Kinodynamic Motion Planning with Neural Networks

Josef Michelsen
Ann and H.J. Smead Aerospace Engineering Sciences
University of Colorado at Boulder
Boulder, Colorado
jomi7243@colorado.edu

Nicolas Perrault
Ann and H.J. Smead Aerospace Engineering Sciences
University of Colorado at Boulder
Boulder, Colorado
nipe1783@colorado.edu

***Abstract*—Unmanned Aerial Vehicles (UAVs) are increasingly utilized due to their potential to automate tasks, reduce operational costs, and minimize human risk. Many real-world UAV applications require navigating cluttered environments, such as urban areas, indoor spaces, or regions with infrastructure like power lines. These scenarios demand motion planning algorithms that account for both kinematic and dynamic constraints. This paper presents a Deep Q-Network (DQN) based approach to kinodynamic motion planning for a UAV modeled as a six-dimensional double integrator. The DQN agent outputs acceleration commands using an epsilon-greedy action selection policy and is trained with a prioritized experience replay buffer. The trained policy is evaluated by deploying it in previously unseen environments to assess its ability to guide the UAV to a goal while avoiding obstacles. Results demonstrate that the learned policy generalizes well, successfully navigating to the goal from most initial positions in an environment without collisions.**

***Index Terms*—Component; UAV; Kinodynamic Motion Planning; DQN; Deep Reinforcement Learning, Priority Buffer**

I. Introduction

Reinforcement Learning (RL) has emerged as a powerful framework for enabling autonomous agents to solve complex control tasks through interaction with their environment. In this work, we explore the application of DQN to the problem of kinodynamic motion planning which is a challenging variant of path planning where an agent must account not only for spatial constraints but also for system dynamics such as inertia and momentum.

We present the implementation of a DQN-based controller for kinodynamic navigation of a simplified UAV operating in continuous two and three dimensional environments populated with static obstacles. The agent is trained to navigate from a starting state to a goal region while avoiding collisions, subject to the vehicle's dynamic constraints.

To model the vehicle's motion, a double integrator was used to integrate acceleration to velocity and then integrate velocity to a change in position. At each time step, the agent takes in information about the environment and selects an acceleration action. This formulation captures the dynamic behavior of the system, requiring the agent to reason about momentum when making decisions.

DQN was selected due to its ability to work in a continuous state space and approximate the expected cumulative reward function (Q-function). This implementation, developed primarily from scratch, includes essential enhancements such as a prioritized buffer replay, mini-batch training, and linearly decreasing ϵ for an ϵ-greedy action picking strategy. The trained agents were evaluated for robustness and their ability to generalize to unseen environments by benchmarking performance across diverse obstacle-rich environments.

This paper details the system architecture, training methodology, and experimental results, demonstrating the potential of deep reinforcement learning for kinodynamic planning, as well as the limitations and challenges that arise in practice.

II. Related Work

Motion planning is a fundamental problem in robotics, and as such, extensive research has been dedicated to developing algorithms that enable efficient and reliable planning. In parallel, RL has gained significant traction in recent years, driven by algorithmic advances and the increasing availability of computational resources capable of training complex agents.

DQN were first introduced by Mnih et al. in [4], where the algorithm demonstrated the ability to learn control policies directly from raw pixel inputs. In their seminal work, DQN achieved human-level or superhuman performance on several Atari 2600 games using only screen images and game scores as inputs. The approach combined Q-learning with deep convolutional networks and introduced key innovations such as experience replay and a target network to stabilize training. This work marked a significant breakthrough in the field of deep reinforcement learning and laid the groundwork for applying RL to complex, high-dimensional control tasks.

In the context of kinodynamic motion planning, many traditional approaches do not rely on neural networks. Some strategies focus on combining geometric and optimization-based planning with control strategies to produce feasible trajectories. For example, in [1], the authors present a traditional motion planning stack for quadrotors that includes: a geometric planner based on sampling to generate a collision-free path, an offline nonlinear trajectory optimization step to produce dynamically feasible trajectories, and a quadratic programming (QP)-based force allocation controller that operates online to track the optimized trajectory. This approach reflects the strengths of modular pipelines that explicitly handle dynamic feasibility and control allocation but can suffer from high

computational demands and limited generalization to novel environments. Others, such as [2] use more standard sampling based techniques like Rapidly-exploring Random Trees (RRT).

For systems with significant nonlinear dynamics, classical motion planning methods may struggle to produce feasible or efficient trajectories. In such cases, the ability of neural networks to approximate complex, nonlinear Q-functions makes them attractive tools for learning-based motion planning, particularly when generalizing across diverse environments.

As discussed in [3], DQN has also been applied to UAV path planning in unknown environments. However, that work does not explicitly account for vehicle dynamics during training, focusing instead on planning feasible paths through cluttered environments. Despite this simplification, DQN was shown to reliably find optimal or near-optimal trajectories to target regions while avoiding obstacles, highlighting its potential as a motion planning framework.

DQN and its variants have also been applied to autonomous UAV navigation using convolutional neural networks to process onboard camera imagery for obstacle avoidance, as demonstrated in [5]. This represents a realistic deployment scenario for DQN-based agents in civilian and military applications such as search and rescue, reconnaissance, and aerial surveying. As deep reinforcement learning agents become more robust and data-efficient, they hold promise for expanding the operational capabilities of UAVs and other autonomous platforms, motivating continued research in this area.

III. Problem

A. Kinodynamic Motion Planning

Consider an agent operating within a bounded workspace $W \subset \mathbb{R}^d$, where $d \in \{2, 3\}$. This workspace contains a finite set of obstacles $\mathcal{O}$, where each obstacle $o \in \mathcal{O}$ is a closed subset of W, i.e., $o \subset W$. The dynamics of the agent's motion is given by

$$\dot{x}(t) = f(x(t), u(t)), \tag{1}$$

where $x(t) \in X \subset \mathbb{R}^n$ and $u(t) \in U \subset \mathbb{R}^N$ are the agent's state and control at time t, respectively, and $f : X \times U \to \mathbb{R}^n$ is the vector field.

In addition to motion constraints defined by the dynamics in (1) and obstacles in $\mathcal{O}$, we consider state constraints, e.g., bound on the velocity. To this end, we define the set of valid states, i.e., states at which the agent does not violate its state constraints and does not collide with an obstacle, as the valid set and denote it by $X_{\text{valid}} \subseteq X$. Then, given initial state $x_{\text{init}} \in X_{\text{valid}}$, time duration $t_f \geq 0$, and control trajectory $\mathbf{u} : [0, t_f] \to U$, a *state trajectory* $\pi : [0, t_f] \to X$ is induced, where

$$\pi(t) = x_{\text{init}} + \int_0^t f(\pi(\tau), \mathbf{u}(\tau))d\tau \qquad \forall t \in [0, t_f]. \tag{2}$$

Trajectory π is called *valid* if, $\forall t \in [0, t_f]$, $\pi(t) \in X_{\text{valid}}$.

In motion planning, the interest is to find a valid trajectory π that visits a given goal set $X_{\text{goal}} \subseteq X_{\text{valid}}$. Therefore, by following this trajectory, the agent is able to respect all of its motion constraints, avoid collisions with obstacles, and reach its goal. In this work, we approach kinodynamic motion planning through the framework of Markov Decision Processes (MDPs), providing a structured method to formulate and solve these complex planning problems.

To systematically analyze the kinodynamic motion planning problem within an MDP framework, we consider four progressively challenging scenarios.

B. Scenario 1

Scenario 1 assumes an obstacle-free workspace, i.e., $|\mathcal{O}| = 0$. Additionally, we assume the initial state x_{init} and goal region X_{goal} are static and remain unchanged across planning episodes. Thus, in each planning instance, the agent starts from the same state $x_{\text{init}} \in X_{\text{valid}}$ and seeks to reach the same goal set X_{goal}. Furthermore, we assume the agent selects control inputs from a discrete, finite subset of the control space U, denoted by $\bar{U} \subset U$.

With these assumptions, we formulate this scenario as an MDP, denoted by the tuple (S, A, T, R), defined as follows:

- **State Space (S):** Defined directly by the agent's state space, $S = X$.
- **Action Space (A):** Defined as a finite discrete subset of available controls, $A = \bar{U}$.
- **Transition Function (T):** Transitions between states are deterministic and defined by equation (2).
- **Reward Function (R):** Given a transition from state $s \in S$ to state $s' \in S$ by action $a \in A$, the reward function is defined as:

$$R(s, a, s') = \begin{cases} +1000 \text{ if } s' \in X_{\text{goal}}, \\ -1000 \text{ if } s' \notin X_{\text{valid}}, \\ -\text{dist}(s', \text{centroid}(X_{\text{goal}})) \text{ o/w.} \end{cases} \tag{3}$$

Here, the reward function provides a large positive reward for reaching the goal region X_{goal}, a large penalty for violating state constraints or leaving the valid region X_{valid}, and otherwise a penalty proportional to the agent's distance from the centroid of X_{goal}.

C. Scenario 2

Scenario 2 expands upon Scenario 1 by relaxing the assumption that the initial state x_{init} and goal region X_{goal} remain fixed across planning episodes. Specifically, this scenario considers the agent starting from any valid initial state $x_{\text{init}} \in X_{\text{valid}}$ and attempting to reach any goal region $X_{\text{goal}} \subseteq X_{\text{valid}}$.

To address this generalized problem, we incorporate goal information into the MDP formulation by augmenting the state representation. Thus, we redefine the **State Space (S)** as follows:

$$S = \{(x, x_{\text{goal}}) \mid x \in X_{\text{valid}}, x_{\text{goal}} \in X_{\text{goal}}\}. \tag{4}$$

In this formulation, each state $s \in S$ encodes both the agent's current state and the goal state's information, enabling the learned policy to generalize across different initial-goal state pairs within the workspace. The action space, transition

function, and reward function remain unchanged from Scenario 1.

D. Scenario 3

Scenario 3 extends Scenario 2 by introducing obstacles into the workspace, i.e., $|\mathcal{O}| > 0$. In this scenario, we assume the obstacle configuration remains fixed across all planning episodes; formally, this means $\mathcal{O}_i = \mathcal{O}_j$ for any two planning instances i, j. Consequently, the definition of the valid state space X_{valid} now explicitly excludes states in collision with obstacles.

Since the obstacles are static across episodes, the MDP formulation established in Scenario 2 does not require modification. Instead, the state validity checks account for obstacle constraints when determining if an agent's state $x \in X_{\text{valid}}$.

E. Scenario 4

Scenario 4 further generalizes Scenario 3 by relaxing the assumption that obstacle configurations remain static across planning episodes. Specifically, obstacle sets now vary between episodes; formally, $\mathcal{O}_i \neq \mathcal{O}_j$ for planning instances $i \neq j$. However, we maintain the condition that the number of obstacles remains constant, i.e., $|\mathcal{O}_i| = |\mathcal{O}_j|$ for all planning episodes.

To account for this increased generality, obstacle information must be explicitly incorporated into the MDP's state representation. Consequently, we redefine the **State Space** (S) as:

$$\begin{aligned} S = \{(x, x_{\text{goal}}, o_1, o_2, \ldots, o_{|\mathcal{O}|}) \\ \mid x \in X_{\text{valid}},\ x_{\text{goal}} \in X_{\text{goal}},\ o_k \in \mathcal{O}\}, \end{aligned} \tag{5}$$

where each obstacle o_k encodes its geometric information. By embedding obstacle information directly into the state space, the learned policy is capable of generalizing across varied obstacle configurations.

IV. Solution

To address the kinodynamic motion planning problem using reinforcement learning a complete training and evaluation pipeline based on the DQN algorithm was implemented. This section outlines the key components of our solution, including dynamics modeling, DQN implementation, neural network architecture, prioritized experience replay, and benchmark testing.

First, a six-dimensional double integrator (6DDI) model was implemented from scratch to simulate the motion of a simplified UAV. This model takes in acceleration commands from the NN and then integrates those to capture translational position and velocity in three-dimensional space. By integrating from acceleration the simulation provides a realistic dynamic system that requires the agent to account for inertia and momentum during path planning. Notably this does not account for the rotational dynamics that affect real UAV systems, these were omitted because the nonlinear relationships between NN output and vehicle behavior would increase in complexity reducing the chances of training a successful agent.

The DQN algorithm was implemented from scratch, excluding the neural network components, which were constructed using the PyTorch deep learning library. This implementation includes a standard epsilon-greedy exploration strategy, where the exploration rate ϵ was linearly decayed over time to balance exploration and exploitation throughout training. The reward function used during training was defined in equation 3 and was designed to encourage goal-reaching behavior while penalizing collisions and inefficient movement. This drives the agent toward the goal while trying to reduce the number of steps it takes because increased steps add penalty. By incentivizing taking a lower number of steps this should help the agent favor motion planning paths that are near-optimal.

Training the agent utilizes mini batch training. Episodes of the agent interacting with the environment are constantly running, meaning each time the agent reaches a terminal state the environment is reset. Once the buffer adds 1024 experiences the NN is updated by the experiences chosen with the priority buffer implementation. mini batch training was implemented to try and smooth the training process by averaging many experiences before back propagating the resulting gradient through the NN.

A. Deep Q-Network Equations

The DQN algorithm combines traditional Q-learning with a neural networks to approximate the Q-function. The key equations used in our implementation are outlined below.

1) Bellman Equation for Q-Learning: At the core of Q-learning is the Bellman optimality equation, which defines the relationship between the Q-value of a state-action pair and the expected return:

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right], \tag{6}$$

where:

- s and s' are the current and next states,
- a and a' are the current and next actions,
- r is the reward received after taking action a in state s,
- $\gamma \in [0, 1)$ is the discount factor.

The discount factor is used to determine how much the agent should prioritize immediate vs future rewards, a value near 1 prioritizes future rewards the most while values near 0 prioritize immediate rewards. This implementation utilized a discount factor of 0.9 because it prioritized long term goals over immediate rewards while still allowing immediate rewards to be accounted for. Immediate rewards are important because with a reward function based on the agents distance to the goal increasing immediate rewards means the UAV is getting closer to the goal. However, long term rewards are more important because it enables the agent to avoid obstacles which might be between it and the goal.

2) Q-Value Update Target: In DQN, the Q-function is approximated using a neural network $Q(s, a; \theta)$ with parameters θ. A separate target network with parameters θ^- is used to compute a stable target value.

$$Q_{\text{target}} = Q(s, a; \theta). \tag{7}$$

This target is treated as the ground truth during training, while the main network learns to minimize the temporal-difference (TD) error between its Q-value predictions and this target. The target is reset to be equal to the main network every 30,000 episodes to ensure the target network is reflecting the best approximation of the Q-function while maintaining it stable enough to train the main network.

$$y = r + \gamma * \max_{a'} Q_{\text{target}}(s', a'; \theta^-). \tag{8}$$

The target value which is used in the loss calculation is based on the action which maximizes the reward from the target network.

3) TD Loss Function: The loss function minimized during training is the mean squared error between the predicted and target Q-values. In the case of prioritized experience replay, the TD loss is weighted by importance-sampling weights w_i that come from that experiences TD:

$$\mathcal{L}(\theta) = \mathbb{E}_i \left[w_i \left(Q(s_i, a_i; \theta) - y_i \right)^2 \right]. \tag{9}$$

4) TD Error for Prioritization: The TD error, used for both learning and updating priorities in the replay buffer, is given by:

$$\delta_i = Q(s_i, a_i; \theta) - y_i. \tag{10}$$

To enhance sample efficiency and learning stability a prioritized experience replay buffer was implemented. This buffer used priority-based sampling where the probability of selecting a transition was proportional to its temporal-difference (TD) error. The priority buffer implementation relies on this measurement to pick which experiences should be used for training from the buffer. The larger the TD error the more "surprising" a measurement is meaning that experience will have a larger gradient and cause a larger update to the weights in the NN. By prioritizing training the NN with experiences with the largest TD the agent should be able to learn faster than if a random sampling from the buffer was used for training.

5) Sampling Probability: The priority p_i of each experience is the absolute value of its TD. The probability of sampling a transition i from the buffer is then computed as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{11}$$

The priorities are normalized in order to convert them to probabilities which enables easier selection of experiences. $\alpha \in [0, 1]$ determines how much prioritization is used: $\alpha = 0$ corresponds to uniform random sampling, and higher values increase prioritization. This implementation used $\alpha = 0.6$ as a middle ground to ensure high priority states were chosen while trying to prevent the NN from biasing too much from those experiences.

6) Importance Sampling Weights: To compensate for the non-uniform probability distribution during training, importance sampling (IS) weights are introduced to correct for bias:

$$w_i = \left(\frac{1}{N \cdot P(i)} \right)^\beta, \tag{12}$$

where N is the buffer size and $\beta \in [0, 1]$ controls how much compensation is applied. This implementation used $\beta = 0.4$ which allowed the loss to be adjusted by the IS weights slightly prioritizing experiences with higher TD values regardless of the sampling bias. Similar to the α value decision this was done in order to make sure training the NN prioritized high TD experiences, specifically those which caused a collision with obstacles, while not biasing the NN such that it reduced its robustness. The IS weights are also normalized so that $\max_i w_i = 1$, ensuring training remains stable.

7) Epsilon-Greedy Exploration: During training, the agent selects actions using an epsilon-greedy strategy:

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q(s, a; \theta), & \text{with probability } 1 - \epsilon \end{cases}, \tag{13}$$

The ϵ is then linearly decreased from a value of 1.0 to 0.05 over 90% of that training rounds episodes. This allows the agent to start off by making completely random actions in order to best explore the environment and as the Q-function approximation of the environment improves ϵ decreases and the agent begins to prioritize exploitation over exploration to make it to the goal more and more often.

The neural network approximating the Q-function was built using PyTorch and consisted of four hidden layers, each with 512 neurons, using the ReLU activation function after each layer. The optimizer chosen was Adam with a learning rate of 0.0005. The NN is trained in the standard manner, where gradients are computed from the TD loss and the NN weights are then updated via backpropagation.

The network input depended on the environment configuration. For environments with no obstacles and a static goal, the inputs included only the UAVs position and velocity. For more complex scenarios involving obstacles and varying goal locations the inputs were extended to include the UAVs relative position to the goal and obstacles giving the NN the information needed to avoid the obstacles and make it to the changing goal region.

Finally, a benchmark testing framework was developed to evaluate the generalization capability of trained policies. This framework systematically tested trained agents across a suite of representative environments, including variations not seen during training, to assess robustness and adaptability of the learned control strategies.

V. Results

We demonstrate the performance of *DQN* in learning sound policies across the four scenarios described in Section III. The underlying dynamical system is a 6D double integrator, with

the state vector defined as $x = (x, y, z, \dot{x}, \dot{y}, \dot{z})$ and control inputs $u = (\ddot{x}, \ddot{y}, \ddot{z})$. The system evolves according to the second-order differential equations:

$$\begin{aligned} \dot{x} &= v_x, & \dot{v}_x &= u_x, \\ \dot{y} &= v_y, & \dot{v}_y &= u_y, \\ \dot{z} &= v_z, & \dot{v}_z &= u_z, \end{aligned} \tag{14}$$

where v_x, v_y, and v_z denote velocities along each axis, and u_x, u_y, u_z are the corresponding control accelerations. In all scenarios, we restrict the action space to six discrete actions, corresponding to unit acceleration in each cardinal direction of the 3D workspace.

A. Scenario 1

For Scenario 1, we evaluated our *DQN* implementation over 23 million training episodes. Figure 1 illustrates the agent's reward per episode. The reward steadily increases as training progresses, and by approximately 15 million episodes, the agent achieves a reward close to 900, indicating convergence toward a near-optimal policy. However, Figure 1 also reveals sporadic negative episode rewards throughout the training period. These negative rewards arise because each training episode begins from the identical initial state x_{init}, causing certain regions of the state space to remain underexplored, and thus occasionally resulting in poor action choices.

The learned policy is visualized in Figure 2, showing the agent following a nearly direct path from the fixed initial state x_{init} toward the goal region X_{goal}. To assess the generalization capability and identify underexplored regions, we benchmarked the trained neural network. Specifically, we conducted 5,000 simulations, each starting the agent at a random workspace location with zero initial velocity. In each simulation, the agent selected actions according to the learned policy to generate trajectories toward the goal region. The neural network achieved a success rate of 86%, where success is defined as reaching the goal region without collisions or constraint violations. These benchmarking results are visualized in Figure 3, highlighting successful and unsuccessful starting states, identifying areas within the state space that require further exploration.

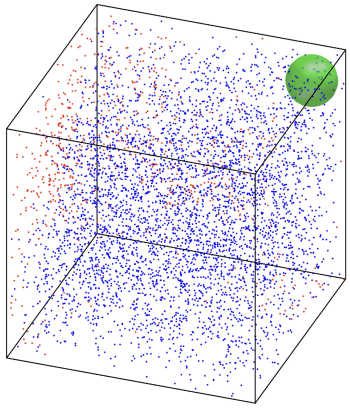


Fig. 3: Benchmark results for Scenario 1. Across 5,000 planning instances, x_{init} was initialized to a random location within the workspace. Blue datapoints represent successful policies, while red datapoints represent failed policies.

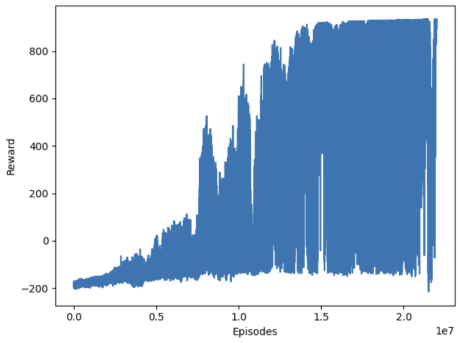


Fig. 1: Simulation reward as a function of episode number in an open, bounded environment with fixed starting state and goal region.

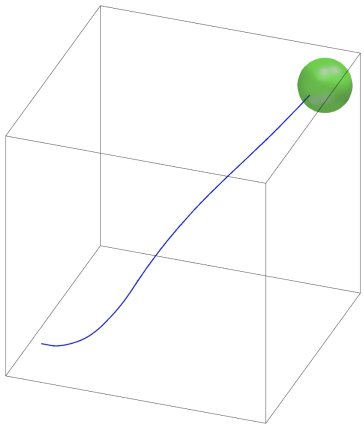


Fig. 2: Trajectory produced by the *DQN* policy for Scenario 1. The blue line represents the trajectory, the green sphere denotes the goal region, and the cube represents the workspace bounds.

B. Scenario 2

For Scenario 2, we encoded the centroid of the goal region into the neural network input, increasing the input dimension from 6 to 9. We trained the DQN for 25 million episodes, resulting in the reward-versus-episode plot shown in Figure 4. This figure illustrates a steady increase in reward without the sporadic negative rewards observed in Scenario 1, due to each episode now beginning from a random initial state $x_{\text{init}} \in X_{\text{valid}}$, thereby ensuring broader exploration of the state space.

Several example trajectories generated by the trained neural network are visualized in Figure 6. Each trajectory demonstrates the agent's performance when initialized from a random state and directed toward a randomly chosen goal region. To directly compare Scenario 2 with Scenario 1, we conducted the same benchmark simulation as before: 5,000 episodes where the agent started from random workspace locations and targeted a fixed-goal region. Under these conditions, the Scenario 2 DQN achieved a success rate of 91%, an improvement over the 86% success rate observed in Scenario 1. The results of this benchmark are visualized in Figure 5. An interesting observation from Figure 5 is that many failed initial positions are concentrated in a single region of the workspace, rather than being distributed around each corner as one might expect.

Additionally, we evaluated the model's generalization capability by performing another benchmark of 5,000 simulations, in which both initial position and goal regions were randomly selected. In this setting, the model successfully reached the goal region in 97% of simulations, a 6% improvement over the case with a fixed goal position. This result is expected because the fixed goal lies in a corner of the workspace, making it more difficult to reach than a typical randomly sampled goal.

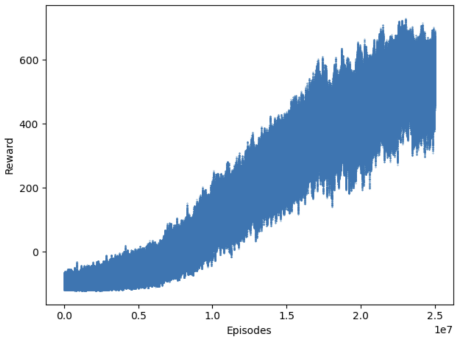


Fig. 4: Simulation reward as a function of episode number in an open, bounded environment with random starting state and goal region.

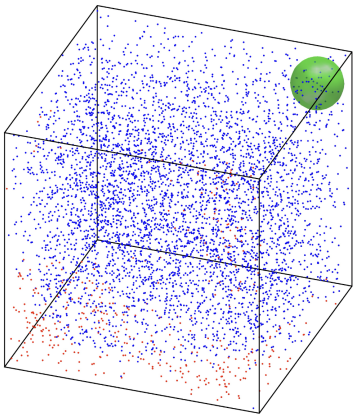
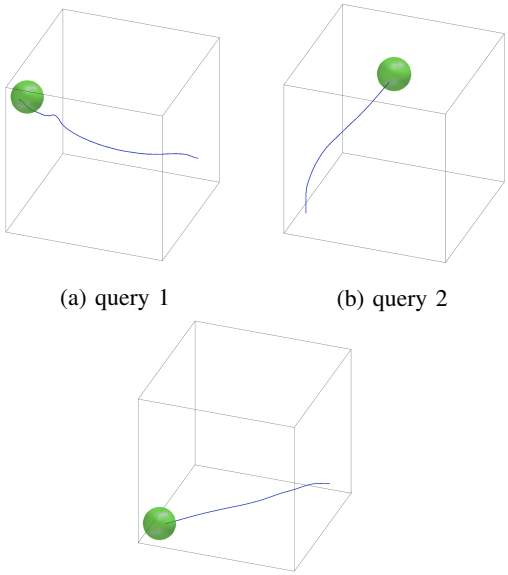


Fig. 5: Benchmark results for Scenario 2. Across 5,000 planning instances, x_{init} was initialized to a random location within the workspace. Blue datapoints represent successful policies, while red datapoints represent failed policies.



(a) query 1 (b) query 2

(c) query 3

Fig. 6: Three planning episodes using the trained neural network for Scenario 2. Each planning query shows a random initial starting state with a random goal region and the agent succefully navigating to that region.

C. Scenario 3

For Scenario 3, the neural network again received a 9-dimensional input. However, in this case, the workspace included four static obstacles that remained fixed in position throughout all training episodes. These obstacles, along with an example of a successful trajectory, are visualized in Figure 7.

Since generating sound policies in cluttered environments is significantly more challenging than in the obstacle-free settings of Scenarios 1 and 2, the number of training episodes for this *DQN* scenario was increased to 30 million. The corresponding reward-versus-episode plot is shown in Figure 8. As shown in Figure 8, the agent achieves a mean reward of approximately 600, which is similar to that observed in Scenario 2, indicating that the state space has been sufficiently explored.

In evaluation across 5,000 trials with randomized agent start states and goal regions, the trained policy achieved a 91% success rate in reaching the goal. Although this performance is relatively high, failures frequently occurred when the agent was required to navigate around obstacles. This is evident in Figure 5, where many failed planning instances (denoted by red datapoints) are located behind or near obstacles. To improve performance, we modified the training setup so that the agent only started from workspace locations that required obstacle avoidance. We initialized this new training with the neural network weights obtained from the original model. However, this approach did not produce improvements, and the network failed to make further progress over an additional 20 million training episodes.

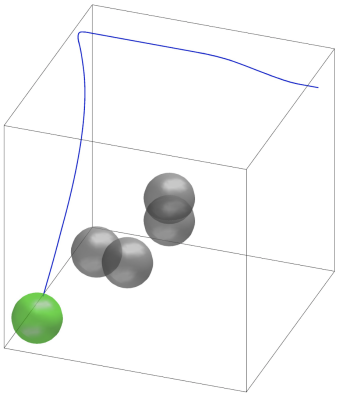


Fig. 7: Trajectory produced by the *DQN* policy for Scenario 3.

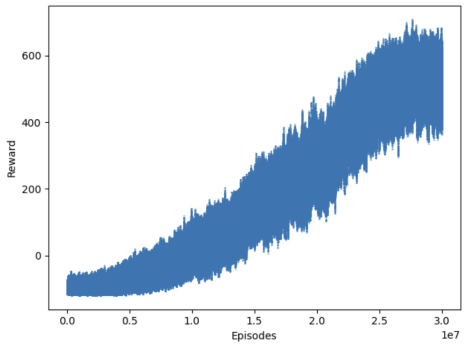
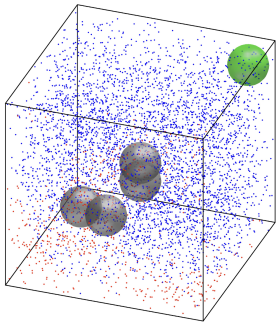
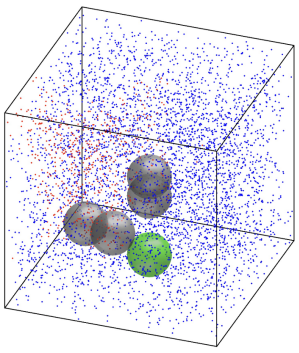


Fig. 8: Simulation reward as a function of episode number in a bounded environment with static obstacles, random initial states, and random goal regions.



(a) query 1



(b) query 2

Fig. 9: Two planning scenarios using the *DQN* in Scenario 3. Red data points indicate failed policies, and blue data points indicate successful policies.

D. Scenario 4

For Scenario 4, we considered an environment with a single obstacle whose position varied across planning episodes. To enable the *DQN* to generate sound policies, the obstacle's position was appended to the neural network input, increasing the input dimension from 9 (6 for the 6D double integrator and 3 for the goal position) to 12 (including 3 for the obstacle position). To accelerate training, we used our implementation of a prioritized experience replay buffer, which promotes the sampling of more informative transitions. The model was trained for 10 million episodes, and the reward as a function of episode number is shown in Figure 10. As seen in Figure 10, after 10 million episodes the agent achieves a mean reward of approximately 600, comparable to the performance of Scenario 3, which required 30 million episodes. This suggests that the prioritized replay buffer improves training efficiency relative to uniform sampling.

Figure 12 illustrates the network's ability to generate valid policies that navigate to randomly selected goal regions while avoiding randomly placed obstacles. We benchmarked this network over 5,000 episodes, with random initial positions, goal regions, and obstacle placements. The resulting success rate was 71%. Similar to Scenario 3, the model struggled in cases where obstacle avoidance was critical for reaching the goal. However, in some configurations of goal and obstacle positions, such as the one shown in Figure 11, the agent performed well, successfully avoiding the obstacle and reaching the goal.

VI. Conclusion

The main goal of this investigation was to train an agent using DQN which is capable of kinodynamic motion planning for a UAV in a cluttered environment. This goal was, mostly, achieved as shown in the benchmarks where the trained agents were deployed in previously unseen environments and succeeded, in most cases, to reach the goal while avoiding the obstacles. Achieving a 100% pass rate on the bench marks is critical for deploying an agent in most real world situation because it would give confidence to the organization deploying the UAV that it could safely and autonomously handle any situation where it would have to avoid an obstacle. However, there may be use cases where the deploying organization may not necessarily require complete coverage in these types of situations, such as the United States Coast Guard which could deploy an agent out at sea, where no obstacles exist at flying altitudes, which may not be able to avoid obstacles in all situations. These results show that DQN with priority buffering is capable of producing agents that can motion plan in unfamiliar environments while taking into account the kinodynamics of the vehicle. The success these agents have shown gives confidence that this approach, with some additional work, can be deployed in real world situations. To continue this investigation an agent could be trained for longer than what was permissible on the current timeline in order to try and get an agent that can complete the benchmarks with 100% passing rate. Additionally, agents should be trained against the Dubins airplane model, which was implemented but never trained on, to incorporate both translational and rotational dynamics in the simulation. The addition of rotation dynamics will bring the simulation even closer to modeling a real world system.

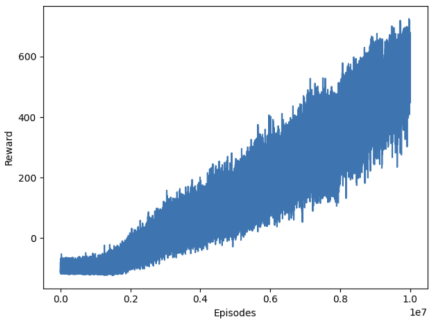


Fig. 10: Simulation reward as a function of episode number in a bounded environment with static obstacles, random initial states, and random goal regions.

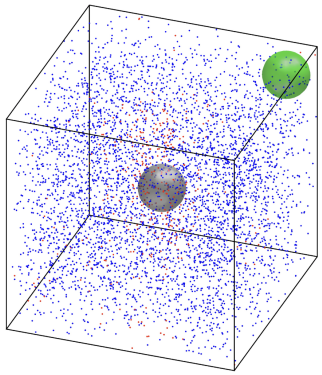
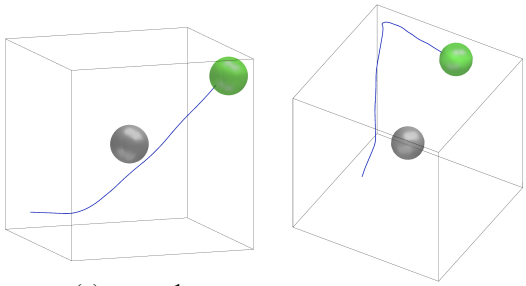


Fig. 11: Benchmark results for Scenario 4. Across 5,000 planning instances, x_{init} was initialized to a random location within the workspace. Blue datapoints represent successful policies, while red datapoints represent failed policies.



(a) query 1

(b) query 2

Fig. 12: Two planning episodes using the trained neural network for Scenario 4. Each query features a random initial state, a random goal region, and a randomly placed obstacle, with the agent successfully navigating to the goal.

VII. Contributions

Josef Michelsen contributions:

1) Writing the DQN implementation
2) Writing the priority buffer implementation
3) Writing the loss function implementation
4) Writing the epsilon greedy implementation
5) Writing the report

Nicolas Perrault contributions:

1) Writing the 6DDI
2) Writing the agent and workspace/environment implementation
3) Writing the agent testing code
4) Producing the plots used in this report
5) Writing the report

VIII. Release

The authors grant permission for this report to be posted publicly.

References

[1] Wahba, K., Ortiz-Haro, J., Toussaint, M., & Hönig, W. (2023). Kinodynamic Motion Planning for a Team of Multirotors Transporting a Cable-Suspended Payload in Cluttered Environments. arXiv preprint arXiv:2310.03394. https://arxiv.org/abs/2310.03394

[2] T. Lai, W. Zhi, T. Hermans and F. Ramos, "Neural Kinodynamic Planning: Learning for KinoDynamic Tree Expansion," 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Abu Dhabi, United Arab Emirates, 2024, pp. 11789-11795, doi: 10.1109/IROS58592.2024.10801948. keywords: Training;Measurement;Costs;System dynamics;Computational modeling;Neural networks;Euclidean distance;Kinematics;Predictive models;Planning,

[3] Guo, Yifan & Liu, Zhiping. (2024). UAV Path Planning Based on Deep Reinforcement Learning. International Journal of Advanced Network, Monitoring and Controls. 8. 81-88. 10.2478/ijanmc-2023-0068.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," NIPS Deep Learning Workshop, Lake Tahoe, NV, USA, Dec. 2013. [Online]. Available: https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf

[5] Wang, F., Wang, F., Zhu, X., Zhu, X., Zhou, Z., & Tang, Y. (2024). Deep-reinforcement-learning-based UAV autonomous navigation and collision avoidance in unknown environments. Chinese Journal of Aeronautics, 37(3), 237–257. https://doi.org/10.1016/j.cja.2023.09.033

[6] M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, Algorithms for Decision Making. Cambridge, MA, USA: The MIT Press, 2022. [Online]. Available: https://algorithmsbook.com/decisionmaking/