🤖

# RxRovers Roaming for Rapid Relief

| Status | Reinforcement Learning |
|---|---|

## Contribution Table

| Team Member | Checkpoint [Contribution %] | Final [Contribution %] |
|---|---|---|
| bhanucha | 50 | 50 |
| charviku | 50 | 50 |

## Report Overview

This report includes a comprehensive overview of our project, "RxRover Roaming for Rapid Relief,"  in building and refining the system, implementing various algorithms, and addressing challenges along the way. We started with establishing a basic environment that included a single rover navigating between a source and a destination. From this foundation, we expanded the scenario to include two RxRovers tasked with delivering medicine to two separate destinations. We employed 6 algorithms for training the rovers: Q Learning (QL), Double Q Learning (Double QL), Deep Q Network (DQN), Double Deep Q Network (DDQN), Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C). we share the outcomes of these implementations along with the hyperparameter setting. We provide the training and evaluation results for each of the algorithms.

## Project Description

"RxRovers: Roaming for Rapid Relief" aims to integrate advanced reinforcement learning (RL) into the healthcare domain, specifically focusing on optimizing medical supply delivery within hospital settings. This seeks to deploy autonomous agents, RxRovers, which are programmed to navigate through hospital corridors efficiently, dodging any potential obstacles (Dynamic and Static) to ensure the timely distribution of medicines, which can significantly enhance patient care and outcomes.

Employing simulated hospital environments for training and evaluation, in the project we create an outline map representations of hospital layouts, with RxRovers as robotic agents, starting points near operations desks, and various obstacles including other RxRovers, humans, and static structures like rooms, walls etc.

This project not only exemplifies the potential of AI and RL in healthcare but also underscores the importance of technological innovation in enhancing patient care.

### Objectives:

- Developing RL agents capable of autonomously navigating hospital environments while delivering medical supplies.
- Optimize path planning strategies to ensure timely and efficient delivery of medicines, while avoiding obstacles such as equipment, humans, and environmental constraints.
- Enhance the visual representation and user experience of the simulated hospital environment to improve engagement and realism.
- Conduct comparative analysis of various reinforcement learning algorithms to identify the optimal approach for medical supply delivery optimization within hospital environments.

## Background Work

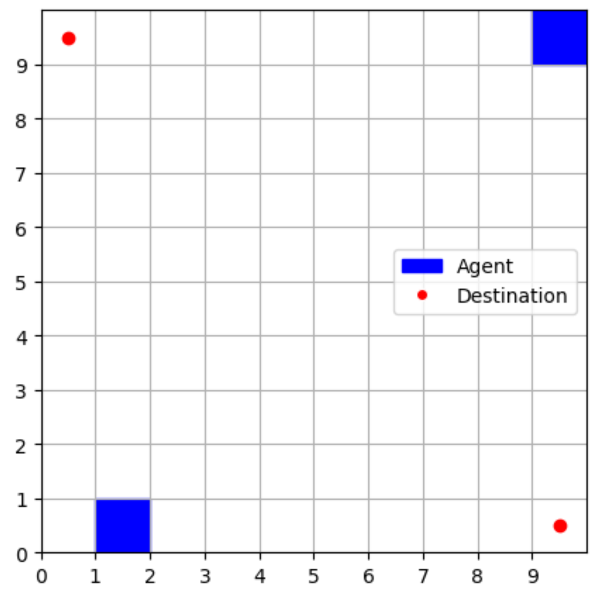
This project draws inspiration from various works in the field of reinforcement learning (RL) and artificial intelligence (AI). One such inspiration is the utilization of deep reinforcement learning (DRL) algorithms, as discussed in works like "Deep Reinforcement Learning DQN for Multi-Agent Environment" by Bruno Centeno [1]. This work highlights the effectiveness of DRL in navigating complex environments with multiple agents, a concept directly applicable to the development of autonomous RxRovers within hospital settings.

From the project "Warehouse Robot Path Planning," our project gains insights into multi-agent path planning and collision avoidance strategies, particularly in warehouse scenarios.  By adapting the  approach used in the "Warehouse Robot Path Planning" project [2], our project can train the agents to navigate the hospital environment. Furthermore, the collision avoidance strategies outlined in the warehouse project provided valuable insights into handling dynamic obstacles and coordinating actions between multiple agents. Our project can leverage these strategies to ensure that RxRovers navigate hospital corridors safely, avoiding collisions with obstacles, personnel, and other RxRovers.

## Environment

### Initial Stage

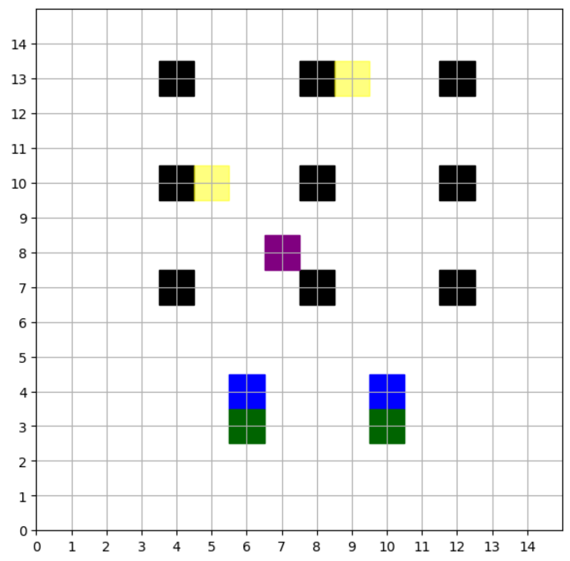
We created a basic environment with 9×9 grid simulating rovers navigating on a grid. Initialized action and observation spaces, set up grid size, starting points, destinations, and parameters such as rewards and penalties for actions and events. The environment allows for resetting to initialize the scenario with rovers at fixed starting points.



During each step, rovers take actions, updating their positions based on movement rules and calculating rewards considering their proximity to the destination and penalties for collisions or inefficient movements. Collision avoidance rules handling scenarios with multiple agents in the same cell and applying penalties for collisions. Observations are generated as grid-based representations of the agents' positions.

## Refined Version

Following the initial setup, we introduced several modifications to the environment. These changes involved incorporating rooms (depicted in black), desks (serving as starting points for the agents and depicted in green), individuals (depicted in violet), and destinations (depicted in yellow) to accurately reflect the hospital setting within our grid representation.



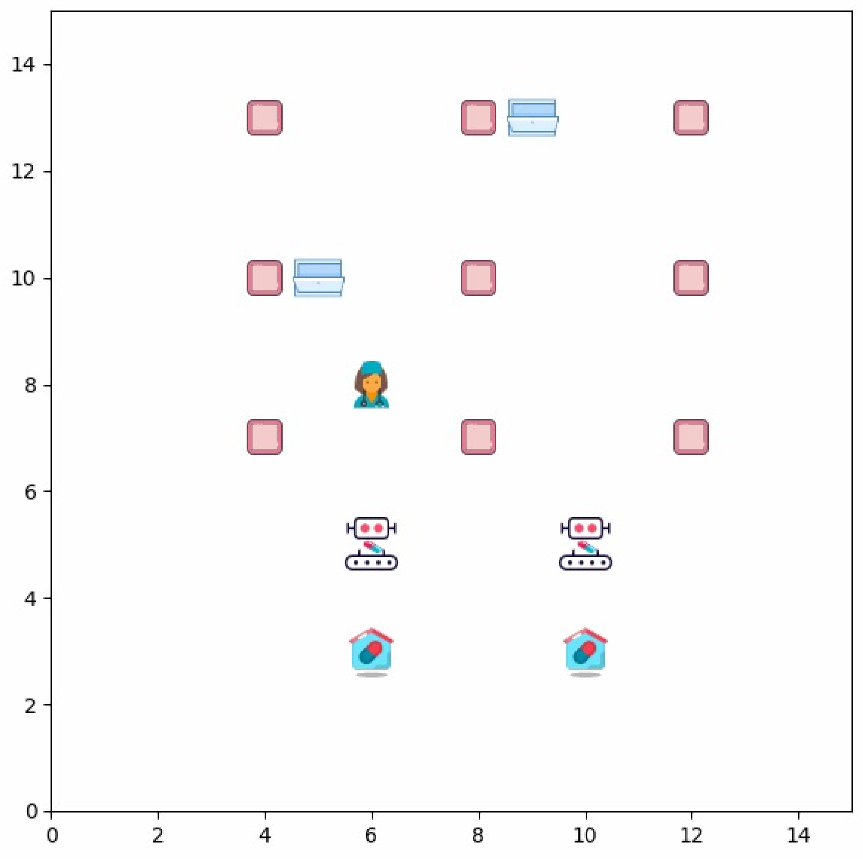
- **Grid Size**: A 15×15 grid represents the hospital environment where the RxRovers operate.
- **Rovers**: Two agents, represented by blue squares, start at predetermined positions on the grid, as shown in figure.
- **Targets**: Each Rover has a yellow target destination where it needs to deliver medicine.
- **Actions**: Rovers can move down, up, left, right, or stay still (5 possible actions).
- **Operation Desks**: Dark green squares represent operation desks that the Rovers from where they are sent with medicines to deliver at the target room.
- **Rooms**: Black squares indicate the locations of rooms that the Rovers must avoid collisions with. These are static obstacles.
- **Human**: A purple square represents a moving human obstacle that the Rovers need to avoid collision. This is a dynamic obstacle.
- **Observation Space**: This includes all the positions of the Rovers and the human on the grid along with the rooms, operation desks and the grid boundary.
- **Rewards**:
    - Rovers are penalized for actions that lead to collisions (-15) or for being too close to the human where it has to take the wait action(-5).
    - Rovers are rewarded for moving closer to their targets (+30) and reaching their destinations (+100).
    - There are penalties for moving away from the targets (-20) or moving out of grid bounds (-15).

- **Termination:** The episode ends if both Rovers reach their targets or the maximum time steps (20 by default) are reached.
- **Human Movement:** Randomly determined, adding dynamic change to the environment with each time step, this also has 5 actions to take from.

💡 **Note:** As the Timestamps are set more than the shortest path, the rewards are higher than the manually calculated values for optimal reward, however our target is to make the agents reach the targeted positions avoiding collisions which is our main agenda. With an increasing trend in the rewards graph, the rx rovers are learning to avoid the collisions with the rooms and dynamic moving human.

💡 Our implementation can be understood from

**bhanucha_charviku_rx_rover_Enhanced_Environment.ipynb**



## Algorithm 1: Q Learning (QL)

💡 **bhanucha_charviku_rx_rover_Q_Learning.ipynb**

Implemented Q-Learning for a rover agent navigating a grid environment.

- Set up Q-Learning parameters such as learning rate (alpha), discount factor (gamma), and exploration rate (epsilon). The rover's exploration rate decreases over time to balance between exploring new paths and exploiting known ones. Iterating through episodes, making decisions, updating Q-values based on rewards received.
- Plot total rewards per episode to track agent's learning progress.
- Assessed agent's performance over 10 episodes and visualized its behavior.

This approach helps the agent learn optimal actions through exploration and exploitation in the grid environment.

## Setting

- **State Representation**: The state is represented by the positions of the two rovers and the human. These positions are concatenated into a tuple to form the state space.
- **Action Space**: Each rover can take one of five actions: move up, move down, move left, move right, or stay in place. Hence, the action space is discrete with five possible actions for each rover.
- **Rewards**: The rewards are defined based on the rover's interactions:
    - A positive reward is given if the rover moves closer to its target.
    - A negative reward is given for collisions with obstacles or going out of bounds.
    - A higher positive reward is given if the rover reaches its target.
    - A negative reward is given if the rover encounters the human.
- **Done Flag**: The episode terminates when either both rovers reach their targets or the maximum number of time steps is reached.

We are training two Q-learning agents, each with its own Q-table, to learn how to navigate the rovers effectively in this environment. During training, the agents select actions based on an epsilon-greedy policy, update their Q-values using the Bellman equation, and decay their exploration rate over time.

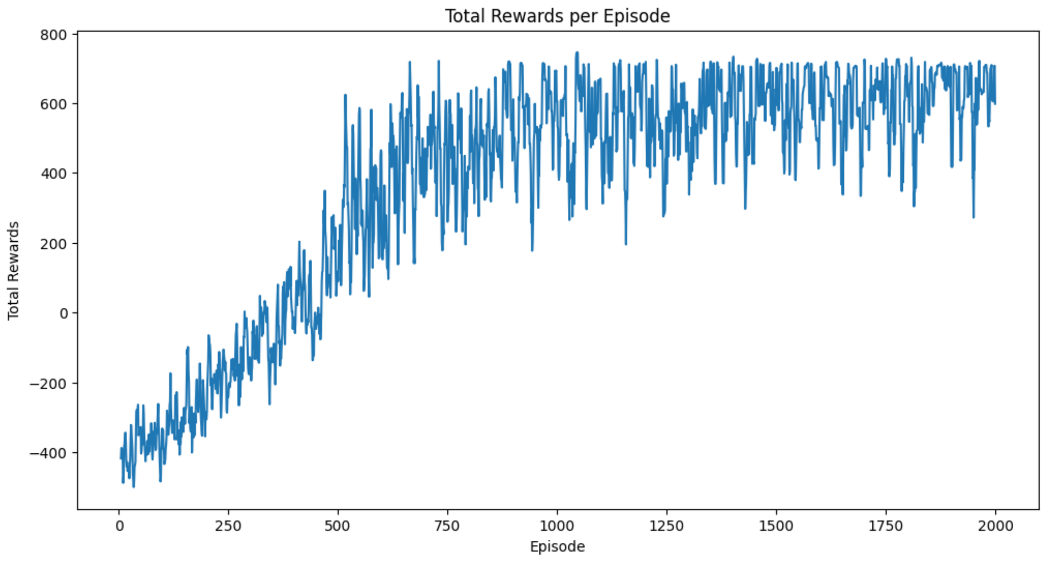
Through multiple episodes of training, the agents learn to maximize their cumulative rewards by choosing actions that lead to positive outcomes and avoiding actions that result in negative rewards or collisions. Finally, we visualize the learning progress by plotting the total rewards obtained in each episode.

## Hyperparameters:

- **Alpha (Learning Rate)**: 1e-4
- **Gamma (Discount Factor)**: 0.9
- **Epsilon (Exploration Rate)**: 0.5
- **Epsilon Decay**: 0.995
- **Epsilon Minimum**: 0.01

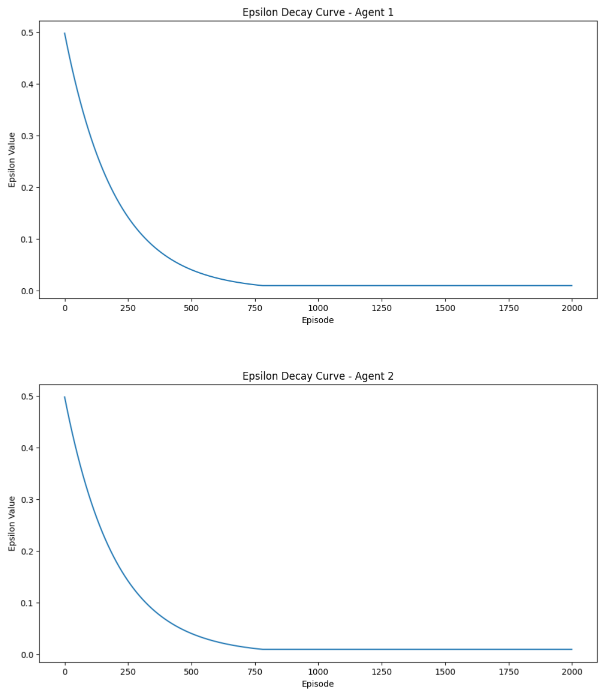
## Training Phase

1. **Total Rewards per Episode (Agent 1 and Agent 2)**:
    - **Structure**: The graphs show the total rewards per episode for each of the agents over 2000 episodes.
    - **Trend**: Initially, rewards fluctuate heavily, showing a mix of negative and positive rewards. As episodes progress, the rewards start to trend upwards, indicating an overall improvement in agent performance, demonstrate a positive trajectory, with fluctuation, but a clear upward slope.
    - **Stability**: By the later episodes, both agents have stabilized with consistent positive rewards, indicating learning and adapting to the environment effectively.



2. **Epsilon Decay Curve**:

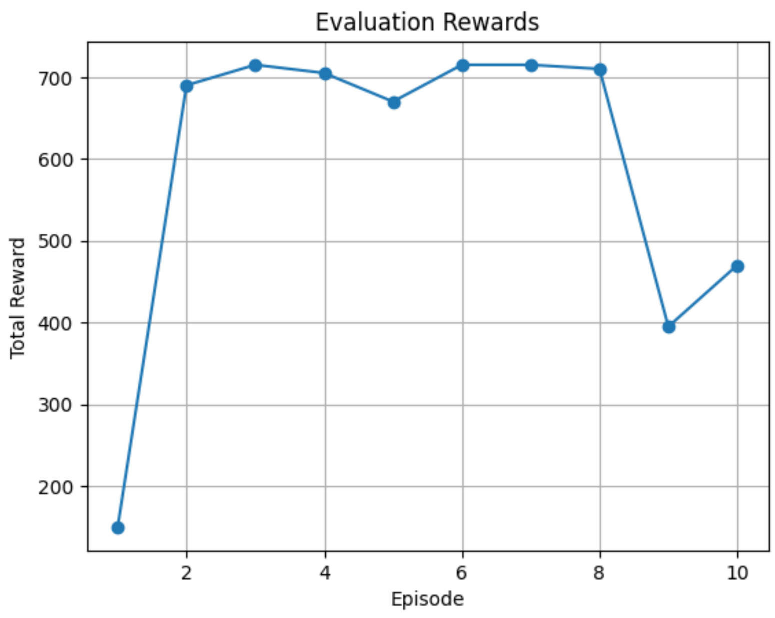
   - **Structure**: These curves depict the epsilon decay for both agents over 2000 episodes.
   - **Trend**: The curves show a gradual decrease from an initial epsilon value of 0.5, representing a balance between exploration and exploitation. By around episode 1000, epsilon values for both agents settle near zero, indicating the agents rely more on their learned policies than random exploration.
   - **Implications**: This suggests that the agents have moved from exploratory actions to utilizing their Q-table and deterministic behavior, which contributes to their stabilization and consistent rewards.



## Evaluation Phase

1. **Evaluation Rewards per Episode**:

   - **Structure**: This graph shows the evaluation rewards for both agents over 10 episodes.
   - **Trend**: The rewards start strong, peaking in early episodes and then showing some fluctuation. By the end of 10 episodes, the agents maintain substantial rewards, although with some inconsistency.
   - **Insight**: This suggests that while both agents have generally learned to navigate the environment effectively, there's room for improvement in their consistency. However, the rewards remain positive and relatively high.



## Algorithm 2: Double Q Learning (DQL)

💡 **bhanucha_charviku_rx_rover_Double_Q_Learning.ipynb**

In the Double Q-learning algorithm, we use two separate Q-tables to decouple action selection from action evaluation, which helps in mitigating overestimation biases.

- **Initialization**: Similar to Q-learning, we initialize two Q-tables (`q1_table` and `q2_table`) with zeros, each representing the expected future rewards for each action in each state.
- **Action Selection**: When selecting an action, we randomly choose between the two Q-tables (`q1_table` and `q2_table`). This randomness helps in exploration. If the random choice selects `q1_table`, we choose the action with the highest value from `q1_table`, and vice versa for `q2_table`.
- **Update Rule**: During the update step, we randomly choose one of the Q-tables and update the corresponding action-value based on the next state's maximum value from the other Q-table. This prevents the algorithm from being overly optimistic or pessimistic about the value of actions.
- **Epsilon-greedy Policy**: Similar to Q-learning, we have an epsilon-greedy policy to balance exploration and exploitation. Epsilon starts high and decays over time to gradually shift the agent's focus from exploration to exploitation.
- **Training**: During training, the agent interacts with the environment, updating its Q-tables based on observed rewards and transitions.
- **Evaluation**: After training, we evaluate the agent's performance over a certain number of episodes, recording the total rewards obtained in each episode and generating GIFs to visualize the agent's behavior.

### Setting

Same as Q learning

1. **Grid Size**: The environment is represented as a grid of size 15×15.

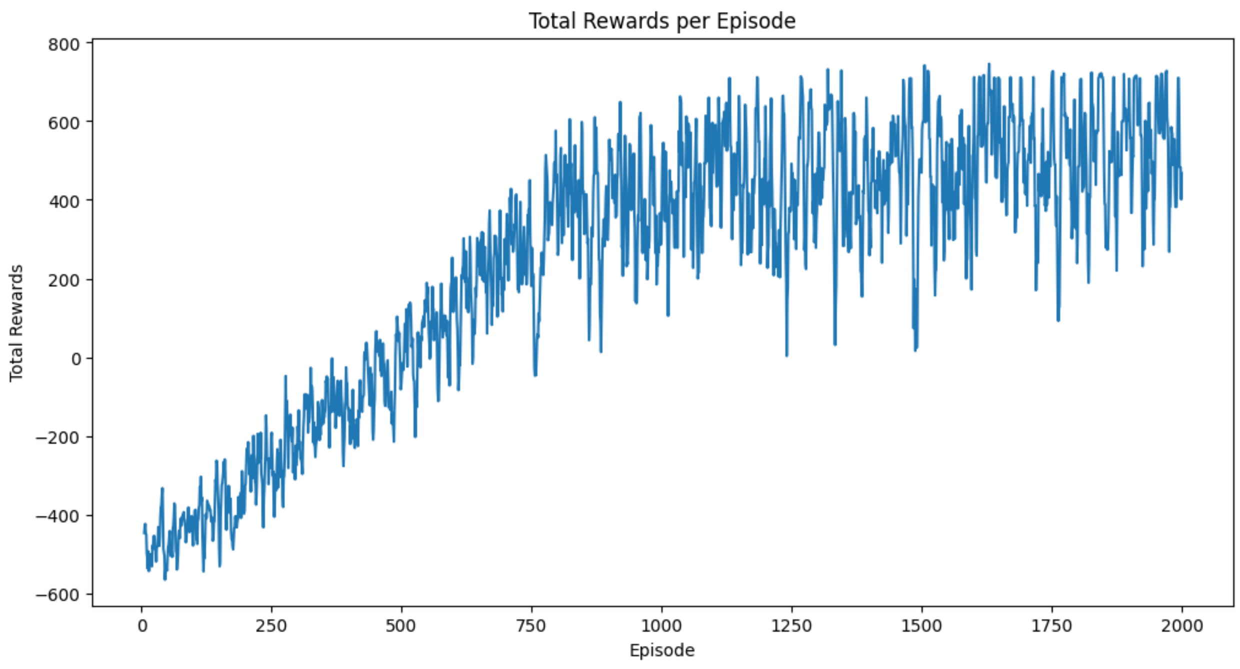
2. **Action Space**: Discrete: Agents can take one of five actions: move up, down, left, right, or stay in place.
3. **Observation Space**: Observations are represented as tuples containing the positions of both rovers and the human.

## Hyperparameters

- **Learning Rate (alpha)**: 1e-4
- **Discount Factor (gma)**: 0.9
- **Initial Epsilon (eps)**: 0.5
- **Epsilon Decay Rate (eps_decay)**: 0.995
- **Minimum Epsilon (eps_min)**: 0.01

## Training Phase

1. **Total Rewards per Episode:**
    - **Structure**: The graphs display the total rewards per episode for agents trained with both Q-learning and Double Q-learning.
    - **Double Q-learning Trend**: Similar to Q-learning, rewards start volatile. However, Double Q-learning shows a more consistent upward trend, albeit with some notable dips. By the end of 2000 episodes, the rewards stabilize in both approaches, indicating effective learning.

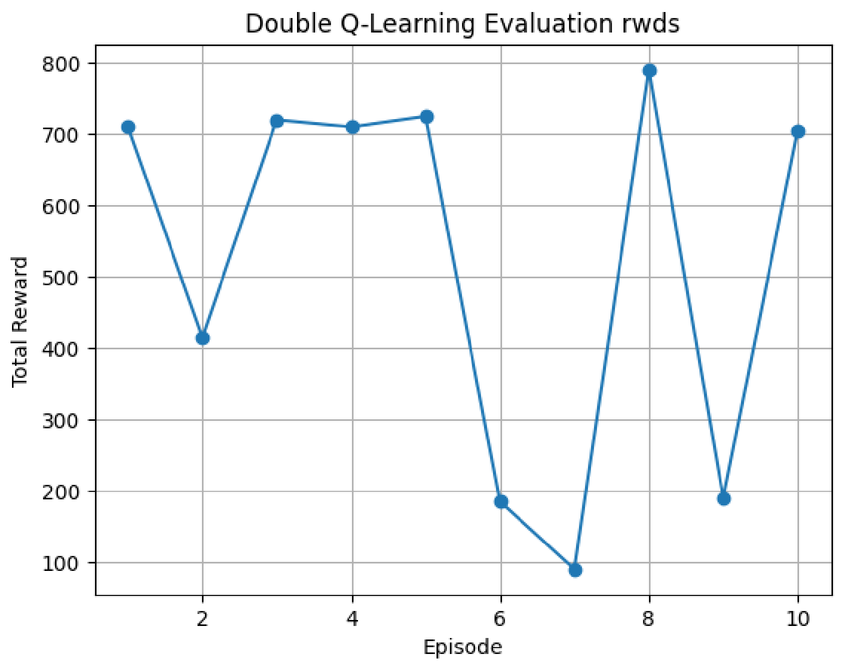


## Evaluation Phase

1. **Evaluation Rewards:**
    - **Double Q-learning Evaluation**: The evaluation rewards for Double Q-learning show a similarly erratic pattern, with sharp peaks and valleys. This suggests potential inconsistencies in the agent's learned

behavior.

- **Training Phase:** Both Q-learning and Double Q-learning show volatile reward trends that stabilize over time. The epsilon decay curves reflect a shift from exploration to policy-based actions, highlighting the agents' learning progress.
- **Evaluation Phase:** Both approaches demonstrate erratic but generally positive rewards, indicating learned behavior with room for improvement.



## Algorithm 3: Deep Q Network (DQN)

💡 **bhanucha_charviku_rx_rover_DQN.ipynb**

In this Deep Q-Network (DQN) implementation for the Rover Grid Environment, we utilize a neural network to approximate the Q-values for each state-action pair.

### Settings

1. **Neural Network Architecture (roverDQN)**:
    - Input Dimension: Determined by the observation space shape of the environment.
    - Output Dimension: Equals the number of actions in the environment = 5
    - Architecture: A feedforward neural network with two hidden layers (256 units and 128 units) followed by ReLU activation functions. The output layer has the same number of units as the action space.
2. **Replay Memory (ReplayMemory)**:
    - Utilized to store past experiences (transitions) for experience replay.
    - Implemented using a deque with a maximum capacity.

3. **Select Action Function (select_act)**:
    - Determines the agent's action based on an epsilon-greedy strategy.
    - Exploration is controlled by epsilon, which decays over time according to an exponential decay schedule.
4. **Optimize Model Function (optimize_model)**:
    - Performs one step of gradient descent on the Q-network.
    - Utilizes a target network to stabilize training.

The agent interacts with the environment for a fixed number of episodes (no_epis). During each episode, the agent selects actions based on the epsilon-greedy policy and stores the experiences (transitions) in the replay memory. The Q-network is trained using batches sampled from the replay memory. The loss is calculated using the smooth L1 loss between the predicted Q-values and the target Q-values. The target network is periodically updated to the current policy network. Epsilon decays over time to shift the agent's focus from exploration to exploitation as training progresses.

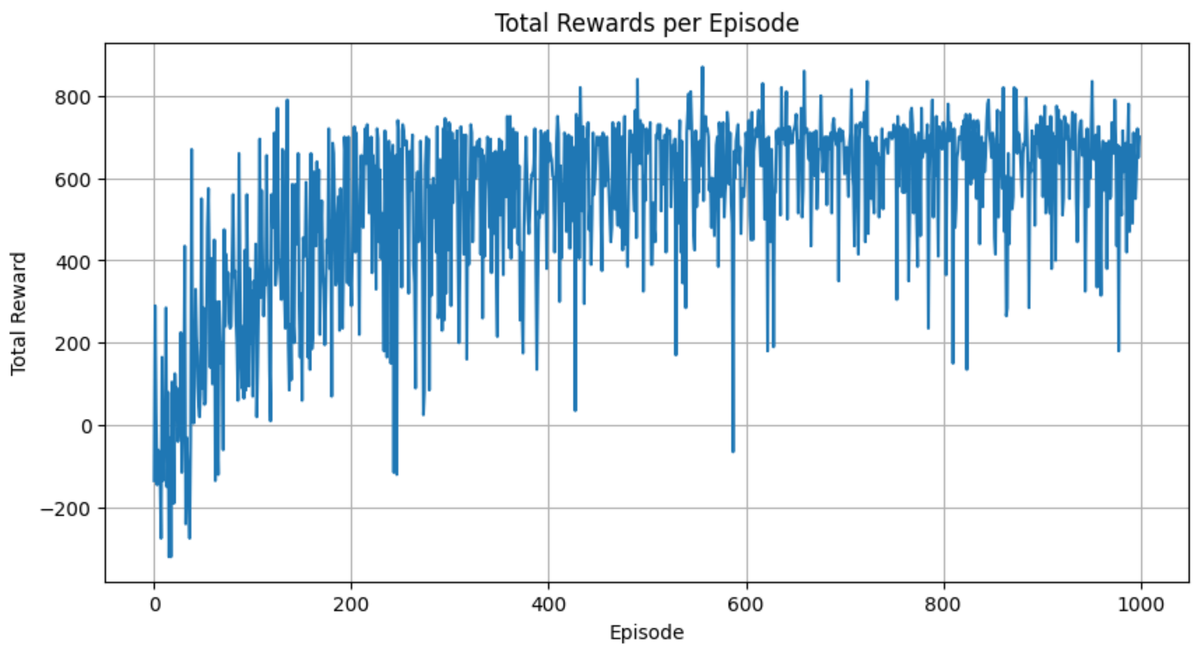
## Hyperparameters

- **Number of Episodes (no_epis)**: 1000
- **Target Network Update Frequency (target_update)**: 10 episodes
- **Batch Size (batch_sizee)**: 256
- **Discount Factor (gma)**: 0.9
- **Learning Rate (learning_r)**: 0.001
- **Epsilon Initial Value (eps_begin)**: 1
- **Epsilon Final Value (eps_end)**: 0.05
- **Epsilon Decay (eps_decay)**: 10000
- **Maximum Timestamps per Episode (max_ts)**: 30

## Training Phase

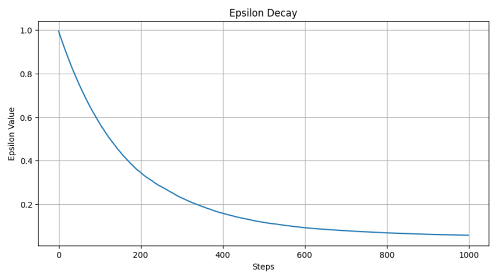
1. **Total Rewards per Episode Graph:**

    The fluctuating pattern indicates that the agents learn from the environment, experiencing both successes and failures before stabilizing their performance. The overall upward trend illustrates how the agents' strategies improve through repeated interactions with the environment.



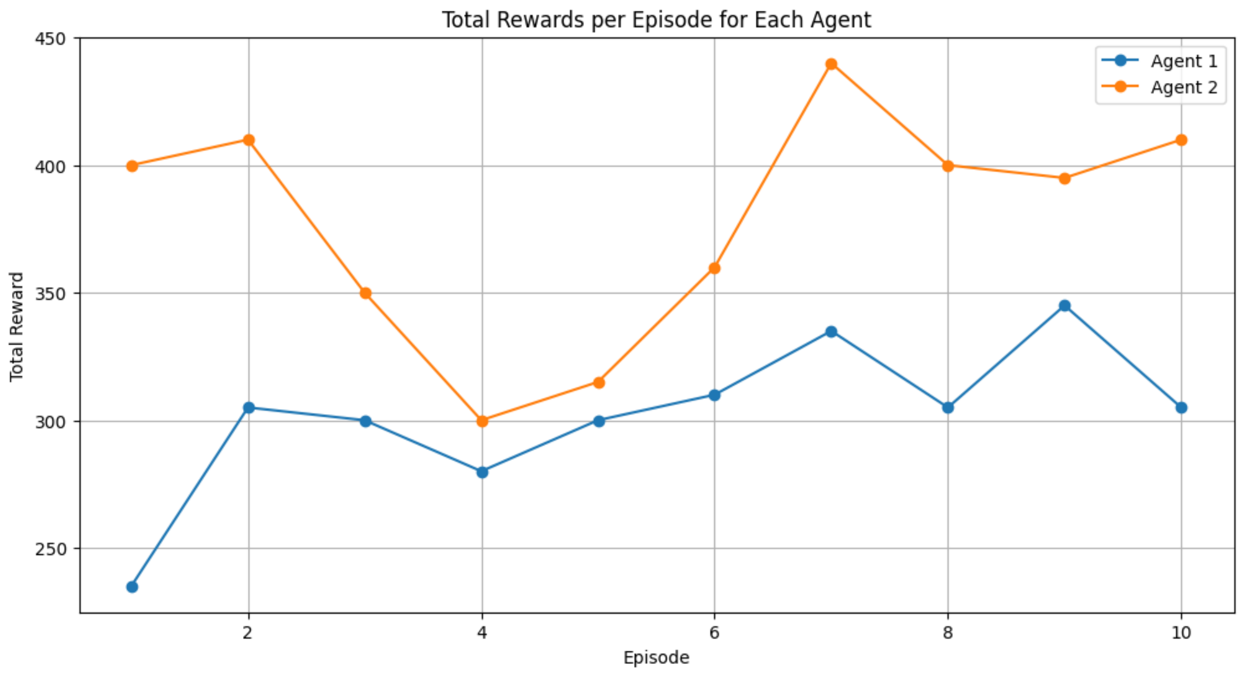
1. **Epsilon Decay Graph:**

   The epsilon value determines the exploration-exploitation balance, controlling how often agents explore new actions versus exploiting known ones. The decay indicates the agents shift from exploring to exploiting over time. The consistent decline suggests the agents gradually move towards exploitation, leveraging learned strategies for consistent performance.



## Evaluation Phase

- **Evaluation:** The agents' progress is tracked through rewards, their exploration-exploitation balance is managed by epsilon decay, and they undergo a detailed evaluation process to measure their performance and compare between agents. The plot showing both the agents getting optimal values each over all the evaluation episodes.



## Algorithm 4: Double Deep Q Network (DDQN)

💡 **bhanucha_charviku_rx_rover_DDQN.ipynb**

Double Deep Q-Networks (DDQNs) represent an evolution of the standard Deep Q-Network (DQN) architecture, addressing its tendency to overestimate action values by decoupling selection and evaluation of the action.

### Settings

The given `optimize_model_ddqn` function encapsulates the core of the DDQN update step:

1. A batch of experiences is sampled from the replay memory. Each experience contains the state, action, next state, reward, and completion flag.
2. The experiences are processed to separate non-final states and compute the expected Q-values for the next states using the target network.
3. the policy network's output is used to select the greedy action for the next state, while the target network is queried to estimate the Q-value of taking that action in the next state.
4. The loss is computed as the difference between the currently estimated Q-values from the policy network and the expected Q-values from the target network (adjusted by the reward and discounted for future rewards).
5. The policy network's weights are adjusted to minimize this loss using gradient descent, with gradients being clipped to avoid too large updates that could destabilize learning.

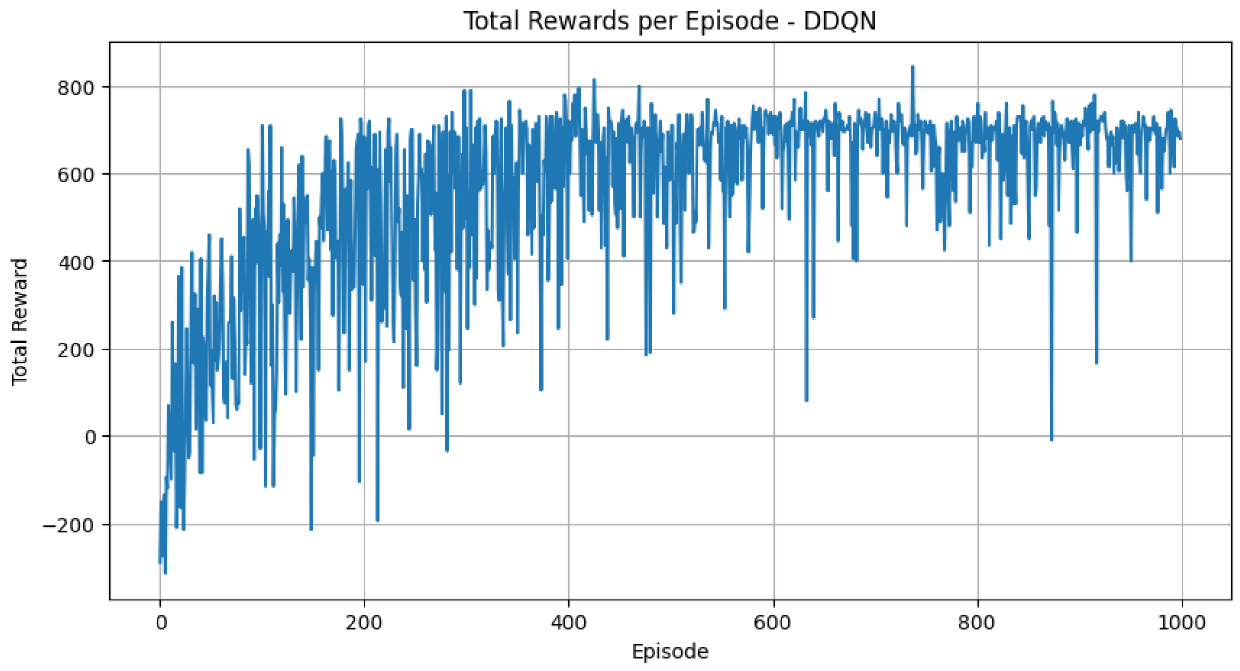
### Hyperparameters

The key hyperparameters for the DDQN training are:

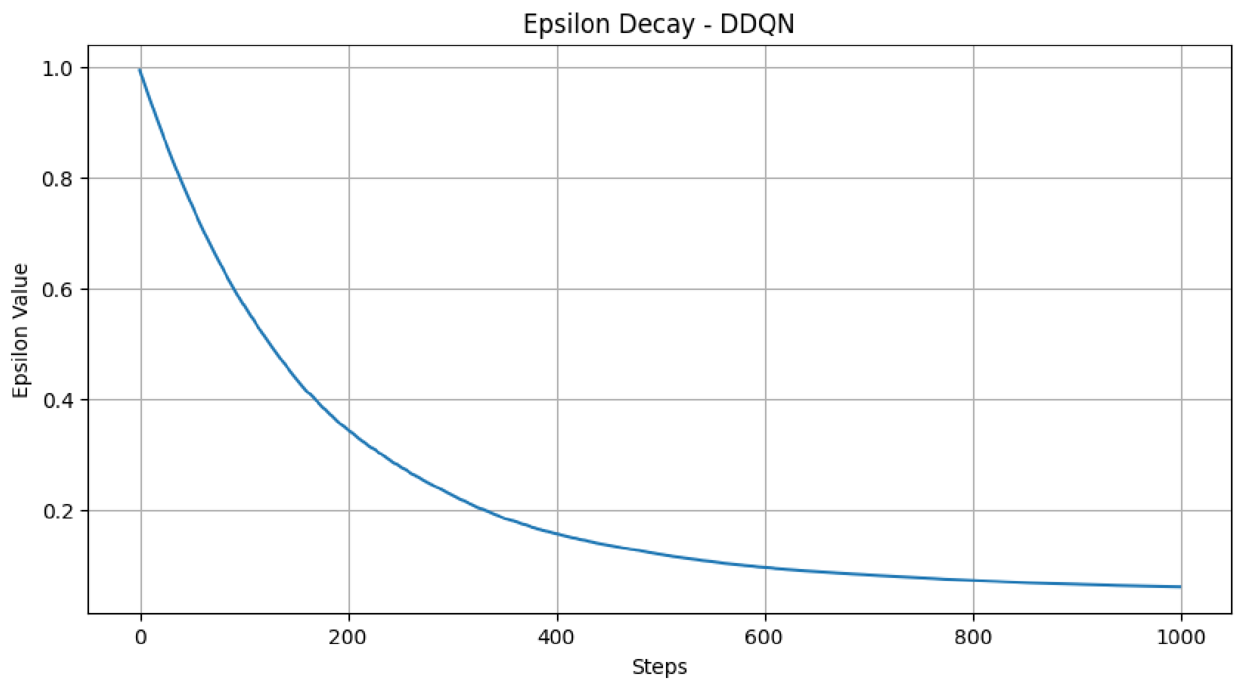
- **BATCH_SIZE**: Number of experiences used per update, 256.
- **GAMMA**: Discount factor for future rewards, 0.9.
- **LR**: Learning rate for the optimizer, 0.001.
- **EPS_START**: Initial exploration rate, 1.
- **EPS_END**: Final exploration rate, 0.05.
- **EPS_DECAY**: Rate of exploration decay, 10,000.
- **NUM_EPISODES**: Total training episodes, 1000.
- **TARGET_UPDATE**: Frequency of syncing the target network, every 10 episodes.

## Training Phase

- The Total Rewards per Episode for DDQN graph shows the agent's performance variability during training. An initial upward trend suggests learning progress, followed by stabilization with continued with very minor fluctuations in rewards. The presence of negative rewards slowly stopped as the training goes.



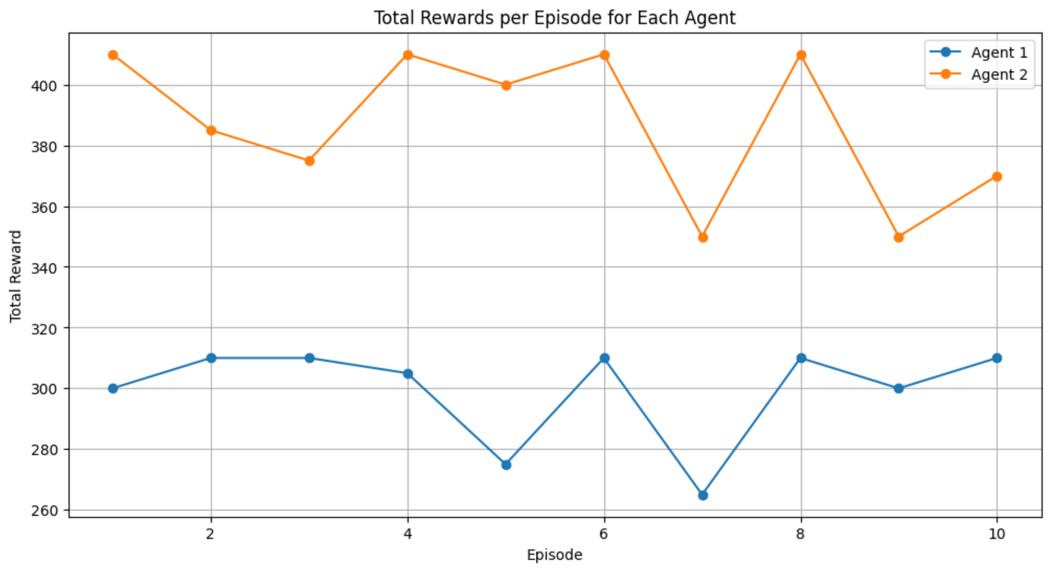
- **Epsilon Decay**: The graph shows the decay of the epsilon value over the steps.  The plot demonstrates a typical exponential decay, meaning the agent is initially more exploratory, randomly sampling the action space, and as learning progresses, it becomes more confident in its policy, thus exploiting the learned knowledge.



## Evaluation Phase

The evaluation graph depicts the total rewards per episode for two different agents (Agent 1 and Agent 2) during a DDQN training sequence.

- **Agent 1 (Blue Line)**: This agent shows it has received optimal reward values for most episodes. The overall trend suggests that Agent 1 consistently performs well.
- **Agent 2 (Orange Line)**: Agent 2's rewards are higher compared to Agent 1 as it is more far for it to reach the target position, and collect more rewards along the way. The performance is  consistent through later episodes.



## Algorithm 5: Proximal Policy Optimization (PPO)

💡 **bhanucha_charviku_rx_rover_PPO.ipynb**

Proximal Policy Optimization (PPO) is a robust and efficient algorithm developed by OpenAI. PPO strikes a balance between policy optimization and stability, offering an effective approach to training complex models.

### Settings

The PPO model is implemented with a two-layer feedforward neural network:

- The *actor* head outputs policy logits, from which a Categorical distribution is created to sample actions.
- The *critic* head outputs a scalar value indicating the value function estimate.
- **Training Loop**: The training process consists of several key steps:
    - States, actions, rewards, values, and log-probs are collected in a replay buffer.
    - Batches of transitions are sampled from the buffer, and advantages are computed.
    - The model is trained in epochs, with separate losses computed for the policy and value function.
- **Loss Functions**: The total loss combines policy and value losses:
    - *Policy Loss*: Calculated using the surrogate objective, with clipping to stabilize updates.
    - *Value Loss*: Mean Squared Error between critic estimates and discounted rewards.

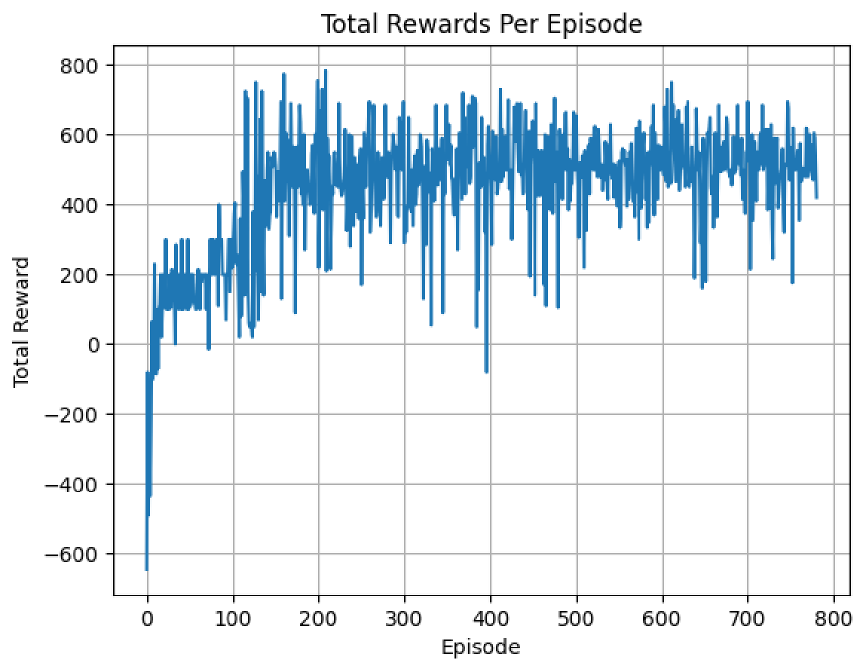
### Hyperparameters

The key hyperparameters for the PPO training are:

- **Total Timesteps(**50,000): The total number of timesteps for training.
- **Gamma**: 0.99: The discount factor for future rewards.
- **Lambda_**: 0.95: The Generalized Advantage Estimation (GAE) parameter for balancing bias and variance in advantage computation.
- **Epsilon**: 0.2: The clipping threshold for the surrogate objective, preventing large policy updates.
- **Epochs**: 3: The number of epochs per training iteration, during which the model is optimized on each sampled batch.
- **Batch Size**: 64: The size of batches sampled from the replay memory for each training iteration.
- **Learning Rate**: 0.001: The learning rate for the Adam optimizer used in training.

### Training Phase

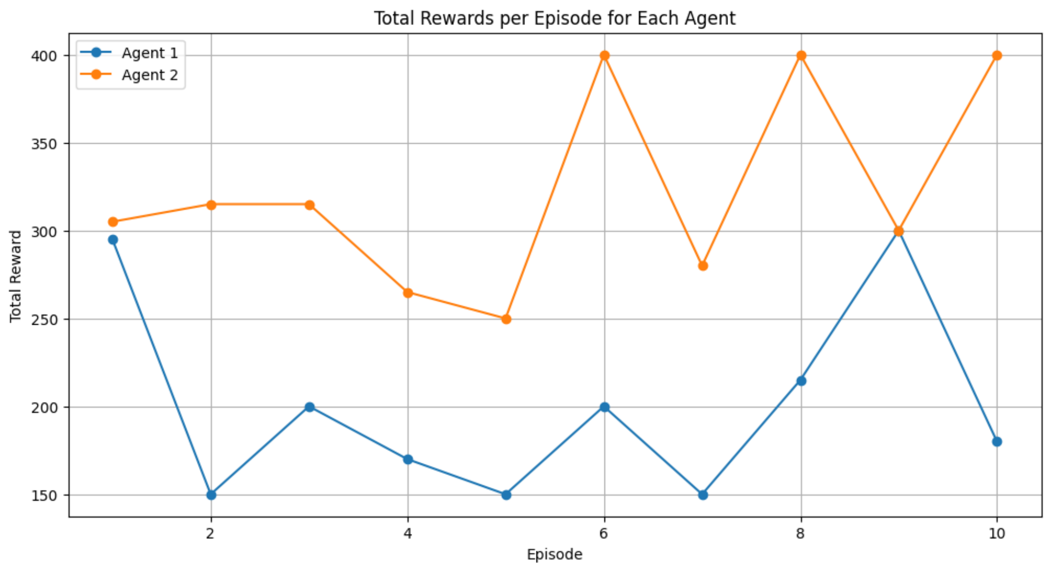
- The Total Rewards per Episode for PPO graph shows the agent's performance variability during training. An initial upward trend suggests learning progress, followed by stabilization with continued with very minor fluctuations in rewards. The presence of negative rewards slowly stopped as the training goes. Around episode 50 to 100, the rewards show a noticeable upward trajectory, indicating that the model is learning to take more rewarding actions.



### Evaluation Phase

The evaluation graph depicts the total rewards per episode for two different agents (Agent 1 and Agent 2) during a PPO training sequence.

Both agents have consistently achieved their optimal rewards. Agent 1 (Blue curve) being closer to the target position collected comparatively less rewards hence the differentiation. The Agent 2 (orange line) dis receiving optimal rewards based on dynamic obstacles coming on the way.



## Algorithm 6: Actor Critic (A2C)

💡 **bhanucha_charviku_rx_rover_A2C.ipynb**

A2C is a reinforcement learning algorithm that combines elements of both policy-based methods (Actor) and value-based methods (Critic). It maintains two networks: an actor network that learns a policy, and a critic network that estimates the value function.

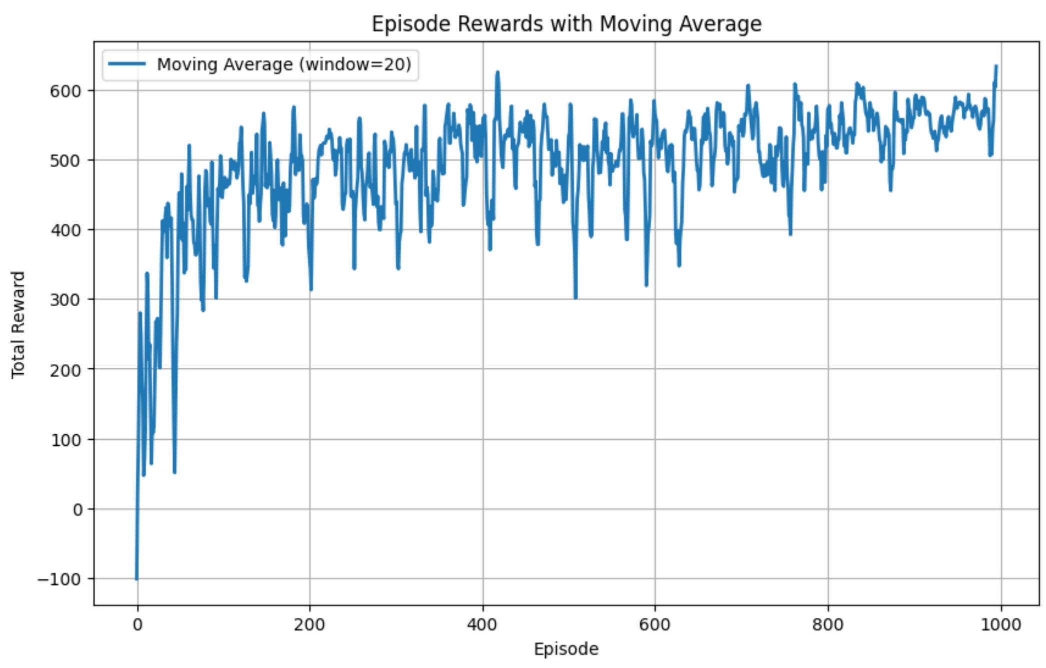
## Settings

Key Components:

1. **Actor Network**: The actor network learns a policy π(s) that selects actions based on the current state. In the context of the Rover Grid Environment, the actor network takes the state (observation) as input and outputs a probability distribution over actions for each agent.
2. **Critic Network**: The critic network learns the value function V(s), which estimates the expected return (total future reward) from a given state. In the Rover Grid Environment, the critic network takes the state as input and outputs a value estimate for each agent.
3. **Advantage**: The advantage function A(s, a) measures how much better or worse an action is compared to the average action. It is calculated as the difference between the observed reward and the expected reward (value) from the critic network.
4. **Policy Update**: The policy (actor) is updated using the advantage function to encourage actions that lead to higher rewards. The advantage function helps in determining whether an action was better or worse than expected, allowing the policy to be updated accordingly.
5. **Value Function Update**: The value function (critic) is updated to better estimate the expected return from a given state. It helps in reducing the error between the predicted value and the observed return.
6. **Network Architecture**: In the given implementation, both the actor and critic networks are implemented using neural networks. The actor network outputs a softmax distribution over actions, while the critic network outputs a single value.
7. **Advantage Calculation**: The advantage is calculated using the observed rewards and the estimated values from the critic network. It helps in determining the quality of actions taken by the agent.
8. **Policy and Value Function Updates**: The actor and critic networks are updated using gradient descent. The actor is updated to maximize the advantage-weighted log probabilities of actions, while the critic is updated to minimize the mean squared error between predicted values and observed returns.

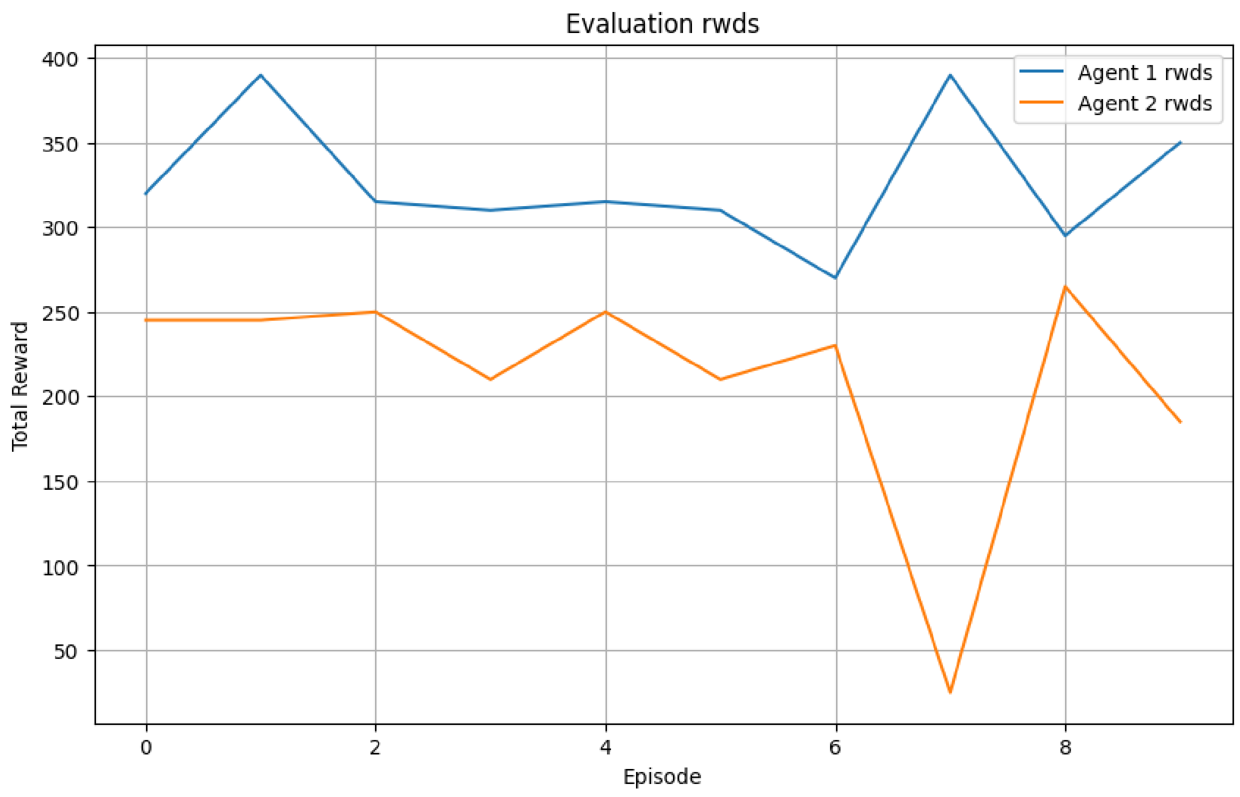
## Training Phase

- The graph shows an initial upward trend, with rewards increasing significantly over the first few hundred episodes, indicating the agents learn to optimize their strategies effectively. Beyond this initial phase, rewards stabilize and exhibit some oscillation around a steady level.
- The moving average (with a window of 20) illustrates a clear upward trend, smoothing out the episodic fluctuations. This indicates a general progression in the agents' performance over time.
- The training phase graph demonstrates that the agents learn to perform better in the RoverGridEnv environment over time, achieving higher and more consistent rewards as they progress.



## Evaluation Phase

- **Agent 1 vs. Agent 2:**
    - **Agent 1:** Initially achieves higher rewards than Agent 2, ranging between 300 and 400.
    - **Agent 2:** Starts strong, and stabilizes around 250 rewards.
- The graph shows fluctuation in the rewards for both agents, with Agent 1 experiencing more significant oscillations than Agent 2. The evaluation graph highlights differences in strategies and performance between the two agents. While Agent 1 generally outperforms Agent 2, the difference is consistent in both agents.



## Comparison

Comparing different reinforcement learning algorithms and how they performed is a bit challenging to say as due to the variety of factors involved in their design, implementation, and performance:

**General Comparison:**

| Aspect | Q-Learning | Double Q-Learning | DQN | DDQN | PPO | A2C |
|---|---|---|---|---|---|---|
| Algorithm Type | Model-free, Value-based | Model-free, Value-based | Value-based, Deep NN | Value-based, Deep NN | Actor-Critic, Policy Gradient | Actor-Critic, Policy Gradient |
| Exploration-Exploitation | Epsilon-greedy | Epsilon-greedy | Epsilon-greedy | Epsilon-greedy | Continuous | Continuous |
| Stability & Learning Rate | No target network | No target network | Target network | Double updates, target network | Adaptive, Gradient Clipping | Adaptive, Entropy Bonus |
| Convergence | Slow | Moderate | Fast | Faster | Fast | Moderate |
| Memory Requirements | Low | Low | High | High | Moderate | Moderate |
| Adaptbility | Moderate | Moderate | Low to Moderate | Low to Moderate | High | Moderate |
| Generalization | Low to Moderate | Low to Moderate | Low | Low | High | Moderate |

**Performance:**

- **QL:** Slow initial learning, had to train the model for 2000 episodes compared to others for convergence and to learn optimal policy. And while evaluation results were much more consistent compared to double q learning
- **Double QL:** Faster initially learning, but fluctuated more while learning compared to Q learning. May be due to over estimations in between. While evaluating there was sometimes a dip in rewards earned.
- **DQN:** Steady increase in rewards during training, stabilizing around 500 episodes, with some fluctuation. this fluctuates more compared to double dqn. For evaluation it has learned and is consistently taking optimal rewards.
- **DDQN [BEST]:** Rapid learning, stabilizing earlier than DQN and maintaining higher rewards, reducing overestimation. Evaluations all the 10 episodes are perfectly learning and choosing the best path and avoiding obstacles and optimal rewards.
- **PPO:** Rapid increase, stabilizing after 100 episodes, with consistent performance, though with some fluctuations. Its learnt too early. and evaluation showed higher fluctuations.
- **A2C:** Steady increase but higher fluctuation around 200 episodes, with a moving average indicating consistent performance. evaluations both agents taking optimal rewards.

## Challenges Addressed

In our RxRovers project, we faced several  challenges while aiming to revolutionize medicine delivery in hospitals using reinforcement learning.

- **Navigating a Dynamic World:** Hospitals are lively places. People and machines move around all the time, making navigation tricky. Our RxRovers had to learn to dodge these moving obstacles smoothly, requiring us to invent ways for them to understand and react to this ever-changing environment.

  Approach Taken: By designing proper reward structure for taking wait action around dynamic obstacles at the same time get penalized for waiting, so that it will choose to move forward.
- **Custom Paths for Diverse Layouts:** No two hospitals are the same, and our rovers needed to find their way in each unique setting. This means it requires prior training of the hospital map for creating a smart system that could learn to figure out the best routes on its own.

  Solution: Our environment can be easily updated to accomodate various settings. The reward actions and implementation can be extended to other hospital scenarios.
- **Learning Optimal Path:** The agents have to take the shortest route to the delivery rooms as specified by its targets by simultaneously avoiding obstacles.

  We are making sure the agent is learning the shortest distance based on euclidean distance to the target and reward dynamics handle the approach.
- By carefully adjusting the learning parameters, especially reward structure and for the advanced reinforcement learning techniques used: DQN and DDQN, PPO, A2C, we enhanced our RxRovers' decision-making skills, ensuring they're both fast and reliable.
- They are successfully reaching their targets following an optimized path at the same time avoiding the dynamic and static obstacles collisions. When the human is around the corners of the agent, it chooses the wait action, it would continue to wait longer if given positive rewards for not colliding

## BONUS

The "RxRovers: Roaming for Rapid Relief" is a real-world application of reinforcement learning (RL),

### Real-World Application [5 points]:

1. **Healthcare System Integration:** Our project directly addresses a real-world healthcare problem: delivering medical supplies efficiently within hospital environments. The efficient distribution of medical supplies, including medication, equipment, and resources, is crucial in hospital settings. This project integrates RL into the healthcare domain to enhance this delivery system, directly benefiting patient care and outcomes.
2. **Simulating Hospital Layouts:** We creates a simulated hospital environment that mirrors real-world scenarios. The environment includes features such as:
    - **Hospital Corridors:** Representing a real-world hospital layout with multiple corridors, rooms, and operation desks.
    - **Obstacles:** Incorporates dynamic obstacles (humans, other rovers) and static obstacles (rooms, walls), accurately reflecting the unpredictable nature of hospital environments.
    - **RxRovers:** Representing autonomous agents, programmed to navigate these environments, avoid obstacles, and reach designated destinations.
3. **Complex Navigation Challenges:** The project's challenges mirror real-world navigation complexities in hospital settings:
    - **Dynamic and Static Obstacles:** The rovers navigate through hospital corridors, avoiding dynamic (moving humans) and static obstacles (rooms, walls), accurately representing real-world scenarios.
    - **Path Planning:** The project optimizes path planning to ensure timely and efficient delivery of medical supplies, which is a critical factor in healthcare delivery.
4. **Adaptable Environment:** The project's environment can be modified to reflect various real-world hospital layouts, demonstrating its adaptability to different healthcare settings.

## REFERENCES

[1]

Deep reinforcement learning DQN for multi-agent environment
AI Bots
https://medium.com/yellowme/deep-reinforcement-learning-dqn-for-multi-agent-environment-5f4fae1a9ff5

[2] https://github.com/LyapunovJingci/Warehouse_Robot_Path_Planning

- https://github.com/atb033/multi_agent_path_planning

Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.2.1+cu121 documentation
This tutorial shows how to use PyTorch to train a Deep Q Learning (DQN) agent
on the CartPole-v1 task from Gymnasium.
https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Deep Q Learning (DQN) using Pytorch

Reinforcement Learning

https://medium.com/@vignesh.g1609/deep-q-learning-dqn-using-pytorch-a31f02a910ac