2AMC15 — Data Intelligence Challenge

1. Introduction

The increasing use of autonomous systems in various industries has driven significant interest in the development of delivery robots for tasks such as food delivery in restaurants. These robots promise to enhance operational efficiency, reduce labor costs, and improve customer service (Hossain (2023)). Reinforcement learning (RL), particularly deep reinforcement learning (DRL), offers a powerful framework for developing systems capable of learning optimal behaviors from interactions with their environment. Previous studies have successfully applied RL techniques to autonomous navigation and delivery tasks (Jahanshahi et al. (2022), Faust et al. (2017)). For instance, Deep Q-Learning (DQN) has been widely used for solving navigation problems in grid environments (Li (2017)). Additionally, research on multi-task learning and dynamic task management in RL contexts provides valuable insights for enhancing the efficiency and robustness of delivery robots (Li (2017)).

The focus of our project is to design an autonomous delivery robot that functions as a waiter in a restaurant setting. The robot's primary tasks are to pick up plates from the kitchen, navigate the restaurant, and deliver food to customers at designated tables.

1.1 Problem Formulation

To address the outlined tasks, we model the problem using a Markov Decision Process (MDP). The MDP is defined by the tuple (S, A, P, R, γ):

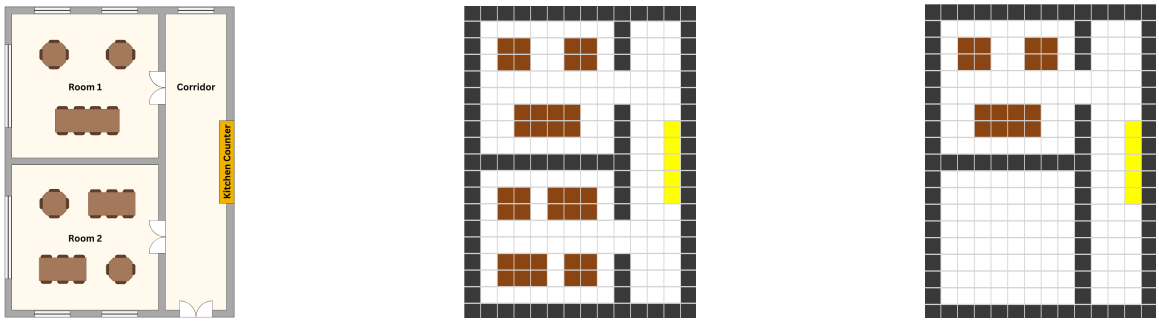
- **States (S)**: A state $s \in S$ represents the robot's current position on the grid and its task status (e.g., whether it is carrying food or not).
- **Actions (A)**: The set of actions $a \in A$ includes moving in four directions (up, down, left, right).
- **Transition Function (P)**: The transition function $P(s'|s, a)$ defines the probability of transitioning to state s' from state s after taking action a. In our deterministic environment, this simplifies to $s' = T(s, a)$, meaning there is no randomness in the state transitions (i.e., $\sigma = 0$).
- **Reward Function (R)**: The reward function $R(s, a)$ assigns rewards based on the current state and action. Positive rewards are given for visiting the kitchen when it is the goal and for visiting the correct tables. Negative rewards are given for moving onto empty tiles or visiting the kitchen when it is not the goal. Severe negative rewards are assigned for hitting walls or obstacles and for visiting the wrong tables.
- **Discount Factor (γ)**: The discount factor $\gamma \in [0, 1)$ represents the importance of future rewards.

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid \pi\right] \tag{1}$$

The goal is to find an optimal policy π^* that maximizes the expected cumulative reward over time, defined in equation 1 In this equation, s_t and a_t represent the state and action at time step t, respectively.

1.2 Environment design

The environment is modeled as a grid-based representation of a restaurant, comprising walls, floors, tables, and the kitchen. The robot must navigate this environment to perform its tasks efficiently. The grid can be adjusted to simulate different restaurant layouts, including single-room and dual-room scenarios connected by a door. An example of a restaurant floor plan and the grid abstraction of it is provided in Figure 1.



(a) Floor plan (b) 2 rooms (c) 1 room

Figure 1: Restaurant and abstractions

In our environment, each table is represented by a number (starting at 1), and the kitchen is represented by a 0. An agent has a capacity c, which indicates the number of plates it can carry at a time. In every training episode, the agent is initialized in a random position and is given a fixed-size list (`tables_to_visit`) of randomly selected tables to visit (size = n). At first, the agent has no plates in its inventory and must find the kitchen to collect plates (this instruction to visit the kitchen is encoded as a single `[0]` in its `inventory`). Once it finds the kitchen, the `inventory` is filled up to its capacity c with tables to visit. It then tries to find the correct tables, removing them from the list once they are found. If the agent visits the kitchen while its inventory is not empty, it will refill up to capacity c. If the agent manages to empty its list and there are still orders left to fulfil in `tables_to_visit`, we again place a `[0]` in its `inventory` to direct it towards the kitchen so that it can refill with its `inventory` with tables. An episode ends when all tables in (`tables_to_visit`) have been visited, or when a specified maximum amount of episode steps is reached.

The state of the agent $s_t = (x_t, y_t, k_t)$, where x_t, y_t represent the robot's position on the grid, and k_t indicates the current task status (i.e `inventory`). The action $a_t \in \{up, down, left, right\}$ determines the transition to the next state s_{t+1}.

The reward function R can be formulated as:

$$R(s_t, a_t) = \begin{cases} -0.1 & \text{if moving onto an empty tile} \\ -1 & \text{if hitting a wall or obstacle,} \\ 1 & \text{if visiting the kitchen when it is the goal,} \\ -2 & \text{if visiting the kitchen when it is not the goal,} \\ -5 & \text{if visiting the wrong table,} \\ 10n & \text{if visiting the correct table, where } n \text{ is the number of times the} \\ & \text{agent has to visit the specific table.} \end{cases}$$

2. Approach

2.1 Q-Learning

In the previous assignment, our Q-Learning agent performed the best out of the three agents we implemented. Thus, we decided that starting with the Q-Learning agent was reasonable. However, due to the increase in complexity, we thought it might not perform as well. Nevertheless, it could be an interesting baseline to compare with a more complex agent. Since the reader may be familiar with Q-Learning and the previous environment setup we had in Assignment 1, only the main changes required for the agent to work on the new task will be highlighted.

The state is defined at any given moment not only by the position on the grid but also by the tables (or kitchen) that the agent has to visit. Hence, the number of states is the product of the number of grid cells and the combinations with repetitions of the number of tables, with sizes ranging from zero to the capacity of the agent. Evidently, the number of states of the Q-Learning agent is substantially higher in the new environment.

Finally, in order to obtain the encoding of the state, we concatenate the coordinates strings and the encoding of the tables. Then, we use a Hash function (SHA-256) and perform the modulo operation on the digest to ensure that the resulting pointer lies between zero and the number of states. Regarding the choice of ϵ we chose an initial value of 0.8 with a decay of 0.95 and minimal value of 0.01.

2.2 Deep Q-Learning

Double Deep Q-Networks (DDQN) build upon the foundation of Q-learning, using a neural network to approximate the Q-values instead of looking them up in a table. However, DQNs suffer from overestimation bias, where the action-value estimates can be overly optimistic. To address this, DDQN introduces two separate networks: the online network and the target network. The online network is used to select the action, while the target network is used to evaluate the Q-value, thus decoupling action selection from action evaluation. The update rule for DDQN is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) - Q(s,a) \right)$$

where θ represents the parameters of the online network and θ^- represents the parameters of the target network. The target network's parameters are periodically updated to match the online network's parameters, which helps in stabilizing the learning process.

The primary advantage of DDQN over DQN is its ability to reduce the overestimation bias, leading to more accurate Q-value estimates and improved policy performance. By

using separate networks for action selection and evaluation, DDQN achieves more stable and reliable learning in complex environments.

A replay buffer is a crucial component in DDQN, which stores experiences (s, a, r, s') encountered by the agent. By sampling mini-batches of experiences from the buffer for training, it allows the agent to reuse past experiences, improving data efficiency. So after the replay buffer has collected a batch of experieneces, it'll pass them to the online network and the training will happen on it.

The loss function in DDQN aims to minimize the difference between the predicted Q-values and the target Q-values. This is typically achieved using the Mean Squared Error (MSE) loss. Given a mini-batch of transitions (s, a, r, s') sampled from the replay buffer, The loss for a mini-batch is :

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(y - Q(s, a; \theta)\right)^2\right]$$

where $\mathcal{D}$ is the replay buffer and θ are the parameters of the Q-network. By minimizing this loss, the network learns to approximate the optimal Q-values.

2.2.1 Comparison

Standard Q-learning methods have the disadvantage of being unsuitable for large or continuous state spaces due to the need to maintain a table of Q-values for every state-action pair. Deep Q-learning on the other hand addresses this limitation by approximating the Q-function using a deep neural network. This allows deep Q-learning methods to generalize across states that are similar to each other. Additionally, since DQL does not need to store the Q-value of every state-action pair in a table explicitly, this method requires significantly less memory. This makes DQL far more scalable to high-dimensional spaces.

2.2.2 Implementation

In this section, we discuss the base setups implemented for training the Double Deep Q-Network (DDQN) agent, including hyperparameters and model input features. **Epsilon (ϵ)**: This parameter balances exploration and exploitation. It starts at 0.4 and decays to 0.01 over time, allowing the agent to explore the environment initially and exploit learned strategies later. **Gamma (γ)**: Set to 0.90, this discount factor determines the importance of future rewards. A high gamma value ensures the agent prioritizes long-term rewards over immediate gains.

The input features of the DQN model include the agent's position, the visit list, and the number of dishes to be delivered. Since the visit list is hard to map directly to tables, we use both the visit list and its positional encoding to provide the model with necessary sequence information.

3. Empirical Study

In this section, we tune the hyperparameters of our DDQN agent, and then perform three experiments: comparing the DDQN agent to the baseline Q-learning agent, comparing the DDQN agent's performance on different grid sizes, and comparing the DDQN's performance with different capacity values. For these experiments, we train each agent until convergence

using early stopping (stopping training once the reward has not increased for a certain number of episodes, or once we reach 1000 episodes), and plot the total reward per episode. Then we run the trained agent for 100 episodes and evaluate it using several metrics: average steps taken, kitchen visits, wrong table visits, percentage of plates delivered, and total reward.

3.1 Hyper parameters tuning

To optimize the performance of our delivery robot, we conducted a series of hyperparameter tuning experiments on the Two-Room setup (Figure 1b). The goal was to identify the best combination of parameters that maximize the robot's efficiency in completing its tasks. The tested hyperparameters, with their values, are reported in Table 1 The maximum steps per episode (max_steps_per_ep) were set to n_plates $\times$ 50, and the decay steps were defined as max_steps_per_ep $\times$ iters $\times$ decay_factor.

Hyperparameter	**Tested Values**	**Default Value**
Epsilon Start (ϵ_{start})	[0.1, 0.5, 0.9]	0.5
Epsilon End (ϵ_{end})	[0.01, 0.05, 0.1]	0.01
Gamma (γ)	[0.5, 0.8, 0.95]	0.95
Decay Steps Factor	[0.2, 0.4, 0.6]	0.4

Table 1: Hyperparameters Tested and Their Default Values

Each hyperparameter was tuned individually while keeping the others fixed at their default values, determined to be optimal in preliminary tests. The robot's performance was evaluated based on the total reward accumulated over training epochs. For each parameter set, we ran multiple training sessions and recorded the total reward per epoch. This allowed us to analyze the impact of each hyperparameter on the robot's ability to learn and perform its tasks effectively.

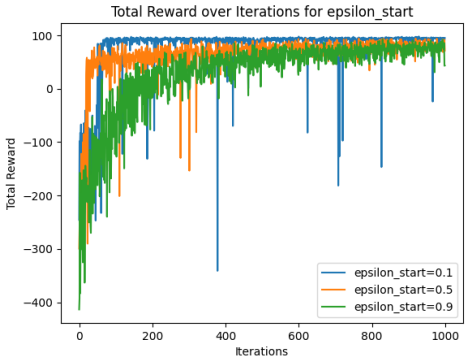
3.1.1 Results and Analysis

The results of the experiments are reported in Figures 2a, 2b, 2c, and 2d.

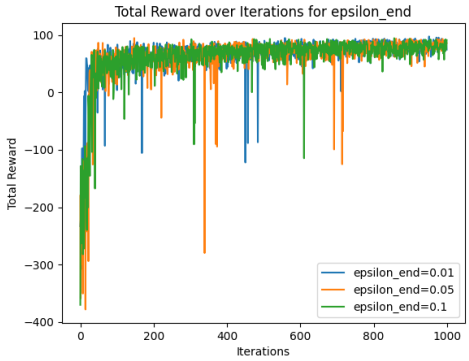
Figure 2a illustrates the total reward vs. epochs for different epsilon start values. While the convergence values are similar, $\epsilon_{\text{start}} = 0.1$ converges faster and is more stable compared to $\epsilon_{\text{start}} = 0.9$, which takes significantly longer. In Figure 2b, the trends for different epsilon end values are nearly identical, indicating minimal impact on learning performance. As shown in Figure 2c, $\gamma = 0.5$ performs poorly, converging to a negative reward, whereas $\gamma = 0.8$ and $\gamma = 0.95$ converge to a positive reward of around 100, demonstrating better long-term performance. Figure 2d reveals that varying the decay steps factors results in very similar performance, suggesting a minor impact. Based on these experiments, the optimal configuration for our delivery robot includes $\gamma = 0.95$, $\epsilon_{\text{start}} = 0.5$, $\epsilon_{\text{end}} = 0.01$, and a decay steps factor of 0.4. This configuration achieves the highest cumulative rewards, indicating effective learning and task execution.

3.2 Single-Room and Two-Room Configurations

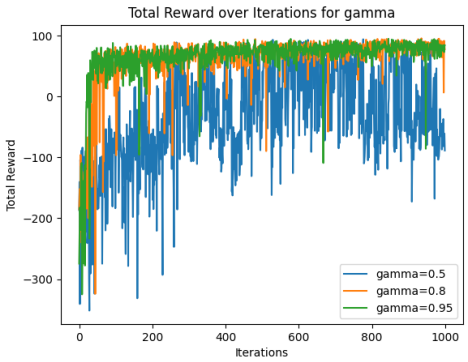
To evaluate our delivery robot agent's adaptability and robustness, we designed an experiment involving two restaurant configurations: a single-room setup (Figure 1c) and a



(a) Total Reward vs. Epochs for Different Epsilon Start Values



(b) Total Reward vs. Epochs for Different Epsilon End Values



(c) Total Reward vs. Epochs for Different Gamma Values

(d) Total Reward vs. Epochs for Different Decay Steps Factors

two-room setup (Figure 1b) connected by a narrow doorway. The one-room restaurant is a simpler environment where the agent's basic navigation and task completion skills are tested. The two-room restaurant adds complexity by requiring the agent to find and navigate through the doorway, testing its ability to handle more intricate environments and longer paths.

3.2.1 Experimental setup

For this experiment, we train one agent for each grid using the hyperparameters given in Table 2.

	N_plates	**Capacity**	**Gamma**	**Start ϵ**	**Early stopping criterion**	**Max training episodes**
1-room agent	9	3	0.95	0.4	100 episodes with no reward increase	1300
2-room agent	9	3	0.95	0.5	200 episodes with no reward increase	1000

Table 2: Hyperparameters for grid comparison experiment

A lower starting ϵ was chosen for the 1-room agent because it is smaller and contains fewer tables, so less exploration is required at the start when compared to the 2-room agent.

Likewise, we use a higher early-stopping value for the 2-room agent so that it has more time to converge. The results for this are shown in Figure 3 and Figure 3. As expected, the 2-room agent takes much longer to converge, since the state space is larger. Both agents learn to deliver all the plates in their inventory, but the 1-room agent performs significantly better in all other metrics. When evaluating the 2-room agent with the GUI, we observe that the agent is usually very adept at serving tables, but sometimes gets stuck in loops between certain points on the grid. This results in a lower average reward and higher average steps taken. The agent visits incorrect tables more often, which in combination with the oscillations indicates that it did not find an optimal policy.

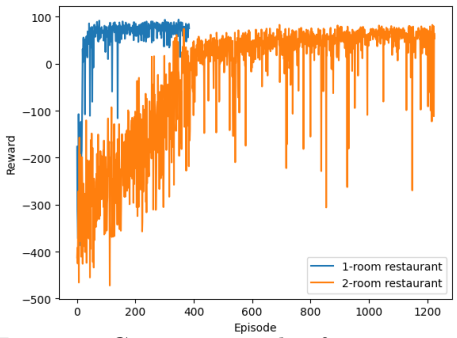


Figure 3: Convergence plot for 1-room vs 2-room grid

	1-room agent	2-room agent
Steps taken	102.9	182
Kitchen visits	4.1	2.9
Wrong table visits	0.2	6.9
Plates delivered (%)	100	96.2
Total reward	92.4	19.4
Steps to table	12.9	43.6

Table 3: Average metrics per 100 episodes for 1-room vs 2-room grids

3.2.2 Comparison with Q-Learning agent

Using the same hyperparameters for the Q-learning agent from Table 2(besides the initial ϵ of 0.8 in both cases), we obtain the following results displayed in Figure 4 and Table 4. The Q-Learning agent struggles to learn the task in the new environment due to the extremely high-dimensional state space. While the agent's overall performance is poor, there's a clear difference between the one-room and two-room scenarios. The success rate for delivering plates drops from about 30% in the one-room setup to around 15% in the two-room setup. However, the results remain significantly inferior compared to those achieved by DDQN (see Figure 3 and Table 3). This confirms our hypothesis that Q-Learning is insufficient for the new, complex task, and that DDQN substantially enhances the baseline performance.

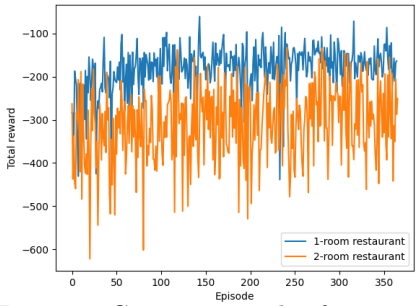


Figure 4: Convergence plot for 1-room vs 2-room grid

	1-room agent	2-room agent
Steps taken	530.77	531
Kitchen visits	2.26	1.47
Wrong table visits	12.17	33.16
Plates delivered (%)	31.1	15.6
Total reward	-170.5	-287.6
Steps to table	263.6	435.8

Table 4: Average metrics per 100 episodes for 1-room vs 2-room grids

3.3 Varying plate capacity

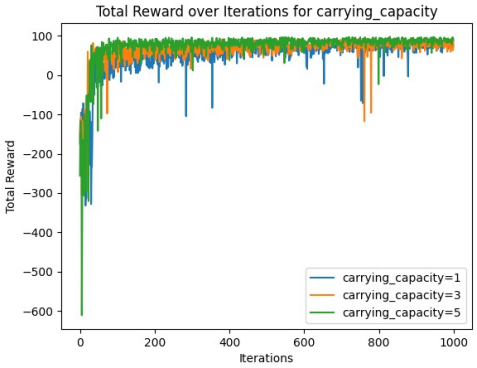


Figure 5: Total rewards for varying carrying capacities

As described in subsection 1.2, the robot must return to the kitchen every three deliveries by default. This represents the amount of plates that the robot can carry simultaneously. In this experiment, we test how the performance changes as the carrying capacity of the robot changes by comparing the standard carrying capacity of three to capacities of one and five. The results of these agents are compared based on the reward function value after all food has been delivered and all agents are trained on the grid with one room as can be seen in figure 1c. The results of this experiment can help restaurants to decide whether they would rather have many robots with small carrying capacities or fewer robots with greater carrying capacities. The obtained results show that larger carrying capacities tend to lead to higher total rewards if the model is trained using relatively few iterations. As the number of iterations increases, the difference in performance between the 3 models decreases. Thus, restaurant owners can customize the number of plates they wish their robot to carry without significantly affecting the performance of the model.

4. Conclusions

In this report, we introduced our solution to the real-world problem of plate delivery in restaurants using a reinforcement learning approach. Traditional methods like Q-learning proved ineffective in our realistic setting, so we developed a functional Double Deep Q-Network agent. Our agent efficiently learns to deliver plates to various tables, handling different numbers of plates and adapting to the number of rooms the restaurant has open. In our complex environment, where multiple tables need simultaneous service, the agent performs proficiently, effectively delivering plates and functioning as an ideal waiter.

4.1 Future work

There are several potential directions for future work on our waiter robot DDQN agent:

1. **More Complex Environments:** Experiment with more complex settings, such as additional rooms, larger room sizes, and an increased number of tables.
2. **Battery Management:** Integrate a charger and battery level feature, allowing the robot to autonomously determine when to recharge and to learn the optimal recharging times.
3. **Multi-Agent Coordination:** Investigate how multiple agents can interact effectively to ensure accurate order delivery.
4. **Direct Customer Delivery:** Explore scenarios where agents deliver directly to customers rather than just to tables.

These enhancements could significantly improve the functionality and efficiency of our waiter robot and are a natural way to proceed.

References

A. Faust, I. Palunko, P. Cruz, R. Fierro, and L. Tapia. Automated aerial suspended cargo delivery through reinforcement learning. *Artificial Intelligence*, 247:381–398, 2017.

M. Hossain. Autonomous delivery robots: A literature review. *IEEE Engineering Management Review*, 51(4):77–89, 2023. doi: 10.1109/EMR.2023.3304848.

H. Jahanshahi, A. Bozanta, M. Cevik, E. M. Kavuk, A. Tosun, S. B. Sonuc, B. Kosucu, and A. Başar. A deep reinforcement learning approach for the meal delivery problem. *Knowledge-Based Systems*, 243:108489, 2022.

Y. Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

5. List of contributions

- Andrei: Implementation of DDQN agent, environment updates, parts of section 1.2 (second paragraph), version control maintenance, code review
- Annika: Implementation of DDQN agent, updates to environment, single-room vs two-room experiment, introduction of section 3.
- Aria: Report:Sections 2.2, 2.2.2, DDQN agent bug fix. Env update(adding new cells to the env, adding them to gui + implementing how kitchen and table work, around half of the work was done by Annika). Early stopping + pos-encoding. Optimizing the agent(architecture and hyperparam wise). README.
- Cris: Report: Sections 2.1, 3.2.2, 4, and 4.1. Restaurant floor plan. Code: grid design, review code, Q-learning baseline, big table management.
- Koen: Report: Sections 2.2, 2.2.1, Table setup experiment section, video script, record video. Code: review code, Q-learning baseline, table block, generations of targets around table, positional encoding.
- Luca: Report: Sections 1, 3.1. Code: hyperparameter tuning experiment, design and implementation, visualisation of results, testing and optimization, code review.
- Patrick: Report: Sections 1, 2.2, 2.2.1, 3.3, review code, hyperparameter tuning experiment, carrying capacity experiment, visualizing experimental results