Self Service Autonomous Mobile Robot for Grocery Store

Jitesh Chandrahas Sonkusare
sonkusare.j@northeastern.edu

Rishabh Kumar
kumar.rishab@northeastern.edu

Pratik Sanjay Baldota
baldota.p@northeastern.edu

Jerin Delbin Sebastian
delbin.j@northeastern.edu

Brinton Cheng
cheng.brin@northeastern.edu

Abstract

The modern retail landscape embraces innovations like cashierless checkout and delivery services such as Instacart. Our project introduces an autonomous self-service shopping cart inspired by warehouse robots, leveraging reinforcement learning and pathfinding to navigate supermarkets, collect items, and deliver them to checkout while avoiding obstacles. By removing human intervention, reducing wait times, and streamlining the process, this solution offers a cost-effective, user-centric, and advanced shopping experience.

1 Introduction

Grocery shopping is often time-consuming and inefficient, with traditional methods leaving room for innovation. While solutions like cashierless checkouts and Instacart streamline the process, they face scalability and cost challenges. This project introduces an autonomous shopping cart that uses a user-generated list, reinforcement learning, and pathfinding to navigate aisles, retrieve items, and avoid obstacles. By addressing the drawbacks of existing systems, it offers a time-efficient, cost-effective, and fully autonomous alternative to traditional grocery shopping.

2 Related Work

Automated systems in retail and warehouses showcase the potential of AI and pathfinding technologies for optimizing navigation and task fulfillment. Amazon Robotics' warehouse robots use algorithms like A* to efficiently navigate and retrieve items in structured layouts, while Instacart's Caper Cart integrates AI to streamline in-store shopping with features like item scanning and personalized recommendations. Unlike these systems, which focus on navigation or partial automation, our project leverages reinforcement learning and pathfinding to develop fully autonomous carts capable of independently navigating stores, retrieving items, and delivering them to checkout.

3 Environment and Rewards

We simulated a supermarket environment using a custom Gym grid-like structure, where the cart (agent) moves in four directions. Trained with reinforcement learning, the cart follows a reward-penalty system: positive rewards for moving closer to or reaching the target item, and penalties for inefficient behaviors such as unnecessary steps, moving away from the item, looping(to avoid local minima), or failing to find a path. This system ensures the cart learns to optimize movements, efficiently navigate obstacles, and complete tasks effectively.

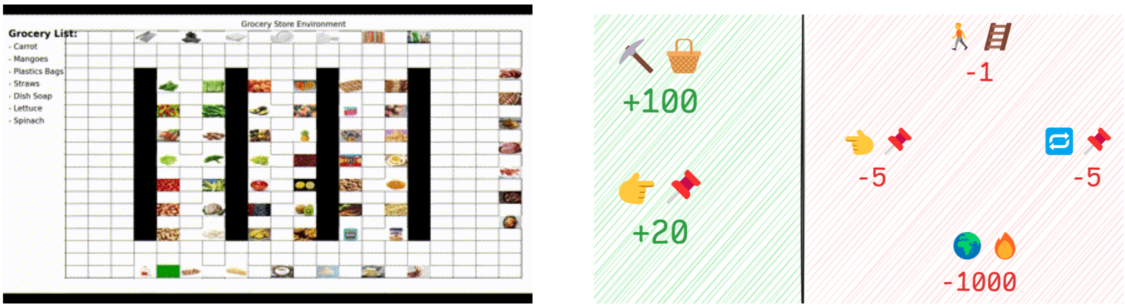


Figure 1: Left: Environment GUI, Right: Reward Structure

4 Methods

In recent years, deep reinforcement learning has been one of the most concerned directions in artificial intelligence. It combines the perceptual ability of deep learning with the decision-making ability of reinforcement learning and directly controls agents' behavior through high-dimensional perceptual input learning. Generally speaking, it applies the neural network structure to the process of reinforcement learning. Nowadays, the major deep reinforcement learning algorithms include Deep Q Network, Deep Deterministic Policy Gradient, Asynchronous Advantage Actor-Critic, Proximal Policy Optimization

(PPO). We used the PPO as our primary AI algorithm in our group project, so we combed the PPO principle in the next section.

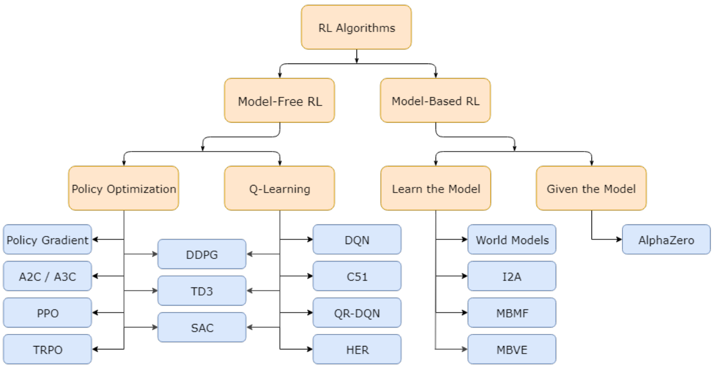


Figure 2: Important Algorithms of Deep Reinforcement Learning

The diagram above provides an overview of various Reinforcement Learning (RL) algorithms, broadly categorized into Model-Free and Model-Based approaches. Within these categories, we observe diverse methods for policy optimization, Q-learning, and leveraging model-based techniques. To comprehensively understand RL, we will focus on two representative algorithms: Proximal Policy Optimization (PPO) and Deep Q-Network (DQN). By discussing these two algorithms, we can cover value-based and policy-based paradigms, ensuring a holistic overview of key approaches in Deep RL.

4.1 PPO - Proximal Policy Optimization

Proximal Policy Optimization (PPO)[14] is an optimization algorithm that can improve the data efficiency and reliable performance of Trust Region Policy Optimization (TRPO) while only using the first-order optimization. PPO is an optimized version based on Policy Gradient and TPRO. Even if it uses the same way to perform multiple optimization steps, Policy Gradient Method may often lead to destructively significant policy updates. TPRO uses a hard constraint instead of a penalty since choosing a fixed penalty coefficient is difficult. In TPRO, the KL penalty coefficient needs to be adjusted to improve the algorithm's performance. The primary objective function of PPO is:

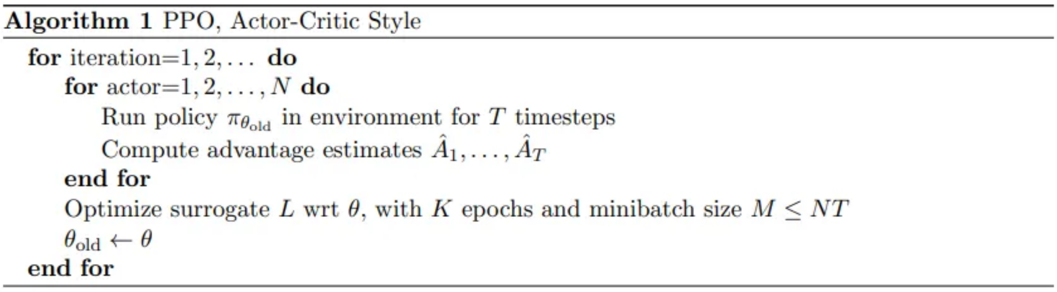
$$L_{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\Big(r_t(\theta)\hat{A}_t, \text{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)\hat{A}_t\Big)\Big] \tag{1}$$

Explanation of Components:

- θ: The parameters of the policy being optimized.
- $r_t(\theta)$: The probability ratio between the new policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$.
- $\hat{A}_t$: Is the estimated advantage at time t.
- ϵ: The clipping hyperparameter, which constrains the probability ratio to the range $[1-\epsilon, 1+\epsilon]$.
- $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$: Clips the probability ratio to ensure it stays within the range $[1-\epsilon, 1+\epsilon]$, reducing the likelihood of harmful policy updates.

In the $L_{\text{CLIP}}(\theta)$ in PPO, if the agent has too large of a policy update, it will be punished, which is different from $L_{\text{CLIP}}(\theta)$ (The objective function in TPRO). The *clip()* function can prevent the incentive factors from moving rt outside the interval [1-ϵ, 1+ϵ]. The primary goal of the PPO objective function is to strike a balance between exploration and exploitation while avoiding drastic updates that could destabilize the learning process. By using a clipped surrogate objective, PPO ensures that the updates to the policy remain within a trust region, improving stability and reliability compared to earlier approaches such as vanilla Policy Gradient or Trust Region Policy Optimization (TRPO).

This pseudo-code shows how PPO works step by step. First, the algorithm runs for multiple iterations. In each iteration, several actors (or agents) work in parallel, interacting with the environment



```
Algorithm 1 PPO, Actor-Critic Style
for iteration=$1, 2, \ldots$ do
    for actor=$1, 2, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 3: PPO Algorithm Pseudo-code

for a fixed number of steps (T) using the current policy. This helps gather data quickly. Next, the algorithm calculates something called "advantage estimates" ($\hat{A}_t$), which tell us how good each action was compared to what was expected. Finally, it updates the policy by optimizing the PPO objective (L_{CLIP}) using small chunks of data (called minibatches) over several rounds. This process makes sure the policy improves steadily without making big, unstable changes.

In conclusion, PPO is a powerful and versatile RL algorithm that effectively balances stability and performance through its clipped surrogate objective. Its simplicity, combined with its ability to handle continuous and discrete action spaces, makes it a popular choice in modern RL applications. While PPO is focused on policy optimization, another cornerstone of RL is value-based methods. To explore this paradigm, we now turn our attention to Deep Q-Networks (DQN), a foundational algorithm in the realm of Q-learning that has revolutionized how RL handles discrete action spaces.

4.2 DQN - Deep Q-Network

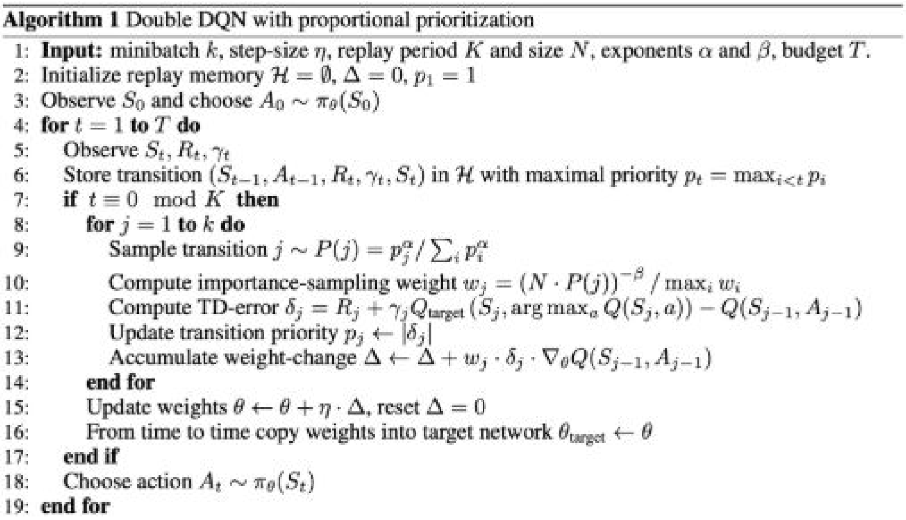
The Deep Q-Network (DQN) is a reinforcement learning system that extends Q-learning by storing action values in deep neural networks rather than tables. This update makes it more suitable for dealing with vast, complicated situations. DQN also employs a technique known as experience replay, in which previous events, such as actions and their outcomes, are saved and randomly utilized during training. With a result, learning grows more regular and stable. To overcome instability due to ongoing updates, DQN employs a separate target system that updates on a regular basis. It functions by rewarding positive acts and condemning negative ones, making it useful for solving challenges. DQN function is

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t = s, a_t = a, \pi\right] \tag{2}$$

Explanation of Components:

- $Q^*(s,a)$: The optimal action-value function, representing the maximum expected cumulative reward for taking action a in state s and following the optimal policy thereafter.
- π: The policy, which maps states to actions and determines the agent's behavior.
- $\mathbb{E}$: The expected value operator, averaging over possible future rewards given the current state and action.
- r_t: The reward received at time step t.
- γ: The discount factor, a value between 0 and 1 that determines the importance of future rewards.
- s_t, a_t: The state and action at time t, respectively.

A robot learning to play a video game is an example of an artificial intelligence system that utilizes a Deep Q-Network (DQN) algorithm to gradually improve its performance. The DQN gathers data about the game, including the current status, possible actions, and the rewards associated with them. It uses this knowledge to make decisions aimed at achieving the best possible outcome. The system learns from its experiences, allowing it to make better decisions over time. The inputs to the system include the store layout, which represents the spatial structure of the store, real-time cart locations, item positions on shelves, and information about obstacles or barriers in the environment. This input data is preprocessed and restructured to be compatible with the DQN. A neural network at the core of the DQN analyzes the processed data to estimate Q-values for each state-action pair, guiding the robot's decision-making process. A reward mechanism is implemented to encourage efficient behavior, such as minimizing travel distance, while penalties are introduced for undesirable actions, like collisions or inefficiencies. The network is trained iteratively using data from the environment, and the loss function, representing the difference between predicted and actual Q-values, is minimized across iterations to enhance the robot's performance.



```
Algorithm 1 Double DQN with proportional prioritization
Input: minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ to $T$ do
    Observe $S_t, R_t, \gamma_t$
    Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
    if $t \equiv 0 \mod K$ then
        for $j = 1$ to $k$ do
            Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
            Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
            Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
            Update transition priority $p_j \leftarrow |\delta_j|$
            Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
        end for
        Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
        From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
    end if
    Choose action $A_t \sim \pi_\theta(S_t)$
end for
```

Figure 4: DQN Algorithm Pseudo-code

The usage of DQN is proportional to prioritization in Double DQN. The initial step of the code is to configure priority replay, memory, and settings. Transition states (state, action, reward, and next state) are recorded at each stage and prioritized based on their significance. Regularly, minibatches are sampled, and Q-values are prioritized and updated using Temporal Difference (TD) error. The fairness and balance of the updated values are ensured through importance sampling. The target network is updated periodically to maintain training stability. By prioritizing Q-values and focusing on significant values, this method improves efficiency and stability.

4.3 A-Star

As a benchmark for comparison with our reinforcement learning approaches, we have also implement the A* algorithm for pathfinding in the Grocery Store environment. A* is a widely used heuristic-based search algorithm that efficiently finds the shortest path between a start and goal state in a graph-based representation of the environment. Implementing A* will provide a valuable baseline for evaluating the performance of our reinforcement learning algorithms. It will allow us to compare the efficiency and adaptability of both approaches in the context of Grocery Store robot navigation

4.4 Hyperparameter Choices

The DQN agent uses a neural network with two hidden layers of 128 units each and ReLU activation functions. It employs epsilon-greedy exploration with an initial epsilon of 1.0, decaying by 0.995 each episode. The agent is trained for 4000 episodes using a discount factor of 0.99 and a replay buffer of size 100000. The PPO agent uses an actor-critic architecture with a shared base network of two hidden layers with 64 units each and ReLU activation functions. The actor network outputs action probabilities using a softmax function, while the critic network estimates the value function. The agent uses a clipping parameter of 0.2 for the PPO algorithm and is trained for 100 iterations with a learning rate of 3e-4. Both implementations use the Adam optimizer and are designed to work with the custom GroceryStoreEnv environment, which has a grid size of 20x20.

5 Results

5.1 Loss vs Iteration

The loss vs iteration graphs is essential for monitoring the training progress of a model, showing how well it learns over time by minimizing errors. In our project, this visualization helps identify convergence, stability, and potential issues like instability or over-fitting, ensuring the model is optimized effectively. Below are the Loss vs Iteration plots for PPO and DQN respectively.

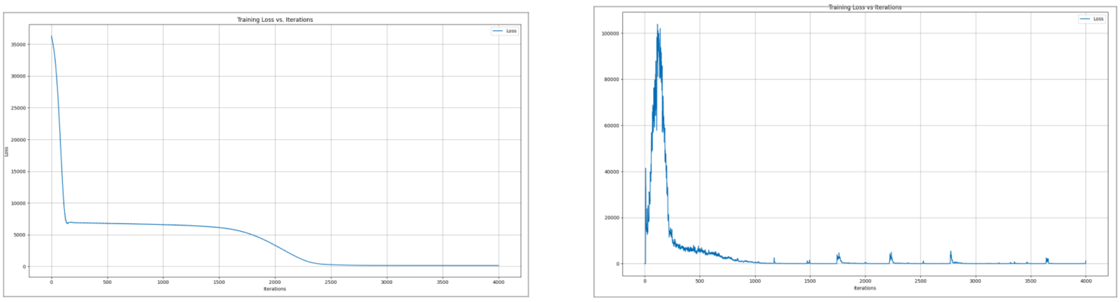


Figure 5: Loss vs Iterations graph for PPO(a) and DQN(b)

It is evident from the above figure PPO(a) that the training loss decreases smoothly over iterations, indicating stable convergence. The initial drop followed by a plateau shows effective training as it eventually goes to 0.

For DQN (b) the training loss shows significant oscillations and increases in the beginning with sharp peaks and valleys. The reason for this is because of the epsilon greedy policy which starts with exploration due to which the agent is randomly roaming around in the environment until it learns. However, the loss decreases over time with larger updates to the Q-value estimates, resulting in a loss reaching 0.

5.2 Rewards vs Episodes

The rewards vs episodes graph is essential for evaluating how well the agent learns to maximize cumulative rewards over time. It tracks the agent's learning progression, stability, and convergence, providing insights into the effectiveness of the training process and algorithm performance in our custom environment.

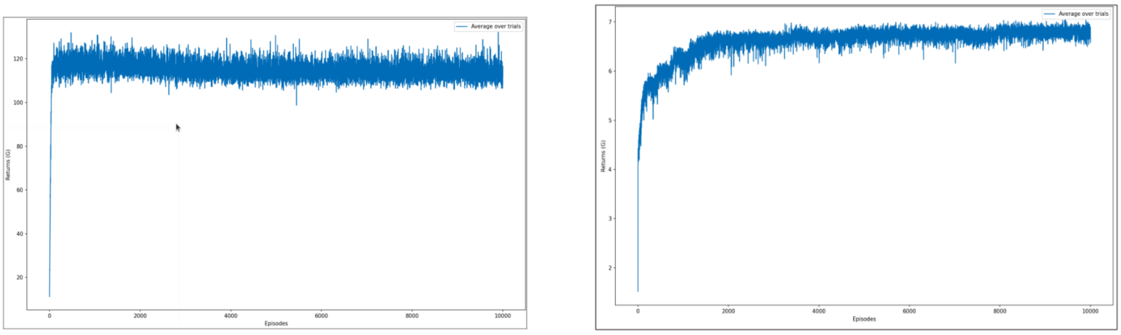


Figure 6: Reward vs Episode graph for PPO(a) and DQN(b)

From the reward vs episodes graph for ppo we can observe that the rewards increase rapidly in the intial episodes, stabilizing aroung a high average value with small fluctuations. This shows steady and efficient learning. If we compare the rewards vs episodes graph of DQN with that of ppo we can observe that the agent is learning to improve its performance like ppo agent but this is happening at a slower and less stable rate. This is typical for DQN as its value-based approach is more sensitive to noisy updates and hyperparameter settings.

The output videos of the 3 agent's performance in our custom environment- `https://github.com/jitesh3023/Artificial_Intelligence_Final_Project/tree/main/Output_Videos`

6 Conclusion

Summary					
Algorithm	Use Case	Pros	Cons	Train	Test
PPO	Control based tasks, Robotic arms (Continuous action space)	Stable, efficient, balanced	Compute heavy, needs large samples	Stable learning	Good for generalization
DQN	Atari games, Discrete Action Space	Efficient for discrete tasks	Poor in dynamic settings, easy compute	Can be unstable	Good in similar environment
A*	Static environment, optimal paths	Guaranteed shortest path	Ineffective in dynamic settings	N/A	Optimal in static environemnt

Table 1: A Summary Table

A* search is one of the best algorithms for finding shortest path, which makes it a good baseline for evaluating the performance of our DQN and PPO models. However, A* cannot handle dynamic shopping lists, where the same items appear in a different order. In such cases, A* can only find the shortest path between consecutive items based on the fixed order in the list, rather than adapting to different permutations of the items. In contrast, a well-trained DQN or PPO model learned to adjust the shop list, enabling more efficient item collection. During the training process, DQN may experience instability, particularly in the early stages, whereas PPO tends to remain stable throughout the entire training process. DQN is better suited for discrete action spaces, while PPO performs well in continuous environments. Since our simulation operates in a discrete environment, the overall performance of DQN and PPO was similar. However, in a real-world scenario which is a continuous environment, PPO would likely outperform DQN.

There are several opportunities for future improvements, including transitioning to a continuous environment to better simulate the real world, assigning weights to grocery items, and incorporating multi-agent interactions.

7 Team Contributions

Jitesh Chandrahas Sonkusare: Created custom environment and A*. Worked on PPO and DQN.
Rishabh Kumar: Worked on PPO. And Helped with parameter tuning.
Pratik Sanjay Baldota: Worked on DQN. And Helped with parameter tuning.
Brinton Cheng: Worked on Metric Calculations and A* Implementation. Helped with DQN.
Jerin Delbin Sebastian: Helped with PPO. Helped with A* Implementation

8 Code Repository

Link: `https://github.com/jitesh3023/Artificial_Intelligence_Final_Project/tree/main`

References

[1] Thompson, C., Khan, H., Dworakowski, D., Harrigan, K., & Nejat, G. (2018). An Autonomous Shopping Assistance Robot for Grocery Stores. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Retrieved from `http://asblab.mie.utoronto.ca/sites/default/files/An_Autonomous_Shopping_Assistance_Robot_for_GroceryStores-Camera_Ready.pdf`

[2] Berkeley, C. (2022). Self-Driving Robots: A Revolution in the Local Delivery. *California Management Review*. Retrieved from `https://cmr.berkeley.edu/2022/04/self-driving-robots-a-revolution-in-the-local-delivery/`