Reinforcement Learning Project

Rojina Kashefi (r1018183)

May 2025

Task 1

1.1

In this task, a 5×5 grid world environment was implemented using Pygame. For the initial demonstration, the agent follows a predefined sequence of actions: right, right, down, down, and left. The outcome of this sequence is illustrated in Figure 1.

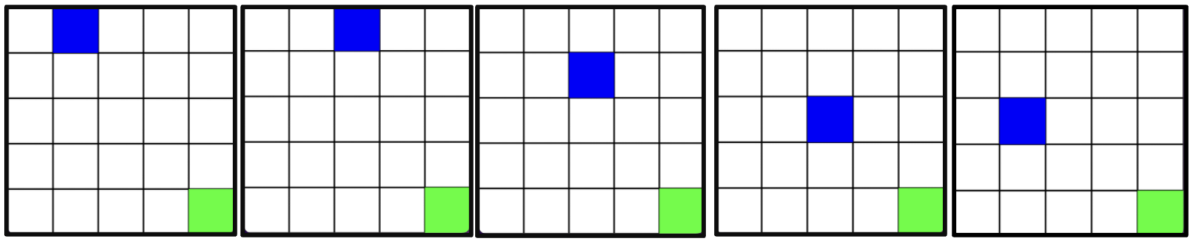


Figure 1: Visualization of the Grid World with predefined actions

For the random agent scenario, the action space consists of four possible moves: up, down, left, and right. At each step, the agent selects an action uniformly at random. The episode continues until the agent reaches the goal.
To enable interaction between the agent and the environment, a reinforcement learning (RL) task was implemented. Figure 2 illustrates the average performance of the random agent over 10,000 episodes, while Figure 3 visualizes the agent's trajectory during the first 10 time steps of a single episode.

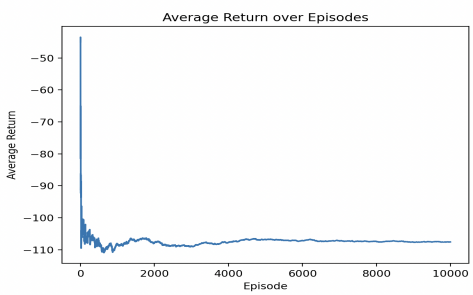


Figure 2: Average results over 10,000 episodes for the random agent

1.2

To interact with the environment, a fixed agent implemented. This agent always moves down until it can no longer do so, and then switches to moving right. Unlike the random agent, the fixed agent accounts for the environment's state, such as avoiding movement into walls or lava tiles. Figure 4 shows the agent's trajectory in the EMPTY ROOM environment. The agent successfully reaches the goal after 7 steps. In contrast, Figure 5 illustrates the agent's behavior in the ROOM WITH LAVA environment. After step 4, the agent gets stuck, as its only available actions are blocked by walls. This demonstrates the need for more advanced decision making.

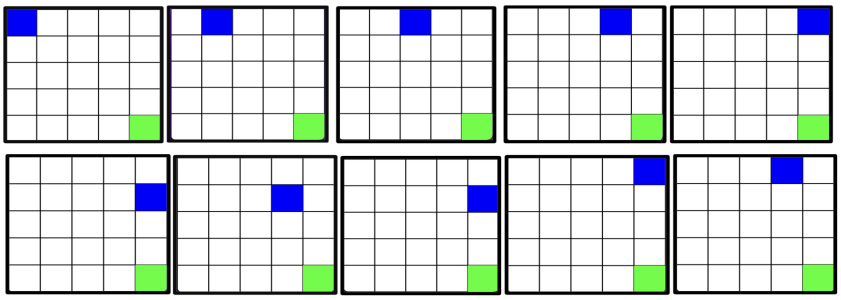


Figure 3: Trajectory of the random agent over the first 10 time steps of an episode

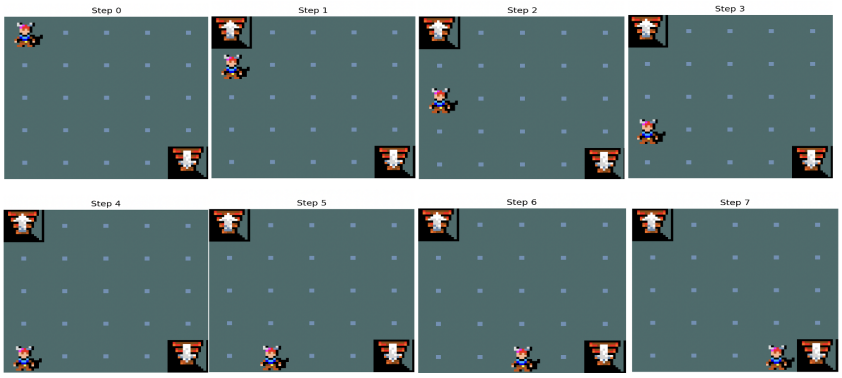


Figure 4: Fixed agent behavior in the EMPTY ROOM environment

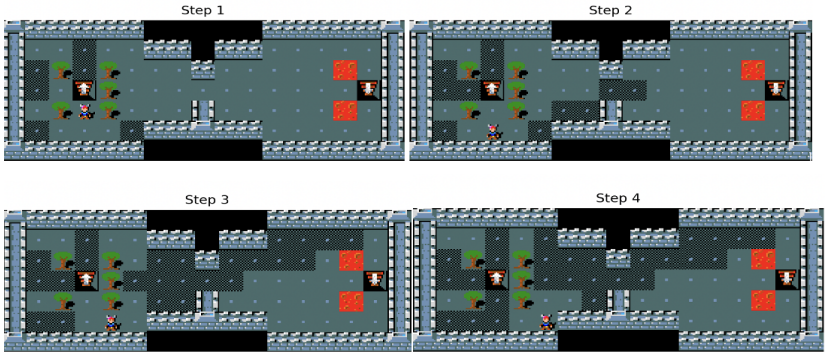


Figure 5: Trajectories learned by Q-learning (top row) and SARSA (bottom row) at various time steps.

Task 2

2.1

In figure 6 average return of montecarlo, sarsa and qlearning on four environments after 10,000 episodes is shown.

On-policy vs Off-policy Learning

In on-policy learning, the same policy is used for exploration, data collection, and improvement, using ϵ-greedy policy. The behavior policy and the target policy are identical. The value of a state under the current policy π is estimated directly from the returns obtained by following that policy:

$$V^{\pi}(s) = E_{\pi}\left[G_t \mid S_t = s\right]$$

where G_t is the return at time step t, and the expectation is taken over trajectories generated by the policy π.

In off-policy learning, the roles are separated. A behavior policy b is used to explore the environment and generate trajectories, which may not be optimal. A different target policy π is then evaluated and improved using the collected data. To estimate the value of the target policy from data generated by the behavior policy, we use importance sampling. This corrects for the distribution mismatch between the

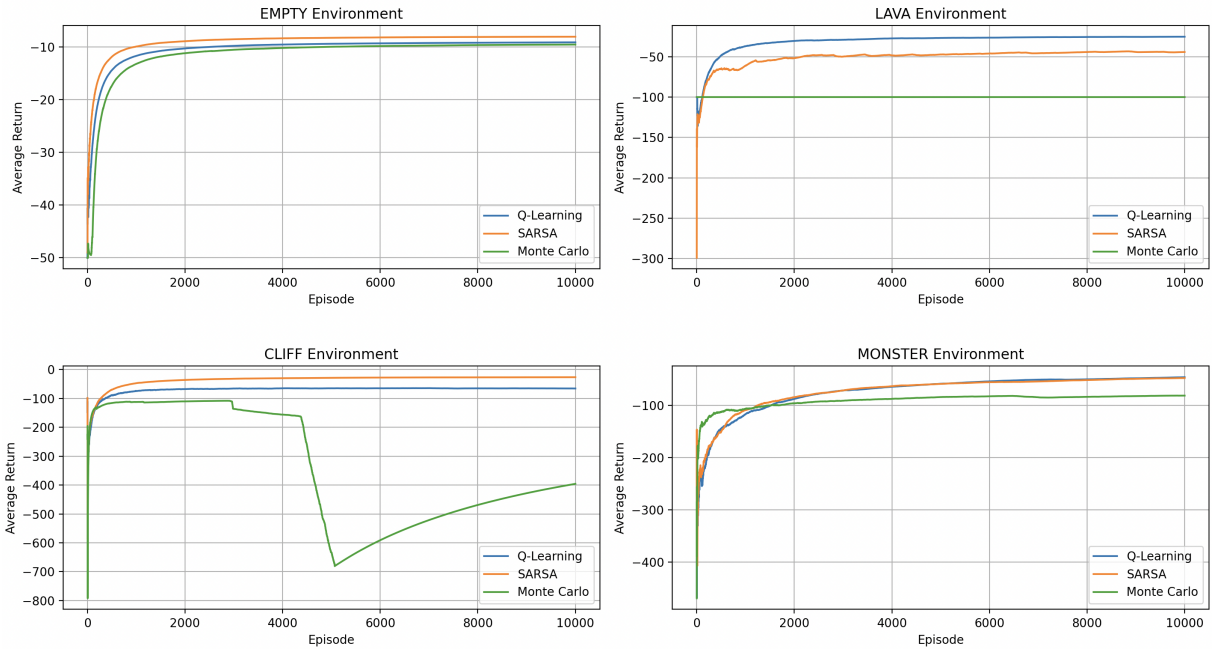


Figure 6: Monte Carlo vs Q-Learning vs SARSA Across Environments

two policies. The value function is estimated as:

$$V^{\pi}(s) = E_b \left[\rho_{t:T-1} \, G_t \mid S_t = s \right]$$

where the cumulative importance sampling ratio is defined as:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(a_k \mid s_k)}{b(a_k \mid s_k)}$$

This reweights the return G_t according to how likely the actions were under the target policy compared to the behavior policy, allowing accurate estimation of $V^{\pi}(s)$.
To compare on-policy and off-policy learning, it is more appropriate to focus on SARSA and Q-learning, as both use the temporal difference (TD) method and rely only on the next step to update their value estimates. Monte Carlo relies on full episode trajectories, which makes direct comparison unfair. In Figure 7, we observe that SARSA (on-policy) outperforms Q-learning (off-policy) in the EMPTY environment, which is a simple and low-risk setting. In the LAVA environment, Q-learning performs better. This is likely due to the room's high variability, where Q-learning benefits from its ability to explore more using a separate behavior policy. In the CLIFF environment, SARSA again outperforms Q-learning. This environment is risky, and Q-learning's tendency to favor high reward actions without considering immediate consequences often leads the agent into the cliff. In contrast, SARSA updates its values based on the actual actions taken, resulting in safer path. In the MONSTER environment, both methods perform similarly, suggesting that neither strategy has a clear advantage in this more dynamic settings. Overall, SARSA tends to perform better in environments with lower exploration demand and higher risk, while Q-learning is more effective in complex or highly variable environments where exploration is crucial.

Monte Carlo vs Temporal Difference:

In Monte Carlo, we estimate state values by averaging returns from complete episodes. To update the value of a state, we must wait until the episode terminates to compute the total return. This makes Monte Carlo learning episodic and slow:

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t - V(S_t) \right)$$

where

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

In contrast, Temporal Difference (TD) methods perform updates after every step. They use the immediate reward plus a discounted estimate of the next state's value. This approach is known as *bootstrapping*.

$$V(S_t) \leftarrow V(S_t) + \alpha \left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right)$$

TD methods are more sample-efficient and suitable for continuing tasks, while Monte Carlo is more accurate in the long run when full trajectories are available.
As shown in the average return plots, Monte Carlo performs worse across risky environments such as LAVA, CLIFF, and MONSTER. This is expected, as Monte Carlo relies on complete episodes to compute returns, which makes it highly sensitive to randomness and variance in the environment. The updates depend entirely on the sampled returns G_t, which can vary significantly across episodes, especially in environments with high penalties or stochastic transitions. In environments like CLIFF, where certain actions can lead to severe negative rewards, Monte Carlo may overestimate or underestimate the value of states due to inconsistent episode returns. This results in unstable and noisy learning. By contrast, Temporal Difference TD methods such as SARSA and Q learning perform updates at each time step using the immediate reward and an estimate of the next state's value. These methods may introduces some bias since they use other estimated values rather than the true return. However, this bias is brings an advantage of lower variance. This bias variance tradeoff works in favor of TD methods. While their updates may be slightly biased, their lower variance leads to more stable and efficient learning. This makes TD methods particularly effective in environments with complex dynamics or high risk, where frequent and incremental updates are necessary. In contrast, Monte Carlo methods tend to perform good only in safe and deterministic environments, where full episode returns are more consistent and reliable.

Different Learning Rates

The learning rate, denoted by alpha, is a key parameter in temporal difference learning that determines how much new information influences the current value estimates. A high learning rate allows the agent to adjust its estimates quickly based on recent experiences and accelerate learning. However, this can also lead to inaccurate updates, as it relies too heavily on the latest estimates and may overreact to noisy rewards. In contrast, a low learning rate results in slower updates but produces more stable and consistent learning over time, although it requires more episodes to reach convergence. Choosing an appropriate learning rate is essential to balance the trade-off between learning speed and stability. As shown in Figure 8, different learning rates can significantly affect the convergence behavior and stability of the learning process.

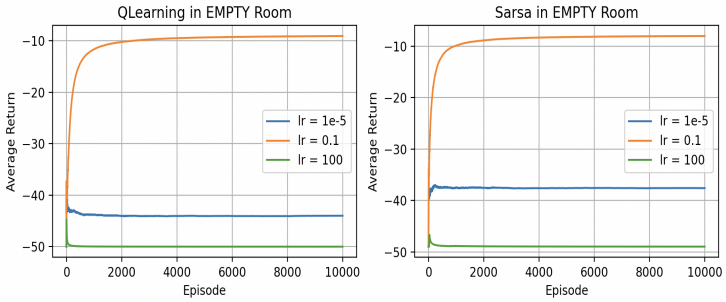


Figure 7: Effect of different learning rates on the convergence of temporal difference learning.

Different Epsilon for Exploration

In Figure 8, we observe the effect of using smaller and larger epsilon values. A smaller epsilon encourages the agent to exploit its current knowledge rather than frequently exploring random actions. This reflects the well-known trade-off between exploitation and exploration. With $\epsilon = 0.01$, the agent selects the action with the highest estimated value. This allows it to refine its policy with lower variance. As a result, the learning process becomes more stable and converges faster toward an optimal policy. In contrast, a high epsilon value, such as $\epsilon = 0.9$, causes the agent to explore a lot, even when effective policies have already been learned. This results in not using the knowledge acquired so far. This high level of randomness introduces noise into the learning process, delaying convergence. This effect is particularly evident in simple environments such as the EMPTY room, where high exploration is unnecessary.

2.2

As illustrated in Figure 9a, Q-learning follows a riskier path, while SARSA goes for a safer route. This difference is because of the nature of their learning strategies. Q-learning is an off-policy method. It learns the optimal greedy policy independently of the agent's current actions. As a result, it often chooses a faster but more dangerous path along the cliff edge. In contrast, SARSA is an on-policy algorithm that learns from the agent's actual behavior, which includes exploration through the epsilon greedy policy. To reduce the chance of falling off the cliff during exploration, SARSA learns to favor a safer path, even if it takes longer to reach the goal.

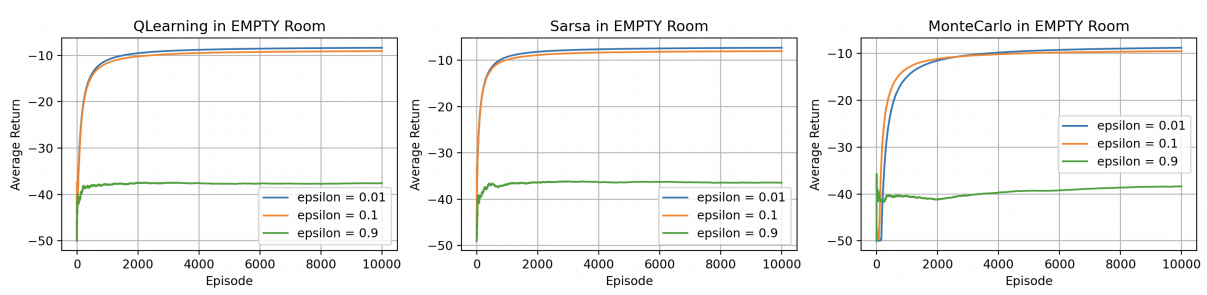
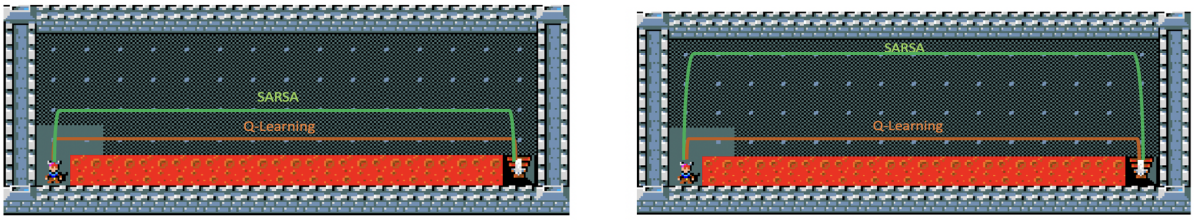


Figure 8: Effect of different epsilon on the convergence of different methods.

As shown in Figure 9b by applying epsilon decay, we can see that SARSA takes a safer path, while there is no change in the strategy learned by Q-learning. This is because SARSA's updates depend on the actual actions taken during training, which are influenced by the current exploration rate. As epsilon decreases, SARSA shifts toward a more deterministic and safer policy. In contrast, Q-learning is off-policy and always updates toward the greedy action, regardless of the actual behavior. Therefore, the decaying exploration rate has no impact on the learned strategy in Q-learning. Figure 10 illustrates



(a) Before epsilon decay

(b) After epsilon decay

Figure 9: Strategies learned by Q-Learning and SARSA in the CLIFF environment after 200 episodes.

sample trajectories of Q-learning and SARSA agents at stages 20, 50, 100 episodes from running 200 episodes in total. At time step 20, both agents shows random behavior due to a high exploration rate. Q-learning behaves inconsistently at this point because it updates its Q-values based on the assumption that future actions will always be greedy, even though the agent is currently exploring. This results in unstable value estimates and an unclear path. In contrast, SARSA updates its Q-values using the actual actions taken, which include exploratory moves. As a result, SARSA begins to form a more cautious policy early in training, even though the policy is not fully effective. By time step 50, Q-learning has started to stabilize and moves more directly toward the goal, while SARSA continues to avoid the cliff area. At time step 100, Q-learning has converged to a greedy policy and efficiently reaches the goal using a shorter but riskier route along the cliff. Meanwhile, SARSA still accounts for exploration in its updates and maintains a more conservative trajectory.

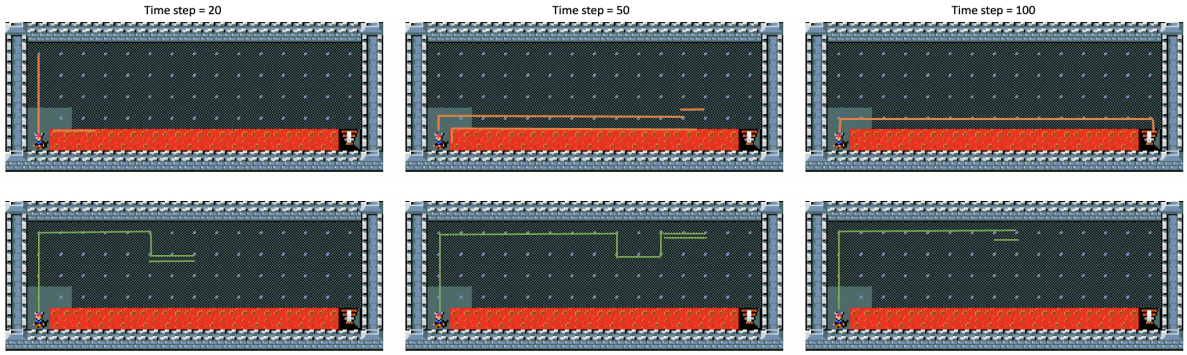


Figure 10: Trajectories learned by Q-learning (top row) and SARSA (bottom row) after 200 episodes.

2.3

Q-learning and Dyna-Q are different in how they learn and use experience. Q-learning is a model-free method, meaning it learns by interacting with the environment. Each time the agent takes an action and observes the outcome, it updates its knowledge directly based on that real experience. It does not build an internal representation or model of how the environment behaves. As a result, Q-learning tends to learn more slowly, especially in complex environments, because it only updates its knowledge one step at a time using actual interactions. In contrast, Dyna-Q is a hybrid approach that combines both model-free learning and model-based planning. Like Q-learning, it updates its action values from real experience. However, it also builds an internal model of the environment by recording the outcomes of its actions. The model is then used to simulate past experiences, allowing the agent to update its values and improve its policy through internal planning, as if interacting with the environment. In summary, Q-learning learns only by doing, while Dyna-Q learns by doing and imagining. By integrating planning steps through the use of a learned model, Dyna-Q accelerates learning as shown in Figure 11 and policy improvement. However, it requires more memory and computation to store and update the model.

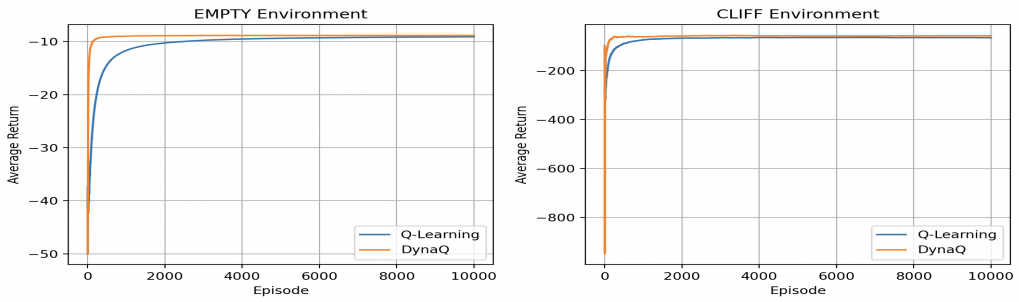


Figure 11: Dyna-Q vs Q-Learning in Empty and Cliff Environment

Task 3

3.1

Classical reinforcement learning provides a theoretical foundation for understanding how agents can learn to make decisions. With function approximation, agents gain the ability to generalize their learning in larger and more complex environments. Deep reinforcement learning builds upon this by improving model architectures and optimization techniques that make the learning process more efficient and stable. However, these model still faces several important challenges. In supervised learning, training data is usually assumed to be independent, and each input has a fixed target. In contrast, reinforcement learning data is gathered through an agent's interaction with the environment, resulting in consecutive states and actions that are highly correlated. This strong correlation makes it difficult to train neural networks and cause overfitting. Additionally, the learning targets in reinforcement learning are not fixed. Because of bootstrapping, the targets themselves keep changing. This means the agent is learning from unstable and moving targets, which can lead to unreliable training.
The Deep Q-Network (DQN) is a reinforcement learning algorithm that combines Q-learning with deep neural networks, enabling agents to learn optimal actions in environments with high-dimensional state spaces. Unlike traditional Q-learning, which relies on a Q-table to store action values for each state, DQN uses a neural network to approximate the Q-function. This also allows it to generalize across similar states. To address the above mentioned issues, experience replay is used, where the agent stores past experiences in a memory buffer and trains on randomly sampled mini-batches. This helps break the correlation between consecutive training samples. Another key technique is the use of target networks, where a second slow network is used to calculate the target Q-values. There are also improved extensions of the basic DQN algorithm. Double DQN reduces overestimation of Q-values by using two network. One to select the best action and another to evaluate its value. Dueling DQN separates the estimation of the state value and the advantage of each action, helping the agent learn more effectively. Lastly, Distributional DQN allows the agent to learn the entire distribution of possible future rewards, rather than just the expected value.
Reinforcement learning is the process of learning how to act. Traditional methods estimate action values (Q-values) and derive a policy from them, whereas policy gradient methods optimize the policy directly with respect to expected return. One foundational method is REINFORCE, which updates the policy using the total return from each episode. However, this approach suffers from high variance due to

randomness in the environment, leading to unstable and slow learning. To mitigate this, REINFORCE with Baseline introduces a baseline, typically the state value function, which reduces variance by centering the updates around how much better an action performs compared to the average. The Actor-Critic method further improves stability by combining a policy network (actor) with a value network (critic) and using the advantage function, which measures how much better an action is than the expected value of the state. However large updates in these methods can cause instability, so techniques like TRPO (Trust Region Policy Optimization) and PPO (Proximal Policy Optimization) have been proposed. TRPO uses a KL divergence constraint to limit policy changes. PPO simplifies optimization with a clipping strategy to prevent large updates, offering better efficiency and stability. Despite these advances, direct policy learning remains sensitive to architecture and prone to early convergence to deterministic policies, which reduces exploration. To maintain stochasticity, KL divergence from a random policy can be minimized. This is closely related to maximizing entropy, which encourages exploration by favoring more diverse and uncertain action choices. This term is then added to PPO objective and create the final loss function.

3.2

For this experiment, I used a multi-layer perceptron neural network as a function approximator. Figures 12 and 13 show the episode rewards for PPO and DQN, respectively, after 50,000 time steps of training in the Empty Room environment. Both algorithms eventually converged to a final reward of -8, which corresponds to the optimal policy. However, PPO reached this optimal performance much earlier,around episode 4,000, while DQN required approximately 15,000 episodes to converge. Although PPO exhibits some fluctuations after convergence, it still demonstrates faster and more stable learning compared to DQN. This indicates that PPO converges more quickly, even though both methods achieve the same final reward.

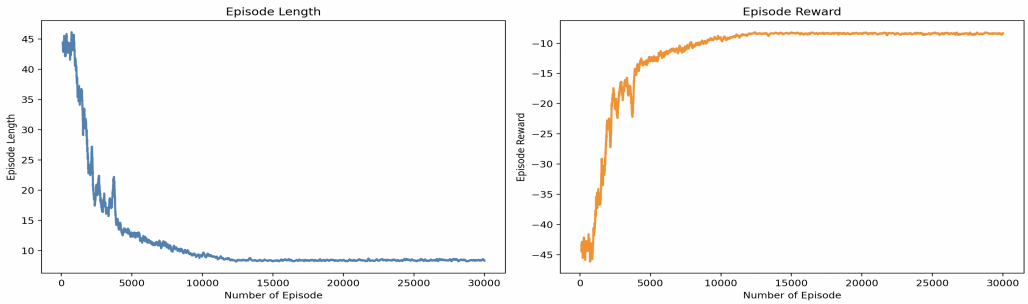


Figure 12: DQN episode reward and episode length in Empty Room

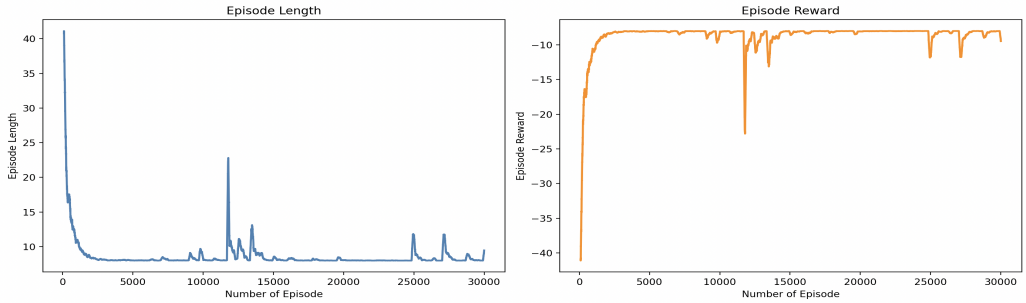


Figure 13: PPO episode reward and episode length in Empty Room

Figures 14 and 15 show the results after 2,000,000 time steps of training in the Multiple Monsters room. In this environment, DQN shows fluctuations and converges significantly slower than PPO. In contrast, PPO reaches stable performance early and maintains it throughout training. Additionally, PPO achieves a final average reward of approximately -12, while DQN settles around -15. This performance can be attributed to the difference in learning strategies. PPO directly optimizes the policy through gradient, whereas DQN estimates Q-values and selects actions based on them. In complex environments like the Multiple Monsters room, this makes DQN more sensitive to instability and delayed convergence.

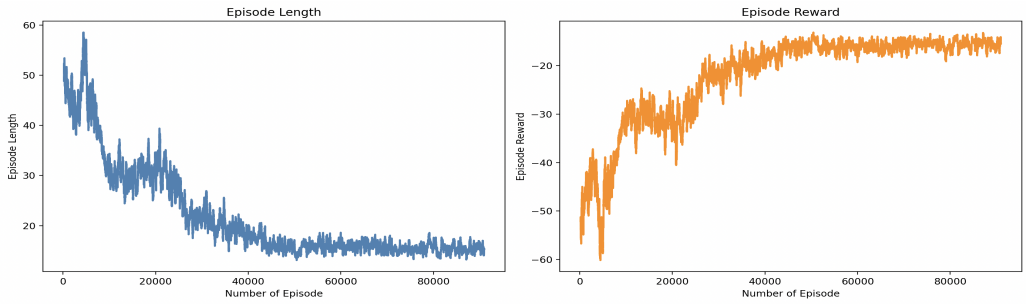


Figure 14: DQN episode reward and episode length in Multiple Monsters room

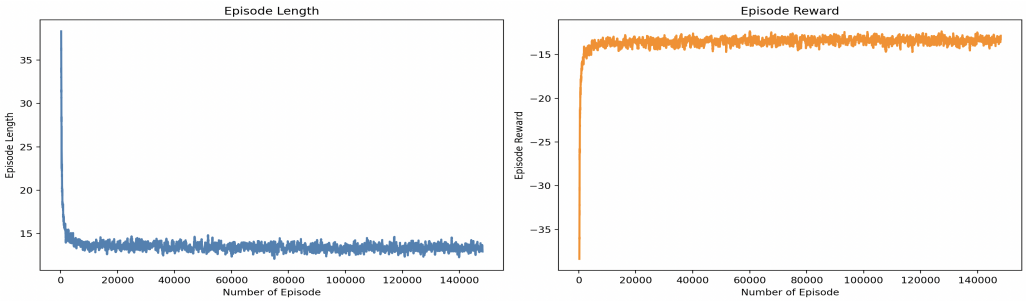


Figure 15: PPO episode reward and episode length in Multiple Monsters room

3.3

Table-based Q-learning is simple, efficient, and well-suited for environments with small, discrete state and action spaces. It maintains a table of Q-values and updates them directly using the Bellman equation, making it easy to implement and interpret. However, this approach lacks scalability and generalization, as it requires every state-action pair to be explicitly stored and updated. This becomes impractical in environments with large or continuous state spaces. In contrast, Deep Q-Networks (DQNs) use neural networks to approximate the Q-function, enabling them to handle high-dimensional and continuous state spaces. They can generalize across similar states, which is a key advantage in complex environments. However, DQNs are more computationally intensive, require extensive hyperparameter tuning, and can be unstable during training without additional techniques such as experience replay and target networks. As shown in Figure 16, tabular Q-learning converges faster than DQN in simple environments like the Empty Room, reaching the optimal policy with fewer episodes and more stable performance. However, in more complex environments with larger state spaces, such as the Monster Rooms,tabular Q-learning struggles to learn effective policies. From Figure 6, we observe that tabular Q-learning converges to a reward of around -80 after 10,000 episodes, while DQN achieves a much better final performance of around -40 in the more complex room with multiple monsters. This demonstrates DQN's ability to handle more complex tasks despite its slower convergence.

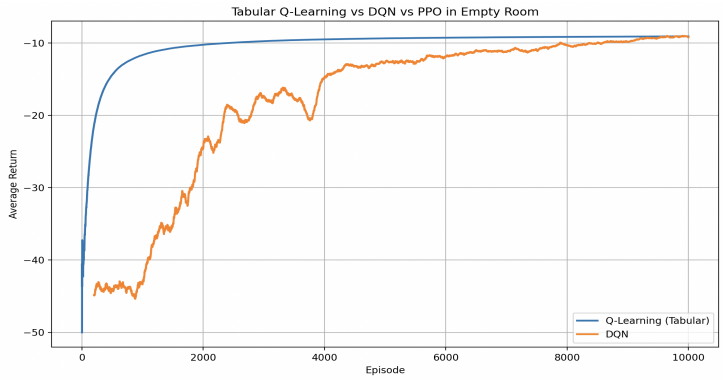


Figure 16: Tabular vs Deep Q-learning approach in the Empty Room