Deep Reinforcement Learning for Motion Planning of Quadrotors Using Raw Depth Images

Efe Camci
School of Mechanical and Aerospace Engineering
Nanyang Technological University
Singapore
efe001@e.ntu.edu.sg

Domenico Campolo
School of Mechanical and Aerospace Engineering
Nanyang Technological University
Singapore
d.campolo@ntu.edu.sg

Erdal Kayacan
Department of Engineering
Aarhus University
DK-8000 Aarhus C
Denmark
erdal@eng.au.dk

***Abstract*—In this work, we introduce a novel, end-to-end motion planner for quadrotor navigation. Informed by a rough path to goal in partially unknown environments, our method creates desirable motion plans using raw depth images from a front-facing camera. It exploits correlations between local spatial portions of these images to generate desirable motion primitive sequences on the fly without conducting explicit sensing-reconstructing-planning. We evaluate our method through an extensive comparison with three competitor algorithms over ten different environments in AirSim simulations. Our method outperforms its competitors in terms of safe navigation distance, navigation time, and crash rate over 50 flights. We also deploy our method for real flight tests with DJI F330 Quadrotor equipped with Intel RealSense D435, and demonstrate its real-time applicability. Our method successfully performs 15 real flights in three different environment settings with increasing complexity. The experiments can be found at https://youtu.be/hw0sxNwliqs.**

I. Introduction

Autonomous navigation of quadrotors requires sensing [1]–[3], planning [4]–[6], and control [7]–[9]. Majority of state-of-the-art navigation methods perform these tasks separately to attain modularity. In this way, each module can easily be designed, developed, analyzed within their own scope, and combined with the existing modules in literature. However, if a module in the pipeline fails, other modules are most likely to fail as well. If individual modules are not developed considering possible fails of other modules, overall navigation can be deteriorated. To circumvent this issue, we develop a single module which provides end-to-end reasoning from raw sensory inputs to motion plans using deep reinforcement learning (RL) [10]–[12].

According to our navigation scenario, the environment is partially known without exact obstacle location information, an initial rough path to the goal is given, and the concatenation of desirable local motion plans for safe and quick navigation is to be found. To this end, we propose an RL agent which solely uses a series of raw depth images and relative position information of a moving setpoint on the initial rough path in order to generate local motion plans. We particularly train a deep Q-network (DQN) with around 400,000 parameters which exploits correlations between these two coarse inputs to yield desirable motion primitive sequences on the fly.

This work is financially supported by the Singapore Ministry of Education (RG185/17) and Aarhus University, Department of Engineering (28173).

The specific contribution of this work is a novel end-to-end motion planner which fuses different types of coarse inputs (raw image and relative position) without any pre-processing to generate desirable motion primitive sequences for a quadrotor.

II. Related Work

The specific problem considered herein has been a topic of recent research in robotics community. In [5], researchers present a conservative trajectory-optimization-based local planner together with a local exploration strategy for quadrotor navigation. They build upon their previous work [13] in which a reactive local planning method is proposed to avoid obstacles using disparity maps. Both methods require an explicit reconstruction of a local map first, and perform planning within this map thereafter.

In [14], researchers introduce a collision avoidance method which exploits point cloud data and samples minimum-time motion primitives. They suggest that the computational burden of building a map can be alleviated using k-d tree but the method still needs some sort of world model representation. Another computationally lightweight method [15] introduces neural network control policies which compute control commands directly from sensor inputs. However, the method requires supervision from time-free model-predictive path-following controller in order to avoid obstacles. In [16], researchers develop a vision-based reactive planning algorithm inspired by model predictive control. This method requires exact representation of obstacles in order to include them in a nonlinear program formulation.

From end-to-end learning perspective, researchers in [17] train a deep RL agent using virtual raw monocular images in a variety of environments within low fidelity simulations to extract collision avoidance policies. While the method circumvents the challenges of 3D reconstruction and demonstrates decent results in terms of collision avoidance, it is yet to be incorporated into a complete path planning framework in which quadrotor navigates to the goal as quick as possible. Another learning-based algorithm is proposed in [18] which follows high-level navigation instructions by direct mapping between images, instructions, pose estimates and control. In [19], researchers train a network for detecting gates in drone racing kind of tracks followed by a desired heading and velocity generation. They demonstrate successful passes through gates in real flights but sensing and planning modules are separate in this method.

Inspired by some of the methods above, our approach sidesteps separate sensing-reconstructing-planning. It provides direct reasoning from sensory inputs to local motion plans. It generates (mostly) smooth motion plans by exploiting Bézier curves as motion primitives. Besides decent collision avoidance, our method caters for quick navigation to goal in dense environments.

III. Approach

The main challenge in developing RL-based algorithms successfully is to formulate the RL problem in a fashion that the agent would be able to derive useful representations of an environment from sensory inputs, and it could exploit them for generalization to new situations. In this regard, success of many RL-based algorithms lies in proper design of main elements: state, action, and reward. We explain each of these elements in the next subsections while depicting them in Fig. 1 as the main portions of our RL system.

A. State

State includes information on the current respective situation of the agent in the environment. It consists of three consecutive 32×32 depth images taken from the front-facing camera and the relative position of the moving setpoint with respect to the quadrotor in its body frame, x_B, y_B, z_B. The former informs the agent about obstacle locations implicitly while the latter gives insight on the proximity of the agent to the moving setpoint. Since the agent's aim is to avoid a priori unknown obstacles while moving towards the goal, these two items constitute necessary information for the agent to reason about its current situation. Depth images are taken directly from the front-facing camera without any image processing and fed to the DQN. Relative position is calculated by simply substracting the quadrotor's position vector from the moving setpoint's (that moves at the approximate navigation speed which is 1 m/s in this work) and transforming the resultant vector into the quadrotor's body frame.

B. Action

Action defines the agent's move at each time step. It is designed to be in the form of motion primitives which are based on Bézier curves. They are parametric curves based on Bernstein polynomials:

$$C : [0,1] \longrightarrow \mathbb{R}, \quad C(t) = \sum_{i=0}^{n} \binom{n}{i} P_i B_{n,i}(t), \tag{1}$$

where P_i are control points and $B_{n,i}(t)$ are Bernstein polynomials of n^{th} degree which are given as:

$$B_{n,i}(t) = (1-t)^{(n-i)}\, t^i. \tag{2}$$

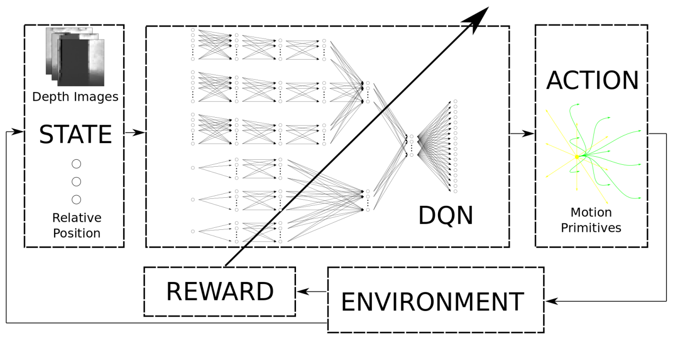

Fig. 1. Overview of the proposed RL system.

The cubic Bézier curves (n=3) are used to represent smooth motion primitives in x_B, y_B, z_B for each finite time step.

The agent selects an action at each time step among the action set which consists of 18 different primitives. This set is constructed on the account that the agent has a forward-biased motion since it has only a front-facing camera as its exteroceptive sensor. The set is depicted in Fig. 2. The length of the curves (and lines) can be adjusted for different case studies based upon the environment density and the quadrotor agility. In this work, since we consider the approximate navigation speed as 1 m/s, we select primitives which result in at most 1 m of navigation in each axis and which are executed at 1 Hz of re-planning.

C. Reward

Reward serves as an instructive feedback for the agent to evaluate its actions. It is designed based upon the relative motion of the quadrotor with respect to the moving setpoint within a time step. The Euclidean distance between the quadrotor and the moving setpoint is calculated at the beginning (d_{t^-}) and end of each time step (d_t). A raw reward is obtained using the change in this quantity through the current time step ($\Delta d = d_t - d_{t^-}$), and it is discounted using the same quantity obtained at the end of the current time step. Besides, a primary logic is added to the reward function to inform the agent about collisions and excessive deviations from the initial rough path. In this sense, if the agent moves further away from the initial path, it gets a lower reward. If it stays closer to the initial path, it gets a higher reward. If it crashes into obstacles, it gets a drastic punishment. If it deviates from the initial path excessively, it gets another milder punishment. This reward logic is designed for the agent to learn how to avoid obstacles safely while following an initial rough path fairly in unknown environments. It is formally defined as:

$$R = \begin{cases} (R_l)\, f(d_t), & \text{for } \Delta d_u < \Delta d \\ \left(R_l + (R_u - R_l)\, \dfrac{\Delta d_u - \Delta d}{\Delta d_u - \Delta d_l} \right) f(d_t), & \\ & \text{for } \Delta d_l \le \Delta d \le \Delta d_u \\ (R_u)\, f(d_t), & \text{for } \Delta d < \Delta d_l \\ R_{dp}, & \text{for } d_{max} < |d_t| \\ R_{cp}, & \text{for collision.} \end{cases} \tag{3}$$

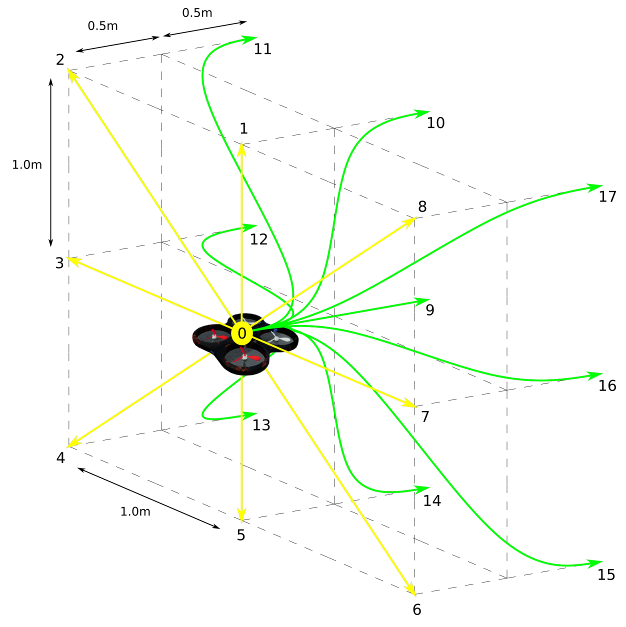

Fig. 2. Motion primitives as the action set.

The variables R_l and R_u are the lower and upper limits on the reward, and they are selected as 0 and 1 in this work. The terms Δd_l and Δd_u are the limits on Δd beyond which the reward is saturated to its bounds, and they are selected as 0 and 1 m, respectively. The term d_{max} is the limit which determines the excessive deviation and it is equal to 5 m. The term R_{dp} is the mild punishment for excessive deviation which is equal to -0.5, and R_{cp} is the drastic punishment for collision which is equal to -1. Lastly, f is a function to regulate the discount rate based upon the maximum allowable deviation (d_{max}). It is introduced both to avoid unreasonably high discount rates when the quadrotor is fairly keeping up with the moving setpoint and to yield very high discount rates if the quadrotor fails to navigate towards the goal. It is formally defined as:

$$f : \mathbb{R} \longrightarrow (0,1), \quad f(d_t) = \frac{1}{2}\left(tanh\left(\frac{2d_{max} - d_t}{d_{max}}\right) + 1\right). \tag{4}$$

While R_l, R_u, R_{dp}, R_{cp}, and f are crafted by trial-and-error, the terms Δd_l, Δd_u, and d_{max} are determined considering the usual operational speeds of our quadrotor in the environments of interest. All of these parameters are flexible to be adjusted for different case studies.

D. Algorithmic Details

The aim of an RL agent is to find the optimal policy which maximizes sum of returns over the long run. The essential idea behind many RL algorithms is to estimate the action-value function $Q(s,a)$ which can then govern the agent's policy by yielding the expected returns for state-action pairs. In theory, if an agent would have infinite number of action trials and would use the Bellman equation as an iterative update, it would converge to the optimal action-value function $Q^*(s,a)$ eventually. However, it is impractical in real world scenarios to have infinite number of trials and estimate the action-value function for each state-action pair separately. Therefore, function approximators have emerged as a useful choice to estimate the action-value function by adding a certain level of generalization [20]. In this work, we use a DQN (Fig. 3) for this purpose, specifically to approximate the following function:

$$Q(s,a) = \mathbb{E}\left(R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') \middle| s_t = s, a_t = a\right) \tag{5}$$

where s_t is the state that the agent is in and a_t is the action taken at the beginning of the time step t. The terms s_{t+1} and R_{t+1} are the next state which the agent reaches and the reward it observes at the end of the time step t, respectively. The term γ is the discount factor which determines the present value of the future rewards [21].

The proposed RL algorithm in this work is an off-policy, episodic training algorithm (Algorithm 1). Instead of updating the network parameters θ_i at each time step, we update them (using Adam [22] in PyTorch with default settings) after each certain number of episodes within which minimum number of interactions (64 in this work) is reached. During the update, all data in experience replay (ER) are used through random minibatches of 64. We use Huber loss while creating gradient for network parameters update [23]:

$$L_\delta\left(y_j - Q\left(s_j, a_j; \theta_i\right)\right) =$$
$$\begin{cases} \frac{1}{2}\left(y_j - Q\left(s_j, a_j; \theta_i\right)\right)^2, & \text{for } \left|y_j - Q\left(s_j, a_j; \theta_i\right)\right| < \delta \\ \delta\left|y_j - Q\left(s_j, a_j; \theta_i\right)\right| - \frac{1}{2}\delta^2, & \text{otherwise.} \end{cases} \tag{6}$$

The variable j refers to data sample indice in a minibatch, δ is a constant for the steepness of the piecewise loss function and it is equal to 1. The terms y_j and $Q\left(s_j, a_j; \theta_i\right)$ are the target and estimated values for the network with parameters θ_i where i stands for the episode number.

IV. Experiments

The experiments for validating the method involve training and testing stages. First, we train an agent in seven virtual environments to have diversified retrospective knowledge. Then, we test the agent in ten virtual environments which are unseen during the training stage to demonstrate sufficient generalization. Lastly, we test the same agent in real flights to demonstrate successful sim-to-real knowledge transfer.

A. Training in AirSim

The simulation software we use is the Microsoft AirSim [24]. It is an open source software for simulating drones, cars, and people. It is based on Unreal Engine game software. It offers high fidelity simulations especially for drones. We create seven training environments in AirSim as depicted in Fig. 4. The environments are diversified regarding their complexity, from obstacle-free environment (Env. 1) and obstacle-free corridors with different widths (Envs. 2 and 3) to left-right (Envs.

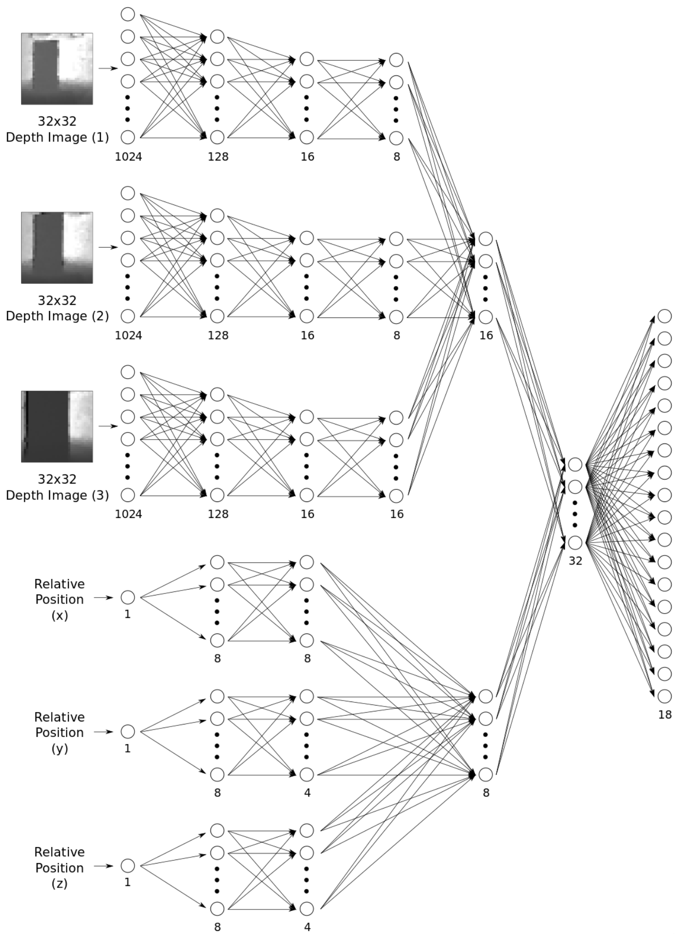

Fig. 3. The proposed DQN architecture. Each element of the state is processed separately in a few layers first and combined in a few other layers thereafter in order to resemble the conceivable feature extraction and decision making paradigms.

4 and 5) and up-down (Envs. 6 and 7) slalom environments. The start points in these environments are on the side where the quadrotor is located in Fig. 4. Accordingly, the goal points are on the other side, and the quadrotor tries to navigate there safely by avoiding obstacles. The distance between start and goal points is 12 m in all the training environments. We merge these environments in a single AirSim session. The agent visits them randomly during the training stage in order to randomize the learning process sufficiently. We find merging them particularly helpful for improved learning performance as compared to training in different environments sequentially. Diversification of data samples during learning is enhanced by this way, which possibly improves the data sample efficiency.

We train the agent for 2000 episodes using an ε-greedy policy. We utilize linear decay on ε from 1.0 to 0.1 for the first half of the training and keep it constant at 0.1 for the second half. We utilize the discount factor γ as 0.5 to have a balance between stable but short-sighted and far-sighted but unstable agents. The average reward throughout the episodes can be seen in Fig. 5. The average reward first increases at the earlier episodes, then stabilizes towards the end when the agent gains sufficient knowledge for near-optimal behaviour.

Algorithm 1 Episodic deep Q-learning

```
for i in episodes do
    observe state $s$
    for t in time steps do
        take action $a$ governed by $\varepsilon$-greedy policy
        observe state $s'$ and reward $R$
        save data sample ($s$,$a$,$s'$,$R$)
        if $R < 0$ then
            break
        end if
        $s$ = $s'$
    end for
    update ER with new data
    if min interaction limit is reached since the last update then
        divide ER into random minibatches of size $n$
        for j in minibatches do
            pick the minibatch with indice j
            apply $y_{1:n}$ = $R_{1:n}$ + $\gamma \max_{a'} Q({s'}_{1:n}, a'; \theta_i)$
            update DQN weights by $L_\delta(y_{1:n}$ - $Q(s_{1:n}, a_{1:n}; \theta_i))$
        end for
    end if
end for
```

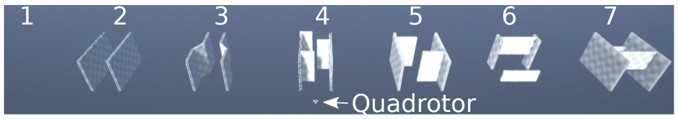

Fig. 4. Training environments: obstacle-free, corridor, and slalom tracks.

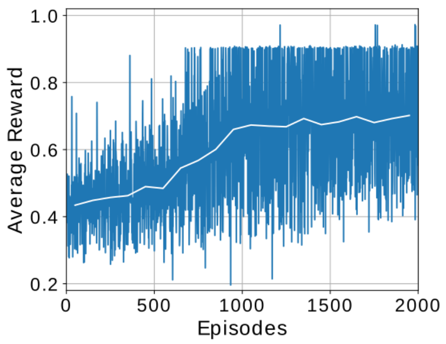

Fig. 5. Average reward through the training episodes. The white line represents the mean value of each 100 episodes over a sliding window.

B. Testing in AirSim

Following the training stage, we deploy the agent in ten different test environments (Fig. 6) which are unseen during training. They consist of obstacles with different shape, size, and orientation as well as corridors with different width and length to assess the generalization capability of the proposed agent. In each environment, the quadrotor starts on the end where it is located in Fig. 6 and tries to navigate towards the other end. The distance between these two ends is 30 m for

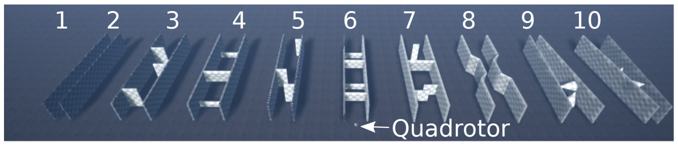

Fig. 6. Test environments with obstacles of different shapes and sizes.

all the test environments[1].

We test the proposed DQN in each environment for five times to have generalized evaluative results. We also compare it with three competitor algorithms. Two of these are the preliminary versions of the proposed DQN: DQN-$\beta.1$ and DQN-$\beta.2$. DQN-$\beta.1$ is a version in which the relative position information of the moving setpoint is omitted in the state definition. DQN-$\beta.2$ is another version in which the relative position information is kept, but paired with only a single depth image instead of three images. By the comparison with these two versions, we aim at an ablation study which justifies our design choices for the proposed DQN. We also include a comparison with a conventional method in literature: potential fields (PF) based planning [25]. In the case of PF, the explicit sensing-reconstructing-planning process is conducted using the depth sensor. We keep the resolution (32×32) and update frequency (1 Hz) of depth images as the same for a fair comparison. We create online point cloud representation of the environment by these depth images and convert it into a probabilistic voxel map with a voxel size of 0.5 m, incrementally. Each voxel creates a repulsive force proportional to its occupancy probability and inversely proportional to the square of its distance to the quadrotor. The moving setpoint creates an attractive force proportional to the square of its distance to the quadrotor[2]. We calculate the overall force vector and map it into linear velocities with maximum velocity being 1 m/s.

Table I states average navigation distance, navigation time, and crash rate for the proposed DQN along with its competitors. In terms of averaged navigation distance over five trials, the proposed DQN either outperforms its competitors or yields a similar performance to them in 8 of 10 environments. PF yields a similar but little downgraded navigation distance performance especially in Envs. 3, 7, and 9. DQN-$\beta.1$ and DQN-$\beta.2$ yields worse results. The former does not have enough guidance to move towards goal, confuses its way towards dense regions, and ends up with a crash. The latter avoids the obstacles at the first glance but does not have a sense of passing them due to the lack of previous depth images. It crashes into the obstacles mostly from the side when they are out of sight.

TABLE I
AVERAGED TEST RESULTS IN AIRSIM.

Env.#	Algorithm	Navigation distance (m)	Navigation time (s)	Crash rate
Env. 1	DQN-$\beta.1$	30.00	33.44	0%
	DQN-$\beta.2$	30.00	33.18	0%
	PF	30.00	33.24	0%
	DQN	30.00	33.70	0%
Env. 2	DQN-$\beta.1$	30.00	33.35	0%
	DQN-$\beta.2$	30.00	33.45	0%
	PF	30.00	34.04	0%
	DQN	30.00	33.43	0%
Env. 3	DQN-$\beta.1$	15.07	-	100%
	DQN-$\beta.2$	23.72	-	100%
	PF	28.59	47.29	20%
	DQN	30.00	35.21	0%
Env. 4	DQN-$\beta.1$	10.36	38.56	80%
	DQN-$\beta.2$	30.00	36.37	0%
	PF	30.00	41.64	0%
	DQN	30.00	35.80	0%
Env. 5	DQN-$\beta.1$	25.97	42.04	60%
	DQN-$\beta.2$	25.36	41.75	40%
	PF	30.00	41.24	0%
	DQN	30.00	34.49	0%
Env. 6	DQN-$\beta.1$	19.43	38.18	60%
	DQN-$\beta.2$	30.00	36.35	0%
	PF	30.00	39.24	0%
	DQN	26.48	35.63	20%
Env. 7	DQN-$\beta.1$	30.00	36.14	0%
	DQN-$\beta.2$	28.29	38.50	40%
	PF	25.38	47.04	40%
	DQN	29.02	35.88	20%
Env. 8	DQN-$\beta.1$	30.00	33.40	0%
	DQN-$\beta.2$	30.00	33.17	0%
	PF	30.00	32.84	0%
	DQN	30.00	33.46	0%
Env. 9	DQN-$\beta.1$	30.00	36.66	0%
	DQN-$\beta.2$	8.59	-	100%
	PF	27.44	36.54	20%
	DQN	30.00	36.49	0%
Env. 10	DQN-$\beta.1$	30.00	37.75	0%
	DQN-$\beta.2$	30.00	36.19	0%
	PF	30.00	35.64	0%
	DQN	30.00	36.01	0%

In terms of averaged navigation time over five trials, the proposed DQN either outperforms the others or yields a similar performance to them in all the environments. Especially when compared with its closest competitor PF, the proposed DQN's performance is superior on navigation time. It can navigate to the goal around 20% faster when the obstacles are densely located (Envs. 3, 4, 5, 6, 7), which naturally decreases the velocities generated by PF. DQN-$\beta.1$ yields slightly slower behaviour than the proposed DQN since it is deprived of the moving setpoint's relative position information as its incentive to move forward. DQN-$\beta.2$ has the incentive, but it probably does not have as much grasp as the proposed DQN over the environments due to the lack of sequential depth images.

In terms of averaged crash rate over five trials, the proposed DQN either outperforms the others or yields a similar performance to them in 8 of 10 environments. Over the 50 flights (5 trials in 10 environments), the number of safe flights is 48

[1]We consider tracks with the same straight length in order to have consistent benchmarking in this work. However, the proposed method is flexible to be employed in different environments with a variety of lengths of straight lines, a concatenation of straight lines, or turns. Once the intermediate goal points are put to create straight path segments as initial rough paths, it is trivial to utilize the method in these environments since the RL system is designed with respect to the quadrotor's body frame.

[2]We tune the respective hyperparameters of PF by trial-and-error for the fastest possible navigation in test environments.

for the proposed DQN, 46 for PF, 36 for DQN-β.2, and 35 for DQN-β.1. Even these results alone reveal the superiority of the proposed DQN to its preliminary versions DQN-β.1 and DQN-β.2. Thus, they justify the proposed state definition consisting of depth images and relative position information. Moreover, a crash rate slightly better than the renowned PF reveals the reliability of the proposed DQN.

C. Testing in Real Flights

Following the extensive comparative testing in AirSim, we deploy the proposed DQN directly on a DJI F330 Quadrotor (Fig. 7) in our lab. Our vehicle is equipped with the flight controller PX4 FMU, the companion computer Nvidia Jetson TX2, and the extrecoptive sensor Intel RealSense D435. The planning codes are running on TX2 utilizing its GPU for DQN in PyTorch with Cuda option. D435 provides live depth images as an input to DQN while PX4 is responsible for the low-level control of the vehicle. Odometry information is obtained using motion capture system. All modules communicate over Robot Operating System (ROS).

We consider three different environment configurations (Fig. 8): one as obstacle-free, one with a short but wide obstacle to test up-down slalom performance, and the other one with a narrow but long obstacle to test left-right slalom performance. Since our lab space is limited to roughly 3 m by 3 m area, we set the initial rough path of 4 m diagonally. We consider the approximate speed of the vehicle as 0.5 m/s in order not to cause dangerous movements of the quadrotor. Accordingly, we decrease the length of motion primitives to a maximum movement of 0.5 m in each axis. The rest of our RL system parameters is exactly the same as in AirSim.

We conduct five trials in each environment for real flights as well to demonstrate reliability. The proposed DQN completes 15 out of 15 flights without a crash. It navigates to the goal point in all the three environments while avoiding the obstacles successfully. A remark in real flights is that the average navigation time is higher as compared to AirSim trials. This is mostly due to the control ability of our quadrotor. We simply use PX4's position controller with default parameters which are quite sensitive to the power available in the battery. As compared to the precise execution of high-level commands in AirSim without considering any motor-ESC-propeller or battery efficiency issues, real flight control ability is obviously less. Still, the proposed DQN yields adequate end-to-end reasoning through relatively noisy depth images from D435 and completes the task in a slower manner but with 100% accuracy.

From the real-time applicability aspect, the proposed DQN requires the highest computation time on TX2 as can be seen in Table III. This computation demand mostly results from the presence of three depth images because it increases the number of connections in the DQN drastically. This can be seen when the computation time for DQN is compared with the one for DQN-β.2 which uses a single depth image. However, this result is obtained without any code or network optimization, and it is still close to the PF's computation time. Further development can bring this number down easily and allow for deployment with low-cost hardware.

TABLE II
AVERAGED TEST RESULTS IN REAL FLIGHTS.

Env.#	Navigation distance (m)	Navigation time (s)	Crash rate
Env.1	4.00	22.60	0%
Env.2	4.00	19.49	0%
Env.3	4.00	21.14	0%

TABLE III
COMPUTATION TIME ANALYSIS ON TX2.

Algorithm	Computation time (s)
DQN-β.1	0.0340
DQN-β.2	0.0161
PF	0.0304
DQN	0.0368

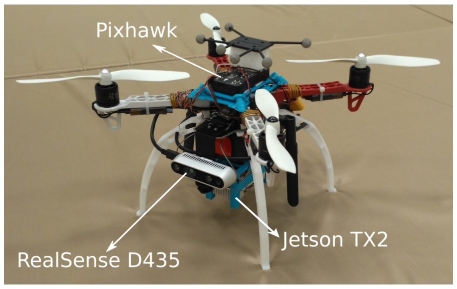

Fig. 7. DJI F330 Quadrotor with Pixhawk as flight controller, Jetson TX2 as companion computer, and RealSense D435 as exteroceptive sensor.

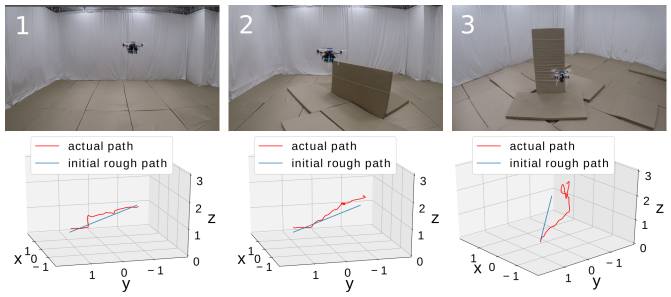

Fig. 8. Real environments with corresponding trajectory results for the trials with the shortest navigation duration.

V. CONCLUSION AND FUTURE WORK

A. Conclusion

In this work, we have developed a novel, end-to-end motion planner based upon deep RL. The proposed method has proven decent generalization capability by performing in a variety of environments which are unseen during training. It has achieved successful flights in 63 of 65 trials in these previously unseen environments. While 50 of these flights were in simulations, 15 of them was in real flights, which proved sufficient sim-to-real knowledge transfer as well. We have also conducted an extensive comparison with three competitors. The proposed method has yielded substantially better performance as compared to its two preliminary versions in all the three metrics: navigation time, navigation distance, and crash rate. It has yielded similar performance to PF-based planner regarding the last two metrics but superior performance regarding navigation time.

B. Future Work

Looking forward, there are a few aspects that can be studied as an extension of this work. Firstly, the current planning frequency of 1 Hz could be increased to possibly operate in much denser areas. Secondly, one can try to create a larger network which takes more depth images possibly from multiple depth sensors around the vehicle to increase the situational awareness. Lastly, the action set can be enhanced to have more freedom during planning. We are particularly interested in the last branch because smoother and safer motions can be achieved in this way.

Acknowledgment

We would like to acknowledge the Air Lab members Rogerio Bonatti, Wenshan Wang, and Sebastian Scherer at Carnegie Mellon University, PA, USA for substantially helpful discussions.

References

[1] A. S. Huang, A. Bachrach, P. Henry, M. Krainin, D. Maturana, D. Fox, and N. Roy, “Visual odometry and mapping for autonomous flight using an rgb-d camera,” in *Robotics Research: The 15th International Symposium ISRR*, vol. 100. Springer, 2016, p. 235.

[2] K. Sun, K. Mohta, B. Pfrommer, M. Watterson, S. Liu, Y. Mulgaonkar, C. J. Taylor, and V. Kumar, “Robust stereo visual inertial odometry for fast autonomous flight,” *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 965–972, 2018.

[3] J. Zhao, C. Hu, C. Zhang, Z. Wang, and S. Yue, “A bio-inspired collision detector for small quadcopter,” in *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2018, pp. 1–7.

[4] E. Camci and E. Kayacan, “Planning swift maneuvers of quadcopter using motion primitives explored by reinforcement learning,” in *2019 American Control Conference (ACC)*. IEEE, 2019, pp. 279–285.

[5] H. Oleynikova, Z. Taylor, R. Siegwart, and J. Nieto, “Safe local exploration for replanning in cluttered unknown environments for microaerial vehicles,” *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1474–1481, 2018.

[6] E. Camci and E. Kayacan, “Learning motion primitives for planning swift maneuvers of quadrotor,” *Autonomous Robots*, vol. 43, no. 7, pp. 1733–1745, 2019.

[7] D. Mellinger and V. Kumar, “Minimum snap trajectory generation and control for quadrotors,” in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 2520–2525.

[8] M. Mehndiratta, E. Camci, and E. Kayacan, “Automated tuning of nonlinear model predictive controller by reinforcement learning,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 3016–3021.

[9] E. Tal and S. Karaman, “Accurate tracking of aggressive quadrotor trajectories using incremental nonlinear dynamic inversion and differential flatness,” in *2018 IEEE Conference on Decision and Control (CDC)*. IEEE, 2018, pp. 4282–4288.

[10] J. Wang, Q. Zhang, D. Zhao, and Y. Chen, “Lane change decision-making through deep reinforcement learning with rule-based constraints,” in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–6.

[11] E. Camci and E. Kayacan, “End-to-end motion planning of quadrotors using deep reinforcement learning,” *arXiv preprint arXiv:1909.13599*, 2019.

[12] M. Gschwindt, E. Camci, R. Bonatti, W. Wang, E. Kayacan, and S. Scherer, “Can a Robot Become a Movie Director? Learning Artistic Principles for Aerial Cinematography,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 1107–1114.

[13] H. Oleynikova, D. Honegger, and M. Pollefeys, “Reactive avoidance using embedded stereo vision for mav flight,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 50–56.

[14] B. T. Lopez and J. P. How, “Aggressive 3-d collision avoidance for high-speed navigation,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 5759–5765.

[15] S. Stevšić, T. Nägeli, J. Alonso-Mora, and O. Hilliges, “Sample efficient learning of path following and obstacle avoidance behavior for quadrotors,” *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3852–3859, 2018.

[16] B. Penin, P. R. Giordano, and F. Chaumette, “Vision-based reactive planning for aggressive target tracking while avoiding collisions and occlusions,” *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3725–3732, 2018.

[17] F. Sadeghi and S. Levine, “Cad2rl: Real single-image flight without a single real image,” *arXiv preprint arXiv:1611.04201*, 2016.

[18] V. Blukis, N. Brukhim, A. Bennett, R. A. Knepper, and Y. Artzi, “Following high-level navigation instructions on a simulated quadcopter with imitation learning,” *arXiv preprint arXiv:1806.00047*, 2018.

[19] S. Jung, S. Hwang, H. Shin, and D. H. Shim, “Perception, guidance, and navigation for indoor autonomous drone racing using deep learning,” *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2539–2544, 2018.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, p. 529, 2015.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[22] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[23] P. J. Huber *et al.*, “Robust estimation of a location parameter,” *The annals of mathematical statistics*, vol. 35, no. 1, pp. 73–101, 1964.

[24] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “Airsim: High-fidelity visual and physical simulation for autonomous vehicles,” in *Field and service robotics*. Springer, 2018, pp. 621–635.

[25] O. Khatib, “Real-time obstacle avoidance for manipulators and mobile robots,” in *Autonomous robot vehicles*. Springer, 1986, pp. 396–404.