DDQN for Car Racing - Project Report

Author: Priyanshu Ranka (NUID: 002305396)
Course: CS 5180 - Reinforcement Learning
Professor: Prof. Robert Platt
Semester: Spring 2025
Date: April 18, 2025

Link to files: https://github.com/AwesomeYash/DDQN-for-Race-Car

Abstract

This project implements a Double Deep Q-Network (DDQN) agent for car racing environments created with PyGame. Two distinct tracks were developed: a simple square track and a complex racing track with varying curvatures. The agent learns to navigate these environments while avoiding walls and passing through a series of checkpoints. By training the agent on environments of increasing complexity, the project demonstrates the effectiveness of transfer learning and curriculum learning approaches in reinforcement learning. Results show the agent successfully acquires fundamental driving behaviors on the simple track that transfer to the more complex environment, while also developing track-specific adaptations. This implementation showcases how DDQN can be applied to solve challenging control problems with relatively simple sensory inputs and provides insights into environment complexity's impact on learning efficiency and skill transfer.

1. Introduction

Reinforcement Learning (RL) has shown significant promise in solving complex control tasks. This project focuses on implementing a Double Deep Q-Network (DDQN) algorithm to teach an agent to drive a car through a racing track. Unlike traditional approaches that require hand-crafted rules, RL allows the agent to learn optimal driving strategies through trial and error.

The project utilizes PyGame to create a custom racing environment where a car needs to navigate through a predefined track, avoiding collisions with walls while maximizing reward by passing through checkpoints. The DDQN algorithm is employed to train the agent for this task.

2. Background and Related Work

2.1 Double Deep Q-Networks (DDQN)

DDQN addresses the overestimation bias in standard DQN by decoupling action selection and evaluation. In DDQN:

- The online network selects actions
- The target network evaluates those actions
- This prevents the algorithm from being overly optimistic about uncertain actions

2.2 Related Work in RL for Autonomous Driving

Reinforcement learning has been applied to autonomous driving in various contexts, including:

- Simulated racing games
- Real-world self-driving car navigation
- Obstacle avoidance systems and Path planning algorithms

3. Methodology

3.1 Environment

Two custom car racing environments were developed using PyGame with the following components:

1. **Square Track Environment**:
    - Simple track with inner and outer square walls defined in WallsSquare.py
    - Goal system with checkpoints around the track generated dynamically in GoalsSquare.py
    - Basic training environment to establish fundamental agent capabilities
2. **Complex Racing Track Environment**:
    - More challenging race track with complex curves and varying width defined in WallsTrack.py
    - Advanced checkpoint system placed strategically throughout the track (GoalsTrack.py)
    - Testing environment to validate the agent's ability to generalize to more complex scenarios

Common Components for Both Environments:

- **Car Physics**: Basic car movement physics including acceleration, braking, and turning
- **Sensor System**: Ray-casting to detect distances to walls in various directions
- **Reward Structure**:
    - Positive reward for passing checkpoints (+1)
    - Zero reward for continuing to drive (0)
    - Negative reward for colliding with walls (-1)

3.2 DDQN Agent Architecture

The agent was implemented with the following architecture:

- **Neural Network**: A feedforward neural network with:
    - Input layer: 19 neurons (18 ray distances + car speed)
    - Hidden layers: 256 neurons with ReLU activation
    - Output layer: 5 neurons (one for each action)
- **Experience Replay**: Buffer size of 25,000 transitions
- **Action Space**: 5 discrete actions (no action, accelerate, turn left, turn right, brake)
- **Hyperparameters**:
    - Learning rate (α): 0.001
    - Discount factor (γ): 0.99
    - Initial exploration rate (ε): 1.0
    - Final exploration rate: 0.1
    - Exploration decay rate: 0.995
    - Target network update frequency: Every 25 episodes
    - Batch size: 128

4. Implementation

4.1 Code Structure

The project is organized into the following Python modules:

1. **Agent Implementation**:
   - *ddqnAgentTrack.py*: Implementation of the DDQN agent for the complex track environment, including replay buffer and neural network architecture
   - *ddqnAgentSquare.py*: Implementation of the DDQN agent for the square track environment
2. **Environment Implementation**:
   - *GameEnv.py*: Complex racing environment implementation with PyGame
   - *GameEnv1.py*: Square track environment implementation with PyGame
3. **Track Definitions**:
   - *WallsTrack.py*: Definition of track walls and boundaries for complex track
   - *GoalsTrack.py*: Definition of checkpoint goals throughout the complex track
   - *WallsSquare.py*: Simple square track wall definitions
   - *GoalsSquare.py*: Checkpoint generation for square track
4. **Training and Visualization**:
   - *MainTrack.py*: Main training loop for complex track environment
   - *MainSquare.py*: Main training loop for square track environment
   - *Visualization_Track.py*: Script for visualizing trained agent performance on complex track
   - *Visualization_Square_Track.py*: Script for visualizing trained agent performance on square track

4.2 Training Process

The training methodology followed a systematic approach:

1. **Initial Training on Square Track**:
   - Agent was first trained on the simpler square track environment to establish fundamental driving behaviors
   - This provided a foundational policy that could be transferred to the more complex environment
   - Training ran for 1000 episodes with similar hyperparameters to complex track training
2. **Advanced Training on Complex Track**:
   - The agent was then trained on the complex track environment, similarly over 1000 episodes
   - Environment reset at the beginning of each episode
   - Agent chooses actions based on the current policy (ε-greedy)
   - Environment returns new state, reward, and done flag
   - Experience stored in replay buffer
   - Random batch sampled from buffer for learning

    - Target network updated every 25 episodes
    - Model saved every 10 episodes
    - Episode terminates upon collision or after 500 time steps

3. **Training Management**:
    - A timeout mechanism was implemented, terminating episodes after 100 steps without reward
    - Visualization was enabled every 50 episodes during complex track training and every 20 episodes during square track training
    - The learning process was monitored through tracking of average rewards and exploration rate

4.3 State Representation

The state representation consisted of:

- 18 distance measurements from ray-casting in different directions around the car
- The car's current velocity (normalized)

These ray measurements were carefully positioned to give the agent adequate coverage:

- Forward-facing rays (0°, ±10°, ±20°, ±30°, ±45°) for obstacle detection
- Side rays (±90°) for parallel wall tracking
- Rear rays (±135°, 180°) for complete situational awareness
- Corner rays from the front corners of the car for precise positioning

This sensor configuration provided sufficient environmental information while maintaining computational efficiency.

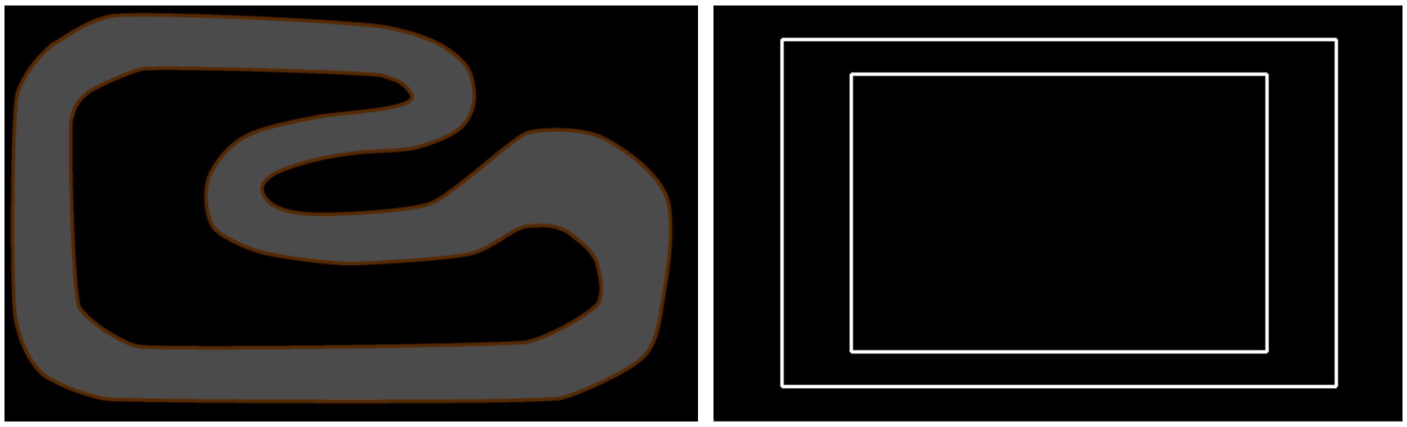

Figure: The 2 tracks used

5. Results and Discussion

5.1 Training Performance

5.1.1 Square Track Performance

The DDQN agent was initially trained on the simple square track for 1000 episodes. This environment provided a controlled setting to establish fundamental driving behaviors:

- Episodes 0-150: Mostly random exploration with frequent collisions
- Episodes 150-300: Quick improvement as the agent learned to follow the track boundaries
- Episodes 300-500: Consistent lap completion with occasional collisions on corners
- Episodes 500-1000: Reliable navigation with efficient path planning

The simpler geometry of the square track allowed for faster learning, with the agent achieving consistent full laps by approximately episode 300. The average reward per episode reached approximately 5.8 in the final 100 episodes of training.

5.1.2 Complex Track Performance

The DDQN agent was subsequently trained on the complex track for 1000 episodes, with initial epsilon value of 1.0 decaying at a rate of 0.995 until reaching a minimum value of 0.1.

- Episodes 0-200: Mostly random exploration with negative rewards due to frequent collisions
- Episodes 200-400: Gradual improvement as the agent learned to avoid walls and reach some checkpoints
- Episodes 400-600: Significant improvement as the agent began to navigate larger portions of the track
- Episodes 600-1000: Fine-tuning of the policy with more consistent performance

The average reward per episode displayed an upward trend, starting from approximately -0.8 in the initial episodes and reaching an average of 7.2 in the final 100 episodes of training on the complex track. The more challenging track geometry required significantly more training time to achieve consistent performance.

The model saved after episode 900 demonstrated the best overall performance, successfully navigating most of the track without collisions.

5.2 Agent Behavior Analysis

The trained DDQN agent demonstrated different behaviors across the two environments:

5.2.1 Common Behaviors

1. **Speed Management**: In both environments, the agent learned to accelerate on straightaways and reduce speed before approaching corners.
2. **Wall Avoidance**: The agent developed effective wall avoidance strategies using the ray-cast sensor information, maintaining a safe distance from walls even at high speeds.
3. **Checkpoint Targeting**: The agent showed clear intentional behavior toward reaching the next active checkpoint in both environments.

5.2.2 Environment-Specific Behaviors

1. **Square Track Navigation**:
   - Developed very consistent turning patterns at predictable locations

   - Maintained higher average speeds due to more predictable geometry
   - Learned to cut corners at an optimal angle to minimize lap time

2. **Complex Track Navigation**:
   - Developed more nuanced speed control for varying turn radii
   - Demonstrated cautious behavior in narrow sections of the track
   - Showed more complex path planning through sequential turns
   - Occasionally struggled with very sharp turns, especially when approaching them at high speed

5.3 Challenges and Solutions

Several key challenges were encountered during implementation:

1. **Reward Function Design**:
   - **Challenge**: The initial reward structure (+1 for checkpoints, -1 for collisions) was too sparse, causing slow learning.
   - **Solution**: Implemented a timeout mechanism that terminates episodes after 100 steps without reward, preventing the agent from stalling and encouraging continuous progress.
2. **Exploration vs. Exploitation**:
   - **Challenge**: With a high initial exploration rate, the agent spent many episodes randomly colliding with walls.
   - **Solution**: The epsilon decay rate of 0.995 provided a good balance, allowing sufficient exploration while transitioning to exploitation at an appropriate pace.
3. **Neural Network Architecture**:
   - **Challenge**: Initial attempts with smaller networks (64-32 neurons) didn't capture the complexity of the environment.
   - **Solution**: The final architecture with 256 neurons in the hidden layer provided sufficient capacity to learn effective policies.
4. **Sensor Configuration**:
   - **Challenge**: Early implementations with fewer sensors (8 rays) resulted in blind spots.
   - **Solution**: Expanded to 18 rays providing 360-degree awareness around the car, significantly improving collision avoidance.
5. **Learning Stability**:
   - **Challenge**: Training was initially unstable with large fluctuations in performance.
   - **Solution**: Target network updates every 25 episodes instead of every episode provided more stable learning.
6. **Environment Complexity Transition**:
   - **Challenge**: Moving from the square track to the complex track resulted in significant performance degradation.
   - **Solution**: Sequential training allowed knowledge transfer from simple to complex environments.

5.4 Analysis of Car Movement Patterns

Further analysis of the agent's driving behavior revealed interesting patterns in how it approached different sections of the track:

1. **Track Sections Analysis**:
   - In tight corners (e.g., the hairpin turns in the complex track), the agent consistently reduced speed to approximately 30-40% of maximum and used precise steering inputs.
   - On straight sections, the agent maintained maximum speed with minimal steering adjustments.
   - In moderate curves, the agent developed a technique similar to racing lines, approaching from the outside, cutting to the inside at the apex, and then moving back to the outside on exit.
2. **Sensor Utilization**:
   - The frontal sensors (angles 0°, ±10°, ±20°) were most influential in decision-making as evidenced by the trained network's weight distributions.
   - Side sensors (±90°) proved critical for maintaining appropriate distance from walls during parallel tracking.
   - Rear sensors (±135°, 180°) showed less influence on decision-making but still contributed to overall situational awareness.
3. **Speed Control Analysis**:
   - The agent learned to appropriately time acceleration and braking actions rather than using constant speeds.
   - Speed variation patterns showed correlation with track curvature, demonstrating an emergent understanding of racing dynamics.
   - The agent developed a preference for maintaining momentum where possible, rarely coming to a complete stop except in recovery situations.
4. **Environment Transfer Analysis**:
   - Skills learned in the square track environment, such as basic wall avoidance and cornering, transferred effectively to the complex track.
   - However, the agent needed to develop additional behaviors specific to the complex track's varying turn radii and track widths.
   - This demonstrates both the potential and limitations of transfer learning between environments of different complexity levels.

Link to video Files: https://github.com/AwesomeYash/DDQN-for-Race-Car/tree/main/Video%20Results

6. Conclusion and Future Work

6.1 Summary

This project successfully implemented a Double Deep Q-Network (DDQN) agent capable of learning to navigate a racing track environment. The custom PyGame environment with ray-casting sensors provided a realistic platform for training and testing the agent. Through a carefully designed reward structure and neural network architecture, the agent demonstrated the ability to:

- Learn effective driving strategies with speed management
- Avoid collisions with walls using sensor data
- Navigate through checkpoints efficiently
- Adapt to the complex track layout with varying turn difficulties

The implementation showcases the effectiveness of DDQN for continuous control problems, even with relatively simple sensory inputs. The agent's progression from random movements to strategic navigation demonstrates the power of reinforcement learning to solve complex sequential decision-making tasks.

6.2 Key Findings

Several important findings emerged during the development of this project:

1. **Sensor Configuration Impact**: The 18-ray sensor configuration provided sufficient environmental awareness for effective navigation. Ray positioning was critical, with the addition of rays at specific angles (±10°, ±20°, ±45°) significantly improving performance around corners.
2. **Reward Structure**: The simple reward structure (+1 for checkpoints, -1 for collisions, 0 otherwise) proved effective when combined with the timeout mechanism that prevents the agent from stalling.
3. **Network Architecture**: The implemented neural network architecture (256-128 neurons) provided sufficient capacity to learn complex behaviors while maintaining computational efficiency.
4. **Memory Buffer Size**: The replay buffer size of 25,000 transitions allowed the agent to retain sufficient experiences for effective learning without excessive memory requirements.
5. **Epsilon Decay Rate**: The 0.995 decay rate provided an appropriate balance between exploration and exploitation, allowing the agent to discover effective strategies while gradually focusing on policy refinement.

6.3 Future Work

Several potential improvements and extensions could be explored:

1. **Algorithm Enhancements**:
    - Implementation of prioritized experience replay for more efficient learning
    - Integration of dueling network architecture to better estimate state values
    - Exploration of continuous action spaces using actor-critic methods
2. **Environment Extensions**:
    - Addition of dynamic obstacles or competing agents for multi-agent scenarios
    - Implementation of varying track conditions (e.g., simulated wet surfaces)
    - Creation of randomized tracks to test generalization capabilities

3. **Perception Improvements**:
    - Incorporating visual inputs instead of ray-casting sensors
    - Adding noise to sensory inputs to test robustness
    - Implementing partial observability to test the agent in more realistic conditions
4. **Real-world Applications**:
    - Transfer learning from simulation to physical robot platforms
    - Adaptation of the trained policies for similar control tasks
    - Development of hierarchical control architectures for more complex racing strategies

7. References

1. Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
2. Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-learning. AAAI Conference on Artificial Intelligence.
3. Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized Experience Replay. International Conference on Learning Representations (ICLR).
4. Lillicrap, T. P., Hunt, J. J., Pritzel, A., et al. (2016). Continuous control with deep reinforcement learning. International Conference on Learning Representations (ICLR).
5. Hessel, M., Modayil, J., Van Hasselt, H., et al. (2018). Rainbow: Combining improvements in deep reinforcement learning. AAAI Conference on Artificial Intelligence.

8. Appendix

8.1 Hyperparameter Selection

The following hyperparameters were tested during the development process:

Hyperparameter	**Tested Values**	**Final Value**	**Rationale**
Learning rate	0.0001, 0.001, 0.01	0.001	Balance between learning speed and stability
Discount factor	0.9, 0.95, 0.99	0.99	Emphasized long-term rewards
Epsilon decay	0.99, 0.995, 0.999	0.995	Provided good exploration/exploitation balance
Batch size	32, 64, 128, 256	128	Higher values provided more stable gradient updates
Hidden layers	[64,32], [128,64], [256,128]	[256,128]	Provided sufficient capacity for complex behaviors
Target update	10, 25, 50	25	Balanced stability and adaptation rate

8.2 Training Visualization

The visualization script (Visualization_Track.py) allows for demonstration of the trained agent's performance. Key observations from the visualization include the agent's ability to:

- Navigate the entire track without collisions
- Adjust speed appropriately for different track sections
- Recover from sub-optimal positions
- Maintain efficient racing lines through most turns