CMPUT628 - Deep Reinforcement Learning
Assignment 2

February 8, 2025

Dikshant
1877098

---

1. What exploration strategy are we using in Mountain Car, and why does it work?

We are using a pure greedy exploration strategy. The environment's unique characteristics make this possible: the car needs to build momentum by moving back and forth to reach the goal, and even a greedy policy can discover this pattern due to several key factors. First, the optimistic initialization (weights set to zero while actual rewards are negative) encourages exploration of different actions. Second, the environment's random starting positions provide natural exploration opportunities. Third, the continuous state space combined with tile coding enables generalization, where learning in one state automatically transfers to similar states. This generalization is crucial because actions that work well in one state are likely to work well in similar states. These elements work together to make pure greedy exploration surprisingly effective in the Mountain Car environment.

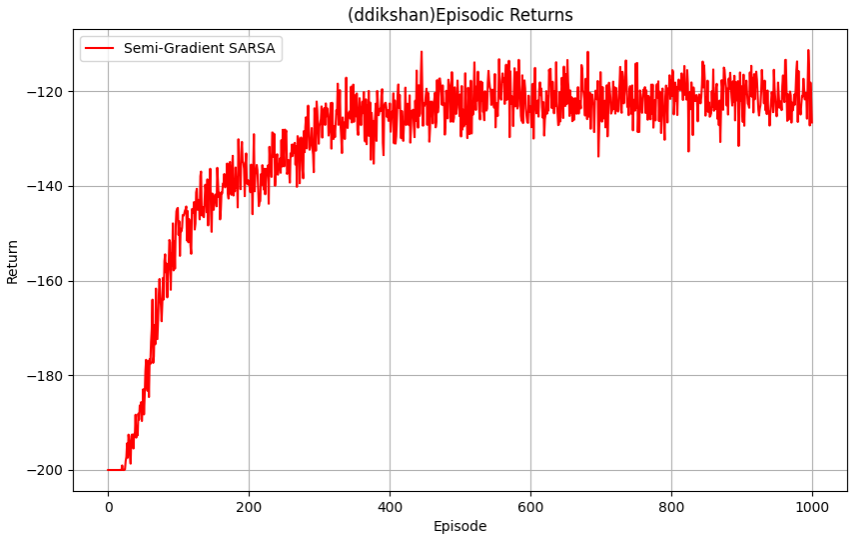

Figure 1: SARSA - Mountain Car

2. Performance of Jumping Task

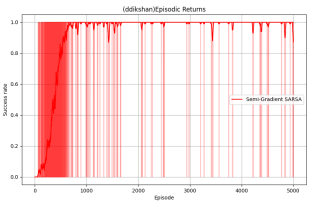

Figure 2: Config 1

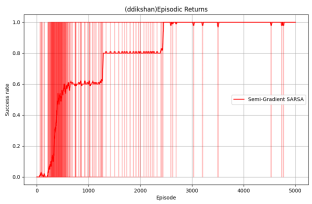

Figure 3: Config 2

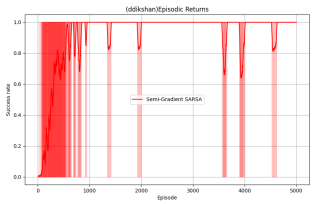

Figure 4: Config 3

Command line to reproduce the results: `python train_jumping_task.py`

3. Features

1. Feature Design

a. *Describe your features and how/why you produced them.*

The features are designed to capture the essential aspects of the environment that are relevant for the agent to make effective decisions. Here are my features:

• `agent_height`: Normalized height of the agent. Helps the agent understand its size relative to the environment.

• `distance_from_goal`: Normalized distance to the right end of the screen. Encourages the agent to move toward the goal.

• `distance_from_obstacle`: Normalized distance to the obstacle. Helps the agent decide when to jump to avoid the obstacle.

• `jump_height`: Normalized maximum height the agent can reach during a jump. Helps the agent understand its jumping capability.

• `in_air`: Binary feature indicating whether the agent is in the air.

• `action_indicator`: Binary feature indicating whether the current action is "up" or "right".

• `optimal_jump_point`: Binary feature indicating whether the agent is at the optimal jump point for the obstacle. For action 0: Checks if `distance to obstacle > obstacle height` (safe to move right) and for action 1: Checks if `distance == obstacle height` (optimal jump point)

For `config 3` with two obstacles, I just add similar features to solve the task.

b, c, d. *Feature Evolution and Generalization*

I will try to answer all three questions together. I figured out a key insight about the jumping mechanics: there exists a unique optimal jumping point where the distance from the agent's rightmost point to the obstacle's leftmost point equals the obstacle height. Thus, the current features works independently of agent/obstacle widths, generalizes across all three configuration, currently assumes unit velocity (1 pixel/time_step) but can be parameterized for different velocities. Rather than maintaining separate feature sets, I developed a unified feature representation that generalizes across tasks through proper normalization and the encoding of this fundamental jumping principle.

e. *Design Considerations*

All distance and height features were normalized by the environment's dimensions to ensure scale invariance, enabling better generalization across different configurations. I made groups of obstacles based on adjacent pixels to identify distinct obstacles and sorts them by distance, though the current features are limited to handling only two obstacles currently.

2. *Solution Description and Challenges*

The implementation uses N-Step Semi-Gradient SARSA (N=7) to address the exploration and credit assignment challenges inherent in this environment. The N-step returns significantly improve credit assignment by allowing reward information to flow back through multiple timesteps, which is crucial for learning the optimal jumping sequences. For exploration, I implemented an adaptive ϵ-greedy strategy starting with ϵ=1.0 and gradually decaying it over time, balancing exploration of the state space with exploitation of learned policies. Adaptive size made the agent learn even quicker but I am not doing that in current implementation. The main challenge that I faced was addressing learning instability, particularly with multiple obstacles. Thus, I used N-step Semi-Gradient SARSA to make credit assignment better. I also wanted same set of hyperparamters to work for all three configs, so I tuned hyperparameters for stability. While the solution handles one or two obstacles effectively, scaling to arbitrary numbers of obstacles remains a challenge due to the fixed-size feature vector implementation.

3. *Challenges with Different Colors and Sprites*

No, it will not work because the current feature extractor relies on specific pixel values (0.0, 0.5, 1.0) to identify the agent, obstacles, and floor. If the environment uses different colors or sprites, the feature extractor may fail to detect these elements. Even with different shapes from rectangle/square, my feature extractor will fail because of the way I am extracting them from the observations.