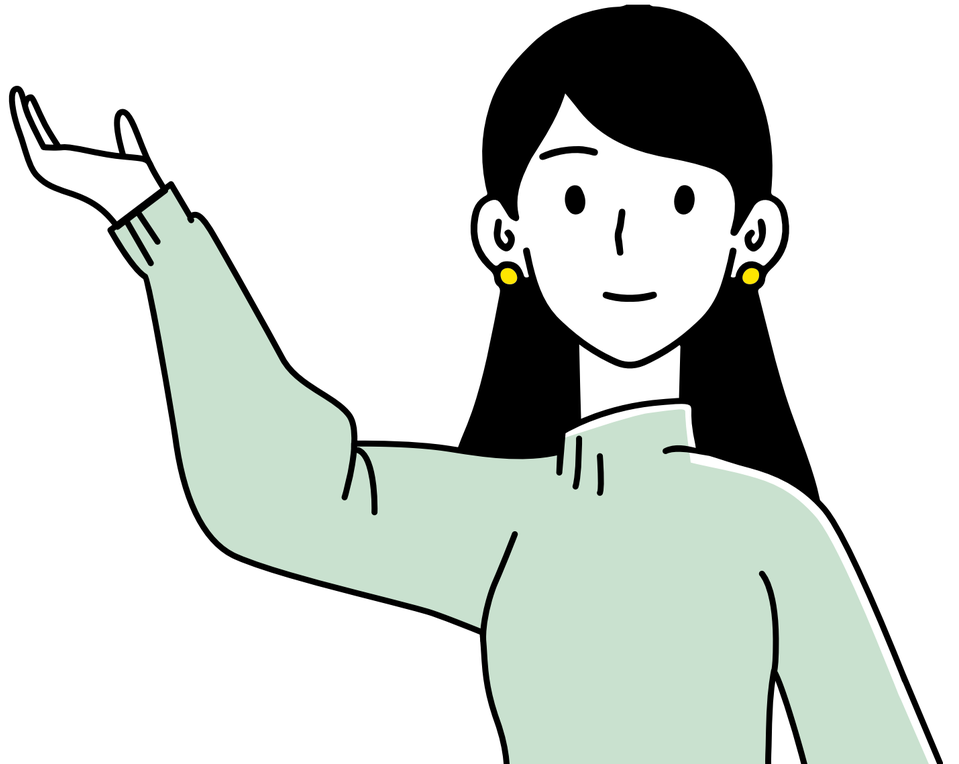
highway-env

Group 9



Today's Agenda

1. What is Lane-Keeping Problem
2. Tabular methods:
3. Function approximation
4. Results and Discussion

Part 1: What is Lane-Keeping Problem

Environment Description

The highway environment simulates a multi-lane highway driving scenario. The ego-vehicle (the agent) is placed among multiple traffic participants and must maintain safe and efficient driving behavior. This env gathers information for decision-making in autonomous driving.The environment offers a discrete action space and a continuous state space, making it suitable for evaluating both tabular and functional approximation reinforcement learning methods

Observations

The environment has a default observation, which can be changed or customised using environment configurations.

Ego-vehicle:

- Speed
- Position in the lane (lateral position)
- Heading direction (angle relative to road)

Other vehicles :

- Relative positions of nearby cars
- Relative speeds of nearby cars

Action

The environment has a different action space type: continuous, discrete and discrete Meta-Actions

ACTIONS :

- 0: LANE_LEFT
- 1: IDLE
- 2: LANE_RIGHT
- 3: FASTER
- 4: SLOWER

The agent’s goal is to:

- Maintain lane center as long as possible.
- Avoid collisions with other vehicles.
- Drive efficiently by keeping a target speed.

To guide this behavior, a shaped reward function is used, with components that:

- Penalize collisions and off-road behavior.
- Encourage staying in the right lane.
- Reward forward movement at optimal speed

Constraints and Challenges:

- Continuous state space
- Sparse rewards
- Dynamic interactions

Evaluation Metrics:

- Average cumulative reward per episode.
- Collision frequency across evaluation episodes.
- Lane deviation and time spent centered in lane.

Part 2: Methodology

Tabular Methods

Step 1: State Discretization

The highway-env environment provides a continuous observation space. To enable tabular learning, ***we discretized*** the ego vehicle's features:

- position (x), lane (y) and forward velocity (vx). Using ***np.digitize()*** with fixed bin intervals. This transformation yielded a compact and finite state representation: (x bin, y bin, vx bin).

Dynamic Hyperparameter Tuning:

- ϵ-greedy exploration was initialized at 1.0 and decayed exponentially to 0.05 using a factor of 0.995.
- The learning rate α started at 0.1 and decayed to a minimum of 0.01 with a decay rateof 0.99

env's Reward

The rewards are normalized to a fixed range, between 0 and 1, which helps stabilize learning and makes the optimal policy invariant to scaling or shifting.

Negative rewards are avoided

Agent optimizes only for one goal, which can lead to unrealistic or unsafe driving

Slower because the agent has fewer clues on how to improve. The agent couldn't generalizeto stable the rewards.

Our shaped reward

A composite reward combining multiple objectives (speed, safety, lane choice, lane stability).

Speed reward= +2 , Collision penalty = -10

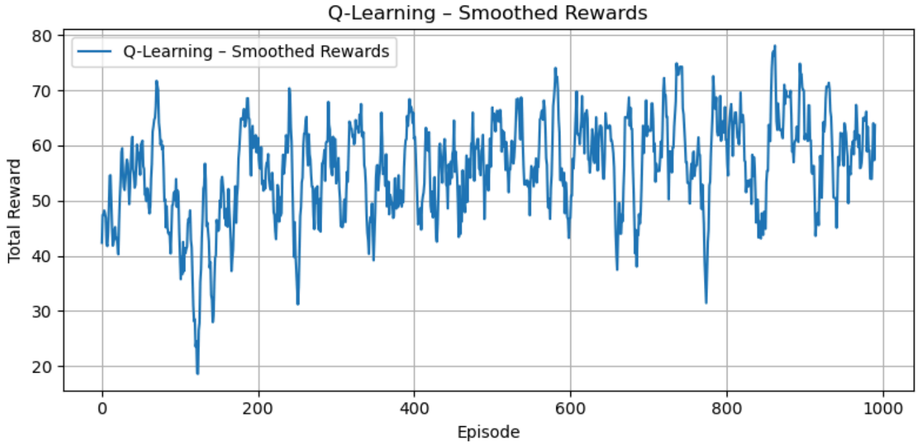
A composite reward combining multiple objectives (speed, safety, lane choice, lane stability).

Produces more balanced and desirable behavior that aligns with real-world driving expectations.

We evaluated three tabular reinforcement learning methods: Q-learning, Double Q-learning, and Prioritized Sweeping. All models used a unified setup involving state discretization, a custom reward function, and dynamic hyperparameter tuning. The agent's state was encoded as a discretized tuple (x bin, y bin, vx bin), and the reward function incentivized efficient, collision-free driving with a preference for staying in the rightmost lane. The agents were evaluated over 1000 episodes, tracking both episodic rewards and episode lengths

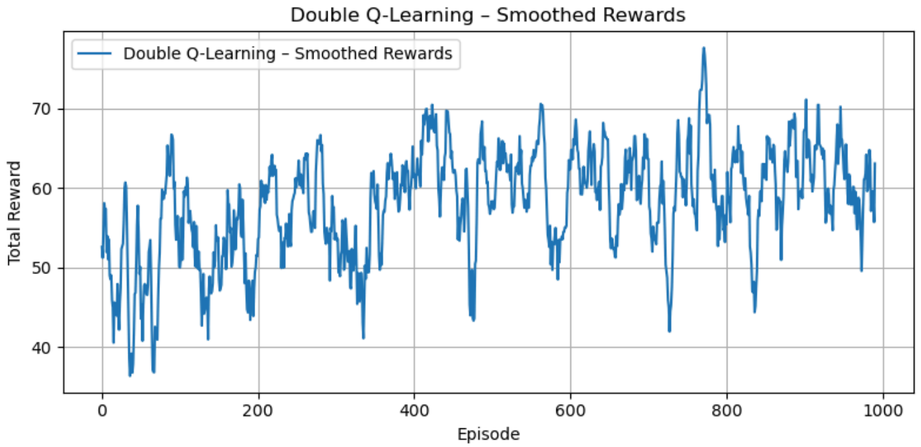
Q-Learning

Showing consistent improvement over time, achieving the highest final reward among the three methods (75.21). Its learning curve showed moderate early-stage variance, but the integration of epsilon and alpha decay helped stabilize policy updates in later episodes.



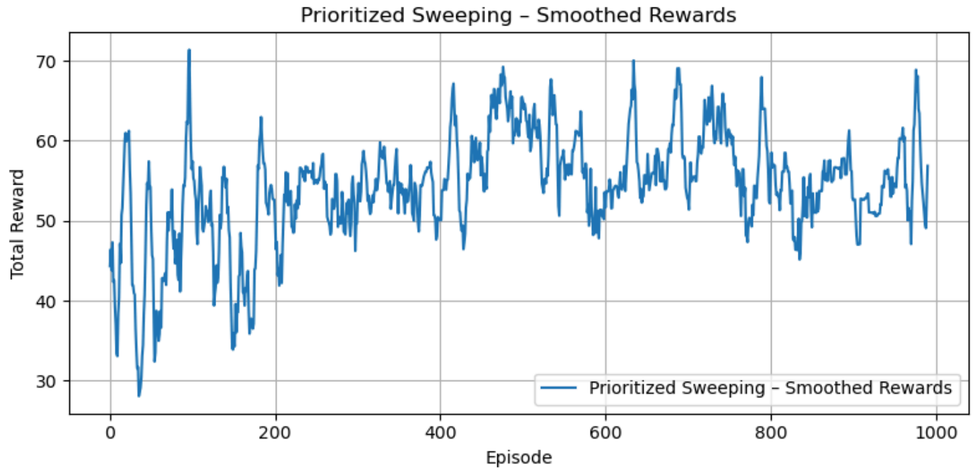
Double Q-Learning

achieved the highest average reward (60.56), indicating strong early performance and rapid initial convergence. However, its final reward dropped significantly to 15.56, suggesting a possible collapse in policy stability. This could be due to the interplay between decaying learning rates and reduced exploration, which may have inhibited sufficient policy refinement in later episodes.



Prioritized Sweeping

It offered a strong balance between performance and stability. While it did not achieve the highest peak reward, it maintained consistent learning with a relatively low standard deviation (13.89). The agent demonstrated efficient planning and early convergence, achieving a final reward of 50.95 and sustaining long survival durations per episode



Comparison of Tabular Methods

Q-learning achieved the highest final reward, indicating strong long-term learning, but had high variability during training. Double Q-learning had the highest average reward and fast early learning but suffered a sharp drop in final performance, suggesting stability issues. Prioritized Sweeping was the most stable overall, with steady performance and low variability, though its final reward was lower than Q-learning’s.

Method	Avg Reward	Std Dev	Final Reward
Q-learning	59.52	26.64	74.24
Double Q-learning	61.69	18.42	86.67
Prioritized Sweeping	54.68	13.45	53.25

Function Approximation

Tile Coding

Defines how continuous states are transformed into discrete features.
Uses multiple overlapping tilings to improve generalization.

```python
class TileCoder:
    def __init__(self, low, high, bins, tilings):
        self.low = np.array(low)
        self.high = np.array(high)
        self.bins = np.array(bins)
        self.tilings = tilings
        self.dimensions = len(bins)
        self.offsets = [
            np.linspace(0, 1, tilings + 1)[1:] * (1 / (b * tilings)) for b in bins
        ]
        self.total_tiles = np.prod(bins) * tilings
```

Why Tile Coding?

- Computational Efficiency: Tile coding is less computationally intensive compared to Fourier basis functions, making it suitable for environments with limited resources.
- Simplicity: It is straightforward to implement and tune, especially in environments with a moderate number of state variables.
- Effective Generalization: By discretizing continuous state spaces into overlapping tiles, it allows for effective generalization across similar states.

Expected SARSA + Tile Coding

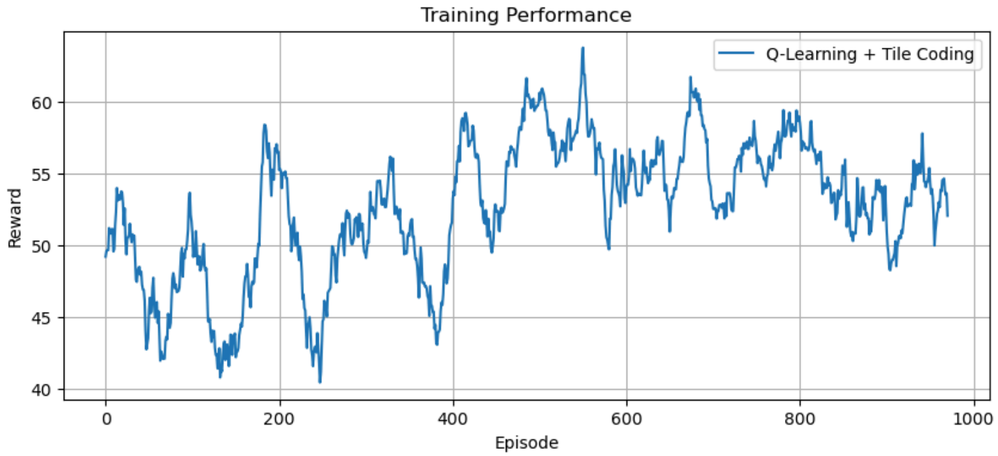
Expected SARSA (State-Action-Reward-State-Action) is a temporal-difference learning method that updates the action-value function by computing the expected value over all possible actions under the current policy, instead of relying on a sampled next action (like in SARSA) or max value (like in Q-learning).

How we used it

- **Function Approximation with Tile Coding**
  - highway-env has a continuous observation space.
  - We used tile coding to convert each state-action pair into a sparse binary feature vector.
  - Features were generated by overlapping tilings, allowing generalization across similar states.
- **Semi-Gradient Updates:**
  - Maintained a weight vector w, one per feature.
  - Updated only active features using the semi-gradient of the Expected SARSA TD error.
- **Policy and Action Selection**
  - Used an ε-greedy policy based on current Q-values.
- **Hyperparameter Tuning**
  - Learning rate (α), discount factor (γ), and exploration rate (ε) were tuned experimentally.
  - Best performance achieved with α = 0.05, γ = 0.99, and ε decayed gradually from 1.0 to 0.05.
  - Number of tilings: 8–16, balancing generalization and sensitivity.

Q-Learning + Tile Coding

An off-policy temporal difference learning method that updates the action-value function based on the maximum reward achievable from the next state, regardless of the agent's current policy. This method is useful for learning optimal policies independently of the agent's actions



Thank you!