( self , action ) :
105         self.action = action
106
107     def take_step (self) :
108         if self.action == 0 : # Forward
109             self.omega_l = 1.7
110             self.omega_r = 1.7
111         elif self.action == 1 : # Backward
112             self.omega_l = -1.7
113             self.omega_r = -1.7
114         elif self.action == 2 : # left
115             self.omega_l = -1.7
116             self.omega_r = 1.7
117         else : # right
118             self.omega_l = 1.7
```

```
            self.omega_r = -1.7
        self.move(self._delta)

    def get_discretized_state ( self) :
        x_discrete = math.floor ( ( ( self. pos_x ) /self.
   env_width) *24)
        y_discrete = math.floor ( ( ( self. pos_y ) /self.
   env_length) *24)
        theta_discrete = np.arctan2 (math. sin ( self. theta
   ) , math. cos ( self. theta ) )
        # print(theta_discrete *180/np.pi)
        theta_discrete = math.floor(theta_discrete/(2 * np.pi
   ) * 20)
        # print(theta_discrete)

        return ( x_discrete , y_discrete , theta_discrete )

    def assign_discretized_state ( self , x , y ) :
        self.pos_x= ( ( x ) /24) *self.env_width
        self.pos_y= ( ( y ) /24) *self.env_length
        env = gym.make('pathfinding-obstacle-25x25-v0')
env.seed(1)

# env.seed(2) # for full-deterministic environment
env.reset()

done = False
Q_values = ''
with open(r'training.txt','r') as f:
    for i in f.readlines():
        Q_values = i
Q_values = eval(Q_values)
s = env.getPlayer()
actions = [0,1,2,3]
points = []
points.append(s)
plt.figure(1)
plt.xlim(0,25)
plt.ylim(25,0)
# ax = plt.subplots()
# ax.add_patch(Rectangle((2, 2), 1, 3,color="black"))

while not done :
    plt.scatter(s[1] ,s[0],c= 'lime',edgecolors= 'lime' )
```

```
159     env.render()
160     sleep(0.005)
161 
162     # if all actions have same value, then select randomly
163     mx_nxt_reward = -999
164     action = ""
165     print(s)
166     if len(set(Q_values[s].values())) == 1:
167         action = np.random.choice(actions)
168     else:
169         for a in actions:
170             nxt_reward = Q_values[s][a]
171             if nxt_reward >= mx_nxt_reward:
172                 action = a
173                 mx_nxt_reward = nxt_reward
174 
175     _ , r, done, _ = env.step(action)
176     s = env.getPlayer()
177     points.append(s)
178 
179 plt.scatter(s[1] ,s[0],c= 'r',edgecolors= 'r' )
180 print(points)
181 plt.show()
```

Bibliography

[1] Mohammed Alhawary. Reinforcement-learning-based navigation for autonomous mobile robots in unknown environments. Master's thesis, University of Twente, 2018.

[2] Hee Rak Beom and Hyung Suck Cho. A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning. *IEEE transactions on Systems, Man, and Cybernetics*, 25(3):464–477, 1995.

[3] XIA Chen and Abdelkader El Kamel. A reinforcement learning method of obstacle avoidance for industrial mobile vehicles in unknown environments using neural network. In *Proceedings of the 21st International Conference on Industrial Engineering and Engineering Management 2014*, pages 671–675. Springer, 2015.

[4] Mohammad Abdel Kareem Jaradat, Mohammad Al-Rousan, and Lara Quadan. Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, 27(1):135–149, 2011.

[5] Lazhar Khriji, Farid Touati, Kamel Benhmed, and Amur Al-Yahmedi. Mobile robot navigation based on q-learning technique. *International Journal of Advanced Robotic Systems*, 8(1):4, 2011.

[6] Krzysztof Kozłowski and Dariusz Pazderski. Modeling and control of a 4-wheel skid-steering mobile robot. *International journal of applied mathematics and computer science*, 14:477–496, 2004.

[7] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. Packt Publishing, Birmingham, UK, 2018.

[8] Madson Rodrigues Lemos, Anne Vitoria Rodrigues de Souza, Renato Souza de Lira, Carlos Alberto Oliveira de Freitas, Vandermi João da Silva, and Vicente Ferreira de Lucena. Robot training and navigation through the deep q-learning algorithm. In *2021 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6, 2021.

[9] Kristijan Macek, Ivan PetroviC, and N Peric. A reinforcement learning approach to obstacle avoidance of mobile robots. In *7th International Workshop on Advanced Motion Control. Proceedings (Cat. No. 02TH8623)*, pages 462–466. IEEE, 2002.

[10] Enrico Marchesini and Alessandro Farinelli. Discrete deep reinforcement learning for mapless navigation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10688–10694, 2020.

[11] Jawad Muhammad and Ihsan Ömür Bucak. An improved q-learning algorithm for an autonomous mobile robot navigation problem. In *2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE)*, pages 239–243, 2013.

[12] Sudharsan Ravichandiran. *Hands-on Reinforcement Learning with Python: Master Reinforcement and Deep Reinforcement Learning Using OpenAI Gym and TensorFlow*. Packt Publishing Ltd, 2018.

[13] Xiaogang Ruan, Dingqi Ren, Xiaoqing Zhu, and Jing Huang. Mobile robot navigation based on deep reinforcement learning. In *2019 Chinese control and decision conference (CCDC)*, pages 6174–6178. IEEE, 2019.

[14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[15] A. Turiot. *An openai gymimplementation for the pathfinding problem*. 2019.

[16] Guo-Sheng Yang, Er-Kui Chen, and Cheng-Wan An. Mobile robot navigation using neural q-learning. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.04EX826)*, volume 1, pages 48–52 vol.1, 2004.

[17] Jeremy Zhang. Reinforcement learning examples implementation and explanation. https://github.com/MJeremy2017/reinforcement-learning-implementation, 2019.