Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 56-2 (2023) 10126–10131

Neural motion planning in dynamic environments ⋆

Bernhard Wullt * **Per Matsson** ** **Thomas B. Schön** **
Mikael Norrlöf *

* *ABB Robotics, Sweden, {bernhard.wullt, mikael.norrlof}@se.abb.com.*
** *Department of Information Technology, Uppsala University {per.mattsson, thomas.schon}@it.uu.se*

Abstract: Motion planning is a mature field within robotics with many successful solutions. Despite this, current state-of-the-art planners are still computationally heavy. To address this, recent work have employed ideas from machine learning, which have drastically reduced the computational cost once a planner has been trained. It is mainly static environments that have been studied in this way. We continue along the same research direction but expand the problem to include dynamic environments, hence increasing the difficulty of the problem. Analogously to previous work, we use imitation learning, where a planning policy is learnt from an expert planner in a supervised manner. Our main contribution is a planner mimicking an expert that considers the future movement of all the obstacles in the environment, which is key in order to learn a successful policy in dynamic environments. We illustrate this by evaluating our approach in a dynamic environment and by comparing our planner with a conventional planner that re-plans at every iteration, which is a common approach in dynamic motion planning. We observe that our approach yields a higher success rate, while also taking less time and accumulating less distance to reach the goal.

Keywords: Data-driven control, Learning for control, Robots manipulators, Motion planning, Imitation learning

1. INTRODUCTION

Motion planning is the problem of finding a collision-free path for a robot from a given start configuration to a goal configuration. The traditional setting is to consider a work space where the obstacles are static. Under this setting, the problem has extensively been explored resulting in a set of methods. To be able to solve higher dimensional planning problems, the sampling based motion planners (SMP) were introduced (Lynch and Park, 2017). Fundamental algorithms in this family include rapidly-exploring random trees (RRT) (LaValle, 1998) and probabilistic road maps (PRM) (Kavraki et al., 1996). The common idea behind these methods is to solve the problem by sampling collision-free configurations and connect these to compile a graph or tree from which a shortest path can be found. The sampling nature enables the planners to find collision-free paths in higher dimensional spaces, but the downside is that they are quite computationally expensive.

The inherent high computational cost that is associated with the SMPs and recent advancements in deep learning motivated researchers to instead explore the problem from a perception approach, resulting in a group of planners called neural motion planners (NMP) (Qureshi et al., 2021; Jurgenson and Tamar, 2019; Strudel et al., 2020). The NMPs are given an observation of the robots environment together with the robots current and goal configuration. The planners then map this data to a next configuration which takes the robot closer to the goal. This mapping is parameterized by a deep neural network, which needs to be trained offline before being used. Once trained the NMPs efficiently solves the problem at a fraction of the time compared to state-of-the-art SMPs, independent of the dimensionality of the problem.

Qureshi et al. (2021) presents a planner called motion planning network (MPNet), which consists of two networks, an encoding and a planning network. MPNet represents the obstacle space as a point cloud with a fixed amount of measurements and encodes this by a series of fully connected layers. The network is learned by imitation learning (IL), which is also done in our contribution. Jurgenson and Tamar (2019) instead uses reinforcement learning (RL), training an agent with DDPG (Silver et al., 2014) and HER (Andrychowicz et al., 2017), leveraging an expert planner to speed up the learning. To encode the obstacle region, the view of the robots workspace is encoded by a series of convolutional neural networks. However, only a static setting is considered. The RL approach was continued by Strudel et al. (2020), who shows the importance of how the obstacles are represented. The authors suggest to use PointNet (Qi et al., 2017) as an encoding network, using raw point cloud measurements together with their surface normals from the robots work space. This is similar to MPNet, but using PointNet makes the encoder invariant to input permutations, which is missing in MPNet. Strudel

⋆ This research was partially supported by the *Wallenberg AI, Autonomous Systems and Software Program (WASP)* funded by Knut and Alice Wallenberg Foundation.

10.1016/j.ifacol.2023.10.885

et al. (2020) further show the importance of learning method, reporting that the point cloud representation with normals together with the PointNet encoding trained by using RL outperforms all previous approaches. The authors mainly studied a static setting, but reports that their RL-agent, trained in a static environment, is also successful in a dynamic environment.

Inspired by previous work in the NMPs, we continue to explore the setting with dynamic obstacles. To do this we use IL and mimic an expert planner with full knowledge of the future movements of the obstacles. To mimic this expert we produce a labeled data set that encodes the expert's behavior and then train a deep neural network on this data set in a supervised learning manner. The expert used in our work is an RRT planner adapted to plan in a space-time volume. As obstacle representation, we follow Strudel et al. (2020) and use point cloud measurements, where the points are sampled along the boundary of the obstacles. Contrary to previous work, our main setting is dynamic and to be able to perceive the motion of the obstacles, we encode multiple consecutive measurements. As encoding, we use the PointNet architecture proposed by Strudel et al. (2020). However, since the original implementation only encodes one single point cloud measurement, we have expanded the network by first encoding multiple point clouds with PointNet and then simply concatenating the outputs into one latent-space vector. Our approach is verified in an environment consisting of a two degree of freedom (DoF), SCARA type robot, that should pass a moving obstacle. In this environment we study the performance difference of stacking one point cloud measurement compared to stacking four. Furthermore, to show the importance of mimicking an expert who considers future movement, we also benchmark our IL-agent with a re-planning method, where the planner makes no assumption on the movement of the obstacle.

2. PROBLEM FORMULATION

A robot is defined as a N degree of freedom (DoF) serial manipulator. The N joint angles of the robot are referred to as the configuration, q, and the set of feasible configurations is referred to as the configuration space, $\mathcal{C}$. The robot acts in a world $\mathcal{W} \subset \mathbb{R}^{N_{\mathcal{W}}}$, where $N_{\mathcal{W}} = 2$ or $N_{\mathcal{W}} = 3$. The world contains obstacle regions $\mathcal{U}(t) \subset \mathcal{W}$ which are time dependent and time is denoted by $t \subset \mathbb{R}_+$.

2.1 The motion planning problem

In the following problem formulation, we assume full knowledge of the future movement of the obstacles in the environment. Therefore, the state, x, is defined by a tuple, $x = (q, t)$, and the state-space is the cartesian product $\mathcal{X} = \mathcal{C} \times \mathbb{R}_+$. The set of states where the robot is in collision with any obstacle in the world is defined as $\mathcal{X}_{\text{obs}}$ and the set of obstacle free states is $\mathcal{X}_{\text{free}} = \mathcal{X}_{\text{obs}} \setminus \mathcal{X}$.

A state-transition is defined by applying a velocity vector, $v \in \mathcal{V}$, during a time interval, Δt, which results in a subsequent state $x' = (q + \Delta t\, v, t + \Delta t)$. Thus, it is only possible to transition to future time and with bounded velocity.

We define a start state as x_I. A goal configuration is denoted as q_G, from which we define a goal state as $\mathcal{X}_{\text{goal}} = \{(q, t)|(q, t) \in \mathcal{X}_{\text{free}}, \|q - q_G\|_2 < \varepsilon\}$. The motion planning problem is to find a continuous path $\tau : [0, 1] \mapsto \mathcal{X}_{\text{free}}$, that starts in x_I, $\tau(0) = x_I$, ends in the goal region, $\tau(1) \in \mathcal{X}_{\text{goal}}$ and such that the velocity constraints are satisfied in all state-transitions.

The neural motion planning problem A neural motion planner (NMP) is defined by its policy, $\pi : \mathcal{O} \mapsto \mathcal{V}$, which is a mapping of a given observation, $o \in \mathcal{O}$, to an action, which in our case is a velocity vector, $v \in \mathcal{V}$. The observation that is given is a tuple, $o = (q, q_G, h_t)$, containing the current configuration of the robot, q, the goal configuration, q_G, and some representation of the obstacles, h_t, at time t. Contrary to the motion planning problem, the NMP-agent is given no information of the future movement of the obstacles.

The problem that the NMP-agent should solve is to transition the robot to the goal region while avoiding collisions.

3. IMITATION LEARNING AGENT

To approach the NMP-problem, we used IL. We parameterize our policy by a deep neural network, which we denote by $\pi(o; \theta)$, where θ is the network weights. The network is learned by minimizing the mean squared error loss $L = \sum_{(o_i, v_i) \in \mathcal{D}} (\pi(o_i; \theta) - v_i)^2$, from a labeled data set $\mathcal{D}$ that has been annotated by an expert.

The network we chose is an adapted version of the PointNet architecture introduced by Strudel et al. (2020) which is illustrated in the right part of Figure 1. The need for adaptation comes from that PointNet only supports one point cloud measurement, which is reasonable in a static environment. But since our environment is dynamic, the motion of the obstacles becomes an important factor to consider and since it is not explicitly given, it is instead estimated by encoding consecutive measurements. In our case we chose to stack four consecutive point clouds. Thus our obstacle representation is defined as $h_t = (\mathcal{P}_{t-3}, \ldots, \mathcal{P}_{t-1}, \mathcal{P}_t)$. where $\mathcal{P}_t$ is a point cloud measurement at time t. Each point cloud consists of points, $p_j \in \mathcal{P}_i$, that are sampled on the boundary of the obstacles, this is illustrated in the left part of Figure 1. The network processes the four point cloud measurements separately with the PointNet encoding from Strudel et al. (2020) and it is done as follows. For each point within each point cloud, we concatenate the measurement with the current and goal configuration of the agent and input this to a multi-layer perceptron (MLP) block, which outputs 256 local features for each measurement. All local features from the same point cloud are then stacked and a max pooling transformation is applied along the feature dimension, which produces a global feature vector that encodes that point cloud. Our adaptation is that we concatenate all the global feature vectors from the point clouds, resulting in a latent space representation, z_t, which is then processed by a second MLP block, yielding a velocity v. To distinguish the case when we encode one point cloud measurement

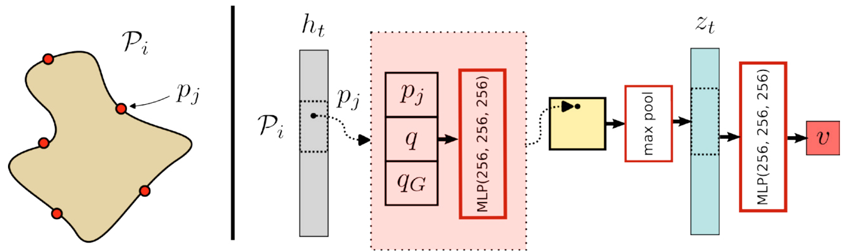

Fig. 1. (Left) Presents a point cloud, $\mathcal{P}_i$, and the corresponding boundary points, p_j. (Right) Illustrates the used network architecture. The network takes as input four point cloud measurements, h_t, as well as the current configuration of the agent q and the goal configuration q_G. Each point cloud measurement is encoded by the PointNet encoding. Thus, for each point cloud, $\mathcal{P}_i$, each point, p_j, is concatenated with q and q_G, and inputted to a shared MLP which outputs local features corresponding to that point. All local features produced from the point cloud are then transformed by max pooling along the feature dimension to produce a global feature vector. This process is repeated for all point clouds. Our contribution is that we concatenate all global feature vectors to one final latent space vector, z_t. This is next inputted to a second MLP that produces a velocity v. The network becomes identical to PointNet if one point cloud is used.

compared to stacking four, we denote the former as PointNet and the latter as StackedPointNet.

4. EXPERT PLANNER

4.1 Background

Traditionally, RRT is used in a static environment and in this context the state is usually the configuration of the robot. RRT uses a tree data-structure to represent the collision-free space with the start state as the root. The main innovation behind RRT is that it can rapidly explore the collision-free space. This is done by sampling new states uniformly from the collision-free state space and then finding the nearest state in the tree. From this state, a local planner moves within a configured max distance in the direction of the sampled state, collision checking the transition, reporting back the state which it reached and if the transition resulted in a collision. If the transition was collision-free, then the reached state is added to the tree. This exploration procedure has the effect of pulling the tree outwards to new unexplored regions. Due to its sampling approach, RRT is probabilistically complete, meaning that it will report back a path to the goal region with probability 1, if there exists one, when the number of samples approaches infinity. The paths that RRT reports back are in general suboptimal. Since RRT's introduction, many new variants have been introduced. Kuffner and LaValle (2000) introduced RRT-Connect, which improved the convergence rate of RRT by using bi-directional search. Thus, a tree from both ends is grown and at each iteration the algorithm tries to connect the trees. Karaman and Frazzoli (2011) improved the suboptimal property of the RRT, by introducing RRT*, which rewires the tree each time a new state is added to the tree. This rewiring is done using a two stage approach. First, the neighbor that can be transitioned to collision-free and has the smallest cost when travelling through is connected with the new state that is added to the tree. Next, the surrounding neighbors that can transition to the new state without collision are checked if they can travel to the start state with a smaller cost through the newly added state, and if so, they are rewired. This rewiring makes the RRT* approach the optimal path each time a new sample is added.

Even though the usual setting is static, RRT has been modified to run in dynamic environments. Methods like ERRT (Bruce and Veloso, 2002), Dynamic RRT (Ferguson et al., 2006) and MP-RRT (Zucker et al., 2007) are re-planning methods that initialize the search with a normal RRT search and then reuses the generated path or tree in a re-planning step, thus being able to adapt to changes in the environment. Dynamic RRT and MP-RRT rely on that the effected edges that become infeasible when the environment has changed can efficiently be removed, and re-grows the tree until a new path is found. ERRT however, only uses the generated path as waypoints to bias the search when re-planning, assuming that the plan does not change too much between iterations. These methods rely on that the collision detection can be done efficiently, which generally is not the case, and that the environment does not change too much between the iterations. Furthermore, some sort of prior knowledge of the obstacle shapes is required in order to approximate them with collision geometries.

Contrary to the re-planning approach, planning in a space-time volume, where the planner has complete knowledge of the future movement is done in Grothe et al. (2022). The authors introduces a planner called ST-RRT*, which is probabilistically complete and asymptotically optimal with respect to arrival time. The search is done in a bi-directional fashion, inspired by RRT-Connect, and rewires part of the tree similar to RRT*.

4.2 RRT planning in space-time

To be able to plan in a time-varying environment, we have adapted a basic RRT planner such that it is compatible with our problem formulation in Section 2.1. Our implemented planner is similar to ST-RRT* in that it plans in the same kind of space, with velocity constraints, and employs a technique ST-RRT* uses to find paths with smaller arrival times. However, our planner lacks all the extra features and asymptotically optimality properties that ST-RRT* has, but it is sufficient for our purpose. We present our planner in Algorithm 1.

We have highlighted in red where the adaptation has been made to the original RRT and where new functionally has been added. We use the same core components that are used in RRT, which are a sampler, a nearest neighbor module and a local planner. The orchestration of these components is the same as in RRT, but the inner workings of the components have been adapted.

As input, we take a start state, x_{start}, a goal region, $\mathcal{X}_{\text{goal}}$ and a max time, $t_{\max}$, that initially upper-bounds the search space in the time-domain. The search is first initialized on line 3, by initializing the nodes, $\mathcal{N}$, and the edges, $\mathcal{E}$, of the tree. In the search loop, a collision-free state is first uniformly sampled within the bounded time-interval $(t_{\text{start}}, t_{\max})$, yielding x_{sample}.

Algorithm 1 Implementation of the adapted RRT-planner for space-time problems.

1: **function** SpaceTimeRRT(x_{start}, $\mathcal{X}_{\text{goal}}$, $t_{\max}$)
2: $(t_{\text{start}}, q_{\text{start}}) = x_{\text{start}}$
3: $\mathcal{N} = \{x_{\text{start}}\}$, $\mathcal{E} = \{\}$
4: **while** $\mathcal{N}$ is not full or time is not out **do**
5: $x_{\text{sample}} =$ Sample(t_{start}, $t_{\max}$)
6: $x_{\text{nearest}} =$ FindNearest($\mathcal{N}$, x_{sample})
7: $(\text{status}, x_{\text{new}}) =$ PlanLocally($x_{\text{nearest}}, x_{\text{sample}}$)
8: **if** status is collision free **then**
9: $\mathcal{N} = \mathcal{N} \cup \{x_{\text{new}}\}$
10: $\mathcal{E} = \mathcal{E} \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$
11: **if** $x_{\text{new}} \in \mathcal{X}_{\text{goal}}$ **then**
12: $(q_{\text{new}}, t_{\text{new}}) = x_{\text{new}}$
13: $t_{\max} = \text{MIN}(t_{\text{new}}, t_{\max})$
14: **end if**
15: **end if**
16: **end while**
17: **return** GetPathFromTree($\mathcal{N}$, $\mathcal{E}$, $\mathcal{X}_{\text{goal}}$)
18: **end function**

Next, the state that is closest to x_{sample} is found by calling the nearest neighbor function FindNearest, which can be summarized as solving the following problem

$$\underset{x'}{\operatorname{argmin}} \left\{ \rho(x_{\text{sample}}, x') \,|\, (q', t') \in \mathcal{N}, t' > t_{\text{sample}} \right\}, \quad (1)$$

where the following distance function is used

$$\rho(x, x') = \|q - q'\|_2 + \gamma \, |t - t'| \, . \quad (2)$$

In the distance function above, γ, is a parameter to weight the importance of states closer in terms of configuration or time. In RRT, usually all states are considered in the nearest search and the state with the smallest norm is reported as the nearest. However, since we are also considering time, a standard implementation could potentially report a state in the past, which is unreachable by the transition dynamics. With a valid nearest state found, x_{nearest}, a local planner is employed at line 7 starting from x_{nearest}, moving in the direction of x_{sample}. A basic implementation of the local planner is simply to interpolate a straight line path, starting from x_{nearest} and extending to x_{sample}. However, this planning policy could potentially violate the velocity limit that is defined for our problem. Hence, to adhere to the velocity constraints, our planner starts from x_{nearest}, computes the largest valid velocity that will transition the agent in the direction of x_{sample}, and then applies the resulting velocity, v, according to the transition dynamics, yielding a subsequent state. The transition is checked for collision and if it is valid, the process is continued until x_{sample} is reached, the final time step equals t_{sample}, some constraint in number of time steps or accumulated configuration distance is exceeded or collision occurs in any transition. Thus, in the end the local planner reports back if the plan ended in collision and the last collision-free state, x_{new}.

If the local plan to x_{new} is collision-free, then the state is added to the tree (line 9 and 10). The newly added state is then checked if it is in the goal set, line 11. If this is the case, we use a technique from ST-RRT*, where we push down the upper-bound of the time horizon by taking the smallest time. In the original RRT the search would in this case be stopped, but here we continue the search to try to find paths that reach the goal set in less time.

5. EXPERIMENTAL RESULTS

5.1 Environment

The environment used in the verification experiments is presented in Figure 2. The environment consists of an agent which is a two DoF manipulator and another two link robot which represents the dynamic obstacle. The goal configuration is visualized using transparent blue in the figure. The agent has two joints, θ_1 and θ_2, and their rotation directions are defined in the figure. The obstacle has a mobile base, thus in total it is assigned four DoF. The obstacle moves along a deterministic trajectory that is generated by cubic via-point interpolation from randomly generated via-points and velocities. Each time when the environment is reset a new start state, x_I, goal configuration, q_G, and obstacle trajectory is randomly selected.

To make the planning problem more challenging we introduce some constraints. We limit the agent's first joint to lie within the interval $\theta_1 \in (-\pi, \pi)$, thus forcing it to steer past the obstacle in order to reach the goal region. Furthermore, to hinder the agent from simply folding itself into one geometry, the second joint is limited to lie in the interval $\theta_2 \in (-3\pi/4, 0)$. The configuration space of the agent therefore becomes $\mathcal{C} \subset \mathbb{R}^2$ and the set of feasible velocities is defined as $\mathcal{V} = \left\{ v \,|\, v \in \mathbb{R}^2, \, \|v\|_\infty \leq 0.1 \right\}$. In the simulations, time is discretized with $\Delta t = 1$ time unit.

However, since the environment is initialized with random start state and goal configuration, it may be possible for the agent to solve the problem by directly steering to the goal without having to maneuver past the obstacle. Therefore, an uninformed policy that always steers directly to the goal will have an overall success rate of roughly 60 %. The cases where this policy is successful are referred to as simple, otherwise it is referred to as complex.

5.2 Implementation and evaluation details

Data set In order to learn a policy with IL, a data set was compiled by running Algorithm 1 in 6883 instances of the environment. For each instance, 100 random start and goal configurations were solved by the planner from which all the paths were collected. For each path the corresponding

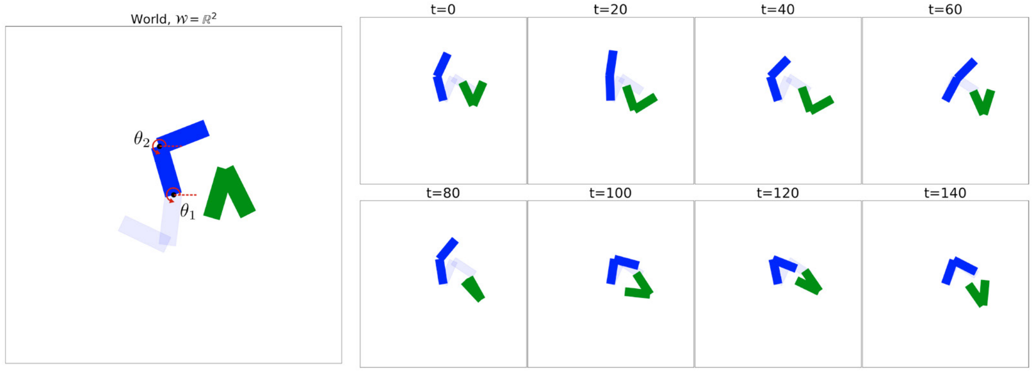

Fig. 2. (Left) View of the world. The blue solid blocks represents the agent, with θ_1 and θ_2, being the joint angles of the agent. Its goal configuration is the transparent blue blocks. The green blocks are the dynamic obstacle. (Right) Illustrating a typical sequence of movements in the environment.

velocities that generated the transitions were calculated. All states within a time interval of 300 time steps were used, resulting in 31×10^6 data points. Regarding the point cloud measurements, we sampled 128 boundary points for each point cloud.

Training To study if our choice of encoding has any effect on performance, we trained both PointNet and StackedPointNet on the expert data set. During training we used a learning rate of 10^{-3} with ADAM (Kingma and Ba, 2015) as optimizer. The performance was evaluated after each epoch by running the model on 100 random instances of the environment and calculating its success rate. The model with the highest success rate was saved and used for further evaluation.

Benchmark As a benchmark to our IL-agent, we use a re-planning strategy similar to a planner like ERRT, which was implemented by assuming that the obstacle is static at each time step. We used RRT-Connect to find a path to the goal and from this, a velocity is computed which moves the agent the longest distance along the path during one time step. This velocity is used to transition the agent to a next time step in the environment and the procedure is then repeated. We don't set any real-time constraints for each re-planning, simply giving the planner enough time to find a valid path. Since the obstacle is assumed to be static, it is possible to run into the case where the problem becomes unsolvable, due to the fact that the pose of the obstacle could cut-off the collision-free space in two halves. When this happens, we calculate the smallest distance to the obstacle and if the distance is larger than a certain threshold, then we simply apply zero velocity. Otherwise, we apply a simple collision avoidance policy, which is to apply the velocity that gives the largest distance to the obstacle. We denote this re-planning policy as ReRRT-C.

Evaluation To study the performance of all the planners, 1000 random instances of the environment were generated and all planners were then evaluated on the same instances.

5.3 Results

The results from the evaluation of all the planners are presented in Table 1.

Table 1. Evaluation results from the planners. Path length and time steps are mean values from all evaluations were the trials were successful.

Planner		All	
	Success rate	Path length	Time steps
ReRRT-C	89 %	2.44	56
PointNet	78 %	1.82	39
StackedPointNet	93 %	2.35	57
		Simple	
ReRRT-C	100 %	1.16	12
PointNet	98 %	1.17	15
StackedPointNet	100 %	1.17	15
		Complex	
ReRRT-C	73 %	5.06	145
PointNet	49 %	3.81	112
StackedPointNet	83 %	4.50	132

First, we compare the performance between PointNet and StackedPointNet. We see a clear difference in performance, where PointNet is underperforming, not being able to solve the problem at a satisfactory rate. Hence, this shows the importance of stacking frames. Next, we study the difference between ReRRT-C and StackedPointNet. If we consider the overall performance, we see that StackedPointNet performs slightly better. However, if we dissect the cases into the simple and complex cases, we first see that there is no real difference in the simple cases. The difference is much more notable in the complex cases. Here we see a much clearer difference in the success rate, where there is a distinct 10 percentage point gap in success rate between the planners, where StackedPointNet is performing better. Furthermore, if we consider the mean path length and mean time steps taken, we see that ReRRT-C attains higher mean values, which could be due to that the planner is repelled by the obstacle when it comes too close or that the planner in some cases approaches the goal configuration too early and has to step back if the obstacle obstructs its path. However, without the obstacle avoidance policy, the success rate would be lower, thus there is a trade-off here between path length and success rate. The IL agent however, has learned a policy which considers the future movement of the obstacle, and therefore ends up accumulating less distance and time steps.

6. DISCUSSION

Strudel et al. (2020) reports that they obtained better results when they used boundary points together with their corresponding surface normals in the point cloud representation for PoinNet. We have also evaluated this extended representation, but we observed no improvement in performance. Furthermore, the authors report that RL gave much better performance compared to using IL and that their RL agent trained in a static environment also worked well in dynamic environments. We have tested our best performing agent with the authors approach and have observed better performance with our IL agent, but we do not include these results here since it is hard to make a fair comparison, and only mention that our IL approach is competitive for NMP in dynamic environments.

One of the main advantages with the NMPs that has not been considered is its computation time. This aspect was not considered due to that it is hard to make a proper comparison, considering all levels of optimization that can be done.

7. CONCLUSION AND FUTURE WORK

Our work proposes to use IL to train an NMP to plan the movement of a DoF robot in a dynamic environment. We do this by learning from an expert motion planner with complete knowledge of the future movement of all obstacles in the environment. To parameterize our policy, we use an adapted version of PointNet that encodes multiple point cloud measurements from the environment. To demonstrate the importance of considering future movements in the expert planner, we compared our NMP with a re-planning strategy that assumes a static environment at each time step. From the experiments, we first showed the importance of stacking multiple observations, observing that if only one point cloud was used resulted in low performance. When we compared our NMP with the re-planning approach, we observed a distinct difference in terms of success rate in the complex cases, where the re-planning approach resulted in a 73 % success rate compared to 83 %.

For future work, we plan to study the performance if we imitate a planner like ST-RRT*, with the goal of obtaining a more optimal policy, which was not considered in this work. Furthermore, we intend to study other learning methods, like RL, to see what works best. Finally, we will develop a more general environment and work with higher DoFs in order to make the problem more realistic.

REFERENCES

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O., and Zaremba, W. (2017). Hindsight experience replay. *Advances in neural information processing systems (NeurIPS)*.

Bruce, J. and Veloso, M. (2002). Real-time randomized path planning for robot navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Ferguson, D., Kalra, N., and Stentz, A. (2006). Replanning with RRTs. In *Proceedings IEEE International Conference on Robotics and Automation (ICRA)*.

Grothe, F., Hartmann, V.N., Orthey, A., and Toussaint, M. (2022). ST-RRT*: Asymptotically-Optimal Bidirectional Motion Planning through Space-Time. In *2022 International Conference on Robotics and Automation (ICRA)*.

Jurgenson, T. and Tamar, A. (2019). Harnessing reinforcement learning for neural motion planning. In *Robotics: Science and Systems XV (RSS)*.

Karaman, S. and Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7), 846–894.

Kavraki, L.E., Svestka, P., Latombe, J.., and Overmars, M.H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4), 566–580.

Kingma, D.P. and Ba, J. (2015). Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun (eds.), *3rd International Conference on Learning Representations (ICLR) 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kuffner, J. and LaValle, S. (2000). RRT-connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation (ICRA)*.

LaValle, S. (1998). Rapidly-exploring random trees : a new tool for path planning. *The annual research report*.

Lynch, K. and Park, F. (2017). *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press.

Qi, C.R., Su, H., Mo, K., and Guibas, L.J. (2017). Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

Qureshi, A.H., Miao, Y., Simeonov, A., and Yip, M.C. (2021). Motion Planning Networks: Bridging the Gap Between Learning-Based and Classical Motion Planners. *IEEE Transactions on Robotics*, 37(1), 48–66.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International conference on machine learning (ICML)*. PMLR.

Strudel, R., Garcia, R., Carpentier, J., Laumond, J.P., Laptev, I., and Schmid, C. (2020). Learning Obstacle Representations for Neural Motion Planning. In *Conference on Robot Learning (CoRL) 2020*. Cambridge MA / Virtual, United States.

Zucker, M., Kuffner, J., and Branicky, M. (2007). Multipartite RRTs for Rapid Replanning in Dynamic Environments. In *Proceedings 2007 IEEE International Conference on Robotics and Automation (ICRA)*.