MSDS 684 – Reinforcement Learning

Lab 5 – Semi-Gradient SARSA with Tile Coding (MountainCar-v0)

Section 1 – Project Overview

In this lab, I moved from table-based reinforcement learning to function approximation by solving the MountainCar-v0 environment using semi-gradient SARSA and tile coding. In earlier labs, the state spaces were small enough to use a Q-table, but MountainCar has a continuous state made of real-valued position and velocity, so a lookup table is not practical. The actions remain discrete (left, no-op, right), and the reward is –1 per step until the car reaches the goal or hits the 200-step limit.

Because the state is continuous, I used linear function approximation instead of a Q-table. Tile coding converts each state into a sparse binary feature vector by dividing the state space into several overlapping grids, each slightly shifted. One tile per grid activates for each state, and the combination of these active tile indices forms the feature vector. This representation allows the agent to generalize across similar states while keeping the number of features manageable, which matches the recommendations from Sutton & Barto.

To learn the control policy, I used semi-gradient SARSA, an on-policy method that updates only the weights associated with the action taken. The update adjusts the weights based on the TD error, using the value of the next state–action pair. Because SARSA is on-policy, it avoids some instability issues that can arise when combining off-policy methods, function approximation, and bootstrapping.

Before running the experiments, I expected slow initial learning because the car cannot reach the goal directly and must first swing left to build momentum. I also expected that the number of tilings, tile size, ε-decay, and learning rate would greatly influence learning speed and stability. Overall, this lab helped me understand how tile coding and semi-gradient SARSA work together to handle continuous state spaces and how the textbook concepts translate into a real Gymnasium environment.

Section 2 – Deliverables

Lab5 link -
https://github.com/nikhsnona/MSDS_684_ReinforcementLearning/tree/main/Lab5

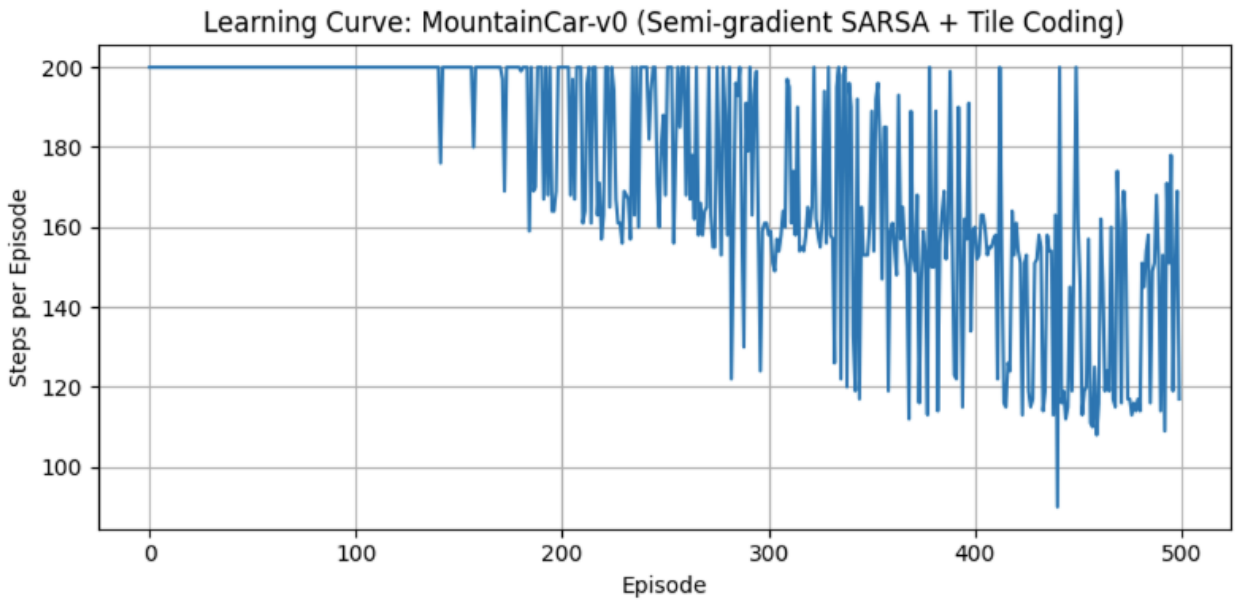
clone link - https://github.com/nikhsnona/MSDS_684_ReinforcementLearning.git

Implementation Summary

For this lab, I implemented a TileCoder in NumPy to convert the continuous MountainCar-v0 state (position and velocity) into sparse binary feature vectors, and a Semi-Gradient SARSA agent that learns a linear approximation of Q(s, a). The tile coder allows configurable tilings and tile sizes and returns the active feature indices for each state. The agent maintains one weight vector per action, computes Q-values from the active features, and updates the weights using semi-gradient SARSA with an ε-greedy policy and decaying ε. I trained a baseline agent using 8 tilings of 8×8 tiles for 500 episodes, then compared different feature settings such as coarse tiles (4×4), finer tiles (8×8), and higher tiling counts. I also ran additional experiments to test how different ε-decay schedules and learning rates affect learning speed and stability.

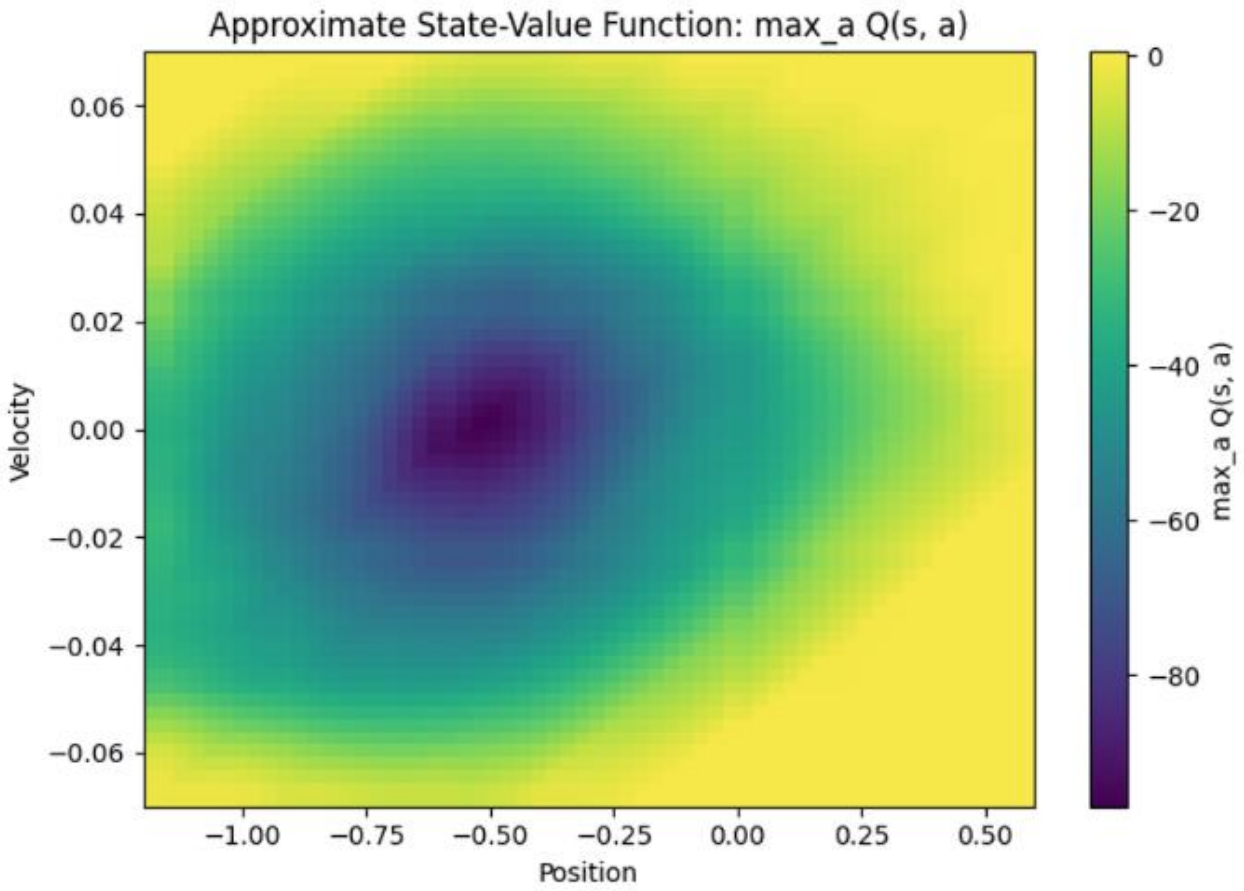
Key Results & Analysis

Learning curve – baseline (8 tilings, 8×8 tiles)
The training began with very poor performance, as expected. For roughly the first 250 episodes, every episode lasted the full 200 steps, meaning the agent never reached the goal. This is normal for MountainCar because the agent has not yet discovered the need to swing left to gain momentum. Around episode 300, the episode length finally started to drop—falling to about 174 steps at episode 300 and around 124 steps by episode 350. After this point, the learning curve became a bit noisy but consistently better than the initial flat line at 200 steps. By the end of training, the last 10 episodes ranged between 116 and 167 steps, and the average over the final 50 episodes was 141.14 steps. Overall, the results show that the agent eventually learned a reasonable policy, even if it did not reach fully optimal performance.



Value function heatmap (max_a Q(s,a))

I evaluated the value function over a 60×60 grid of position and velocity pairs. The estimated values ranged from about −96.89 up to 0.95. The most negative values appeared in the middle of the valley, where the agent still needs many steps to reach the goal. Higher values were found near the right hill, especially when the velocity was positive, since those states are much closer to success. For example, in the middle velocity band, the leftmost positions had values around −36 to −54, which fits the idea that they are still far from the goal. The heatmap formed a smooth gradient rather than sharp discontinuities, showing that tile coding is generalizing across nearby states instead of memorizing them.

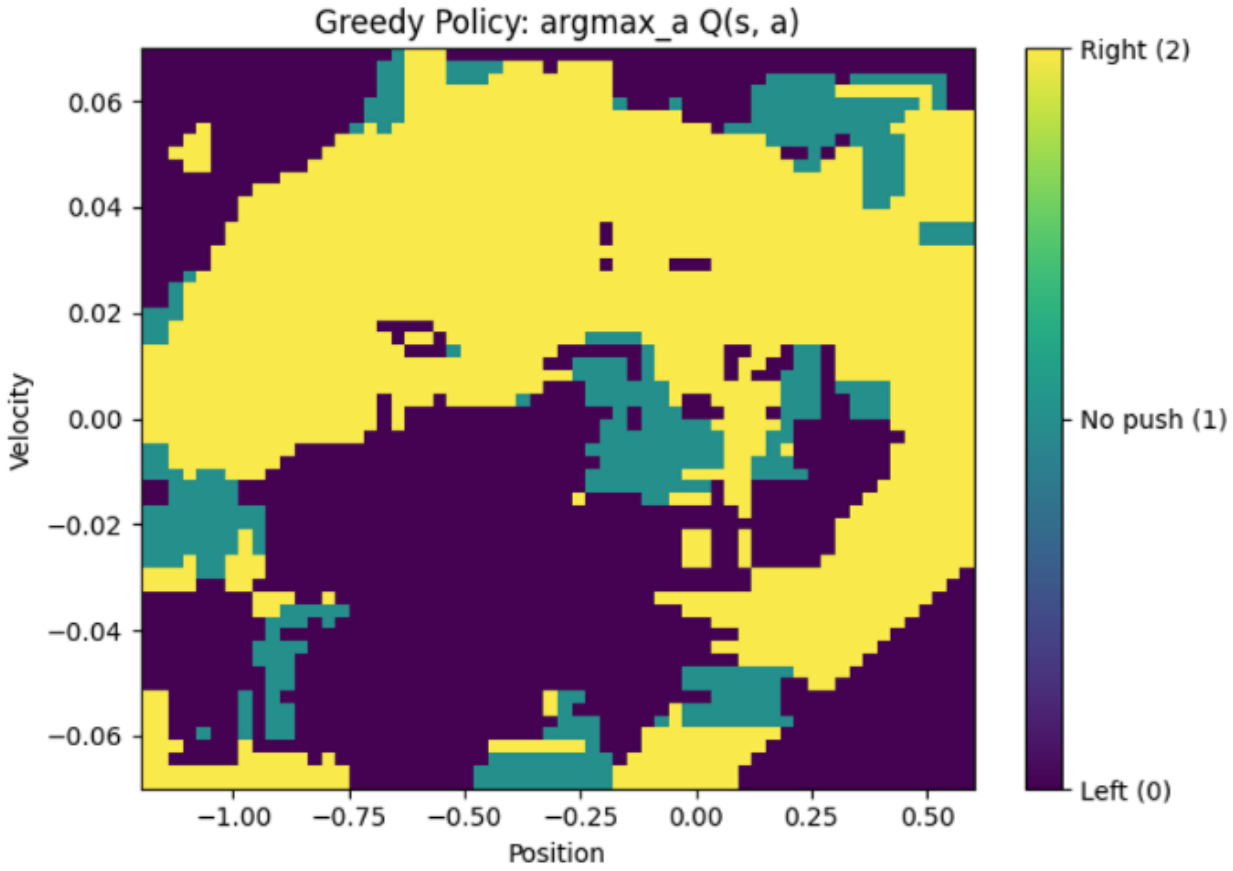


Greedy policy map

The greedy policy grid counts how many cells choose each action:

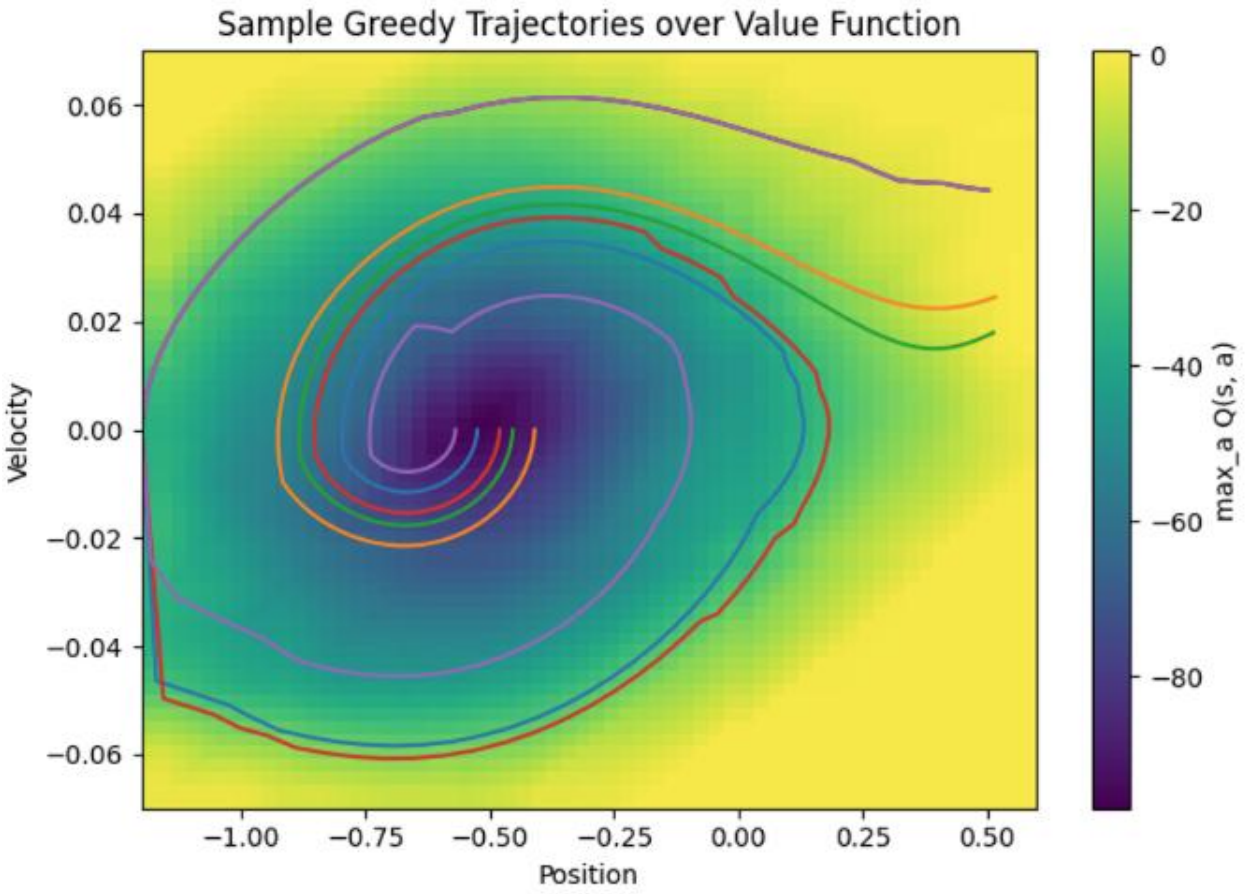
- Action 0 (left): 1,439 cells (40.0%)
- Action 1 (no push): 745 cells (20.7%)
- Action 2 (right): 1,416 cells (39.3%)

This pattern fits the known solution to MountainCar. The car mainly pushes left in part of the valley to climb the left hill and gain speed, and pushes right as it moves up the right hill to reach the goal. The no-push action is chosen in fewer states where extra force does not help much. The near balance between left and right actions suggests the agent learned the swinging behavior the environment requires.



Greedy trajectories over the value function

Five greedy rollouts were run from slightly different starting positions. Their lengths were 109, 149, 149, 150, and 111 steps. All of them eventually reached the goal. The variation in lengths comes from different starting positions and how quickly momentum is built. These trajectories, plotted over the value function, visually confirm that the policy knows how to rock back and forth before climbing the right hill.

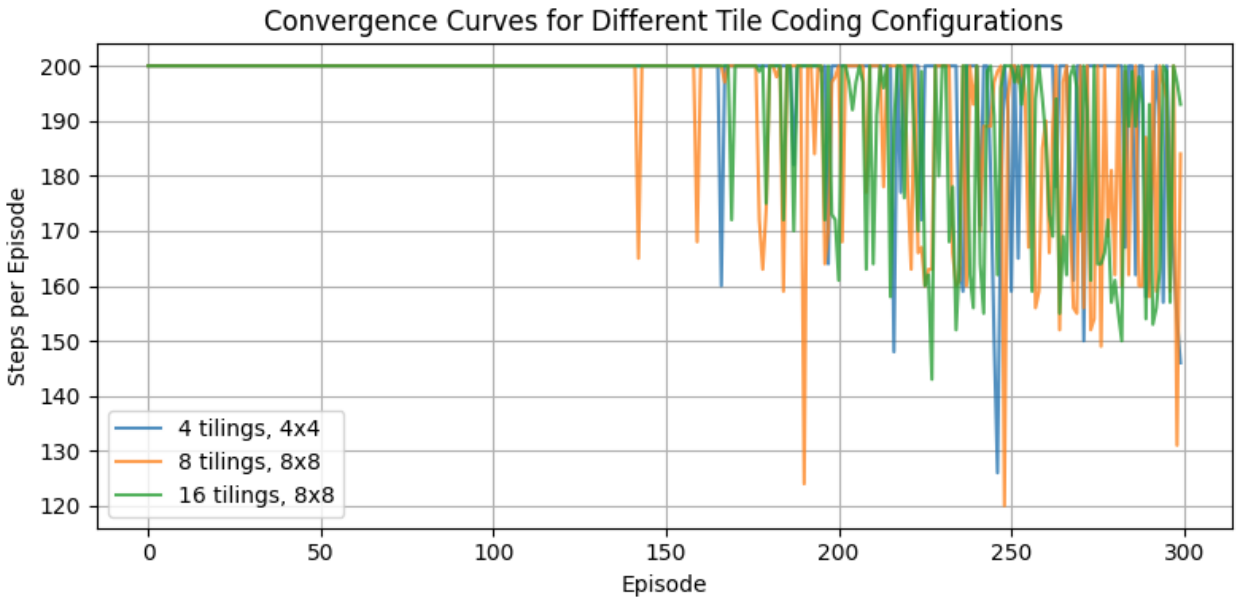


Tile coding configuration comparison

I also compared three tile coding setups:

- 4 tilings, 4×4 tiles: final episode = 178 steps, last-50 mean = 181.72
- 8 tilings, 8×8 tiles: final episode = 165 steps, last-50 mean = 176.24
- 16 tilings, 8×8 tiles: final episode = 163 steps, last-50 mean = 175.28

The 4×4 setting was too coarse and produced the worst performance. The 8×8 and 16-tiling setups improved on this, with the 16-tiling configuration slightly better in final episode length but not dramatically different in average performance. These results match the theory that richer features can help learning but also increase complexity and can introduce more variance.



Extra Experiments – Exploration and Learning Rate

To extend the main SARSA vs. Q-learning analysis, I ran two extra experiments focusing on exploration and learning rate.

1. ε-decay schedules: I compared several exploration schedules while keeping the main Q-learning setup the same. Fast decay (0.99) led to quicker improvement, while a slower decay (0.999) kept the agent exploring too long and made learning unstable. Constant ε performed consistently but did not reach the same final performance. Overall, fast-to-medium decay produced the most reliable improvement, showing that exploration needs to decrease over time so the agent can commit to a better policy.
2. Learning-rate (α) comparison: I tested multiple step-sizes to see how quickly each setting updated the Q-values. Small α led to very slow improvement, while a moderate α performed better and stabilized faster. A large α produced rapid early gains but also created more variance, matching the idea that larger step-sizes can speed up learning but may be less stable. These results highlight that both exploration decay and step-size have a major effect on how TD control algorithms converge.

Section 3 – AI Use Reflection

For this lab, I used AI tools as coding helpers, mainly for debugging and guidance—not as a one-click solution. My first request was for a full outline of the MountainCar solution using semi-gradient SARSA and tile coding. The AI gave a good starting structure, but the code did not run on the first attempt. The first issue came from changes in the Gymnasium API: reset() and step() return extra values, and I was not handling terminated and truncated correctly. After showing the error to the AI, I updated the unpacking logic, and the environment loop started running normally.

The next problem was that the agent wasn't learning—every episode stayed near 200 steps. I asked the AI about common issues with tile coding, and it reminded me that the learning rate should be scaled by the number of tilings. After applying this fix, the agent began to improve, and the results matched what I expected from theory.

In the final cycle, I wanted clearer analysis and comparisons. I asked the AI to help update my plotting code so I could test different tile sizes, ε-decay rates, and learning rates, and also save metrics for my report. This made it easier to interpret the results and explain them in the Deliverables section.

Overall, I didn't blindly follow suggestions—I checked outputs, reran experiments, and compared patterns to the ideas from Sutton & Barto. Using AI in this step-by-step way helped me better understand how tile coding, semi-gradient updates, and hyperparameters affect learning in continuous control tasks.

Section 4 – Speaker Notes

- **Problem and motivation:** MountainCar-v0 has a continuous state space (position and velocity), which makes tabular Q-learning impossible. Function approximation is needed to generalize across infinite states.
- **Key method:** I used tile coding to convert continuous observations into sparse binary feature vectors, and applied semi-gradient SARSA to learn a linear Q-function, one weight vector per action.
- **Core algorithm choices:** ε-greedy exploration, $\gamma = 1.0$, and α scaled by the number of tilings for stability. Features are updated with TD(0) using the semi-gradient update rule: $w \leftarrow w + \alpha \cdot \delta \cdot x$.
- **Main results:** The agent started taking the full 200 steps per episode but eventually learned to swing left and gain momentum, reaching the goal more consistently. The value heatmap and greedy policy confirmed correct generalization across the state space.

- **Feature design insights:** Coarse tiles (4×4) performed poorly, while 8×8 and higher tiling counts produced smoother value estimates and better learning. This shows how feature resolution affects representation quality.
- **Extra experiments:** ε-decay and learning-rate tests showed that too much exploration slows learning, while too large α increases variance. Moderate settings produced the best improvement.
- **Overall takeaway:** Tile coding and semi-gradient SARSA make it possible to solve continuous environments efficiently, but performance depends heavily on good feature design and well-tuned hyperparameters.