Warehouse Robo Using Q-Learning and DQN

Arjun Goyal
School of Computer Science and Engineering, Vellore Institute of Technology - AP,
Amravati, Andhra Pradesh, India
arjun.22bce9036@vitapstudent.ac.in

Abstract- *This study presents a detailed comparison between Q-Learning and Deep Q-Networks (DQN) in the context of robotic navigation within warehouse environments. The Q-Learning technique, applied to a static grid-based map, showed rapid convergence and clarity in state-action mappings. In contrast, the DQN model leveraged a neural network to predict Q-values from more complex, dynamic warehouse layouts structured via CSV input files. The results highlight Q-Learning's effectiveness in simpler, well-defined settings, while DQN proves advantageous in handling higher-dimensional and adaptive tasks.*

Keywords— *Deep Q-Networks, Q-Learning, Reinforcement Learning, Warehouse Robotics, Pathfinding*

I. Introduction

Reinforcement Learning (RL) is a machine learning paradigm where agents learn optimal actions by interacting with an environment to maximize cumulative rewards. This project applies RL algorithms to guide a robot through warehouse-like grid environments containing obstacles, with the goal of reaching a predefined destination efficiently.

Two distinct RL strategies were implemented: Q-Learning and Deep Q-Networks (DQN). The Q-Learning approach utilized a table to store and update Q-values for each state-action combination, making it ideal for simple, fixed-size grid maps. It offers transparency in decision-making and requires relatively low computational resources.

In contrast, DQN employed a neural network to approximate Q-values, allowing it to generalize across a wider range of scenarios. Environments were loaded dynamically from CSV files, enabling the DQN agent to adapt its policy to varying map structures. This flexibility makes DQN better suited for more complex, real-world warehouse navigation problems.

This work offers a comparative assessment of both approaches, examining their strengths, limitations, learning performance, and suitability across static and dynamic pathfinding tasks.

II. Ease of Use

A. *Choosing the Right Learning Method*

Selecting a reinforcement learning strategy depends on environmental complexity and available computational resources. In this project, both Q-Learning and DQN were explored for solving robotic path planning within a warehouse setting.

Q-Learning is straightforward to implement and ideal for smaller grids with discrete state spaces. It relies on a table-based approach to iteratively update values based on agent interactions. However, its scalability is limited when applied to environments with continuous or large-scale states.

DQN, on the other hand, merges deep learning with reinforcement principles. By using a neural network to estimate Q-values, it can operate effectively in complex or variable environments. Our implementation involved training the DQN agent on map configurations imported from CSV files, enhancing its generalizability across different layouts.

B. *Preserving Learning Stability and Method Integrity*

Each method requires tailored components to ensure learning remains stable and effective. For Q-Learning, it is important to maintain a clear reward structure, manage the exploration-exploitation trade-off using strategies like ε-greedy, and monitor convergence.

DQN demands a more intricate setup. Stability is maintained through experience replay buffers that decorrelate learning samples, target networks that prevent unstable updates, and well-chosen loss functions—such as mean squared error—for training. Additionally, model performance depends heavily on the choice of hyperparameters, including the learning rate and discount factor.

By following best practices for both methods, this project ensured consistent, interpretable results during robotic navigation tasks within a warehouse-like environment.

III. PREPARING YOUR MODEL BEFORE TRAINING

A. *Data Preprocessing and State Representation*

Before training either Q-Learning or DQN models, proper state representation and preprocessing are critical:

For Q-Learning:

- **Tabular State Encoding**:
    - States are discrete grid coordinates (e.g., (x, y) positions in a warehouse).
    - One-hot encoding may be used for categorical states (e.g., "obstacle present" = [1, 0], "clear" = [0, 1]).
- **Normalization**: Not required for tabular methods but ensures consistency if combining with neural networks.

For DQN:

- **High-Dimensional Inputs**:

- Raw pixel data from cameras or LiDAR scans require flattening or convolutional layers.
- Normalize inputs to [0, 1] or [-1, 1] for neural network stability.

- **Feature Engineering**:
  - Use edge detection or dimensionality reduction (PCA) for complex sensor data.

Key Rule:

- Keep raw data (e.g., grid maps, sensor logs) separate from preprocessed training data.
- Avoid redundant transformations (e.g., normalizing already normalized states).

B. Hyperparameter Tuning

- Reward shaping is essential in Q-learning and Deep Q-Network (DQN) to guide agents toward optimal behavior. Sparse rewards may slow learning, while overly dense or poorly designed rewards can lead to suboptimal policy learning.
- For Q-learning, using a carefully designed reward structure is important since it directly affects the Q-value updates. Misleading rewards can cause convergence to incorrect policies.
- For DQN, reward clipping (e.g.,

  $$R_t = \max(\min(R_t, 1), -1))$$

  helps stabilize learning by avoiding large Q-value updates and controlling gradients during neural network training.
- Evaluation Metrics: Commonly used metrics include cumulative reward, learning/convergence time, average reward per episode, and success rate to monitor performance during training and testing.

C. Ensuring Model Stability and Effective Update Mechanisms

Maintaining stable training dynamics is critical for both Q-Learning and Deep Q-Network (DQN) approaches. This section highlights the mechanisms and techniques employed to ensure robustness, as well as common challenges encountered during the learning process.

- Q-Learning Stability Strategies
  - Stability in Q-learning is enhanced by gradually decreasing the **learning rate (α)** and **exploration rate (ε)** over time. This helps the agent shift from exploration to exploitation as it gains experience.
  - Successful convergence depends on tuning the **discount factor (γ)** and learning rate carefully. Excessively high or low values may lead to instability or delayed learning.
- Stabilization Techniques for DQN
  - DQN requires additional components to ensure reliable learning. One critical method is the use of a **target network**, which is updated periodically to reduce oscillations during value updates.
  - The target Q-value is computed using the formula:

    $$Q_{target}(s, a) = r + \gamma \, a' \max Q(s', a')$$

    where:
    - Q_{target} is the target Q-value,
    - s' is the next state,
    - a' is the best action in that state.

- The **experience replay buffer** stores previous interactions and samples them randomly to minimize correlations between sequential experiences, thus improving generalization and training stability.
- It's crucial to maintain **consistent input formats** across episodes. Changes in state representation (e.g., dimension mismatch or inconsistent preprocessing) can significantly disrupt learning.

D. Common Pitfalls in Model Training

- **Overfitting**:
  - In **Q-learning**, overfitting can arise when small state spaces are over explored without sufficient randomness, causing the agent to learn rigid behaviors that don't generalize.
  - In **DQN**, excessive reliance on a static replay buffer or insufficient exploration may cause the neural network to memorize patterns, reducing adaptability to unseen environments.
- **Reward scaling** Pitfalls:
  - **Policy collapse** in DQN, where the agent repeatedly selects the same actions regardless of state context.
  - **Distorted Q-value updates** in Q-learning, especially when the reward magnitude overwhelms or undervalues the impact of the state transitions.
- **Handling unfamiliar states**:
  - For both Q-learning and DQN, ensure fallback strategies (e.g., default Q-values, exploration) when encountering new states not yet represented in the Q-table or training data.
- **Efficiency Consideration**:
  - **Q-learning** suffers from inefficiency in large or continuous environments due to the exponential increase in state-action pairs stored in the Q-table.
  - **DQN** training can slow down with large replay buffers and frequent target network updates; efficient sampling strategies and buffer size management help mitigate this.

IV. Finalizing the Model Implementation

After establishing and training both the **Q-Learning** and **Deep Q-Network (DQN)** agents within the notebook, the final phase involves solidifying the model structure, tracking experiments, and preparing the system for evaluation and deployment. This section outlines implementation practices, performance visualization techniques, and best practices that improve reproducibility, scalability, and real-world applicability.

A. *Model Implementation and Experiment Tracking*

To ensure reproducibility and modularity, the Jupyter Notebook has been structured into clearly defined sections:

- **Environment Setup**: Using OpenAI Gym, the environment is initialized with deterministic or stochastic settings.
- **Q-Learning Implementation**: A tabular Q-table is defined and updated using the Bellman equation. Hyperparameters such as learning rate (α), discount factor (γ), and exploration decay (ε) are clearly defined and tested.
- **Deep Q-Network (DQN) Implementation**: The neural network architecture is designed using PyTorch, consisting of an input layer matching state dimensions, two hidden layers (128 units), and an output layer corresponding to possible actions. The model is trained using experience replay, epsilon-greedy policy, and a separate target network updated periodically to stabilize training.

Tracking and Logging Experiments :

To understand training dynamics and performance over time:

- Reward progression is tracked after each episode and plotted using matplotlib.
- Training parameters such as total episodes, learning rate, discount factor, and exploration decay are easily modifiable for experimentation.

B. *Organizing Code and Documentation*

Efficient code organization plays a crucial role in simplifying both training and debugging phases. To maintain clarity and enhance collaboration or reproducibility, code and documentation should follow a structured format.

- **Logical Segmentation**: The implementation is separated into modular blocks based on functionality – environment setup, Q-Learning module, DQN architecture, training routines, and result visualization.
- **Model-Specific Documentation**:
  - **Q-Learning**:
    - Uses a simple dictionary-based Q-table for tabular learning.
    - Q-value updates follow the formula:

$$Q(s, a) = Q(s, a) + \alpha \,.\, (r + \gamma \,.\, \max(Q(s', a')) - Q(s, a))$$

  - **DQN**:
    - Uses neural networks for function approximation.
    - DQN stores agent experiences in a replay memory, and updates are performed by sampling mini-batches from this memory.
    - Periodically updates a **target network** to stabilize training.
- **Training Pipeline**:
  - Q-Learning updates values online after every step.
  - DQN stores agent experiences in a replay memory, and updates are performed by sampling mini-batches from this memory
  - Both agents use an **epsilon-greedy exploration strategy** to trade off between trying new actions and exploiting known good ones.
- **Evaluation Metrics**:
  - **Cumulative Reward** over episodes to assess learning efficiency.
  - **Convergence Time** (number of episodes until the agent achieves consistent success).
  - **Episode Success Rate**, especially for deterministic or goal-based environments.

C. *Visualizing Model Performance*

a) *Reward Graphs and Performance Charts*

- The matplotlib library is used to plot the **reward trend over time**, revealing how both agents learn.
- X-axis represents the number of episodes, and the Y-axis shows the total reward received in each episode.
- These plots help identify issues such as non-convergence, reward sparsity, or unstable updates.

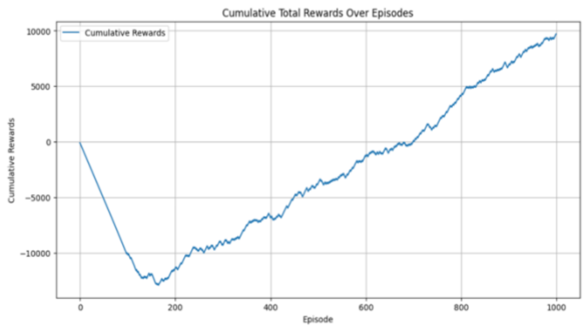


Fig. 1. Cumulative rewards over Episodes using Q-learning

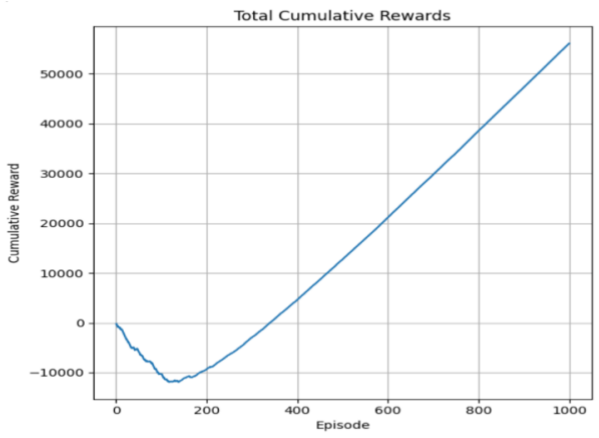


Fig. 2. Cumulative rewards over Episodes using DQN

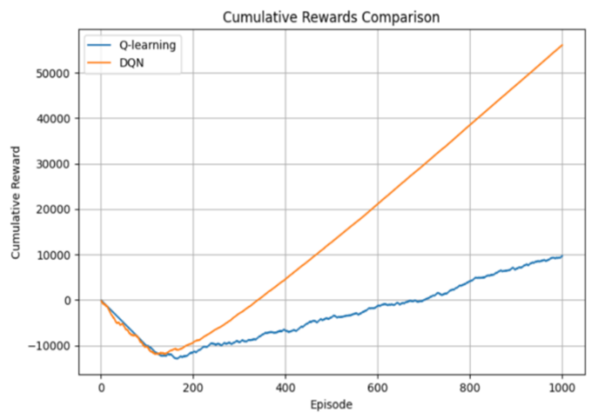


Fig. 3. Comparison of Q-learning and DQN over cumulative rewards

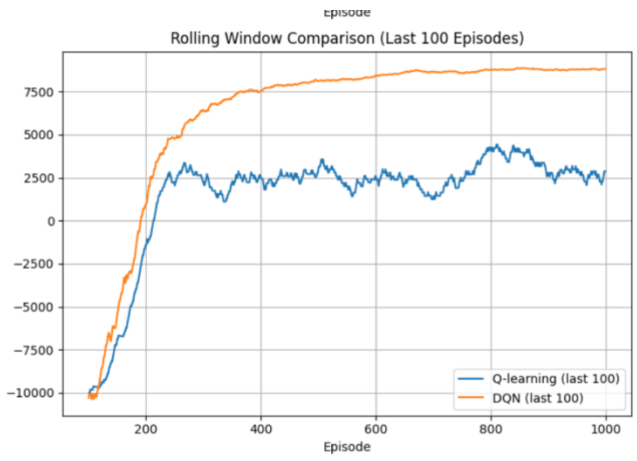
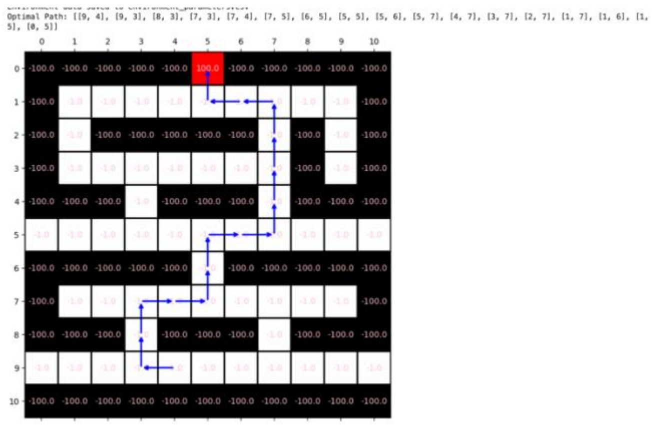


Fig. 4. Comparison of Q-learning and DQN over Rolling window Reward



```
Optimal Path: [[9, 4], [9, 3], [8, 3], [7, 3], [7, 4], [7, 5], [6, 5], [5, 5], [5, 6], [5, 7], [4, 7], [3, 7], [2, 7], [1, 7], [1, 6], [1, 5], [0, 5]]
```

Fig. 5. Optimal Path

b) Tables and Captions for Reporting

To compare model efficiency, training time, and complexity:

TABLE I → COMPARATIVE ANALYSIS OF Q-LEARNING AND DQN

Model Type	Training Time	Key Characteristics	Computational Complexity
DQN	2-5 min	Neural network (128 hidden units), experience replay, target network	O(3,28,000) ops
Q - Learning	30-90 sec	Tabular method, direct Q-table updates	O(11,000) ops

Acknowledgment

We express our sincere gratitude to the open-source community and developers behind **OpenAI Gym**, whose environments enabled rigorous testing of reinforcement learning agents. Special thanks to the contributors of Q-Learning and Deep Q-Learning research whose foundational concepts have guided this implementation. This project also benefits from numerous community tutorials and forums that provided support during the debugging and evaluation stages.

References

[1] V. Mnih et al., "Achieving human-level performance in control tasks via deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] R. Sutton and A. Barto, *Introduction to Reinforcement Learning*, 2nd ed., MIT Press, 2018.

[3] A. Gershman, "A review of reinforcement learning integrated with memory-based approaches," *Current Opinion in Behavioral Sciences*, vol. 38, pp. 1–7, 2021.

[4] J. H. Joseph, "Real-time decisions in dynamic settings using instance-based learning," unpublished manuscript.

[5] H. Van Hasselt, "Using Double Q-learning to address overestimation bias," in *Neural Information Processing Systems (NeurIPS)*, 2010.

[6] C. Watkins and P. Dayan, "Foundational work on Q-learning algorithms," *Machine Learning*, vol. 8, no. 3–4, pp. 279–292, 1992.

[7] J. D. Kelly, P. Gmytrasiewicz, and S. Singh, "Combining reinforcement learning with instance-based methods," *Journal of Artificial Intelligence Research*, vol. 27, pp. 1–25, 2006.

[8] M. Bowling and M. Veloso, "Stochastic game theory for learning in multi-agent systems," *Artificial Intelligence*, vol. 150, no. 1, pp. 1–30, 2003.

[9] D. Kingma and J. Ba, "Adam optimizer: A gradient-based method for stochastic updates," *International Conference on Learning Representations (ICLR)*, 2015. Available: https://arxiv.org/abs/1412.6980

[10] OpenAI, "Gym: A development toolkit for reinforcement learning algorithms," 2016. [Online]. Available: https://arxiv.org/abs/1606.01540

[11] S. Gu et al., "Off-policy updates for robotic manipulation using deep reinforcement learning," in *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2017.

[12] OpenAI. (2023). *ChatGPT: Language model for conversation and code assistance* [Large language model]. Retrieved from https://openai.com/chatgpt.