Mildly Conservative Q-Learning in Offline Reinforcement Learning for Grid World Navigation

Renhong Zhang, Ruixiang Wang, Udit Ekansh
Purdue University, West Lafayette

***Abstract*—Offline reinforcement learning (RL) enables policy training using pre-collected datasets, minimizing online exploration. This project evaluated offline RL algorithms for grid-world navigation, with a custom dataset generated using the A* path-planning algorithm on a 2D occupancy grid derived from a realistic environment map created in the Gazebo Simulation Environment using TurtleBot3. The Mildly Conservative Q-Learning (MCQ) algorithm and other methods, including Behavior Cloning (BC), Advantage-Weighted Regression (AWR), Batch-Constrained Q-Learning (BCQL), and Conservative Q-Learning (CQL), were implemented and compared. Results showed that MCQ and CQL failed to create policies that effectively guided the robot to goal, achieving 0% success rates despite hyperparameter tuning using Optuna. BC achieved a 79% success rate, which improved to 94% with tuning, while AWR reached 88%. Initial results with BCQL showed promise with a 52% success rate, but its hyperparameters could not be fully tuned due to time constraints.In addition, Twin Delayed Deep Deterministic Policy Gradient + Behavior Cloning (TD3+BC) was tested in the PointMaze environment, revealing significant reward degradation when faced with out-of-distribution (OOD) actions. These results highlight the challenges of offline RL in navigation tasks and emphasize the need for robust algorithms and effective hyperparameter tuning. Future work includes validating policies on TurtleBot3 in Gazebo and addressing out-of-distribution scenarios.**

***Index Terms*—Q-Learning, Behavior Cloning, Path Planning, 2D occupancy Grid**

I. Introduction

One of the key challenges in Offline Reiforcement Learning is addressing out-of-distribution (OOD) actions—actions that fall outside the distribution of the training dataset. Algorithms that naively maximize rewards in such settings risk poor generalization or catastrophic failures. To mitigate this, conservative algorithms such as Mildly Conservative Q-Learning (MCQ) and Conservative Q-Learning (CQL) have been proposed, which aim to penalize OOD actions while still learning effective policies. However, these methods remain challenging to implement and tune, particularly in grid-world navigation tasks where precise movement and path optimality are critical.

A. Motivation

The motivation of this project can be summarized below:

- **Bridging the gap between simulation and reality:** Numerous studies focus on simulated environments; however, there is a notable gap in the literature that demonstrates successful applications of offline RL on actual robotic platforms. Through this project, we aim to bridge this gap and explore the limitations of offline RL in practice.
- **Addressing real-world challenges:** Implementing offline learning methods like MCQ on physical robots presents several challenges, such as noisy sensor data, limited data availability, distribution shift, and computational limits. Addressing these challenges in a controlled, yet realistic, setting will establish a foundation for future developments in robot learning.

B. Objective

The primary objectives of this project were:

1) To reimplement the Mildly Conservative Q-Learning (MCQ) algorithm and compare its performance with other offline RL algorithms in a simplified grid-world derived from a Gazebo map.
2) To validate the offline learning results on TurtleBot3 within the Gazebo simulation environment.

MCQ was successfully implemented but did not perform as expected, potentially due to various factors that will be discussed in later sections. Alongside MCQ, we successfully trained and evaluated the performance of other offline RL algorithms, including Behavior Cloning (BC), Advantage-Weighted Regression (AWR), Batch-Constrained Q-Learning (BCQL), and Conservative Q-Learning (CQL). We also extended our exploration by evaluating TD3+BC in a PointMaze environment, analyzing its behavior under out-of-distribution (OOD) scenarios.

The remainder of this paper is organized as follows. Section II reviews related work on offline RL and its challenges. Section III details the methodology, including dataset generation, algorithm implementation, and evaluation metrics. Section IV presents the environmeent setup to run our experiments.Section V presents the results and analysis, discussing the performance of each algorithm. Finally, Section VI concludes the paper and outlines future work.

II. Related work

- **Mildly Conservative Q-Learning for Offline Reinforcement Learning: [4]**
  Mildly Conservative Q-Learning (MCQ) introduces the *Mildly Conservative Bellman (MCB)* operator to balance conservatism and generalization in offline RL. By assigning pseudo-target values to out-of-distribution (OOD) actions, MCQ prevents overestimation while enabling effective learning.

MCB Operator: The MCB operator is defined as:

$$T_{\text{MCB}}Q(s,a) = \begin{cases} TQ(s,a), & \text{if } \mu(a|s) > 0, \\ \max_{a' \in \text{Support}(\mu)} Q(s,a') - \delta, & \text{else.} \end{cases}$$

Where:

– T is the Bellman backup operator.
– $\mu(a|s)$ is the behavior policy.
– $\delta > 0$ introduces mild pessimism for OOD actions.

Key Properties:

– **Convergence:** The MCB operator is a γ-contraction, ensuring convergence to a unique fixed point.
– **Performance Guarantee:** The policy induced by the MCB operator satisfies:

$$Q_\mu \leq Q_{\text{MCB}} \leq Q_{\mu^*},$$

where Q_μ is the behavior policy's value function and Q_{μ^*} is the optimal policy's value function.
– **Pessimism Control:** Mild pessimism allows OOD actions to generalize better compared to overly conservative approaches like CQL.

Practical Implementation: The MCB operator is integrated with Soft Actor-Critic (SAC) using a Conditional Variational Autoencoder (CVAE) to model the behavior policy. The critic loss includes both in-distribution and OOD updates:

$$L_{\text{critic}} = \lambda \mathbb{E}[(Q(s,a) - y)^2] + (1-\lambda)\mathbb{E}[(Q(s,a_{\text{OOD}}) - y')^2],$$

where λ balances in-distribution and OOD training.

Relevance to Project Evaluating MCQ and comparing it to other baseline Offline RL algorithms was the primary objective of this project.

- **A Framework for Behavior Cloning: [2]** Behavioral Cloning (BC) is a supervised learning technique to replicate the behavior of an expert by mapping states to actions using recorded demonstrations. The approach is suitable for scenarios where access to expert policies or actions is available, but interaction with the environment is limited.

  BC Objective: The BC objective minimizes the difference between the expert's actions a^* and the predicted actions $\hat{a}$ for given states s. The loss function is defined as:

$$L(\pi) = \mathbb{E}_{(s,a^*)\sim\mathcal{D}}\left[\|\pi(s) - a^*\|^2\right],$$

  where:

  – $\pi(s)$ is the learned policy.
  – a^* are the expert actions from the dataset $\mathcal{D}$.

  Key Properties:

  – **Simplicity:** Requires only supervised learning on state-action pairs, with no additional interaction with the environment.
  – **Limitations:** BC is prone to compounding errors due to distributional shift when the learned policy deviates from the expert's trajectory.

  Implementation: BC typically uses deep neural networks to model the policy $\pi(s)$, trained using supervised learning techniques (e.g., gradient descent) on the provided dataset of expert demonstrations.

  Experimental Results: BC performs well in structured environments with dense expert data but struggles in scenarios requiring significant generalization beyond the dataset.

  Relevance to Project: BC was implemented to establish a baseline to compare MCQ and other algorithms to.

- **'A Minimalist Approach to Offline Reinforcement Learning' [6]:** Twin Delayed Deep Deterministic Policy Gradient + Behavior Cloning is a reinforcement learning algorithm designed to improve sample efficiency and stability in offline reinforcement learning settings. Building on the foundation of TD3, which mitigates overestimation bias in deterministic policy gradients through techniques like clipped double Q-learning and delayed policy updates, TD3+BC introduces an additional behavior cloning component to leverage static offline datasets effectively. This augmentation helps the policy align closely with the observed data distribution, enabling robust learning from limited or non-interactive datasets. For out-of-distribution (OOD) scenarios, we expect this algorithm to be less effective compare with MCQ, for it has behavior cloning term which might not learned from an expert. In this literature, the author discussed TD3+BC algorithm as follow. The policy is generated according to the following equation:

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\lambda Q(s,\pi(s)) - (\pi(s) - a)^2\right] \quad (1)$$

  where $\mathcal{D}$ is the dataset, λ is a scaling parameter, and Q is the estimated Q-function. Observe that a Mean Squared Error (MSE) behavior cloning term is added to policy generation, which should have lesser performance in out-of-distribution scenario compared to MCQ, as stated. [4] The loss function for the Q-value updates is defined as:

$$L(\phi,\mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}}\Big[\big(Q_\phi(s,a) - \big(r + \gamma(1-d) \cdot \max_{a'} Q_\phi(s',a')\big)\big)^2\Big] \quad (2)$$

  where r is the reward, d is the done indicator (0 or 1), and γ is the discount factor.

  The target action $a'(s')$ is clipped to remain within the action bounds:

$$a'(s') = \text{clip}\big(\mu_{\theta_{\text{target}}}(s') + \text{clip}(\epsilon, -c, c), a_{\text{Low}}, a_{\text{High}}\big) \quad \epsilon \sim \mathcal{N}(0,\sigma) \quad (3)$$

  We will implement TD3+BC using algorithm discussed in this paper on Python.

- **Conservative Q-Learning for Offline Reinforcement Learning: [3]** CQL introduces a conservative Q-function framework to address overestimation in offline reinforcement learning (RL). By penalizing Q-values for out-of-distribution (OOD) actions, CQL ensures the expected

value of a policy under the Q-function lower-bounds its true value.

Conservative Q-Learning Objective: The Q-function is trained with an augmented loss:

$$L(Q) = \alpha \cdot \mathbb{E}_{(s,a)\sim\mu}[Q(s,a)] - \mathbb{E}_{(s,a)\sim\mathcal{D}}[Q(s,a)] + \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[(Q(s,a) - \hat{B}^{\pi}Q(s,a))^2\right]. \quad (4)$$

where:

- $\mathcal{D}$ is the dataset and $\mu(a|s)$ is a chosen distribution.
- $\hat{B}^{\pi}$ is the empirical Bellman backup operator.
- $\alpha > 0$ balances conservatism and standard Bellman error.

Key Properties:

- **Lower Bound:** Ensures $Q(s,a)$ underestimates true values for OOD actions.
- **Gap Expansion:** Penalizes overestimated OOD actions, keeping policies close to the dataset distribution.
- **Safe Improvement:** Offers theoretical guarantees for safe policy improvement:

$$J(\pi) \geq J(\pi_\beta) - \zeta,$$

where ζ depends on sampling error and policy divergence.

Practical Implementation: CQL augments standard actor-critic or Q-learning methods by replacing the Bellman error term with the CQL loss. Implementation requires only minor code changes in existing frameworks, such as SAC or QR-DQN.

Experimental Results: CQL outperforms baselines like BEAR, BRAC, and BC on benchmarks such as D4RL (MuJoCo, Adroit, AntMaze) and Atari games. It achieves up to $2-5\times$ higher returns in complex, multi-modal datasets.

Relevance to Project CQL was chosen as one of the algorithms to compare the performance of MCQ against

- **Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning [5]**

AWR simplifies reinforcement learning by formulating policy updates as supervised regression, leveraging advantage-weighted samples for improved off-policy learning.

Objective: The policy is updated by solving:

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim\mu}\left[\log \pi(a|s) \cdot \exp\left(\frac{A_\mu(s,a)}{\beta}\right)\right],$$

where:

- $A_\mu(s,a) = R_\mu(s,a) - V_\mu(s)$ is the advantage.
- $R_\mu(s,a)$ is the return, and $V_\mu(s)$ is the value function.
- β is a temperature hyperparameter controlling advantage weighting.

Key Properties:

- **Off-Policy Learning:** Efficiently uses static datasets via experience replay.
- **Simple Updates:** Employs standard supervised learning for both policy and value updates.
- **Stability:** Reduces variance in policy updates through clipped advantage weights.

Implementation Steps: Each iteration consists of:

1) Compute value function $V(s)$ by minimizing:

$$L_V = \mathbb{E}_{(s,a,r)\sim\mathcal{D}}\left[\left(R_\mu(s,a) - V(s)\right)^2\right].$$

2) Update policy $\pi(a|s)$ using weighted regression:

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\log \pi(a|s) \cdot \exp\left(\frac{R_\mu(s,a) - V(s)}{\beta}\right)\right].$$

Experimental Results: AWR achieves competitive performance on OpenAI Gym benchmarks, surpassing on-policy methods (e.g., PPO, TRPO) in sample efficiency and achieving comparable asymptotic performance to off-policy methods (e.g., SAC, TD3).

Relevance to Project AWR was chosen as one of the algorithms implemented in this algorithm.

III. Methodology

This section outlines the methodology followed in this project, including preliminary tests, dataset preparation, and the implementation and evaluation of offline RL algorithms.

A. Preliminary Tests on a Simple Map

Initial tests were conducted on a simple map to evaluate the feasibility of the approach and identify potential challenges. The process included the following steps:

1) Loaded the map and processed it to create a 2D occupancy grid suitable for path planning.
2) Generated a dataset containing 4,000 A* trajectories, which included state, action, reward, next state, and done flag information.
3) Modeled the behavior policy using a Conditional Variational Autoencoder (CVAE). The policy achieved a 53% success rate during evaluation.
4) Trained a target policy using the Mildly Conservative Q-Learning (MCQ) algorithm, which resulted in a 0% success rate.

Upon analysis, the potential reasons for the poor performance of the target policy were identified as:

1) Suboptimal behavior policy performance.
2) Hyperparameters not tuned effectively.
3) An inappropriate reward structure for the task.

Additionally, upon revisiting the literature, we realized that the CVAE and MCQ approach is better suited for continuous action spaces. This realization prompted a shift in methodology, leading to the adoption of Conservative Q-Learning (CQL), which is more appropriate for discrete action spaces.

B. Experiments on a Gazebo-Derived Map

To address the limitations identified in the preliminary tests, we switched to a more complex map derived from a Gazebo simulation environment and followed these steps:

1) Loaded the Gazebo-derived map and processed it to create a 2D occupancy grid.

2) Inflated obstacles on the grid to account for safety margins, considering the radius of TurtleBot3.
3) Generated a dataset of 10,000 trajectories using the A* algorithm, resulting in approximately 1.5 million transitions.
4) Modeled the behavior policy using BC, which was evaluated to establish a baseline.
5) Used Optuna, a hyperparameter optimization framework, to carry out extensive hyperparameter trials for all algorithms except Batch-Constrained Q-Learning (BCQL), where time constraints prevented tuning. The best hyperparameters were selected to maximize the success rate for each policy.
6) Trained the target policy using Conservative Q-Learning (CQL) and evaluated its performance. The results showed a 0% success rate, which will be discussed in the Results section.
7) Trained and evaluated these additional algorithms:
   - Advantage-Weighted Regression (AWR).
   - Batch-Constrained Q-Learning (BCQL).
   - Behavior Cloning (BC)

C. TD3+BC in the PointMaze Environment

In addition to experiments on the grid-world environment, we evaluated the TD3+BC algorithm in the PointMaze environment from the D4RL dataset. This experiment aimed to analyze the impact of out-of-distribution (OOD) actions on reward functions. The results revealed that the reward function was significantly affected by OOD actions, highlighting the importance of handling distributional shifts in offline reinforcement learning.

D. Gazebo Modular Navigation Framework

A modular navigation framework was developed in ROS2 using Python and integrated into the Gazebo simulation environment with TurtleBot3 [1]. This framework facilitates interchangeable path-planning algorithms, enabling flexibility for experimentation and integration of offline RL models.

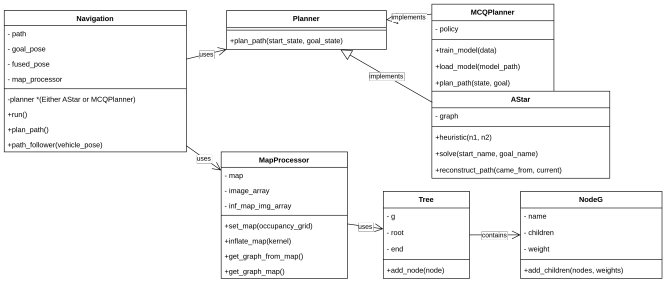


Fig. 1: Modular Code Framework with Navigator and Planner.

a) Framework Components: The framework comprises the following key components:

- `MapProcessor`: Converts occupancy grid data into binary maps, inflates obstacles, and generates navigable graphs.
- `Graph Representation`: Builds graphs connecting map nodes for efficient pathfinding.
- `AStar Path-Planning Module`: Implements A* with a modular interface for flexible algorithm integration.
- `Navigation Controller`: Guides the robot along planned paths, ensuring smooth trajectory tracking.

This modular design streamlines the integration of offline RL algorithms or advanced features like dynamic obstacle avoidance, offering a versatile platform for navigation research.

b) Testing and Evaluation: The A* algorithm was implemented to process occupancy grids generated from Gazebo's simulated laser scans. Key steps included:

- Converting sensor data into binary occupancy grids.
- Inflating obstacles for safe navigation margins.
- Creating navigable graphs and computing optimal paths using A*.

The navigation controller effectively guided the robot along these paths, correcting heading and cross-track errors as needed. Figure 2 depicts TurtleBot3 navigating within the Gazebo environment.

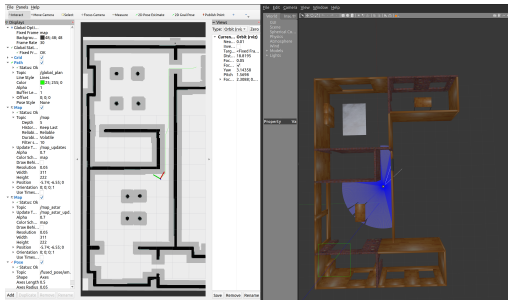


Fig. 2: TurtleBot3 navigating in the Gazebo environment using the modular framework.

This modular framework, coupled with its robust testing, ensures adaptability for integrating RL-based planners and supports ongoing improvements in navigation tasks.

E. Policy-to-Path Translation

1) **From Actions to Spatial Trajectories**: After training, the MCQ policy outputs an action (one of the four discrete directions) at each state. To create a navigable route, we roll out the policy from a chosen start position to generate a sequence of discrete moves. These action sequences are then translated into a set of connected grid coordinates.
2) **Path Smoothing and Refinement**: The raw policy output often contains stepwise transitions. To produce a more realistic and navigable path for the robot, basic smoothing techniques are applied. This involves interpolation methods that reduce zig-zagging behavior and improve overall path feasibility.

F. Integration into ROS2 Navigation Framework

1) **Map Acquisition and Occupancy Grid Handling**: The smoothed path, derived from the MCQ policy, is mapped

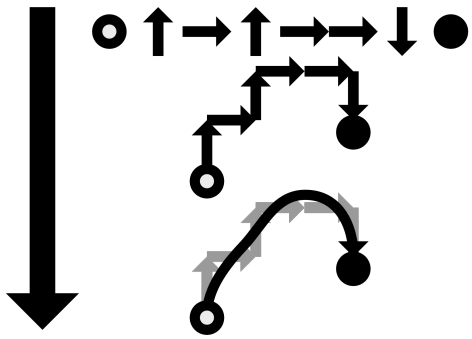


Fig. 3: Process of Policy-to-Path Translation: from Sequence of Actions, to a Smoothed Trajectory

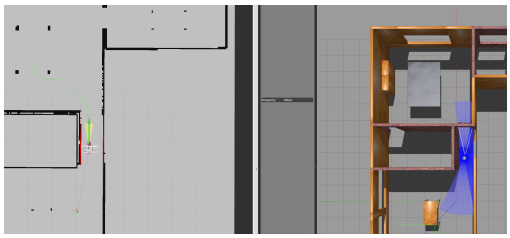


Fig. 4: Path is Followed and Verified in the Simulation Environment

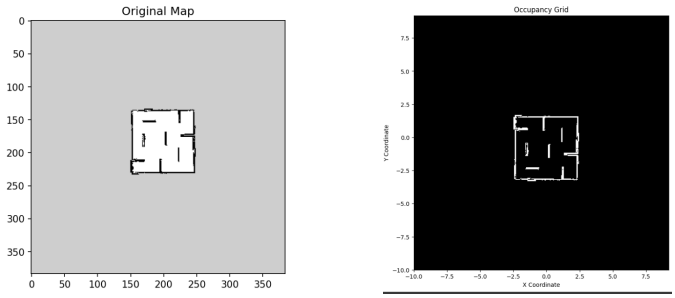
onto the coordinate system used by ROS2 navigation. The occupancy grid, obtained from a map server, provides the global frame of reference for both the path and the TurtleBot's pose.

2) **ROS2 Path Message Construction**: The discrete path from the policy is converted into a standard ROS2 `Path` message. Each point along the path is represented as a `PoseStamped`, providing positions (and optionally orientations) consistent with the robot's coordinate frame.
3) **Path Execution and Control**: The `Path` message is published to the ROS2 navigation stack. A separate trajectory-tracking controller subscribes to this topic and computes velocity commands for the TurtleBot. The robot thus attempts to follow the MCQ-derived path without requiring online replanning during execution.

IV. Environment Setup

A. Initial Tests on a Simple Map

The initial tests were conducted on a simple map (Figure 5a) to evaluate the feasibility of the proposed approach. The map contains three types of spaces: free, unknown, and obstacles. Free space (white) represents navigable areas, obstacles(black) denote non-navigable regions, and unknown space represents areas with undefined navigability (gray.) For simplicity, unknown spaces were treated either as free spaces, and the agent was assumed to only spawn in free spaces. This simplification ensured valid trajectory generation. The map was processed into a 2D occupancy grid (Figure 5b), where free spaces are represented by white cells and obstacles (including treated unknown spaces) by black cells. This discretization simplifies navigation tasks by defining clear boundaries between navigable and non-navigable areas. In later Gazebo experiments, this assumption was explicitly handled to ensure no trajectories were generated from invalid (unknown) map states.



(a) Original map used for initial tests.

(b) Occupancy grid created from the original map.

Fig. 5: Comparison of the original map and its occupancy grid representation.

1) Dataset Generation: The dataset was generated using the A* path-planning algorithm. A total of 4,000 trajectories were collected, with each trajectory representing a sequence of states and actions leading from a random start position to a fixed goal on the 2D occupancy grid. The fixed goal was placed at a specific location in the grid, chosen to ensure sufficient path diversity and to encourage trajectories of varying lengths. The starting positions were selected randomly from the free spaces of the occupancy grid to avoid invalid or unreachable states.

The dataset structure is summarized as:

- **State**: (x, y) coordinates of the agent in the grid.
- **Action**: Discrete movement directions, represented as:
  - 0: Up (-1, 0)
  - 1: Down (1, 0)
  - 2: Left (0, -1)
  - 3: Right (0, 1)
- **Reward**: -1 for every step to penalize long paths, and +100 for reaching the goal to incentivize task completion.
- **Next State**: (x', y') coordinates after taking the action.
- **Done Flag**: Boolean indicating whether the goal has been reached.

Example: A tuple for a transition might look like:

$$((1.5, 2.3), 3, -1.0, (1.5, 2.4), \texttt{False})$$

Here, the agent moves right from state $(1.5, 2.3)$ to $(1.5, 2.4)$, receives a reward of -1.0, and has not yet reached the goal.

Transition Function: In response to the feedback received in the progress report, the transition function is explicitly defined as:

$$P((x, y), a) = (x', y'),$$

where (x, y) is the current state, $a \in \{0, 1, 2, 3\}$ is the discrete action taken, and (x', y') is the resulting state. The relationship between the states and actions is given by:

$$(x', y') = (x, y) + \mathbf{v}_a,$$

where $\mathbf{v}_a$ is the action vector associated with the action a:

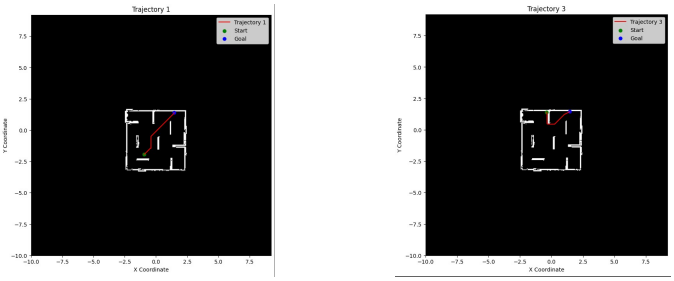
$$\mathbf{v}_a = \begin{cases} (-1, 0), & \text{if } a = 0 \text{ (Up)} \\ (1, 0), & \text{if } a = 1 \text{ (Down)} \\ (0, -1), & \text{if } a = 2 \text{ (Left)} \\ (0, 1), & \text{if } a = 3 \text{ (Right).} \end{cases}$$

The grid-world boundaries and obstacles impose constraints such that invalid transitions (e.g., moving into an obstacle or outside the grid) result in no state change:

$$P((x, y), a) = (x, y) \quad \text{if the transition is invalid.}$$

This deterministic transition function defines how the agent's state evolves based on its current position and the action taken, while adhering to the constraints of the grid-world environment.

Two sample A* trajectories generated on the occupancy grid are shown in Figure 6. These trajectories illustrate optimal paths from random starting positions to the fixed goal. The paths avoid obstacles and demonstrate the deterministic behavior of the A* algorithm in finding the shortest path.



(a) Trajectory 1: Starting position (10, 10) to goal (90, 90).

(b) Trajectory 2: Starting position (70, 30) to goal (90, 90).

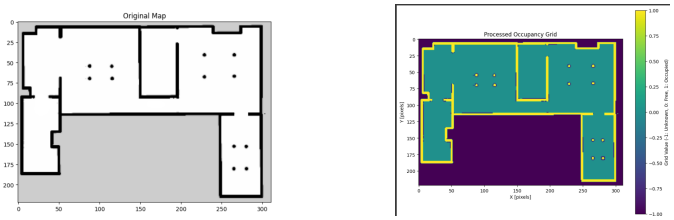
Fig. 6: Sample A* trajectories generated on the occupancy grid.

Each trajectory consists of a sequence of states (x, y) and corresponding actions, as defined by the transition function. The visualizations highlight the deterministic, obstacle-avoiding nature of the A* algorithm, which ensures that every path is optimal with respect to the defined grid-world environment.

B. Gazebo generated map

To evaluate the different policies in a realistic simulation environment, a custom map was derived from the Gazebo environment, and its corresponding occupancy grid was processed for path planning and navigation tasks. The grid was further inflated to account for the radius of TurtleBot3, ensuring safety during navigation.

a) Map and Occupancy Grid: Figure 7 illustrates the original map and the corresponding occupancy grid created using simulated sensor data. The occupancy grid accurately reflects the environment, differentiating between free space and obstacles, which is critical for path planning and navigation.



(a) Original Map

(b) Occupancy Grid

Fig. 7: Gazebo Environment Map and Generated Occupancy Grid.

b) Dataset Generation: Figure 8 illustrates a sample start and goal point overlaid on the inflated occupancy grid. The start point (blue) was randomly sampled from free space within the inflated grid to ensure validity and safety during navigation experiments. The goal point (red) was a fixed point chosen arbitrarily within the inflated grid.

For each sampled start point, an A* path was generated to the fixed goal point, and these paths were used to create the dataset. The dataset structure remained consistent with the initial experiments, consisting of state, action, reward, next state, and done flag (s,a,r,s',d). In total, 10,000 trajectories were generated, resulting in a dataset of 1,566,932 transitions and 18,210 unique states.

This figure also highlights the explicit separation of free, unknown, and occupied cells within the grid. By ensuring that start and goal points were located in valid free-space cells, the experiments minimized the risk of generating invalid trajectories originating from unknown or occupied regions.

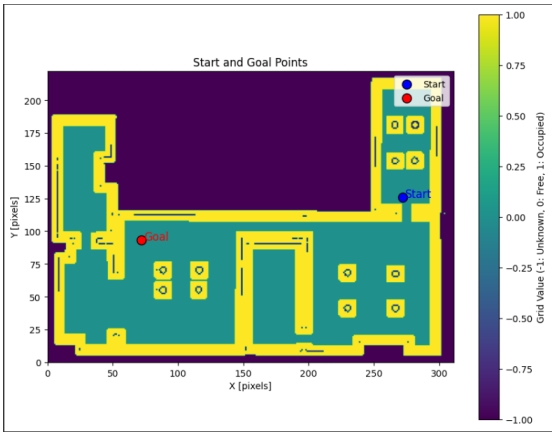


Fig. 8: Start and Goal Points Overlaid on the Inflated Occupancy Grid.

The careful selection of start and goal points, combined with the use of the inflated occupancy grid, ensured that the dataset captured realistic navigation tasks while explicitly addressing assumptions made during preliminary tests.

V. Results and Discussion

The results of this project provide valuable insights into the performance of various offline RL algorithms for grid-

world navigation. The initial tests on a simple map revealed the limitations of using a CVAE to model a behavior policy for MCQ in discrete action spac, prompting a methodological shift to CQL and other algorithms. Subsequent experiments on a Gazebo-derived map demonstrated the scalability of the approach and provided insights into the role of supervised learning techniques like BC in addressing navigation tasks. The results suggest that supervised learning methods may offer a simpler yet effective alternative for certain aspects of this problem. Additionally, experiments with TD3+BC on the D4RL dataset highlighted challenges in handling OOD actions. Detailed results for each stage of the project are presented in the following subsections.

A. Behavior Policy Modeling Using CVAE

a) Hyperparameter Selection: The effect of hyperparameters, including latent dimension (d), hidden dimension, and learning rate, was manually studied. Figures 9a and 9b show the KL divergence loss and reconstruction loss, respectively, for different combinations of these hyperparameters. The following configuration was selected as it minimized both losses:

- **Latent Dimension** (d): 4
- **Hidden Dimension**: 128
- **Learning Rate**: 0.001

b) Training Details: The model was trained on a dataset of 4,000 trajectories generated using the A* algorithm. States and next states were normalized using a standard scaler to ensure numerical stability. The training process involved:

- Optimizing the loss function using the Adam optimizer with the selected learning rate.
- Batch size: 128
- Number of epochs: 10

The trained model successfully minimized both reconstruction and KL divergence losses.

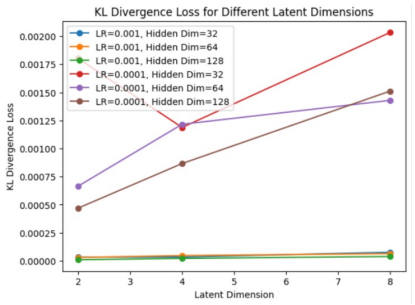
1) Behavior Policy Performance Evaluation: The behavior policy was evaluated using a dataset of trajectories generated during the A* algorithm simulations. The evaluation metrics included:

- **Success Rate**: The percentage of trajectories that successfully reached the fixed goal.
- **Collision Rate**: The percentage of trajectories that ended in a collision with obstacles.
- **Average Cumulative Reward**: The mean total reward obtained across all trajectories.
- **Average Path Length (Successful)**: The average length of successful trajectories, providing insight into the policy's efficiency.

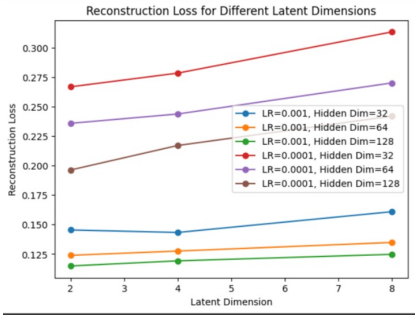
All subsequent algorithms were evaluated using the same metrics to ensure consistency and comparability.

a) Results: The results of the behavior policy evaluation are summarized below:

- The success rate of the behavior policy was 53%, while the collision rate was 44%. These metrics highlight moderate effectiveness in reaching the goal and a significant risk of collisions (Figure 14a).



(a) KL Divergence Loss for different hyperparameter settings.



(b) Reconstruction Loss for different hyperparameter settings.

Fig. 9: Effect of hyperparameters on KL Divergence and Reconstruction Loss.

- The average cumulative reward across all trajectories was 19.73, reflecting a balance between successes and failures (Figure 14b).
- Among successful trajectories, the average path length was 34.04, with lengths ranging from 10 to 70 steps, as shown in Figure 14c.

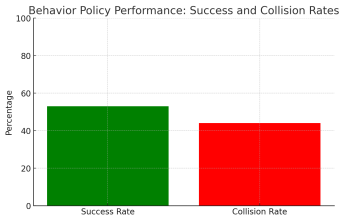
b) Insights: The behavior policy achieved a moderate success rate. The relatively high collision rate suggests the need for a more conservative policy or improved obstacle avoidance strategies. The cumulative reward distribution and path length variability establish a baseline for evaluating subsequent algorithms.

B. Training Target Policy Using MCQ

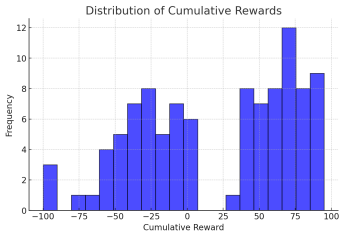
The Mildly Conservative Q-Learning (MCQ) algorithm was used to train the target policy. MCQ balances conservatism and exploitation by penalizing overestimation of Q-values for out-of-distribution (OOD) actions. The training process utilized a pre-collected dataset $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, r, \mathbf{s}', d)\}$, generated using the A* algorithm. Since the true behavior policy was unknown, it was approximated using a Conditional Variational Autoencoder (CVAE) trained on the dataset.

a) Hyperparameters: The following hyperparameters were used for training:

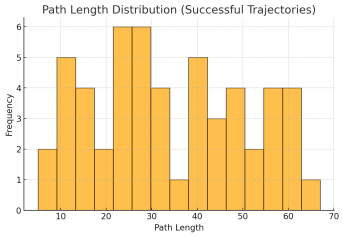
- **Discount Factor** (γ): 0.95
- **Conservatism Weight** (λ): 1.0
- **Target Update Rate** (τ): 0.005
- **Learning Rates**:



(a)



(b)



(c)

Fig. 10: Behavior Policy Performance: (a) Success and Collision Rates, (b) Cumulative Reward Distribution, (c) Path Length Distribution for Successful Trajectories.

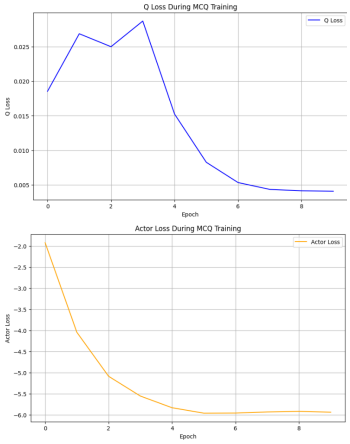
  - Q-Networks: 3×10^{-4}
  - Policy Network: 1×10^{-4}
- **Latent Dimension (d) for CVAE**: 8
- **Hidden Dimension for Networks**: 256
- **Number of Epochs**: 10
- **Number of Policy Samples (N)**: 10

b) Training Results: After 10 epochs, the Q-networks and policy network were successfully trained. The Q-loss and actor loss during training are shown in Figure 11. The Q-loss decreased steadily after an initial fluctuation, indicating that the Q-function approximations were progressively refined. Similarly, the actor loss decreased consistently, suggesting that the policy network effectively optimized its objective.

Despite the reduction in both losses, the target policy failed to generalize effectively to the task, as evidenced by low success rates during evaluation. This suggests potential limitations in the behavior policy approximation or insufficient dataset coverage, which may have led to overestimation of Q-values for out-of-distribution (OOD) actions.

Target Policy Performance Evaluation

The target policy exhibited poor performance, achieving a success rate of only 2%, with 98% of trajectories resulting in failure, as shown in Figure 12. This outcome highlights significant challenges in the ability of the target policy to generalize to the grid-world navigation task.



(a) Q-Loss and Actor Loss During MCQ Training.

Fig. 11: Training losses for MCQ over 10 epochs. The top plot shows the Q-loss, and the bottom plot shows the actor loss. Both losses decrease over time, indicating convergence.

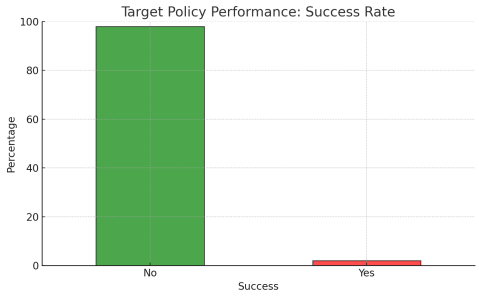


Fig. 12: Target Policy Performance: Success Rate.

c) Possible Reasons for Poor Performance: The following factors may have contributed to the target policy's poor performance:

- **Behavior Policy Approximation:** The CVAE used to approximate the behavior policy may not have accurately reconstructed actions from states. This discrepancy could lead to suboptimal actions being learned by the target policy. With just a 53% success rate of for the behavior policy, this is likely the biggest contributing factor to the poor performance.
- **Hyperparameter Tuning:** The hyperparameters for the MCQ algorithm were not extensively tuned, which might have limited the ability of the Q-networks and policy network to effectively learn from the dataset.
- **Dataset Size:** The dataset used for training may have been too small, resulting in insufficient coverage of the state-action space and limiting the target policy's generalization ability.

d) Suitability of CVAE and MCQ: Further inspection suggests that the CVAE and MCQ methods are more suited to environments with continuous action spaces. Their application

to discrete action spaces, as in this project, may introduce inefficiencies or misalignments with the intended use cases.

C. Behavior Policy Modeling using Behavior Cloning

a) Model Architecture: The behavior policy was implemented as a feedforward neural network with the following structure:

- **Input Layer:** The state representation s, consisting of 2 dimensions (x, y coordinates).
- **Hidden Layers:** Two fully connected layers with 128 units each, using ReLU activations.
- **Output Layer:** A fully connected layer with 4 units, corresponding to the discrete action space:
  - 0: Up $(-1, 0)$,
  - 1: Down $(1, 0)$,
  - 2: Left $(0, -1)$,
  - 3: Right $(0, 1)$.

b) Hyperparameters Used: The following default values for hyperparameters were used for initial training:

- **Learning Rate (η):** 1×10^{-3}, optimized using the Adam optimizer.
- **Batch Size:** 64, balancing computational efficiency and gradient stability.
- **Number of Epochs:** 20, ensuring sufficient training iterations.
- **Hidden Dimensions:** 128 units in each hidden layer.
- **Loss Function:** Cross-entropy loss, suitable for discrete action prediction.

c) Training Process: The model was trained on the dataset using the Adam optimizer. During each epoch:

1) States s were passed through the model to predict action logits.
2) The cross-entropy loss was computed between the predicted logits and the ground truth actions a.
3) Backpropagation was performed to update the model weights, minimizing the loss.

The behavior policy was trained for 20 epochs using BC. The cross-entropy loss was computed during each epoch to monitor the training progress. As shown in Figure 13, the training loss decreased steadily over the epochs, indicating that the behavior policy successfully learned to approximate the mapping from states to actions.

1) Behavior Policy Evaluation: The behavior policy modeled using BC was evaluated to assess its effectiveness in generating trajectories that reach the goal while avoiding collisions. The evaluation results are presented in terms of success rate, collision rate, average cumulative reward, and average path length, as previously defined.

The behavior policy demonstrated the following performance:

- **Success Rate:** 79%
- **Collision Rate:** 21%
- **Average Cumulative Reward:** 9.32
- **Average Path Length:** 70.68

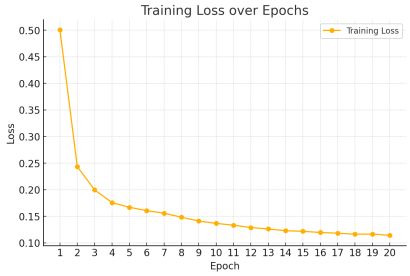


Fig. 13: Behavior Cloning Training Loss Over Epochs. The loss decreases steadily, demonstrating the model's convergence.

The evaluation results are visualized in Figure 14. These plots highlight key aspects of the behavior policy's performance:

- Subfigure 14a shows the success and collision rates, highlighting the policy's ability to reach the goal in most cases.
- Subfigure 14b depicts the distribution of cumulative rewards, with most trajectories achieving positive rewards.
- Subfigure 14c presents the distribution of path lengths for successful trajectories, indicating that most paths are of moderate length.

The results indicate that the behavior policy successfully learns to generate feasible trajectories, although further optimization may be required to reduce the collision rate and improve overall efficiency.

2) Hyperparameter Study using Optuna: To improve the performance of the behavior policy, a hyperparameter optimization study was conducted using Optuna. The objective function for the optimization was to minimize the training loss of the behavior policy. The hyperparameters explored during the study included:

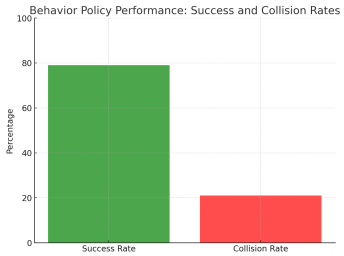
- **Hidden Dimension:** Number of units in each hidden layer.
- **Number of Layers:** Total number of hidden layers in the network.
- **Learning Rate:** The learning rate for the optimizer.

a) Optimization Results: The optimization history is visualized in Figure 15a, which shows the objective value for each trial and the best value achieved during the optimization process. The importance of each hyperparameter is depicted in Figure 15b, highlighting the significant influence of the number of layers and the learning rate on the training loss.

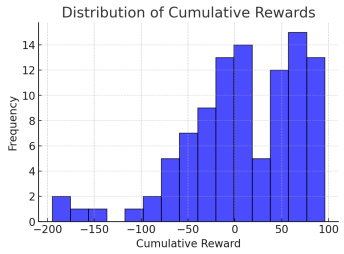
b) Best Hyperparameters: The best hyperparameters identified through the study are:

- **Hidden Dimension:** 256
- **Number of Layers:** 3
- **Learning Rate:** 0.000129

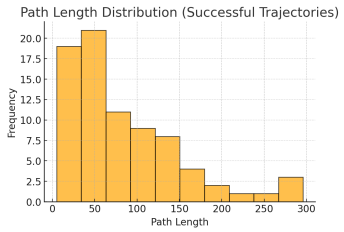
c) Retrained Behavior Policy: Using the optimized hyperparameters, the behavior policy was retrained for 50 epochs. The training loss during this retraining process is visualized in Figure 16. The plot indicates a steady decrease



(a)

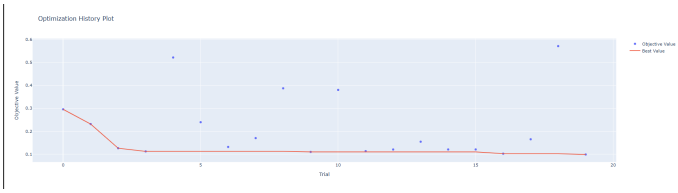


(b)

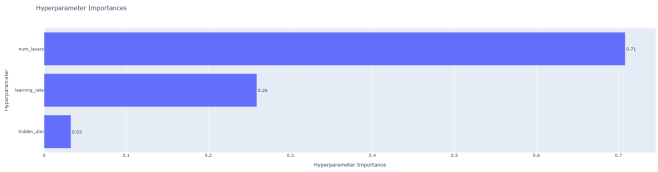


(c)

Fig. 14: Behavior Policy Evaluation Results. The plots highlight the policy's success and collision rates, reward distribution, and path efficiency.



(a) Optimization History.



(b) Hyperparameter Importance.

Fig. 15: Results of Hyperparameter Study Using Optuna.

in loss, demonstrating the effectiveness of the optimized hyperparameters.

The behavior policy was then evaluated after hyperparameter tuning, and the results demonstrated significant improvements over the initial training.

The tuned behavior policy achieved the following performance:

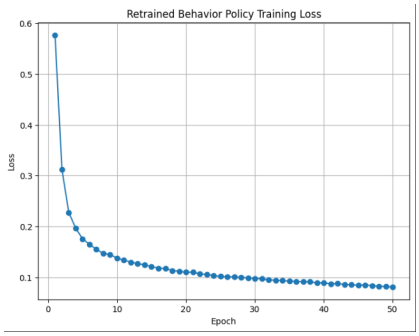


Fig. 16: Retrained Behavior Policy Training Loss with Optimized Hyperparameters.

- **Success Rate:** 94.0%
- **Collision Rate:** 6.0%
- **Average Cumulative Reward:** -5.52
- **Average Path Length:** 106.82 steps

The results are visualized in Figure 17,:

- Subfigure 17a shows the success and collision rates, emphasizing the high success rate and low collision rate achieved by the tuned policy.
- Subfigure 17b illustrates the distribution of cumulative rewards, showing a shift toward higher rewards compared to the untuned policy.
- Subfigure 17c presents the distribution of path lengths for successful trajectories, with an increase in the average path length.

The improvements in success rate and reduction in collision rate underscore the importance of careful hyperparameter tuning. The slight reduction in reward and longer path lengths in comparison to the untuned behavior policy can simply be attributed to the higher number of paths found. However, a detailed analysis of this claim has been skipped for brevity.

D. Training Target Policy Using CQL

The target policy was trained using the **Conservative Q-Learning (CQL)** algorithm, which minimizes overestimation of out-of-distribution (OOD) actions by introducing a conservative regularization term. The objective was to optimize the target policy to generalize effectively using an offline dataset.

a) Training Details: Using the best hyperparameters identified through Optuna:

- **Hidden Size:** 512
- **Batch Size:** 256
- **Learning Rate:** 0.00137
- **Conservative Regularization Weight** (α)**:** 0.00071
- **Discount Factor** (γ)**:** 0.99

The CQL model was trained for 100 epochs. The training dataset contained $1,253,545$ samples, while $313,387$ samples were used for validation.

Unfortunately, the optuna optimization plo generation process errored, so those plots have been skipped since this process took very long training times (4 hours.)

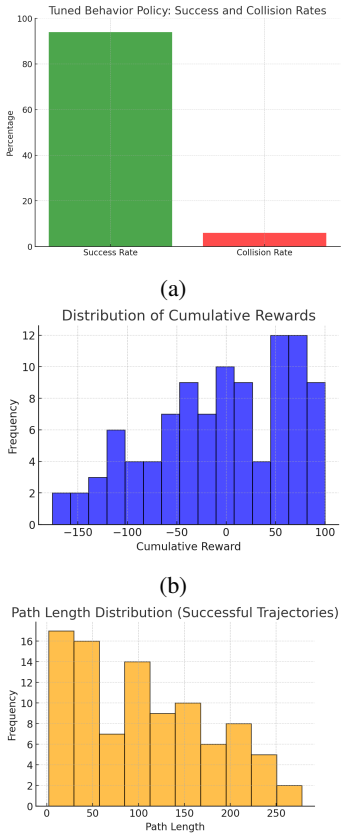


Fig. 17: Evaluation Results for the Tuned Behavior Policy. The plots highlight improved success rates, better cumulative rewards, and the distribution of path lengths for successful trajectories.

b) Loss Curve: Figure 18 shows the training loss curve for the CQL model. While the initial epochs demonstrate significant fluctuations, the loss stabilized in later epochs, indicating convergence.

Fig. 18: Training loss curve for the target policy trained using CQL

The trained model was saved after 100 epochs, and further evaluations were performed to assess its generalization and navigation capabilities. The results of the target policy's performance are presented in the subsequent section.

1) Evaluation of Target Policy Trained with CQL: The performance of the target policy trained using the Conservative Q-Learning (CQL) algorithm is summarized as follows:

- **Success Rate:** 0.0%
- **Collision Rate:** 49.0%
- **Average Cumulative Reward:** –58.87
- **Average Path Length:** Not applicable (no successful trajectories)

The evaluation metrics indicate suboptimal performance in navigating the environment. The success rate highlights the policy's inability to reach the goal. The collision rate of 49% that the policy doesn't always cause the robot to collide, but fails to effectively guide it towards goal within a reasonable number of steps.

To save space, the plots visualizing CQL performance metrics have been omitted. A few potential reasons for the algorithm's bad performance include:

- **Implementation Challenges:** Developing the algorithm from scratch in PyTorch may have introduced subtle bugs or inefficiencies, as opposed to leveraging well-tested implementations from established libraries like `d3rlpy` or `ACME`.
- **Dataset Limitations:** The dataset structure used in this study was not directly compatible with standard offline RL libraries, preventing us from using these tools and necessitating a custom implementation.

These factors were not thoroughly explored, as the focus was shifted to other offline RL algorithms such as Advantage-Weighted Regression (AWR) and Batch-Constrained Q-Learning (BCQL), which are detailed in subsequent sections. Future work could benefit from aligning the dataset format with established offline RL frameworks, which may simplify implementation and improve performance through better-tested algorithmic components.

E. Actor-Critic Policy with AWR

The Actor-Critic policy was trained using Advantage Weighted Regression (AWR), which combines supervised regression with policy iteration. The process involved the following steps:

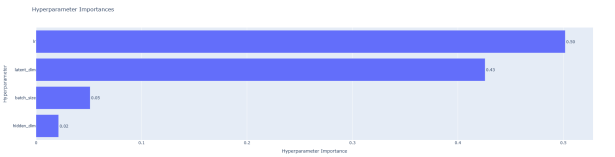
a) Hyperparameter Study: Before training, a comprehensive hyperparameter optimization was conducted using Optuna. The objective of the study was to maximize the success rate of the policy by tuning key hyperparameters. The hyperparameters included:

- **Beta (β):** Controls the degree of advantage weighting.
- **Gamma (γ):** Discount factor for future rewards.
- **Learning Rate (η):** Step size for the Adam optimizer.
- **Hidden Dimensions:** Number of units in each hidden layer of the policy network.
- **Number of Layers:** Depth of the policy network.

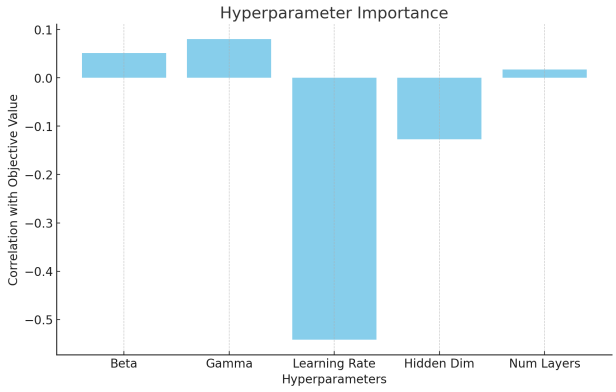
The hyperparameter importance and the results of the study are visualized in Figure 19. The best hyperparameters identified were:

- **Beta (β):** 0.7

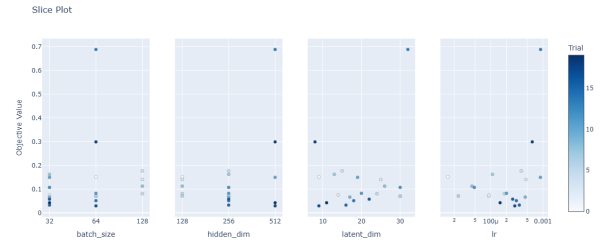
- **Gamma (γ):** 0.95
- **Learning Rate (η):** 5×10^{-4}
- **Hidden Dimensions:** 256
- **Number of Layers:** 2



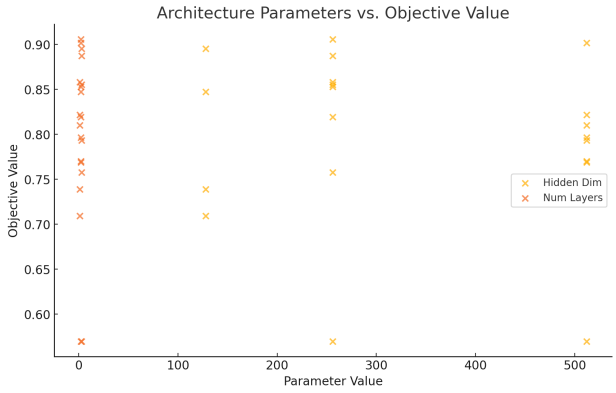
(a)



(b)



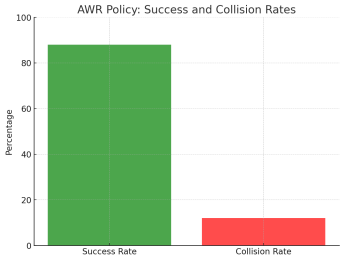
(c)



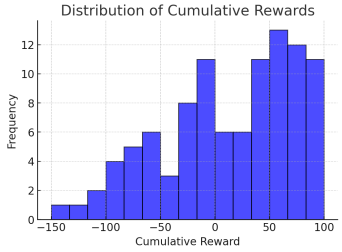
(d)

Fig. 19: AWR Hyperparameter Study: (a) Hyperparameter Importance (b) Hyperparameter Correlaton Plot(c) Slice plot showing the impact of individual hyperparameters. (d) Objective value based on architectural parameters.

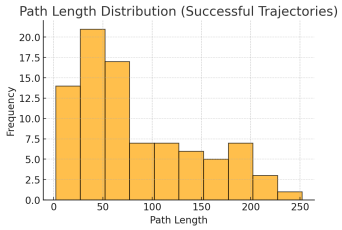
b) Evaluation of AWR: The results of AWR evaluation are presented in Figure 20. The training achieved a final success rate of 88%, with a collision rate of 12%. The average path length was 13.43 steps, and the average cumulative reward received was 83.99.



(a)



(b)



(c)

Fig. 20: Performance of the AWR policy, showcasing success and collision rates (a), average cumulative rewards (b), and c(c) path length distribution.

F. BCQL Training

a) Training Procedure: The BCQL training was divided into three stages:

- **VAE Training:** The VAE was trained for 50 epochs with a batch size of 256 and a learning rate of 0.001. It reconstructed actions while regularizing the latent space using the KL divergence.
- **Q-Network Training:** The Q-networks were trained using the Bellman error loss for 50 epochs with a batch size of 256 and a learning rate of 0.001.
- **Perturbation Network Training:** The perturbation network was trained to refine the actions generated by the VAE while remaining close to the behavior policy. This stage also lasted for 50 epochs.

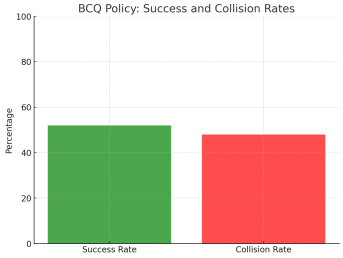
b) Limitations: Due to time constraints, a formal hyperparameter optimization study was not conducted. Default

values for all hyperparameters were used, which may have impacted the overall performance.

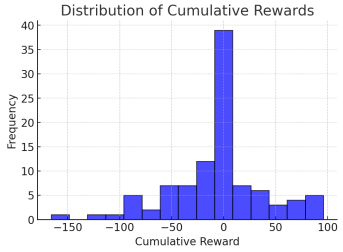
c) Implementation Details: The following parameters were used:

- **State Space:** A 2-dimensional space representing (x, y) coordinates.
- **Action Space:** A discrete space with 4 actions (up, down, left, right).
- **Latent Dimension:** The VAE utilized a latent space of size 32.
- **Device:** Training was performed on a GPU where available.

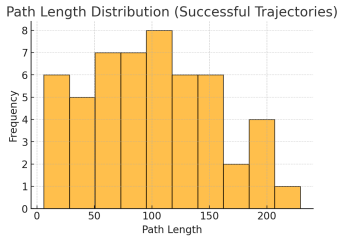
d) Evaluation: The results of BCQ evaluation are presented in Figure 21.The BCQL policy achieved a success rate of 52%, a collision rate of 48%, a cumulative reward of -6.41, and average path length of 98.65.



(a)



(b)



(c)

Fig. 21: Evaluation metrics for the BCQ policy, including success rates (a), cumulative rewards (b), and path lengths (c).

G. TD3+BC Experiment on D4RL dataset

- For TD3+BC experiment, we used Minari(D4RL) dataset and gymnasium on python to render as in Figure 22, in anaconda3 customized environment. I trained three policy with 1,000 epochs, 10,000 epochs, and 100,000 epochs. I used these policy to render the simulation. The Point Maze domain involves moving a force-actuated ball (along the X and Y axis) to a fixed target location. The observation consists of the (x, y) location and velocities. The dataset consists of one continuous trajectory of the agent navigating to random goal locations, and thus has no terminal states. The agent uses a PD controller to follow a path of waypoints generated with QIteration until it reaches the goal. The reward function is the negative Euclidean distance between the goal and the agent.

1) Insights: Since the Point Maze environment is learning from a PD controller, there is a convergence issue when the agent is near the goal. For all three policies, the agent navigates near the goal and then oscillates around the goal or is stuck in some position nearby. Based on the average reward of 10 samples for each policy, we can see a trend that with more training, the performance of the actor is improving, as in Figure 23. However, in a barely trained policy, the number of trajectories from the dataset is significantly reduced, leaving randomly generated goal and starting point out of distribution. In this situation, we often observe the agent rush in segmental fashion, or stuck in a corner trying to pass through obstacles. On the other hand, even with 2.7 hour training, 100,000 epochs, the variance for the reward is proportionally increasing, as in Figure 24. This imply even with a policy trained on many trajectories, the effect from out of distribution starting point and goal still affect gravely on the reward.

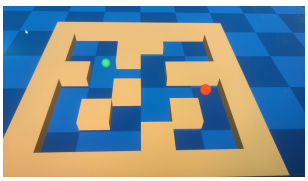


Fig. 22: Mujoco rendered simulation snapshot

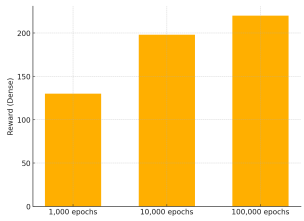


Fig. 23: Average reward for each policy

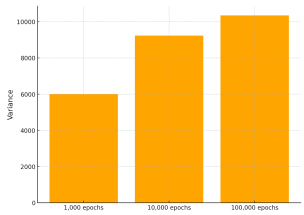


Fig. 24: Variance for each policy with 10 samples

H. Gazebo Simulation Test Results

The RL policy demonstrated a success rate comparable to its performance in the grid-world test. In most scenarios, the TurtleBot successfully navigated to the target, even when minor collisions occurred. However, the policy struggled significantly when the environment changed dynamically due to these collisions, such as shifting a table. In such altered scenes, the policy failed to adapt and often collided with obstacles repeatedly, highlighting a key limitation of training in static environments.

When comparing the RL policy (BC) to the A* algorithm, the RL policy achieved 87% of A*'s success rate on pre-tested successful grid-world paths. Failures were mainly due to environment changes (11%) where untrained scenarios disrupted navigation, while a smaller proportion (2%) were attributed to path-following controller errors, causing execution issues even on valid paths.

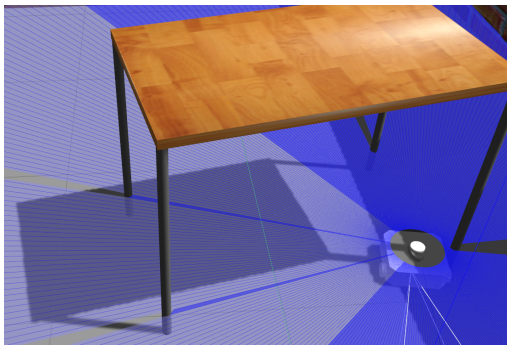


Fig. 25: Failed and Crash to Movable Obstacles

1) Example of Successful Navigation: The image below illustrates an example of a successful navigation episode using the RL policy. The TurtleBot follows a smooth, collision-free trajectory to the target, showcasing the policy's effectiveness under ideal conditions.

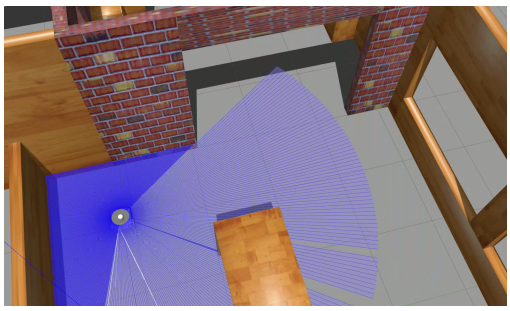


Fig. 26: Successful Navigation Episode

VI. Conclusion and Future Work

Conclusion: This project explored the application of offline reinforcement learning (RL) algorithms for grid-world navigation. While methods like Behavior Cloning (BC) and Advantage-Weighted Regression (AWR) demonstrated success, Mildly Conservative Q-Learning (MCQ) and Conservative Q-Learning (CQL) struggled due to challenges such as behavior policy approximation and hyperparameter tuning. The results highlight the importance of dataset quality, algorithm suitability, and robust implementation for offline RL tasks.

Future Work: To address the observed limitations, the following directions are proposed:

- Implement algorithms using standard libraries, such as `d3rlpy` or `ACME`, to reduce implementation overhead and improve reliability.
- Extend Gazebo experiments by incorporating dynamic obstacles to test policies in more realistic environments.
- Investigate the performance of MCQ in continuous action spaces, where TurtleBot3 could use position and heading as state inputs and output velocity commands as actions.

These steps aim to enhance adaptability, scalability, and real-world applicability of offline RL algorithms.

References

[1] Robin Amsters and Peter Slaets. Turtlebot 3 as a robotics education platform. In *Robotics in Education: Current Research and Innovations 10*, pages 170–181. Springer, 2020.

[2] Michael Bain and Claude Sammut. A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129, 1995.

[3] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1179–1191. Curran Associates, Inc., 2020.

[4] Jiafei Lyu, Xiaoteng Ma, Xiu Li, and Zongqing Lu. Mildly conservative q-learning for offline reinforcement learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 1711–1724. Curran Associates, Inc., 2022.

[5] Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177*, 2019.

[6] Xin-Yu Xu, Yang-Yang Chen, and Tian-Run Liu. Td3-bc-ppo: Twin delayed ddpg-based and behavior cloning-enhanced proximal policy optimization for dynamic optimization affine formation. *Journal of the Franklin Institute*, page 107018, 2024.