Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 53-2 (2020) 15602–15607

Real Time Path Planning of Robot using Deep Reinforcement Learning

Jeevan Raajan [1], **Srihari P V** [1], **Satya Jayadev P, B Bhikkaji, Ramkrishna Pasumarthy**

Department of Electrical Engineering, Indian Institute of Technology Madras, Chennai, 600036, India (e-mail: {raajan.jeevan,sriharivenkat95}@gmail.com)

Abstract: This paper considers finding a path in real time for a robot from the given initial position to the goal position. The environment is assumed to be mapped (known completely) and the resulting path should avoid all the obstacles, both static and dynamic in the mapped environment. The robot's (agent) dynamics is assumed to be discrete LTI with process noise and is controlled with a finite set of inputs. An MDP formulation and a solution based on Deep Reinforcement Learning framework are presented for the problem. Numerical experiments are performed for the proposed method using Deep Q-Network algorithm and the results are compared with the state of the art sampling based path planning algorithms for both static and dynamic environments. It is shown that even though the proposed algorithm provides a sub-optimal path, the computational time is shown to be significantly faster compared to the traditional methods of path planning.

Keywords: Trajectory and Path Planning, Reinforcement Learning, Q-Learning, Autonomous Vehicles, Learning and Adaptation in Autonomous Vehicles

1. INTRODUCTION

In the field of robotics, path planning is one of the most basic and crucial component wherein the objective is to determine a collision free path for the robot to the goal in an obstacle rich environment. In early years, geometric algorithms like visibility graph, cell decomposition and grid based algorithms enjoyed great success in low dimensional path planning problems. However, these algorithm tend to be computationally intractable in higher dimensions involving complex constraints. In the last two decades, sampling based methods have been widely used for solving path planning problems (LaValle (2006)) in higher dimensions. Algorithms like Rapidly Exploring Random Tree* (RRT*) (Karaman and Frazzoli (2011)) and Fast Marching Trees* (FMT*) (Janson et al. (2015)) deals with finding a feasible path in a fixed goal environment whereas a variant of RRT* called the Real Time RRT* (RT-RRT*) (Naderi et al. (2015)) deals with finding a path in dynamic environments involving changing obstacle and goal positions. Furthermore, in order to account for the uncertainties involved in the system and environment, Luders (2014) provides an improved version of RRT* called the Chance-constrained RRT* which allows for probabilistic constraints to be placed on the path and proposes a similar improvement to the FMT* named the Robust-FMT*. However the real-time versions of all the above mentioned algorithms involve significant computational overhead for finding the path while performing collision checks, especially in dynamic environments.

In recent years, the field of Reinforcement Learning (RL) combined with deep neural networks has seen phenomenal progress as it has enabled tackling decision making tasks involving complex environments. (Mnih et al. (2015), Van Hasselt et al. (2016)). Recently, RL approaches have been used to solve Real time path planning problems Lei et al. (2018). However, most of the existing deep-RL algorithms focus on collision avoidance rather than planning a path for a dynamic environment Kahn et al. (2017). In this paper, a Deep-RL based methodology using Deep Q-Network is proposed to solve the path planning problem described in Section 2.

2. PROBLEM STATEMENT

Consider a mobile robot in a 2-D environment of $l \times l$ m^2 as shown in Figure 1 with the top left corner being the origin. The robot is assumed to have the dynamics

$$q_{t+1} = Aq_t + Ba_t + w_t,\ t = 1, 2, \ldots \tag{1}$$

where $q_t = (x_t, y_t)^T \in \mathbb{R}^2$ denotes the 2-D position of the robot at time t, $a_t \in \mathbb{A}$ is the input vector, with $\mathbb{A}$ being a finite subset of $\mathbb{R}^2$. The matrices $A, B \in \mathbb{R}^{2\times 2}$ with A being Hurwitz. The process noise w_t is assumed to be a i.i.d Gaussian process with mean zero and having positive definite covariance P_w.

The robot consists of m distance sensors placed symmetrically around it and their corresponding readings at time t are denoted by $d_t^{(i)}, i = 1, 2, \ldots m$. The maximum range of the distance sensor is d_{max}. A sensor reading $d_t^{(i)} < d_{max}$ indicates the presence of an obstacle in that direction.

[1] Authors contributed equally in this paper.

10.1016/j.ifacol.2020.12.2494

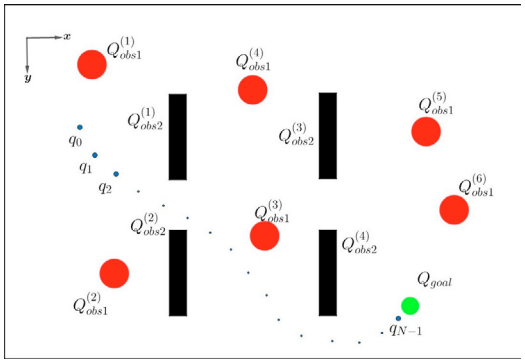


Fig. 1: Environment Schematic

Let Q denote the total configuration space of the robot and q_0 be the initial position of the robot (refer Figure 1). The obstacles in the environment are categorized under two sets

$$Q_{obs1,t} = \bigcup_{i=1}^{k_1} Q^{(i)}_{obs1,t}, \tag{2}$$

$$Q_{obs2} = \bigcup_{j=1}^{k_2} Q^{(j)}_{obs2}, \tag{3}$$

where $Q_{obs1,t}$ denotes the set of obstacles whose positions can vary with time and Q_{obs2} denotes the set of obstacles which are fixed in the environment. In (2) $Q^{(i)}_{obs1,t}$ is taken to be a ball of fixed radius $r^{(i)}_{obs1}$ with centre $q^{(i)}_{obs1,t}$ at time t and in (3) $Q^{(j)}_{obs2}$ is assumed to be a bounded convex polygon. Refer to Figure 1 for an illustration with $k_1 = 6$ and $k_2 = 4$.

The dynamics of $Q^{(i)}_{obs1}$ obstacle in terms of its center is given by:

$$q^{(i)}_{obs1,t+1} = q^{(i)}_{obs1,t} + c_{i,t} \ \forall i \in Q_{obs1}. \tag{4}$$

In (4) + operator represents the set translation and $c_{i,t}$, the time dependent translation for the obstacle, is uniformly distributed over a finite set $\mathcal{U}$. Furthermore, it is assumed that the initial position of the robot is not in the vicinity of the obstacles from set Q_{obs1}.

The goal of the path planning is to find the sequence of inputs $a_0, a_1, \ldots, a_N \in \mathbb{A}$ to go from an initial state q_0 to the terminal state q_N in minimal number of time steps such that:

$$P(q_N \notin Q_{goal}) < \delta_1, \tag{5}$$

$$P(q_t \in (Q_{obs1,t} \cup Q_{obs2})) < \delta_2 \quad t = 1, 2, \ldots, N-1. \tag{6}$$

Here P denotes the probability, $Q_{goal} \subset Q_{free}$ is a ball with centre q_{goal} and radius r_{goal}. And δ_1 and δ_2 are arbitrarily small user-defined value in $(0, 1)$. In words, the aim is to find a path from q_0 to Q_{goal} with the probability of collision with obstacles being less than δ_2.

3. MARKOV DECISION PROCESS (MDP)

A Markov Decision Process (MDP) is characterized by the tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, r >$. Here $\mathcal{S} \subset \mathbb{R}^n$ is the state space, $\mathcal{A} \subset \mathbb{R}^m$ is a finite set referred to as the action space, $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the state transition probability which defines the probability distribution over the next states given the current state and action.

In this framework, at a given time instant t the agent's state $s_t \in \mathcal{S}$. The agent performs an action $a_t \in \mathcal{A}$ from state s_t and transitions to a new state s_{t+1} with probability $\mathcal{P}(s_{t+1}|s_t, a_t)$. The agent then receives a reward $r(s_t, a_t, s_{t+1})$ from the environment for the action taken and the state transitioned. In an MDP, the next state s_{t+1} depends only on the present state s_t and action a_t. The action a_t is defined by a mapping $\pi : \mathcal{S} \to \mathcal{A}$, *i.e.*, $\pi(s_t) = a_t$, referred to as a policy. The collection of all possible policies is denoted by Π. A sequence

$$\omega^{(\pi)}_t = (s_t, \pi(s_t), s_{t+1}, \pi(s_{t+1}), \ldots, s_{N-1}, \pi(s_{N-1}), s_N), \tag{7}$$

with a terminal state s_N, is called a π sample path. The discounted cumulative reward (return) corresponding to this sample path is given by

$$R^{\pi}(\omega_t) = \sum_{n=t}^{N-1} \gamma^{n-t} r(s_n, \pi(s_n), s_{n+1}), \tag{8}$$

where $\gamma \in [0, 1]$ is the discount factor.

As the state transitions are random, the expected return from state s,

$$V^{\pi}(s) = \mathbb{E}(R^{\pi}(\omega_t)|s_t = s), \tag{9}$$

The goal is to determine a policy π^* such that

$$\pi^* = \underset{\pi}{\text{argmax}} V^{\pi}(s_t) \tag{10}$$

3.1 MDP Formulation of the Agent

State: The state vector s at time t comprises of the measurements from the distance sensor $(d^{(i)})$ and the 2-D positions (x, y) of the agent (q_t), the obstacles $(q^{(i)}_{obst,t})$ and goal (q_{goal}).

$$s_t = \{q_t, d^{(i)}_t, q^{(j)}_{obst,t}, q_{goal}\} \ , i = 1, 2 \cdots 8, \ j = 1, 2 \cdots 6$$

At each time t there are 24 state variables. Also note that the distance sensors help detect the obstacles in the surrounding vicinity.

Action Space: The action space comprises of 8 discrete actions $\mathcal{A}$={(0,1),(0,-1),(1,0),(1,1),(1,-1),(-1,0),(-1,1),(-1,-1)}. Note each action $a^{(i)} \in \mathcal{A}$, $i = 1, 2, \ldots, 8$ has a component along x and y directions.

Reward: The objective of our path planning problem is to take a sequence of actions in order to reach the goal. The terminal reward on reaching the goal is accordingly defined as +10. In order to minimize the length of the traversed path to the goal, a penalty is given for each step taken by the agent -0.01. If however the agent collides with an obstacle or the environment edges, a reward of -10 is given and the episode is terminated.

Remark:

(1) The user-defined values δ_1 and δ_2 mentioned in Section II are not dealt explicitly in this framework. However it can be handled by an appropriate choice of the reward function.

4. RL BASED METHODOLOGY

RL is a framework used to find an optimal policy π^* for an MDP. To solve the path planning problem formulated as an MDP, a popular RL algorithm called Q-Learning is used (Watkins and Dayan (1992)). Many advanced variants of Q-learning have been developed in recent years and one of them called the Deep Q-Network (DQN) is applied in this work.

4.1 Q-Learning

For a policy π, the Q-value for a state-action pair at any time t is defined as

$$Q^{\pi}(s_t, a_t) = \int_S \Big(r(s_t, a_t, s_{t+1}) + \gamma V^{\pi}(s_{t+1}) \Big) \mathcal{P}^{\pi}(s_{t+1}|s_t, a_t) ds_{t+1}. \quad (11)$$

In words, Q-value for a policy π is the expected return obtained by applying an action a at the state s and following it up with policy π. The objective of the Q-learning algorithm is to find Q-values for the optimal policy π^* *i.e.*, to find $Q^{\pi^*}(s, a)\ \forall s, a$,

$$Q^{\pi^*}(s_t, a_t) = \int_S \Big(r(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1} \in \mathcal{A}} Q^{\pi^*}(s_{t+1}, a_{t+1}) \Big) \mathcal{P}^{\pi}(s_{t+1}|s_t, a_t) ds_{t+1}. \quad (12)$$

4.2 Deep Q-Network

In this method a neural network based function approximator is used to estimate the Q-values for the state-action pairs, (Mnih et al. (2015)). The neural network is a non-linear function, $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where n and m are the dimension of the state space and action space respectively. Note the optimal Q-function satisfies

$$Q^*(s, a) = \mathbb{E}_{s'}[r(s, a, s') + \gamma \max_{a'} Q^*(s', a')|s, a]. \quad (13)$$

which is (12) in a compact form. The neural network is used to estimate the above Q-function such that, $Q(s, a, \theta) \approx Q^*(s, a)$. Here, θ are the parameters (weights) of the neural network. The neural network is trained with the states as inputs and Q-values for each action as output. The states and the actions are as mentioned in Section 3.1. In general to learn the optimal Q values, the parameter θ is updated using a gradient descent with the learning rate α

$$\theta_{i+1} = \theta_i - \alpha \nabla_{\theta_i} L(\theta_i), \quad (14)$$

where

$$L(\theta_i) = \mathbb{E}[(y_i - Q(s, a; \theta_i))^2], \quad (15)$$

being the cost function to be minimized,

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}[(y_i - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)] \quad (16)$$

its gradient with respect to θ,

$$y_i = \mathbb{E}[r(s, a, s') + \gamma \max_{a'} Q(s', a'; \theta_i^-)|s, a] \quad (17)$$

the target Q value and $Q(s, a; \theta_i)$ is the Q-value predicted by the neural network. The target Q-values are estimated using a separate neural network. This network is a copy of the original network but with parameters θ^-. Henceforth, the network used for estimating the target Q-values is called the target network while the other network is called the online network. To improve the stability of learning and improve convergence, the target network weights (θ^-) are updated using the weights of the online network (θ) periodically after every τ steps. In order to ensure all the states are visited for learning their action-value pairs, the environment is initially explored by taking random actions and the corresponding Q-values are updated. The amount of exploration is slowly decayed and the environment is then exploited for high rewards by using a greedy strategy to select an action a in state s that maximizes $Q^*(s, a; \theta_i)$.

Advancements including Double-DQN (Van Hasselt et al. (2016)), Prioritized experience replay (Schaul et al. (2015)) and Dueling Q-Networks (Wang et al. (2016)) were used in order to improve the efficiency of the DQN.

4.3 Neural Network Architecture

The number of neurons in the input layer $n^{[0]}$ is equal to the number of states s considered, *i.e.*, 24. It is followed by 2 hidden layer having $n^{[1]} = n^{[2]} = 512$ neurons each. The hidden layer is followed by a Dueling architecture which involves a layer having $n_V^{[3]} = 1$ neuron for estimating state-value $V(s)$ and $n_{Adv}^{[3]} = 8$ neurons for estimating the advantage function for each action $A(s, a)$. Finally the output layer consisting of $n^{[4]} = 8$ neurons combines these two functions to give $Q(s, a)$. The activation function used is relu and the optimizer used is Adam.

5. REFERENCE ALGORITHMS

To evaluate the quality of the path returned by the deep RL method, simulation trials were performed and compared with popular algorithms like RRT*, FMT*, CC-RRT*, Robust-FMT* and RT-RRT*.

In RRT* and FMT* the environment is randomly sampled and the samples are connected based on some rule, thereby forming a tree. Then the path to the goal from the start is found by searching the constructed tree. A sampled point will be discarded if it falls inside an obstacle or if its connection to the tree passes through an obstacle. In RRT*, the random samples are drawn one at a time while in FMT* the algorithm begins only after the environment is completely sampled. Both of the above algorithm works well in static environment. In RT-RRT* a path from initial position to goal is planned similar to RRT*. But at each time step the agent (robot) will get an update regarding the environment and will re-plan the path if necessary on the go thereby handling dynamic constraints and goal positions. In CC-RRT*, the environment is randomly sampled as mentioned above but each sample point is drawn from a distribution on its own. The randomly sampled points are then connected to the means of their distribution, thereby forming a tree of state distributions. Obstacle

Algorithm 1

Given $A, B, \mathcal{U}, P_w, k_1, k_2$
Initialize an online network θ and a target network θ^- randomly such that $\theta = \theta^-$.
Initialize $\epsilon, \gamma, \alpha, \tau, \lambda_0$ (Refer Sections 6.4 6.5.1)
for each episode i until convergence **do**
    Randomly initialize the environment as per (18),(19),(20)
    for $j = 1$ to N **do**
        With probability ϵ, take random action a_j and with probability $1 - \epsilon$, select $a_j = \underset{a}{\text{argmax}}\, Q(s_j, a_j; \theta)$
        Store tuple $< s_j, a_j, s_{j+1}, r_j >$ in replay buffer
        Sample prioritized mini-batch of size m from replay buffer
        Estimate $L(\theta)$ using the mini-batch samples
        Update θ using Adam optimizer
        For every τ steps,update target network $\theta^- \leftarrow \theta$
    end for
    $\epsilon = \epsilon - \epsilon_{decay}$
    $\lambda_{i+1} = \lambda_i + \Delta\lambda$
end for

collision checking is based on computing the probability of collision with the help of the probability ellipses defined by the covariance matrices of the distributions. Robust-FMT* works similar to CC-RRT*, except that the environment is sampled completely at the start of the algorithm.

6. SIMULATION AND DISCUSSIONS

6.1 Simulation Platform

The simulations are performed on a 7th gen intel i7 processor with 16GB DDR4 RAM. A 11GB RTX 2080 Ti graphics card is used for training the neural network. In order to keep the simulation trials impartial, the trained neural network and reference algorithms used for comparison are evaluated using the computational power of i7 processor only. The graphical environment is designed using Pygame. Here we consider a 2D environment of 600x600 pixels with the top left most corner being the origin. The robot is considered to be a point mass with obstacles enlarged accordingly.

6.2 Environment Dynamics

The simulations are performed using the dynamics in (1) with $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 30 & 0 \\ 0 & 30 \end{bmatrix}$. The input to the system, a is provided as given in Section III. The covariance matrix of the Gaussian process noise is assumed to be $P_w = \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix}$. It is worth emphasizing that the noise in the system was taken to be Gaussian process to facilitate comparison with algorithms like CC-RRT* and Robust-FMT* (as they can handle only Gaussian noise). The Deep RL method can be generalized to learn any type of noise distribution.

The obstacle dynamics are simulated as mentioned in (4). For simplicity it is assumed $\mathcal{U} = \kappa\{$(0,1),(0,-1),(1,0),(1,1),(1,-1),(-1,0),(-1,1),(-1,-1)$\}$. Here, κ is a scaling factor.

6.3 Simulation Environments

The environments used for simulation are classified into static and dynamic. In the former, the positions of obstacles and the goal are fixed for one trial (episode). While in the latter the obstacle (Q_{obs1}) dynamics follows (4) and the goal position could be changed with respect to time.

Static Environment In each episode, the position of the agent, the 6 dynamic obstacles (Q_{obs1}) and the goal are assigned randomly and these positions are fed as states to the neural network. Over the course of training, the RL agent learns to generalize planning a path for various positions of agent, goal and dynamic obstacles in the environment. In cases without process noise, the agent learned to plan optimal paths, some of which being exceedingly close to the obstacles. However, in the presence of process noise in the system, the stochasticity in the position of the next step of the RL agent sometimes resulted in collision with the obstacles when passing near it, thereby resulting in a large negative reward. This implicitly encouraged the agent to maintain a safe distance from the obstacle while planning a path.

$$||q_t - q_{goal}|| < \lambda_i, \tag{18}$$

$$||q^k_{obs1} - q^l_{obs1}|| < \lambda_{min}\ \forall k, l = 1 \dots 6, k \neq l, \tag{19}$$

$$||q^k_{obs1} - q^l_{obs2}|| < \lambda_{min}\ \forall k = 1 \dots 6, l = 1 \dots 4 \tag{20}$$

Dynamic Environment The generalizing capability of the neural network is used to extend the path planning from a static environment to a dynamic environment. In this case the neural network is trained with the obstacle dynamics as mentioned in 6.2. This trains the RL agent to account for the dynamic movement of the obstacles in the environment.

6.4 Training

The DQN algorithm is used as mentioned in section 4. The learning rate α is set to 0.0001. A discount factor γ of 0.99 is chosen. The target network is updated for every 5000 steps τ. The training is done using a simple ϵ greedy policy to choose actions. Here, ϵ was decayed from 1 to 0.1 over the course of 2,00,000 episodes. Once ϵ is decayed to 0.1, an additional 50,000 episodes are trained to improve the efficiency of the final path taken. Let the initial distance between the agent and goal be denoted by λ. Now the training is done as mentioned in Algorithm I.

6.5 Discussions

In this section, the challenges faced during the simulation of the environment and the corresponding strategies used to tackle them are discussed.

Sparse goal state The probability of reaching the goal state without colliding with other obstacles during the exploration phase is very small resulting in the terminal state being the goal state, a sparse occurrence. Prioritized experience replay helps overcome this by prioritizing such transitions. Also, the training time is further improved by keeping the distance between the agent and the goal λ small in the initial stages of learning and gradually

increasing it as the training progresses. This significantly increases the sample efficiency thereby leading to faster convergence.

Static obstacle avoidance Terminating the episode in case of collision with the static obstacles (Q_{obs2}) during training resulted in agent failing to learn obstacle avoidance. This problem is overcome by continuing the training episode even after such collisions.

Loops in the path Initial efforts to train the Q-network largely resulted in the agent being unable to find paths to the goal and instead tracing a loop. This is largely due to very similar states nearby each other resulting is almost similar Q-value predictions. The dueling architecture improved the Q-value predictions for such cases, thereby overcoming the problem of loop formation.

Variable number of obstacles In general, the input dimensionality of a neural network is fixed. However, a variable number of obstacles in the environment implies variable number of input states. Therefore to fix the input dimensionality, dummy obstacles assumed outside the environment boundaries are provided as obstacle states.

7. RESULTS AND COMPARISON

The proposed method was compared with the sampling based algorithms for over 100 different environments. In order to keep the paper concise the comparison results were provided only for one such environment.

7.1 Static environment case

In this case, the Deep RL approach is compared to RRT*, FMT* and RT-RRT*. Figure 2 shows the path computed by various algorithms for the environment. Table 1 provides details regarding the quality of the path determined by the above four algorithm for an environment. Note all sampling based algorithms used in the comparison are initialized to 5000 nodes each.
The parameters determining the quality of the path are the path cost and path duration. The path cost is the total euclidean cost for reaching the goal from the starting position of the robot. The path duration is the total time taken in seconds for computing the path. For the sampling based algorithms the computational time includes the time taken for sampling the environment, growing the tree and searching the tree to find the path. For Deep RL the computational time involves the time taken for forward propagation of the neural network. For effective comparison, 50 trials were performed for the three algorithms and the mean values for the parameters are used. From Table 1, it can be inferred that though the sampling based algorithms compute a lesser cost path, they are not consistent. But the proposed method finds a unvarying path at better computational time.

In case of process noise in the system, the Deep RL algorithm is still able to handle this and compute a path. The performance in this case can be evaluated by comparing it with CC-RRT* and Robust-FMT*. The path quality in this case involves two new parameters called the maximum risk and accumulated risk. The former parameter is the maximum among all the risks of collision involved in the solution path while the latter is a summation of the risks at each time step in the final path. Here, risk is a numerical estimate for the agent's probability of collision with obstacles and is computed as mentioned in Luders (2014). Table 2 provides the comparison results. From Table 2 it is understood that the proposed method computes a lesser cost path much faster than the robust algorithms but with a higher overall path risk. And from Figure 2 it can be seen that the computed path for the proposed method avoid the obstacles by a higher margin compared to non-robust sampling based algorithms like RRT* and FMT*.

Furthermore, the effect of following a particular policy in a static environment under Deep-RL method can be understood by plotting the magnitude and direction of the Q-values as shown in Figure 2.f and Figure 2.g. The arrows in the environment denote the direction the agent will move if it is placed at that location. It can be observed that these arrows converge on to the goal while deviating away from the obstacles. Also the length of the arrows represents the magnitude of the Q-values at that location. It can be seen that the arrows have longer length near the goal elucidating the fact that the regions near the goal have higher Q-values.

7.2 Dynamic environment case

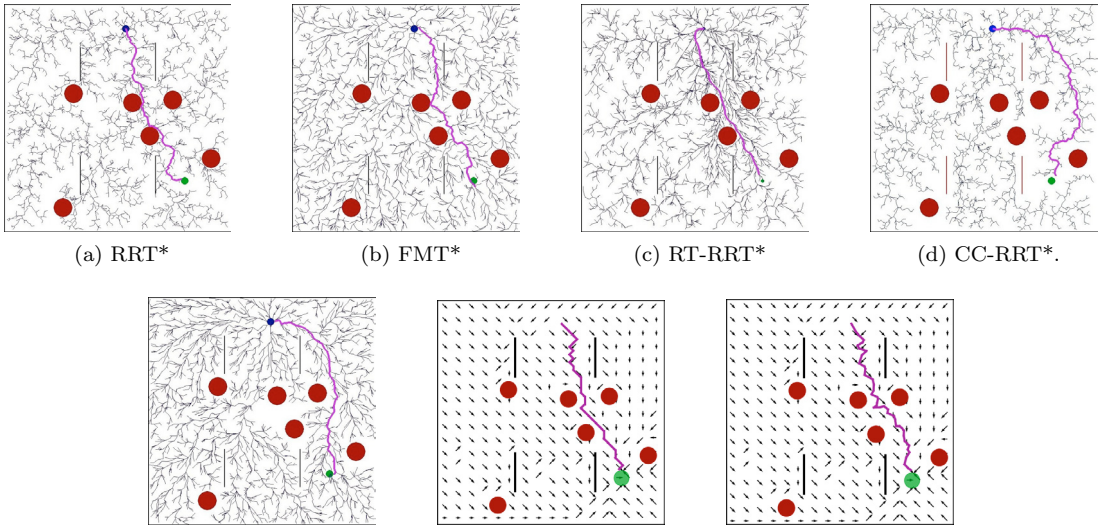
In this case the proposed method is compared with RT-RRT* and CC-RRT*. In all three methods the path to the goal is recalculated at every time step. However the difference being that in RT-RRT* and CC-RRT*, the entire path from current position of the agent to the goal is calculated, whereas for the RL agent, the Q-network only predicts the next optimal action to take at each time step thereby reducing the computational overhead. Here each time step of the environment is treated to be a static case and the neural network finds the next action to take. A video presenting the working of the proposed method in various dynamic environments is provided in `https://youtu.be/8STUwjcKl2Q`. Note for a dynamic environment the Q-value magnitude and direction plot as shown in

Table 1: Path Quality for the Environment without noise

Parameters		**RRT***	**FMT***	**RT-RRT***	**Deep-RL**
Path Cost	Mean	646.78	522.03	521.03	514.26
	SD	63.87	40.10	19.50	-
	Min	492.02	491.38	495.38	514.26
	Max	687.11	591.78	551.78	514.26
Path	Mean	13.37	13.01	9.91	0.151
Duration(s)	Min	12.79	12.76	8.76	0.128
	Max	13.90	13.33	11.93	0.176

Table 2: Path Quality for Environment with noise

Parameters		**CC-RRT***	**Robust-FMT***	**Deep-RL**
Path Cost	Mean	726.92	657.290	550.97
	SD	113.68	64.35	21.8
	Min	565.04	567.41	519.12
	Max	912.22	797.99	584.16
Path Duration(s)	Mean	20.05	18.45	0.162
	Min	19.02	18.01	0.147
	Max	20.90	19.08	0.184
Maximum Risk	Mean	0.0006	0.0007	0.038
	SD	0.0002	0.0002	0.031
	Min	0.00001	0.00003	0.0002
	Max	0.0010	0.0014	0.096
Accumulated risk	Mean	0.0016	0.0018	0.091



(a) RRT* (b) FMT* (c) RT-RRT* (d) CC-RRT*.

(e) Robust-FMT*. (f) Deep RL-without Noise (g) Deep RL-with Noise

Fig. 2: Path computed by various algorithms for an Environment.

figure 2 changes with every time step corresponding to the changing dynamics of the environment components. This allows the RL-agent to choose optimal actions accordingly. It is apparent that RT-RRT* provides path with lower path cost (in terms of euclidean distance) to the goal (Naderi et al. (2015)) but the path considered could lie close to the obstacles. Whereas the path provided by CC-RRT* have less risk of collision but has a higher path cost (Luders (2014)). Also, in both the algorithms the path determined depends heavily on the samples in the environment. The proposed method does not require the environment to be sampled and it exhibits a better performance with lesser risks of collision with obstacles compared to RT-RRT* and lesser path cost as compared to CC-RRT*.

8. CONCLUSIONS AND FUTURE WORK

Contributions of this paper involve developing a RL based framework using Q-network to solve the given path planning problem involving static and dynamic obstacles. Numerical and graphical simulations are performed and compared with various sampling based path planning algorithms. Simulation results suggest that with a marginal drop in path optimality (path cost), the neural network based path planner outperforms the traditional methods in terms of computational speed.

Future work involves developing better encoding methods to handle variable number of obstacles. Further, precision control can be explored which involves moving to a continuous action space involving control signals such as steering and acceleration. Finally, we aim to translate the proposed approach onto a hardware setup and test the efficiency of the approach on a real-time robot.

REFERENCES

Janson, L., Schmerling, E., Clark, A., and Pavone, M. (2015). Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *The International journal of robotics research*, 34(7), 883–921.

Kahn, G., Villaflor, A., Pong, V., Abbeel, P., and Levine, S. (2017). Uncertainty-aware reinforcement learning for collision avoidance. *arXiv preprint arXiv:1702.01182*.

Karaman, S. and Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7), 846–894.

LaValle, S.M. (2006). *Planning algorithms*. Cambridge university press.

Lei, X., Zhang, Z., and Dong, P. (2018). Dynamic path planning of unknown environment based on deep reinforcement learning. *Journal of Robotics*, 2018.

Luders, B.B.D. (2014). *Robust sampling-based motion planning for autonomous vehicles in uncertain environments*. Ph.D. thesis, Massachusetts Institute of Technology.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529.

Naderi, K., Rajamäki, J., and Hämäläinen, P. (2015). Rt-rrt*: A real-time path planning algorithm based on rrt. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, 113–118. ACM.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *CoRR*, abs/1511.05952.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, 1995–2003.

Watkins, C.J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4), 279–292.