Autonomous Driving Car Using DDQN

Tanay Nagarkar

May 10, 2025

1 Introduction

In this project, I developed an autonomous racing agent using Double Deep Q-Networks (DDQN). I designed a simulation that integrates a reinforcement learning loop with a physics-based environment, ray-based perception, and reward shaping. Below, I provide a detailed breakdown of each file involved in the system.

2 File-by-File Breakdown

2.1 main.py – Training Script

Purpose
This is where I implemented the main training loop for the DDQN agent. It controls environment interaction, experience storage, model training, and saving.

Key Components

- **Initialization:**
  I began by setting core training parameters and initializing necessary components:

```
TOTAL_GAMETIME = 1000  # Max steps per episode
N_EPISODES = 10000     # Total training episodes
REPLACE_TARGET = 50    # Target network update frequency
```

  These constants control how long each episode lasts, how many episodes to run, and how frequently the target network is updated.

- **DDQN Agent Setup:**
  I then created the agent with the following parameters:

```
ddqn_agent = DDQNAgent(
    alpha=0.0005,       # Learning rate
    gamma=0.99,         # Discount factor
    n_actions=5,        # Discrete actions
    epsilon=1.0,        # Initial exploration rate
    epsilon_end=0.1,    # Minimum exploration
    epsilon_dec=0.9995, # Decay per step
    replace_target=REPLACE_TARGET,
```

```
    batch_size=512,
    input_dims=19,      # 18 rays + velocity
    fname='ddqn_model.h5'
)
```

This configuration allowed the agent to balance exploration and exploitation over time.

- **Training Loop:**
  For each episode, I reset the environment, selected actions using `ddqn_agent.choose_action()`, stored transitions in memory, and updated the model using:

```
ddqn_agent.learn()
```

- **Model Saving:**
  I saved the model every 10 episodes to ensure progress was not lost:

```
if e % 10 == 0 and e > 10:
    ddqn_agent.save_model()
```

2.2 main_test_model.py – Evaluation Script

Purpose
Here, I evaluated the agent's performance using a trained model and a near-zero exploration rate ($\varepsilon = 0.02$).

Key Differences from Training:

- **Model Loading:**
  I loaded the pre-trained model and synced the target network:

```
ddqn_agent.load_model()
ddqn_agent.update_network_parameters()
```

- **Sound Integration:**
  I added engine sound for realism:

```
engine_sound = pygame.mixer.Sound(SOUND_PATH)
engine_sound.play(loops=-1)
```

- **Rendering:**
  I enabled full-time rendering to visualize the agent's decisions.

2.3 GameEnv.py – Racing Environment

Purpose
This file defines the simulation environment, including car dynamics, perception, and reward logic.

Key Elements:

- **Car Class:**

- I implemented acceleration, deceleration, and turning using realistic constraints (e.g., `maxvel = 15`).
- I used 18 ray sensors to detect nearby obstacles, simulating LIDAR.
- I wrote the collision detection using line-rectangle intersection math.
- I gave rewards based on whether the car passed goals or crashed.

- **RacingEnv Class:**
  - I defined 5 discrete actions (accelerate, brake, turn left/right, or no-op).
  - I returned the state as a vector of 18 normalized ray distances + velocity.
  - The reward logic was handled as follows:

```
if car.score(goal): reward += GOALREWARD
if car.collision(wall): reward += PENALTY
else: reward += LIFE_REWARD
```

2.4 ddqn_keras.py – DDQN Implementation

Purpose
In this file, I implemented the agent logic including neural networks, replay buffer, action selection, and training updates.

Key Components:

- **ReplayBuffer Class:**
  I used this class to store experiences and sample random batches during learning:

```
def sample_buffer(self, batch_size):
    batch = np.random.choice(max_mem, batch_size)
    return states, actions, rewards, new_states, terminal
```

- **DDQNAgent Class:**
  - **Action Selection:**
    I used an epsilon-greedy policy:

```
if rand < self.epsilon:
    action = random.choice(action_space)
else:
    action = np.argmax(self.brain_eval.predict(state))
```

  - **Learning:**
    I implemented Double Q-Learning and used Huber loss for stable updates.

- **Brain Class (Neural Network):**
  I defined a simple feedforward network:

```
model = Sequential([
    Dense(256, input_dim=NbrStates, activation='relu'),
    Dense(NbrActions, activation='linear')
])
```

I trained this using the Adam optimizer with a learning rate of 0.0005.

2.5 Goals.py & Walls.py – Track Definition

Purpose
These files define the racetrack layout using wall and checkpoint segments.

Key Snippets:

- **Walls.py:**
  I added static wall objects:

```
wall1 = Wall(12, 451, 15, 130)
```

- **Goals.py:**
  I ensured only one goal was active at a time:

```
goals[len(goals)-1].isactiv = True
```

2.6 Exploration vs. Exploitation

I used an epsilon-greedy strategy with decaying epsilon:

$$\epsilon_{t+1} = \max(\epsilon_{\min}, \epsilon_t \cdot \epsilon_{\text{decay}})$$

The agent starts with $\epsilon = 1.0$ and gradually reduces it to $\epsilon = 0.1$ to shift from exploration to exploitation.

3 Experimentations

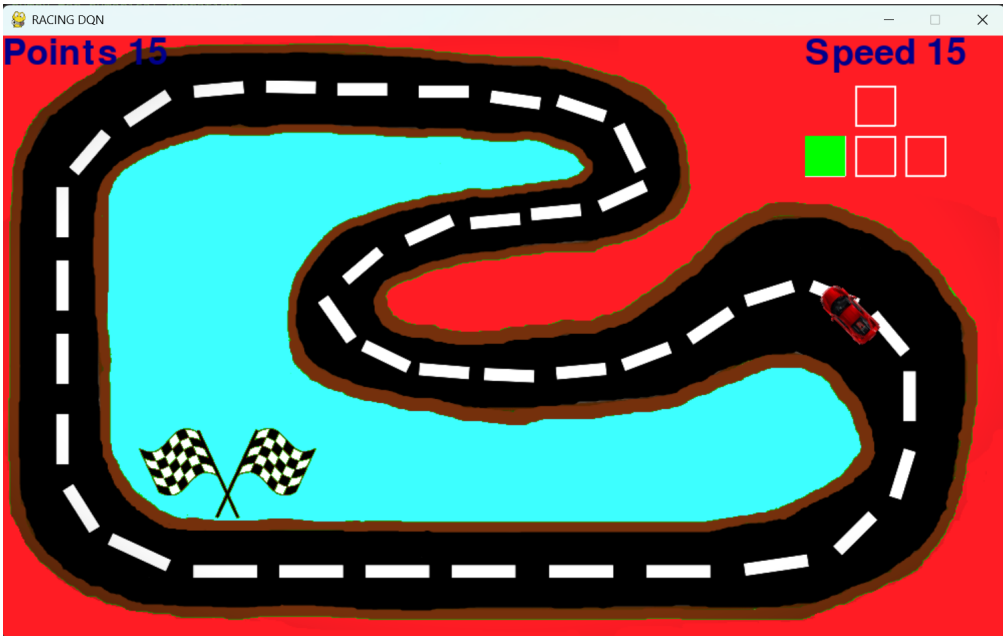
I tried Q-learning and SARSA (State–Action–Reward–State–Action) apart from DDQN. Key findings were that DDQN outperformed Q-learning and SARSA as well. Comparison was done using the scores outputted by the model, then the average of scores was taken and analyzed.

Table 1: Comparison of Reinforcement Learning Algorithms

Metric	Q-Learning	SARSA (LFA)	Double DQN (DDQN)
Total Episodes	34	34	34
Successful Episodes	20	26	34
Success Rate (%)	58.8%	76.5%	100.0%
Maximum Score	90.00	420.00	766.00
Minimum Score	0.00	10.00	21.00
Average Score	30.6	104.1	174.4
Estimated Score Variability	Low–Medium	Medium	High
Consistency Across Episodes	Low	Moderate	High
Failed Episodes	14	8	0
Learning Stability	Fluctuating	Steady	Very Stable
Function Approximator	Tabular	Linear Model	Deep Neural Network
Policy Type	Off-policy	On-policy	Off-policy
Target Network Used	No	No	Yes
Overestimation Bias Handling	No	Partial	Yes

4 Conclusion

The car navigates through the track smoothly without colliding into walls, while displaying the scores on the top-left of the screen. The keyboard's arrow key states are shown on the top right of the screen like this:



5 Problem Formulation of Self-Driving Car as an RL Problem

6 Problem Identification

I identified vehicle navigation in a racing environment as an ideal application for reinforcement learning. This problem involves continuous state spaces, sequential decision-making, and delayed rewards - characteristics that make it well-suited for RL approaches.

7 Mathematical Formulation as a Markov Decision Process (MDP)

Formalized racing car problem as an MDP, defined by the tuple (S, A, P, R, γ) where:

7.1 State Space (S)

I defined the state space as $S \subseteq \mathbb{R}^{19}$, where each state vector $\mathbf{s} \in S$ is represented as:

$$\mathbf{s} = [d_1, d_2, \ldots, d_{18}, v]^T$$

Where:

- $d_i \in [0, 1]$ represents the normalized distance reading from the i-th ray sensor, with $d_i = \frac{1000 - \text{actual\_distance}}{1000}$, transforming raw distance measurements to a normalized representation where 0 indicates maximum distance and 1 indicates minimum distance (proximity to obstacle)
- $v \in [0, 1]$ represents the normalized velocity of the car, given by $v = \frac{\text{current\_velocity}}{\text{maximum\_velocity}}$

7.2 Action Space (A)

A defined discrete action space $A = \{0, 1, 2, 3, 4\}$ where:

- $a = 0$: No action (coasting)
- $a = 1$: Accelerate
- $a = 2$: Turn left
- $a = 3$: Turn right
- $a = 4$: Brake

7.3 Transition Dynamics (P)

The transition function $P : S \times A \times S \to [0, 1]$ describes the probability of transitioning to state $\mathbf{s}'$ from state $\mathbf{s}$ after taking action a:

$$P(\mathbf{s}'|\mathbf{s}, a) = \Pr(\mathbf{s}_{t+1} = \mathbf{s}'|\mathbf{s}_t = \mathbf{s}, a_t = a)$$

In my implementation, the transitions follow deterministic physics-based rules incorporating:

- Car kinematics (acceleration, velocity, and angular velocity)
- Track boundary detection through ray-casting
- Checkpoint detection logic

7.4 Reward Function (R)

Designed a reward function $R : S \times A \times S \to \mathbb{R}$ to encourage desired behavior:

$$R(\mathbf{s}, a, \mathbf{s}') = \begin{cases} R_{\text{goal}} = +1, & \text{if checkpoint reached} \\ R_{\text{life}} = 0, & \text{if continuing normal operation} \\ R_{\text{penalty}} = -1, & \text{if collision detected} \end{cases}$$

This sparse reward structure creates a challenging exploration problem that requires effective RL techniques.

7.5 Discount Factor (γ)

I set $\gamma = 0.99$, balancing the trade-off between immediate and future rewards. This value places significant importance on long-term reward accumulation while maintaining computational stability.

8 Solution Approach: Double Deep Q-Network (DDQN)

To solve this MDP, I implemented a DDQN algorithm with the following formal components:

8.1 Value Function Approximation

I used two neural networks to approximate the action-value function $Q : S \times A \to \mathbb{R}$:

- $Q(\mathbf{s}, a; \boldsymbol{\theta})$: Evaluation network with parameters $\boldsymbol{\theta}$
- $Q(\mathbf{s}, a; \boldsymbol{\theta}^-)$: Target network with parameters $\boldsymbol{\theta}^-$

The neural network architecture I designed consists of:

- Input layer: 19 neurons corresponding to state dimensions

- Hidden layer: 256 neurons with ReLU activation function: $f(x) = \max(0, x)$
- Output layer: 5 neurons with softmax activation: $\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$ for $j = 1, \ldots, K$

8.2 Loss Function

I defined the loss function for gradient descent as the Mean Squared Error between target and predicted Q-values:

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - Q(\mathbf{s}_i, a_i; \boldsymbol{\theta})\right)^2$$

Where the target y_i for DDQN is calculated as:

$$y_i = r_i + \gamma \cdot Q(\mathbf{s}_{i+1}, _{a'} Q(\mathbf{s}_{i+1}, a'; \boldsymbol{\theta}); \boldsymbol{\theta}^-)$$

This decoupling of action selection and evaluation helps prevent overestimation of Q-values.

8.3 Exploration Strategy

I implemented an ϵ-greedy policy with decaying exploration:

$$\pi(\mathbf{s}) = \begin{cases} \text{random action } a \in A, & \text{with probability } \epsilon \\ _{a \in A} Q(\mathbf{s}, a; \boldsymbol{\theta}), & \text{with probability } 1 - \epsilon \end{cases}$$

The exploration parameter decays according to:

$$\epsilon_{t+1} = \max(\epsilon_{\min}, \epsilon_t \cdot d)$$

Where:

- Initial value $\epsilon_0 = 1.0$
- Minimum value $\epsilon_{\min} = 0.1$
- Decay factor $d = 0.9995$

8.4 Experience Replay

I incorporated experience replay using a buffer D of size $|D| = 25,000$ transitions, sampling minibatches of size $N = 512$ for learning:

$$D = \{(\mathbf{s}_j, a_j, r_j, \mathbf{s}'_j, \text{done}_j) | j = 1, 2, \ldots, |D|\}$$

8.5 Target Network Update

I synchronized the target network parameters periodically according to:

$$\boldsymbol{\theta}^{-} \leftarrow \boldsymbol{\theta} \text{ every } C \text{ episodes}$$

Where the update frequency $C = 50$ episodes.

9 Optimization Objective

The ultimate goal of my agent is to maximize the expected cumulative discounted reward:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

Where $\pi_{\boldsymbol{\theta}}$ is the policy derived from the Q-function, defined as:

$$\pi_{\boldsymbol{\theta}}(\mathbf{s}) =_{a \in A} Q(\mathbf{s}, a; \boldsymbol{\theta})$$

10 Experimental Configuration

I conducted training over $N_{\text{episodes}} = 10,000$ episodes with a maximum episode length of $T_{\max} = 1,000$ time steps, evaluating performance through cumulative rewards and successful track completions.