Autonomous Driving Optimization using Reinforcement Learning

Final Report

Team Members:

- Shyam Akhil Nekkanti - 8982123
- Layanika V.S - 8934459
- Samyukth Lalith Lella Gopal - 9005574

GITHUB Repository Link - https://github.com/NoSkhil/highway-env-dqn

1. Introduction

Project Overview

This project implements reinforcement learning to develop autonomous driving agents that optimize vehicle routing and lane management while adhering to safety thresholds. We used Highway-Env's 2D simulation environment to train decision-making agents that can adapt to diverse traffic patterns and make intelligent decisions in complex driving scenarios.

Problem Statement

Development of autonomous driving systems faces two critical challenges:

1. Limited adaptability to diverse traffic patterns
2. Poor decision-making in complex driving scenarios

Traditional rule-based systems cannot adapt to unexpected driving conditions, while supervised learning approaches require extensive labeled data and struggle with edge cases. Our project addresses these limitations by developing RL agents that learn optimal driving policies through interaction with Highway-Env's simulated driving scenarios.

Success Metrics

We evaluate our RL agents based on:

1. Safety performance (collision avoidance rate)

2. Efficiency metrics (travel time, speed maintenance, lane discipline)
3. Scenario completion rate at different traffic densities
4. Training convergence speed
5. Generalization across different Highway-Env scenarios

2. Environment and Setup

Highway-Env Environment

We utilized the Highway-Env simulator, which provides a flexible framework for developing and testing autonomous driving algorithms. The environment features:

- **State Space**: Vehicle positions, speeds, lanes, and surrounding vehicle information
- **Action Space**: Lane changes (left/right), speed adjustments (accelerate/decelerate), and idle
- **Reward Mechanism**: Customizable rewards for safety, efficiency, and appropriate driving behavior

Configuration Parameters

We enhanced the default environment with optimized parameters:

```
env = gym.make(env_name,
    render_mode=render_mode,
    config={
        "observation": {
            "type": "Kinematics",
            "vehicles_count": 10,
            "features": ["presence", "x", "y", "vx", "vy", "heading"],
            "absolute": False
        },
        "action": {
            "type": "DiscreteMetaAction",
        },
        "lanes_count": 4,
        "initial_lane_id": 1,
        "duration": 100,
        "vehicles_density": 1.8,
        "collision_reward": -15,
        "reward_speed_range": [22, 32],
        "simulation_frequency": 15,
        "policy_frequency": 5
    })
```

3. Algorithms and Implementation

Reinforcement Learning Algorithms

We implemented and compared three reinforcement learning algorithms:

1. DQN (Deep Q-Network)

- Deep learning model for Q-value approximation
- Experience replay and target networks
- Suitable for Highway-Env's discrete action spaces

Neural Network Architecture

We developed an enhanced neural network architecture for our DQN implementation:

```
class DQN(nn.Module):
    def __init__(self, observation_space, action_space):
        super(DQN, self).__init__()
        # Calculate input shape based on observation space
        self.input_shape = int(np.prod(observation_space.shape))

        # Enhanced network architecture with more neurons
        self.fc1 = nn.Linear(self.input_shape, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, action_space.n)

    def forward(self, x):
        batch_size = x.size(0)
        x = x.view(batch_size, -1)  # Reshape to (batch_size, flattened_features)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        return self.fc4(x)
```

Custom Reward Function

We designed a sophisticated reward function that balances safety and efficiency using the formula:

$$R(s_t, a_t, s_{t+1}) = w_1(-\alpha \cdot \max(0, (d_{threshold} - d_{collision})^2)/d_{threshold}^2) \\ + w_2(\beta_1 \cdot v_t + \beta_2 \cdot I_{fast\_lane} - \beta_3 \cdot I_{lane\_change})$$

Where:

- w1, w2: Weights for safety vs. efficiency (0.65, 0.35)
- α: Safety scaling factor (8.0)
- dthreshold: Safety distance threshold (0.8)
- β1, β2, β3: Efficiency parameters for velocity, lane preference, and lane changes
- Ifast_lane: Indicator for being in fast lane
- Ilane_change: Indicator for changing lanes

The implementation of this reward function is:

```python
def custom_reward(obs, reward, done, truncated, info):
    # Extract relevant information from observation
    ego_vehicle = obs[0]  # Ego vehicle features
    other_vehicles = obs[1:]  # Other vehicles features

    # Parameters for reward function
    w1 = 0.65  # Weight for safety
    w2 = 0.35  # Weight for efficiency
    alpha = 8.0  # Safety scaling
    beta1 = 0.06  # Velocity scaling
    beta2 = 0.2  # Fast lane bonus
    beta3 = 0.1  # Lane change penalty
    d_threshold = 0.8  # Safety distance threshold

    # Calculate current velocity (in normalized observation space)
    v_t = np.sqrt(ego_vehicle[3]**2 + ego_vehicle[4]**2)  # vx, vy components

    # Calculate distance to nearest vehicle
    min_distance = float('inf')
    for veh in other_vehicles:
        if veh[0] > 0:  # Vehicle is present
            # Calculate Euclidean distance in x,y plane
            dist = np.sqrt((veh[1])**2 + (veh[2])**2)
            min_distance = min(min_distance, dist)

    # If no vehicles detected, set to a large safe value
    if min_distance == float('inf'):
        min_distance = d_threshold + 1.0
```

```
# Safety component: penalize proximity only when closer than threshold
safety_penalty = -alpha * max(0, (d_threshold - min_distance)**2) / (d_threshold**2)

# Determine if in fast lane (rightmost lane in this setup)
in_fast_lane = ego_vehicle[2] > 0.3  # Positive y means rightmost lane

# Determine if changing lanes (based on heading)
changing_lane = abs(ego_vehicle[5]) > 0.1  # Heading indicates lane change

# Efficiency component
efficiency_reward = (beta1 * v_t) + (beta2 * in_fast_lane) - (beta3 * changing_lane)

# Combined reward
combined_reward = w1 * safety_penalty + w2 * efficiency_reward

# Add base environment reward with smaller weight
final_reward = combined_reward + 0.1 * reward

return final_reward
```

Parameters for reward function:

w1 = 0.65 # Weight for safety

w2 = 0.35 # Weight for efficiency

alpha = 8.0 # Safety scaling

beta1 = 0.06 # Velocity scaling

beta2 = 0.2 # Fast lane bonus

beta3 = 0.1 # Lane change penalty

d_threshold = 0.8 # Safety distance threshold

4. Training Process

DQN Agent Implementation

Our DQN agent incorporated several advanced features:

- **Experience Replay Buffer**: Stored transitions for more efficient learning
- **Target Network**: Stabilized training by periodically updating Q-value targets
- **Epsilon-Greedy Exploration**: Balanced exploration and exploitation during training
- **Gradient Clipping**: Improved training stability by limiting gradient magnitudes

Training Parameters

The agent was trained with the following hyperparameters:

- Learning Rate: 0.0005
- Discount Factor (γ): 0.99
- Replay Buffer Size: 100,000
- Batch Size: 128
- Target Network Update Frequency: 1,000 steps
- Epsilon Start: 1.0
- Epsilon Final: 0.01
- Epsilon Decay: 50,000 steps

Training Progression

We trained the model for 500 episodes, with these key stages:

1. **Initial Exploration Phase (Episodes 1-100)**:
   - High exploration rate (epsilon > 0.96)
   - Highly variable rewards as the agent learned the basics of driving
   - Episode rewards between -75 and -15
2. **Intermediate Learning Phase (Episodes 100-300)**:
   - More consistent performance as exploration decreased
   - Growing stability in reward trends

    - Episode rewards stabilizing between -60 and -20

3. **Fine-tuning Phase (Episodes 300-500)**:
    - Lower exploration rate (epsilon < 0.89)
    - More consistent behavior with occasional performance spikes
    - Episode rewards typically between -50 and -20, with some episodes approaching -15

5. Results and Evaluation

Training Metrics

The training process yielded the following results:

1. **Episode Rewards**: Starting from highly negative values, the rewards gradually improved and stabilized as training progressed, indicating the agent was learning effective driving policies.
2. **Training Loss**: The loss curve showed a downward trend with decreasing variance, suggesting that the neural network was successfully converging to a stable policy.
3. **Evaluation Rewards**: Periodic evaluations showed improvement in the agent's performance, with evaluation rewards stabilizing around -50 to -30 in the final stages of training.

Action Distribution Analysis

Analysis of the trained agent's behavior revealed consistent patterns:

- Strong preference for Action 3 (FASTER) in many scenarios, indicating the agent learned that maintaining higher speed is optimal for reward maximization
- Appropriate use of lane-changing actions when necessary for collision avoidance
- Minimal use of SLOWER action except in critical safety situations

Safety Performance

The agent demonstrated proficient collision avoidance capabilities:

- Successfully maintained safe distances from other vehicles in most scenarios
- Appropriate lane changes to avoid potential collisions

- Effective speed management in high-density traffic situations

Efficiency Metrics

The agent showed good performance on efficiency metrics:

- Maintained target velocity in safe conditions
- Utilized faster lanes when available
- Minimized unnecessary lane changes

6. Challenges and Solutions

Implementation Challenges

1. **Environment Configuration**:
   - **Challenge**: Initially encountered issues with the highway-env simulation due to incorrect package installation
   - **Solution**: Properly installed all dependencies and configured environment parameters
2. **Training Stability**:
   - **Challenge**: Inconsistent learning progress in early training iterations
   - **Solution**: Adjusted hyperparameters, particularly learning rate and exploration parameters, to improve stability
   - **Solution**: Implemented gradient clipping to prevent large weight updates
3. **Reward Function Design**:
   - **Challenge**: Balancing safety and efficiency in the reward function
   - **Solution**: Fine-tuned reward weights through experimentation, giving safety a higher priority (0.65 vs 0.35)
   - **Solution**: Added distance-based safety scaling to better represent collision risk

Future Improvements

1. **Algorithm Comparison**:
   - Implement and compare PPO and A2C algorithms against DQN performance
   - Evaluate each algorithm's suitability for different driving scenarios
2. **Environment Expansion**:
   - Test on more complex scenarios such as merge-v0 and intersection-v0
   - Develop custom scenarios with varying traffic densities
3. **Reward Function Optimization**:
   - Implement automated hyperparameter optimization for reward function tuning
   - Explore adding components for lane discipline and smooth driving
4. **Model Architecture Enhancements**:
   - Test different neural network architectures, including convolutional networks for image-based observations
   - Implement dueling networks and prioritized experience replay

7. Conclusion

This project successfully developed and implemented a DQN-based autonomous driving agent capable of navigating through highway scenarios while balancing safety and efficiency concerns. Through careful environment design, reward function engineering, and neural network implementation, we created an agent that demonstrates effective decision-making in diverse traffic conditions.

The trained model shows promising results in collision avoidance, speed management, and lane utilization. While there is room for improvement, particularly in complex scenarios and with alternative algorithms, the current implementation provides a solid foundation for autonomous driving using reinforcement learning approaches.

Our future work will focus on expanding the environment complexity, comparing different reinforcement learning algorithms, and further optimizing the reward function to create even more robust autonomous driving agents.

8. References

1. Highway-Env documentation - https://highway-env.readthedocs.io/
2. Stable Baselines3 - https://stable-baselines3.readthedocs.io/
3. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.
4. Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
5. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.