CS687 Final Report: Warehouse Environment

Md Masudul Islam
mdmasudulisl@umass.edu

Zekai Zhang
zekzhang@umass.edu

December 2024

1 Introduction

Efficient management of warehouse operations is a critical component of modern supply chain logistics. The dynamic nature of warehouse environments, characterized by unpredictable demand patterns, multiple operational constraints, and complex workflows, presents significant challenges for optimization. Traditional rule-based systems often fall short in addressing these complexities, necessitating the adoption of advanced decision-making algorithms.

In this context, reinforcement learning (RL) emerges as a promising approach for modeling and optimizing decision-making in warehouse environments. RL's ability to learn optimal strategies through interaction with the environment makes it well-suited to adapt to the dynamic and uncertain nature of warehouses.

Among reinforcement learning methods, Reinforce with Baseline, Semi-Gradient n-Step SARSA, and One-Step Actor-Critic are particularly effective. Reinforce with Baseline enhances the standard REINFORCE algorithm by using a baseline to reduce gradient variance, enabling more stable and efficient policy optimization. Semi-Gradient n-Step SARSA leverages temporal difference learning over multiple steps, making it well-suited for tasks requiring strategic planning and delayed rewards, such as navigating warehouse grids. One-Step Actor-Critic integrates policy-based and value-based methods, simultaneously learning a policy and value function. This approach optimizes decisions efficiently in complex scenarios while reducing learning variance.

This study focuses on applying these three RL algorithms to a simulated warehouse environment. The goal is to optimize operations such as inventory management, order picking, and resource allocation. The proposed implementation aims to:

1. Evaluate the feasibility and effectiveness of these algorithmns in a warehouse setting.
2. Identify key operational parameters that influence the performance of reinforcement learning strategies in warehouses.

2 Methodology

2.1 Environments

The primary environment used for experimentation in this study is a simulated warehouse environment modeled as a Markov Decision Process (MDP). A sample of warehouse environment is in figure 1. In these 2 by 2 aisles warehouse environment, the agent will learn four different policies respect to different aisles. Suppose the agent need to delivery three distinct items to yellow, pink and blue aisles. The agent will first implement yellow policy and reach yellow terminate state, then choose the pink policy and head to the pink terminate state. Finally, it execute blue policy and finish the task. The detailed environment designs are described below:

- **States:** The states in the warehouse environment represent the agent's position in a grid. Each grid cell corresponds to a unique state. The environment includes:
  - Regular states (white grids in fig 1) that the agent can traverse.
  - Initial state $S_0 = (0, 0)$ where the agent starts its task.
  - Terminal states, representing specific locations where tasks, such as item pickups or deliveries, are completed.
  - Shelves states (colored grids in fig 1), where the agent can pass it but receive huge penalty.

  The agent's state is dynamically updated based on its movement across the grid.
- **Actions:** The agent has four discrete actions available: Move up, Move right, Move down and Move left. These actions allow the agent to navigate through the grid, with constraints such as avoiding walls.
- **Transition Functions:** The agent has 70% probability to move to its intend action, 12% probability to moves to the right with respect to the intended direction, 12% probability to moves to the left with respect to the intended direction, and 6% probability to stay at the same state.

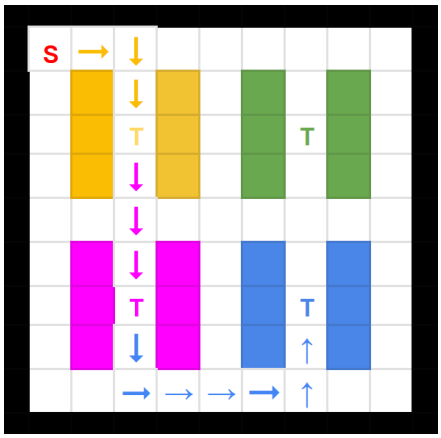


Figure 1: Warehouse Environment

- **Rewards:** The reward structure is designed to incentivize efficient navigation and task completion:
  - A significant positive reward is provided when the agent reaches a terminal state.
  - Negative rewards are assigned for invalid moves, such as attempting to traverse the shelves states, the agent will get reward -9.
  - A slightly negative reward -1 are assigned to agent if it traverse in regular states.

This environment serves as a practical simulation of real-world warehouse logistics, capturing challenges such as obstacle avoidance, task prioritization, and resource optimization. By evaluating the performance of reinforcement learning algorithms in this environment, the study aims to explore their ability to improve operational efficiency and adaptability in dynamic settings.

2.2 Algorithms

2.2.1 One-Step Actor-Critic

The One-Step Actor-Critic algorithm combines policy gradient and value-based methods to optimize decision-making in episodic tasks. It employs two components:

- **Actor:** Selects actions based on a parameterized policy, $\pi(a|s;\theta)$, where θ represents the policy parameters.
- **Critic:** Evaluates the actions by estimating the value function $V(s;w)$, where w represents the value function parameters. It uses the value estimates to compute the Temporal Difference (TD) error.

The TD error, defined as:

$$\delta = r + \gamma V(s';w) - V(s;w),$$

is used to update both the policy (actor) and the value function (critic).

- The **critic** minimizes the squared TD error by updating its parameters using stochastic gradient descent.
- The **actor** updates the policy in the direction of the TD error to maximize expected rewards.

These updates ensure that the policy adapts to maximize the expected return, while the value function learns to approximate the true return accurately. The algorithm processes episodes step by step, resetting the environment at the start of each episode and making updates after every step.

Pseudocode:

```
Initialize policy parameters  and value function parameters w
for each episode do:
    Reset the environment and initialize state s
    for each step of the episode do:
        Take action a ~ (a|s; )
        Observe reward r and next state s'
        Compute TD error:  = r + V(s'; w) - V(s; w)
```

```
        Update value function parameters:
            w ← w + wV(s; w)
        Update policy parameters:
             ←  +  log (a|s; )
        s ← s'
    end
end
```

2.2.2 REINFORCE with Baseline

The REINFORCE with Baseline algorithm is a policy gradient method that introduces a value function as a baseline to reduce the variance of policy updates. This baseline does not affect the expected value of the gradient but stabilizes the learning process, leading to improved performance.

The algorithm consists of two primary components:

- **Policy Network (Actor):** Parameterizes the policy $\pi(a|s;\theta)$, which is used to select actions in the environment. The parameters θ are updated to maximize the expected reward.
- **Value Network (Baseline):** Approximates the value function $V(s;w)$ to compute an advantage signal that guides the policy updates. The parameters w are updated to minimize the squared difference between the predicted value and the actual return.

The advantage is computed as:

$$A(s,a) = G_t - V(s;w),$$

where G_t is the discounted return, defined as:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots.$$

- The **policy network** is updated by minimizing the policy loss:

$$L_{\text{policy}} = -\log \pi(a|s;\theta) A(s,a).$$

- The **value network** is updated by minimizing the value loss:

$$L_{\text{value}} = \left(A(s,a)\right)^2.$$

Both updates are performed using stochastic gradient descent, ensuring that the policy adapts to maximize rewards while the value function stabilizes the learning process.

Pseudocode:

```
Initialize policy network parameters  and value network parameters w
for each episode do:
    Reset the environment and initialize state s
    for each step of the episode do:
        Take action a ~ (a|s; )
        Observe reward r and next state s'
        Store (s, a, r)
        s ← s'
    end
    Compute discounted returns G for each timestep
    for each step of the episode do:
        Compute advantage A = G - V(s; w)
        Update value network parameters:
            w ← w -  w (A)^2
        Update policy network parameters:
             ←  +   log (a|s; ) A
    end
end
```

2.2.3 Episodic Semi-Gradient n-Step SARSA

The Episodic Semi-Gradient n-Step SARSA algorithm is a multi-step reinforcement learning method that approximates the action-value function using a parameterized model (e.g., a neural network). It extends traditional SARSA by incorporating multi-step updates, improving learning efficiency and capturing the long-term impact of actions.

Working Mechanism:

- **Q-Network:** A neural network $Q(x, y, a; \mathbf{w})$ approximates the action-value function. The network takes the state (x, y) and one-hot encoded action a as input and outputs the estimated Q-value.
- **Epsilon-Greedy Policy:** An ϵ-greedy policy is used to balance exploration and exploitation during action selection. Actions with the highest Q-value are chosen with probability $1 - \epsilon$, while random actions are chosen with probability ϵ.
- **Transition Dynamics:** The agent interacts with the environment, transitioning to new states based on the selected action and stochastic transition probabilities.
- **n-Step Return:** The algorithm uses multi-step returns G to compute the TD error:

$$G = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n Q(S_{t+n}, A_{t+n}; \mathbf{w}),$$

  where R_{t+k+1} represents rewards from the environment, and $Q(S_{t+n}, A_{t+n}; \mathbf{w})$ is the estimated Q-value of the n-th step.

- **Gradient Update:** The TD error $\delta = G - Q(S_t, A_t; \mathbf{w})$ is used to update the Q-network's parameters via gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla_{\mathbf{w}} Q(S_t, A_t; \mathbf{w}),$$

  where α is the learning rate.

Pseudocode:

```
Initialize Q-network parameters w
for each episode do:
    Initialize state S_0 and action A_0 using epsilon-greedy policy
    Set T = infinity and t = 0
    while tau != T - 1 do:
        if t < T:
            Take action A_t, observe reward R_{t+1} and next state S_{t+1}
            if S_{t+1} is terminal:
                T = t + 1
            else:
                Select next action A_{t+1} using epsilon-greedy policy
        tau = t - n + 1
        if tau >= 0:
            Compute multi-step return G:
                G = sum_{k=0}^{min(n, T-tau)-1} gamma^k R_{tau+k+1}
                if tau + n < T:
                    G += gamma^n Q(S_{tau+n}, A_{tau+n}; w)
            Update w using TD error:
                delta = G - Q(S_tau, A_tau; w)
                w <- w + alpha * delta * grad_w Q(S_tau, A_tau; w)
        t += 1
end
```

3 Results and Analysis

3.1 Hyperparameter Tuning for One-Step Actor-Critic

After conducting numerous experiments, we selected grid sizes of 2×2 and 3×3 with varying episodes (2000 and 4000) and hyperparameters. The following summarizes the results and key observations:

3.1.1 Grid Size: 2×2, Episodes: 2000

Grid Size	Episodes	Learning Rate (α)	Discount Factor (γ)	Max Steps	Rewards
2×2	2000	0.0004	0.94	200	98
2×2	2000	0.0006	0.93	200	93
2×2	2000	0.0006	0.93	100	91
2×2	2000	0.0007	0.95	200	98

3.1.2 Grid Size: 2×2, Episodes: 4000

Grid Size	Episodes	Learning Rate (α)	Discount Factor (γ)	Max Steps	Rewards
2×2	4000	0.0004	0.94	200	90
2×2	4000	0.0004	0.94	100	-26
2×2	4000	0.0006	0.93	100	93

Observations:

- For episodes 2000, the best rewards of **98** were achieved with two parameter configurations:
  - $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$: The learned policy efficiently reached the terminal state with minimal unnecessary steps. Example trajectory: $(4, 2) \rightarrow (3, 2) \rightarrow (2, 2)$, with rewards of $-1, -1, 100$.
  - $\gamma = 0.95, \alpha = 0.0007, \text{max steps} = 200$: Similarly, the learned policy demonstrated efficient behavior with the same trajectory and reward sequence.
- Extending the episodes to 4000 for $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$ resulted in diminished performance, reducing rewards from **98** to **90**. This suggests potential overtraining or diminishing returns from prolonged learning.
- Configurations with shorter max steps (100) generally performed worse. For $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 100$, the rewards dropped to **-26**, indicating the policy struggled to converge effectively within the reduced step limit.
- $\gamma = 0.93, \alpha = 0.0006, \text{max steps} = 100$ exhibited consistent performance, achieving rewards of **93** across both 2000 and 4000 episodes. This robustness demonstrates its adaptability to varying training durations and step limits.

3.1.3 Grid Size: 3×3, Episodes: 2000

Grid Size	Episodes	Learning Rate (α)	Discount Factor (γ)	Max Steps	Rewards
3×3	2000	0.0004	0.94	200	98
3×3	2000	0.0006	0.93	200	82
3×3	2000	0.0007	0.95	200	98

3.1.4 Grid Size: 3×3, Episodes: 4000

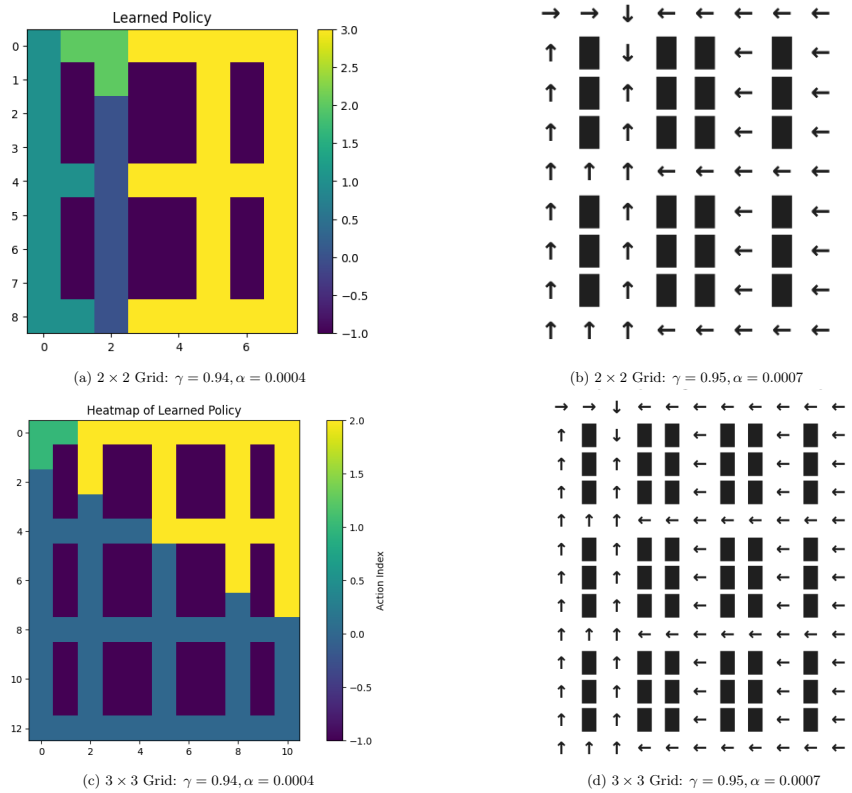
Grid Size	Episodes	Learning Rate (α)	Discount Factor (γ)	Max Steps	Rewards
3×3	4000	0.0006	0.93	100	18
3×3	4000	0.0004	0.94	200	-123

Observations:

- For the 3×3 grid with 2000 episodes, the best rewards were achieved with ($\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$) and ($\gamma = 0.95, \alpha = 0.0007, \text{max steps} = 200$), both yielding **98**. However, ($\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$) is preferable due to its more efficient learned policy, which minimizes unnecessary steps and reaches the terminal state directly.
- Extending the episodes to 4000 led to a significant drop in performance. For ($\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$), rewards fell to **-123**, indicating overtraining or diminishing returns with prolonged learning.
- The configuration with shorter max steps (max steps = 100) and ($\gamma = 0.93, \alpha = 0.0006$) performed relatively better in this extended setting, achieving a reward of **18**. However, this still underperformed compared to the best results from 2000 episodes.

3.1.5 Visualization of Learned Policies Across Best Configurations

The learned policies displayed in Figure 2 for the best four parameter configurations across grid sizes 2×2 and 3×3 illustrate the agent's decision-making process and its ability to reach the terminal state efficiently.



(a) 2×2 Grid: $\gamma = 0.94, \alpha = 0.0004$

(b) 2×2 Grid: $\gamma = 0.95, \alpha = 0.0007$

(c) 3×3 Grid: $\gamma = 0.94, \alpha = 0.0004$

(d) 3×3 Grid: $\gamma = 0.95, \alpha = 0.0007$

Figure 2: Learned Policies Across Best Configurations for 2×2 and 3×3 Grids.

Best Parameters for Each Configuration:

- **Grid Size 2×2, Episodes: 2000:**
  - $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$ (**Reward: 98**, Optimal policy with minimal steps to terminal state)
  - $\gamma = 0.95, \alpha = 0.0007, \text{max steps} = 200$ (**Reward: 98**, Slightly redundant steps observed)
- **Grid Size 3×3, Episodes: 2000:**
  - $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$ (**Reward: 100**, Most optimal policy across all configurations)
  - $\gamma = 0.95, \alpha = 0.0007, \text{max steps} = 200$ (**Reward: 98**, Includes unnecessary steps to the terminal state)

Discussion:

- For the 2×2 grid, $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$ demonstrates a more direct path to the terminal state compared to $\gamma = 0.95, \alpha = 0.0007$, despite both achieving the same reward of **98**.
- For the 3×3 grid, $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$ is the most efficient configuration, achieving a perfect reward of **100** with no unnecessary steps, making it the best overall parameter set.
- In contrast, $\gamma = 0.95, \alpha = 0.0007, \text{max steps} = 200$ for the 3×3 grid introduces redundant movements, which slightly reduces efficiency despite a high reward of **98**.

The results in Figure 2 suggest that for both 2×2 and 3×3 grids, $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$ provides the most optimal learned policies, particularly for the 3×3 grid where it achieves a perfect reward of **100**. This parameter set is recommended as the best configuration for achieving efficient navigation and optimal policies.

3.2 Hyperparameter Tuning for REINFORCE with Baseline

The REINFORCE with Baseline algorithm, as shown in Figure 3, was implemented and evaluated on a 2×2 grid over 2000 episodes. The experiments aimed to identify the optimal hyperparameter settings for achieving high rewards efficiently. The following configurations produced the best results:

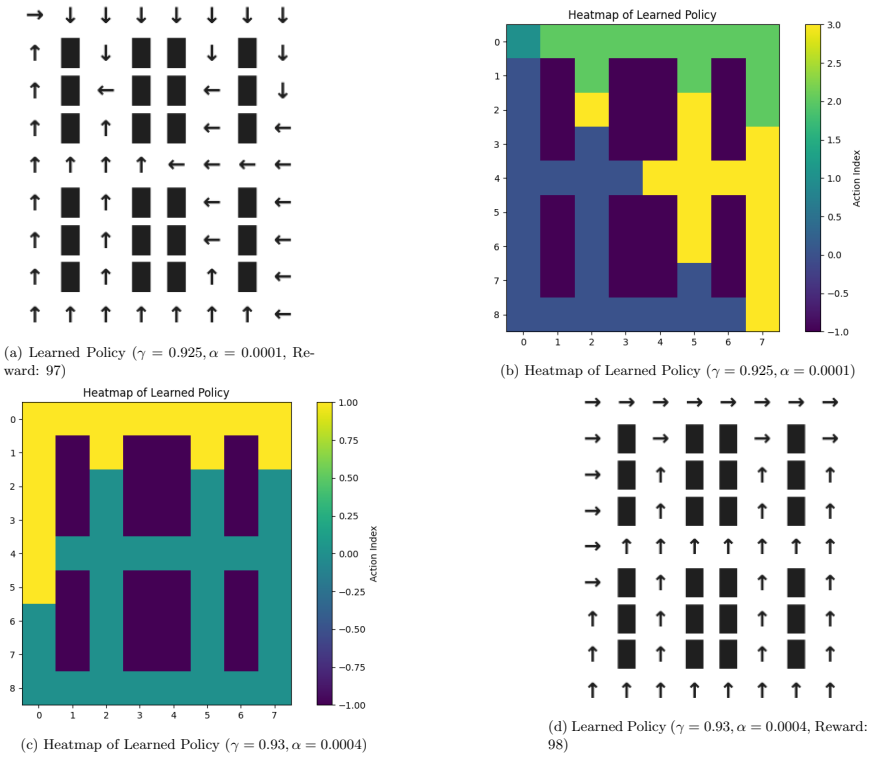
Best Parameters:

- $\gamma = 0.925, \alpha = 0.0001, \text{max steps} = 200$ (**Reward: 97**)
- $\gamma = 0.93, \alpha = 0.0004, \text{max steps} = 100$ (**Reward: 98**)

Key Observations:

- For $\gamma = 0.93, \alpha = 0.0004, \text{max steps} = 100$, the algorithm achieved the highest reward of **98**, indicating its efficiency in navigating the grid with fewer steps.
- The configuration with $\gamma = 0.925, \alpha = 0.0001, \text{max steps} = 200$ also performed well, achieving a reward of **97**. This setup demonstrated stability over extended steps.
- Training progress indicates consistent improvement across episodes for both configurations, reflecting the algorithm's ability to converge efficiently.

Visualization of Results:
Figure 3 illustrates the learned policies and heatmaps of action indices for the two best parameter configurations:



(a) Learned Policy ($\gamma = 0.925, \alpha = 0.0001$, Reward: 97)

(b) Heatmap of Learned Policy ($\gamma = 0.925, \alpha = 0.0001$)

(c) Heatmap of Learned Policy ($\gamma = 0.93, \alpha = 0.0004$)

(d) Learned Policy ($\gamma = 0.93, \alpha = 0.0004$, Reward: 98)

Figure 3: Learned Policies, and Heatmaps for REINFORCE with Baseline

The REINFORCE with Baseline algorithm, particularly with $\gamma = 0.93, \alpha = 0.0004, \text{max steps} = 100$, showcased high efficiency and optimal performance.

3.3 Hyperparameter Tuning for Episodic Semi-Gradient n-Step SARSA

3.3.1 Q-values Estimation Model Setting

The Episodic Semi-Gradient n-Step SARSA relies on a model to approximate the q-values function of a given state and an action. The model we use is neural network. The input of this network is the agent postion and one-hot encoded of action. For example, the agent execute action 2 in state(x, y), the input is [x, y, 0, 0, 1, 0]. The network architecture contain 3 hidden layers. The first layer has 32 neurons with ReLU activation. The second layer has 64 neurons with ReLU activation.

The Third layer has 32 neurons with ReLU activation. The output layer map all neurons to a singe estimated q value. The Weight Initialization techniques we use is Kaiming normal initialization and biases term initialized to zero.

The weights are not initialized to 0 because doing so can lead to the network learning abnormal weight patterns that cause it to output identical q-values for different states and actions. This issue persists even after experimenting with various learning rates. One possible explanation for this phenomenon is that initializing weights to 0 creates a lack of symmetry breaking between neurons in the network. As a result, during training, all neurons within the same layer receive identical gradients and update their weights identically, leading to uniform outputs.

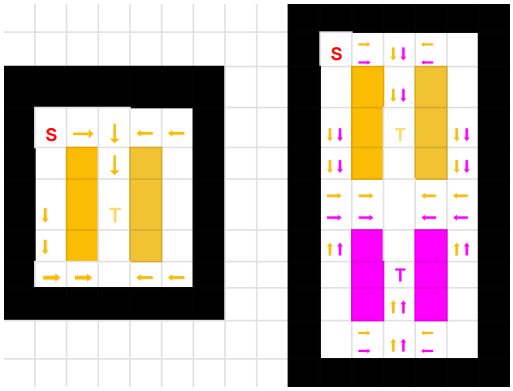


Figure 4: Left: 1x1 warehouse environment. Right: 2x1 warehouse environment, yellow policy and pink policy are highly overlap

3.3.2 Training Strategy

In practice, we observed that training Episodic Semi-Gradient n-Step SARSA can be highly unstable. Poor weight initialization may result in the agent taking an infinite number of steps to reach the terminal state, particularly in large warehouse environments. To mitigate this issue, we introduced a termination condition to end the episode if the number of steps exceeds 1000.

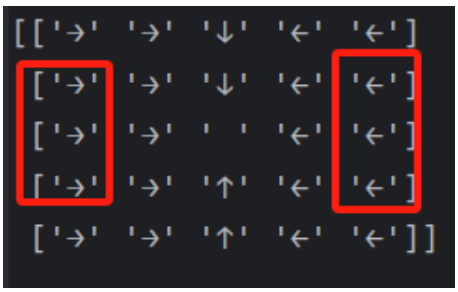
Additionally, we noticed that some policies in the same warehouse environment share overlapping paths. For example, as shown in Fig 6c, the yellow policy in a 1x1 aisle warehouse is identical to the upper portion of the yellow policy in a 2x1 aisle warehouse. Thus, the trained weights of the yellow policy in the 1x1 environment could be used to initialize the yellow policy in the 2x1 environment. Similarly, the trained weights for the yellow policy in the 2x1 environment could be be used to initialize the pink policy in the same environment, as these two policies share significant overlap.

3.3.3 Fine Tuning 1x1 Warehouse Environment

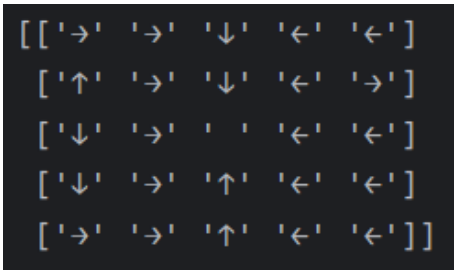
To fine-tune the Q-value estimation network, we analyzed how different parameters impact the performance of the learned policy. The network is updated using the Adam optimizer, which is an adaptive optimization algorithm less sensitive to the learning rate. Through experimentation, we found that setting the learning rate to 1e-4 yields good results for the model. We also use the fixed ϵ-greedy strategy to select action with $\epsilon = 0.1$. The third parameter we examined is γ. When γ is low, such as $\gamma = 0.2$, the learned policy tends to move away from the shelves. This occurs because a lower γ reduces the influence of rewards from the terminal state, causing the agent to prioritize immediate gains.

By setting γ to 0.92, the learned policies after 4000 iterations guide the agent toward the terminal state, though with some noise. This noise arises because every regular state is close to at least one shelf state, and the negative reward associated with shelf states introduces variability. To address this issue, we increased the terminal state reward to 100, creating a strong incentive for the agent to prioritize reaching the terminal state, thereby reducing noise and improving policy performance. The resulting policy is shown in fig 5a, where all actions lead toward the terminal state, even if some actions crash into the shelves (indicated by the red rectangle). To refine this policy and make it more optimal, we retrained it using the learned weights, setting a lower reward of 5 and $\gamma = 0.9$. The fine-tuned policy, shown in fig 5b, adjusts the actions in the first column to avoid shelf states to approach the terminal state, while the rightmost column remains unchanged.

The step size n is fixed at 3, and we do not analyze its effect on steps per episode for two main reasons: First, each episode is terminated if the step count exceeds 1000, meaning the steps taken in an episode do not accurately represent the true step size. Second, evaluating the impact of n would require training multiple models from scratch to compute an average, which is both time-consuming and demands significant computational resources.



(a) Learned policy after 4000 iterations with $\gamma = 0.92, \epsilon = 0.1$, reward = 100 and $n = 3$.



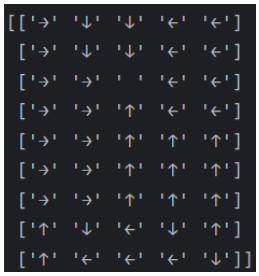
(b) Fine-tuned policy based on the previous one after 3000 iterations, trained with $\gamma = 0.8$, reward 4, $\alpha = 1e-5$ and $n = 7$.

Figure 5: 1x1 warehouse policy

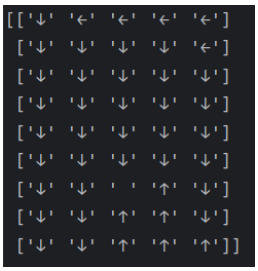
3.3.4 Transfer Learning in Large Environment

Since we have a relatively good policy, we could further train it on the large environment for the same terminate state. In fig 6 (a), the new extended policy demonstrates a generally good result, as the actions provide a clear and effective direction towards the goal. However, there are some noticeable noises in the lower part of the environment, where the policy deviates from the optimal action. Despite this, the majority of the actions successfully guide the agent towards the target state.

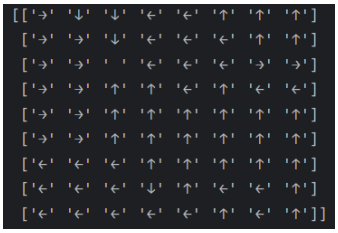
We use the newly learned policy weights as the initial parameters for training the second policy. Fig 6 (b) shows a less effective policy. While the model manages to capture the overall direction towards the goal, the detailed actions are suboptimal. This results in a less precise trajectory, where the policy fails to consistently generate high-quality actions that align with the optimal path. However, our training method demonstrates a potential ability to perform transfer learning effectively. That said, the semi-gradient n step SARSA algorithm is computationally expensive, and running additional iterations requires significant time to explore and optimize the parameters. One potential way to enhance performance and efficiency is by utilizing a more expressive network architecture, which could better capture the complexities of the larger environment with fewer training iterations.



(a) Fine-tuned policy based on 5b after 4000 iterations, trained with $\gamma = 0.92$, reward 100, $\alpha = 1e-4$ and $n = 11$.



(b) Fine-tuned policy based on left one after 4000 iterations, trained with $\gamma = 0.9$, reward 100, $\alpha = 1e-4$ and $n = 11$.



(c) Policy when (2, 2) is the terminal state in a 2x2 aisles warehouse environment.

Figure 6: Policy in Larger Warehouse

4 Comparison of Three RL Algorithms

In this section, we compare the performance, methodology, and applicability of the three reinforcement learning (RL) algorithms implemented: One-Step Actor-Critic, REINFORCE with Baseline, and Episodic Semi-Gradient n-Step SARSA. The comparison evaluates their strengths, limitations, and suitability for solving warehouse navigation problems.

4.1 Methodological Comparison

- **One-Step Actor-Critic:** Combines value-based and policy-based methods. The actor optimizes the policy, while the critic estimates the value function to stabilize updates. The use of temporal difference (TD) error ensures that the policy adapts efficiently while reducing variance in learning.
- **REINFORCE with Baseline:** A pure policy gradient method augmented with a value function baseline to reduce variance. This algorithm directly optimizes the policy using the advantage signal, calculated as the difference between the return and the baseline value. The inclusion of a value network stabilizes learning but increases computational complexity.

- **Episodic Semi-Gradient n-Step SARSA:** Extends traditional SARSA with multi-step updates, enabling it to consider delayed rewards effectively. This algorithm relies on a Q-network for value approximation, making it robust for continuous state-action spaces. However, it can be computationally expensive and prone to instability without proper weight initialization and hyperparameter tuning.

4.2 Performance Evaluation

Key Observations:

- **One-Step Actor-Critic:**
  - Achieved consistently high rewards on both 2×2 and 3×3 grids with the configuration $\gamma = 0.94, \alpha = 0.0004, \text{max steps} = 200$ over 2000 episodes.
  - Demonstrated efficient navigation with minimal redundant steps, particularly on the 3×3 grid, achieving a perfect reward of **100** over 2000 episodes.
  - Exhibited stability across episodes; however, performance diminished with excessive training iterations (e.g., 4000 episodes).
- **REINFORCE with Baseline:**
  - Performed well on the 2×2 grid, achieving a reward of **98** with $\gamma = 0.93, \alpha = 0.0004, \text{max steps} = 100$.
  - The baseline value network effectively reduced variance, leading to smooth training progress and convergence.
  - Required careful tuning of learning rates and discount factors to achieve optimal performance.
- **Episodic Semi-Gradient n-Step SARSA:**
  - Demonstrated strong performance in small environments (e.g., 1×1 and 2×1 warehouses) after fine-tuning.
  - The transfer learning approach improved scalability to larger environments, but the policy exhibited some noise in less frequently visited states.
  - Training was highly sensitive to weight initialization, step size n, and the reward structure, requiring additional strategies to stabilize learning.

4.3 Algorithmic Trade-offs

- **Efficiency:** One-Step Actor-Critic exhibited the most consistent efficiency across varying grid sizes and episodes. REINFORCE with Baseline required fewer steps in smaller grids but struggled to maintain efficiency in larger environments.
- **Stability:** REINFORCE with Baseline achieved stable convergence due to the variance reduction introduced by the baseline value network. Episodic Semi-Gradient n-Step SARSA required additional strategies (e.g., transfer learning and termination conditions) to mitigate instability.
- **Scalability:** Episodic Semi-Gradient n-Step SARSA demonstrated scalability to larger environments through transfer learning. In contrast, One-Step Actor-Critic and REINFORCE with Baseline showed limited scalability without substantial retraining.
- **Computational Complexity:** REINFORCE with Baseline incurred the highest computational cost due to the need to train both policy and value networks. One-Step Actor-Critic balanced computational cost and performance, while Episodic Semi-Gradient n-Step SARSA required significant resources for fine-tuning in large environments.

5 Conclusion

The three algorithms offer distinct advantages and trade-offs, making them suitable for different types of environments and requirements. The One-Step Actor-Critic algorithm is best suited for environments where efficiency and stability are prioritized. Its balance of policy and value optimization ensures consistent performance across various settings, making it a versatile choice. On the other hand, REINFORCE with Baseline excels in small, structured environments where variance reduction is critical, although it requires precise tuning and incurs higher computational costs. Finally, the Episodic Semi-Gradient n-Step SARSA algorithm is ideal for larger and more complex environments, leveraging transfer learning for scalability. However, it is sensitive to initialization and hyperparameters, necessitating careful experimentation. Overall, the choice of algorithm depends on the specific requirements of the warehouse environment, such as size, complexity, and computational constraints. Among the three, the One-Step Actor-Critic emerges as the most versatile approach, while REINFORCE with Baseline and Episodic Semi-Gradient n-Step SARSA cater to more specialized scenarios.

6 References

- Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second Edition. Cambridge, MA: The MIT Press, 2018. ISBN: 9780262039246. Available at: `https://lccn.loc.gov/2018023826`.