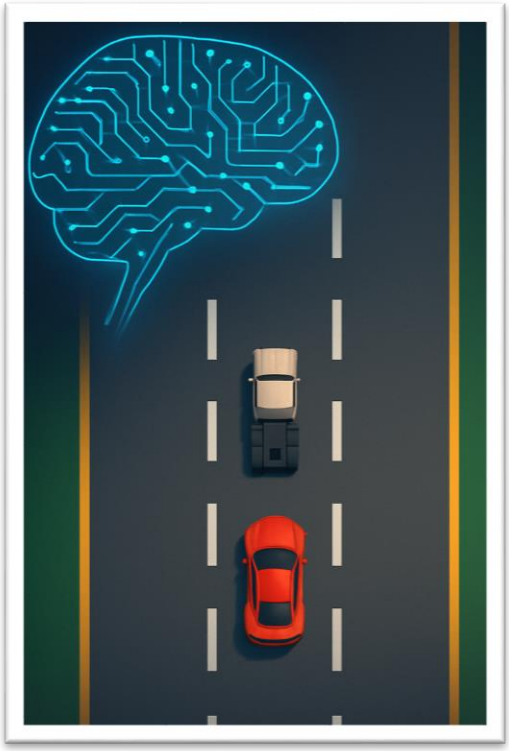

# CENG – 3511

# ARTIFICIAL INTELLIGENCE FINAL PROJECT

## AI Model to Run a Game

SÜLEYMAN EMRE PARLAK – 210709005

**Instructor :** Prof. Dr. Bekir Taner Dinçer

**Teaching Assistant :** Selahattin Aksoy

## INTRODUCTION

In this project, we developed a simple 2D driving game where the player controls a vehicle moving left or right to avoid collisions with incoming traffic and achieve the highest possible score. To enhance gameplay, we implemented both rule-based and reinforcement learning bots that can autonomously control the vehicle. The game was built using Pygame, while Reinforcement Learning model were developed and trained using TensorFlow, the code of the Rule Based method were developed using pure Python. Our goal was to explore how different AI approaches perform in a dynamic and increasingly fast-paced environment.

### SETUP INSTRUCTIONS

- Clone the reposityory. ( git clone https://github.com/paitblack/AI-approaches-on-a-car-game.git )
- Open any IDE you preferred. (Visual Studio etc.)
- Install required packages. (if any)
- To play the demo game run game.py.
- To see the reinforcement learning code, look at the DQNmodel.ipynb file.
- To see how works the rule based approach, run ruleBasedBot.py file.

### TOOLS USED

- **Python** – As the main programming language for the entire project.
- **Pygame** – For building the game interface and handling graphics and events.
- **TensorFlow** – For developing and training the reinforcement learning model.

### GOAL

The goal of this project is to develop an AI-powered self-driving car agent that can navigate a continuously scrolling road without crashing, aiming to achieve the highest possible score. The project explores both rule-based logic and reinforcement learning approaches to enable the agent to make lane-changing decisions intelligently in real time.

### PROJECT FOLDER STRUCTURE

#### game.py

This is the main game file. It initializes the game window using Pygame and handles the core mechanics: rendering the road, controlling the player's car, generating bot vehicles, and detecting collisions. The game increases in speed every 5 points to add difficulty. Player input (left/right keys) allows lane switching, and the game ends upon collision. A basic restart system is also implemented.

- **class Vehicle(pygame.sprite.Sprite):** A base class for all vehicles in the game (both player and bots). It loads and scales the vehicle image, and sets its position on the screen using a rectangle (rect), which is used for drawing and collision detection.
- **class PlayerVehicle(Vehicle):** Inherits from Vehicle. It represents the player's car specifically and loads the player car image (car.png) during initializatio

**ruleBasedBot.py**

The same with the game.py class that has extras such as rule based bot implementation, bot plays the game while running.

- **class BotVehicle(PlayerVehicle):** Inherits from PlayerVehicle. This version of the player car is AI-controlled. It checks the position of other vehicles and tries to change lanes if a collision is likely.

**environment.py**

Car racing game environment designed for RL agents to learn driving by controlling the player car, avoiding crashes, and scoring points.

- **reset(self):** Resets game state, clears vehicles and frame stack, returns initial observation for RL.
- **step(self,action):** Executes action (left, stay, right), updates game, computes reward, checks collision, returns next state, reward, done flag, and info.
- **getState(self):** Captures the screen, converts to grayscale, resizes, normalizes, stacks last 4 frames to form RL observation.
- **updateGameState(self):** Moves road markers, moves vehicles down, removes off-screen vehicles, increases speed and score.
- **spawnVehicles(self):** Adds new vehicles randomly to lanes if below max count, ensuring spacing.
- **render(self):** Draws road, lane markers, player, vehicles, score, and crash image (if game over) on the screen.
- **close(self):** Properly shuts down the Pygame environment.

**DQNmodel.ipynb**

1) **DQNAgent:**

   - CNN-based Q-network with target network for stability.
   - Uses experience replay with a memory buffer.
   - Epsilon-greedy policy for balancing exploration/exploitation.
   - Periodically updates target network weights.
   - Trains on random minibatches from memory.

2) **trainAgent():**

   - Runs episodes interacting with environment.
   - Stores experiences, trains agent, updates target network.
   - Prints scores and saves model weights periodically**.**

3) **testAgent():**

   - Loads saved model with low exploration.
   - Runs one episode to evaluate performance.
   - Renders environment with pygame.

**RULE BASED BEHAVIOR – MY APPROACH**

- **Init:** Inherits position from PlayerVehicle.
- **Update:** Each frame, checks nearby vehicles to decide lane changes.
- **Danger Zone:** Sets safety distance ahead based on speed.
- **Threat Detection:** Finds vehicles close horizontally and ahead within danger zone.
- **Lane Change:** Attempts lane switch to safer lane (right→center, center→left/right, left→center) if not blocked.
- **Lane Block Check:** Returns True if target lane has vehicles too close.
- **Action:** Changes lane by updating horizontal position if safe.

**Result :**


Our bot worked successfully and managed to get a really high score!

In my opinion, the reason it couldn't get higher scores is because the game flow speed is constantly increasing. So I managed to increase the bot score from 1600 to 2400 by increasing the danger zone value in the algorithm in direct proportion to the speed.

**REINFORCEMENT LEARNING - MY APPROACH**

- **Agent-Environment:** Agent observes state, takes action, receives reward and next state.
- **Q-Value Prediction:** Neural network estimates Q-values for all actions**.**
- **Epsilon-Greedy:** Starts fully random (epsilon=1), gradually shifts to learned policy as epsilon decays**.**
- **Experience Replay:** Samples random batches from memory to train, improving stability.
- **Target Network:** Separate target model updated periodically to stabilize learning.
- **Model Update:** Q-learning update applied to adjust predictions towards targets.
- **Save/Load:** Model weights saved and loaded for testing.
- **Testing:** Runs with low epsilon for deterministic policy evaluation.

**The training process of the model was considered to be 1000 episodes, but due to the huge increase in training time due to lack of hardware power, it was stopped after the first 20 episodes were trained. Here we see the results of the model trained with only 20 episodes.**

**Training Outputs of the first 20 episodes:**

| Episode | Score | Epsilon | Episode | Score | Epsilon |
|---|---|---|---|---|---|
| 1 | 0 | 0.15 | 11 | 2 | 0.01 |
| 2 | 0 | 0.10 | 12 | 0 | 0.01 |
| 3 | 1 | 0.01 | 13 | 0 | 0.01 |
| 4 | 3 | 0.01 | 14 | 1 | 0.01 |
| 5 | 0 | 0.01 | 15 | 0 | 0.01 |
| 6 | 2 | 0.01 | 16 | 0 | 0.01 |
| 7 | 2 | 0.01 | 17 | 1 | 0.01 |
| 8 | 2 | 0.01 | 18 | 1 | 0.01 |
| 9 | 3 | 0.01 | 19 | 0 | 0.01 |
| 10 | 1 | 0.01 | 20 | 3 | 0.01 |

**ANALYSIS**

- Since the model was trained on only 20 episodes, we did not get very good results as expected.
- High epsilon at the start (0.15–0.10) means the agent explores a lot, scores fluctuate between 0 and 1.
- As epsilon decreases to 0.01, the agent acts more consistently.
- Scores remain mostly low, with occasional higher scores (3).
- The agent is learning but hasn't stabilized on high scores yet.
- Some fluctuation suggests the environment is challenging or training is still in progress. This because, the game speed is increasing constantly and bot vehicles are spawning randomly.

**The train and the test results on terminal :**

**Train**

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Episode: 20/20, Score: 3, Epsilon: 0.01
```

**Test**

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
Test Score: 1
```

**Some other parameters for hyperparameter tuning**

```
class DQNAgent:
  def __init__(self, stateShape, actionSize):
    self.stateShape = stateShape
    self.actionSize = actionSize
    self.memory = deque(maxlen=10000)
    self.gamma = 0.99
    self.epsilon = 1.0
    self.epsilonMin = 0.01
    self.epsilonDecay = 0.995
    self.lr = 0.0005
    self.model = self._buildModel()
    self.targetModel = self._buildModel()
    self.updateTargetModel()
```