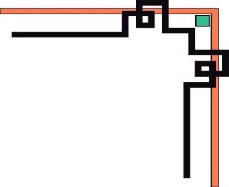
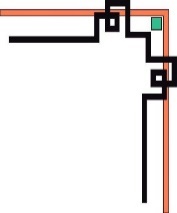
**HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION**



**FACULTY OF MECHANICAL ENGINEERING**

**'Y&&Y'**



**FINAL REPORT**

**COURSE: PRACTICE OF ARTIFICIAL INTELLIGENCE**

**TOPIC:**

**SELF-DRIVING CAR USING REINFORCEMENT LEARNING**

**LECTURER: TRẦN NHẬT QUANG**

**LHP Code: 222PAIN324129E**

**STUDENTS FROM GROUP 8:**

**NGUYỄN CÔNG QUÝ - 20134022**

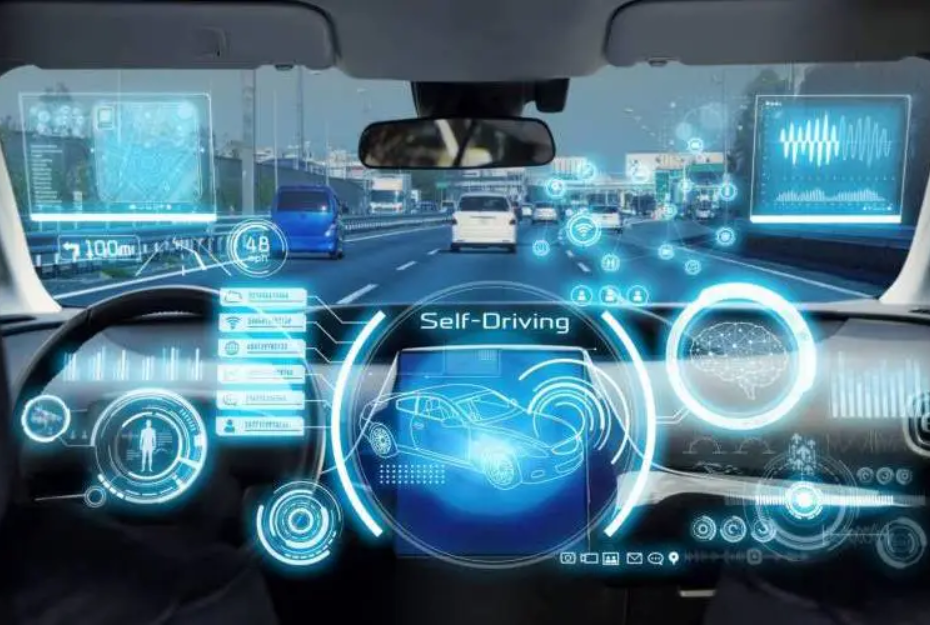
**NGUYỄN TẤN HOÀNG - 20134015**

**HỒ CHÍ MINH CITY , May 25, 2023**

# **1. Introduction**

## **1.1 Introduction to Self-Driving Cars**

Self-driving cars, also known as autonomous vehicles, are vehicles capable of navigating and operating without human intervention. These vehicles leverage advanced technologies, including artificial intelligence (AI) and sensors, to perceive their surroundings, make decisions, and control their movements. The concept of self-driving cars holds great promise for revolutionizing transportation and reshaping our daily lives.



***Figure 1****. Self-driving car illustration*

## **1.2 Importance and Potential Impact**

The emergence of self-driving cars brings forth numerous benefits and potential impacts. Firstly, self-driving cars have the potential to significantly enhance road safety. Human errors, such as distracted driving and fatigue, contribute to a significant number of accidents. By eliminating human factors, self-driving cars can reduce the occurrence of accidents and save lives.

Secondly, self-driving cars offer increased convenience and efficiency. Commuting and transportation can become more efficient as self-driving cars optimize routes, reduce traffic congestion, and allow passengers to engage in other activities during travel. This advancement has the potential to transform our daily routines and productivity.

Additionally, self-driving cars can have positive environmental implications. Through optimized driving patterns and reduced congestion, self-driving cars can contribute to lower fuel consumption and emissions. This aligns with the global goals of mitigating climate change and transitioning to sustainable transportation systems.

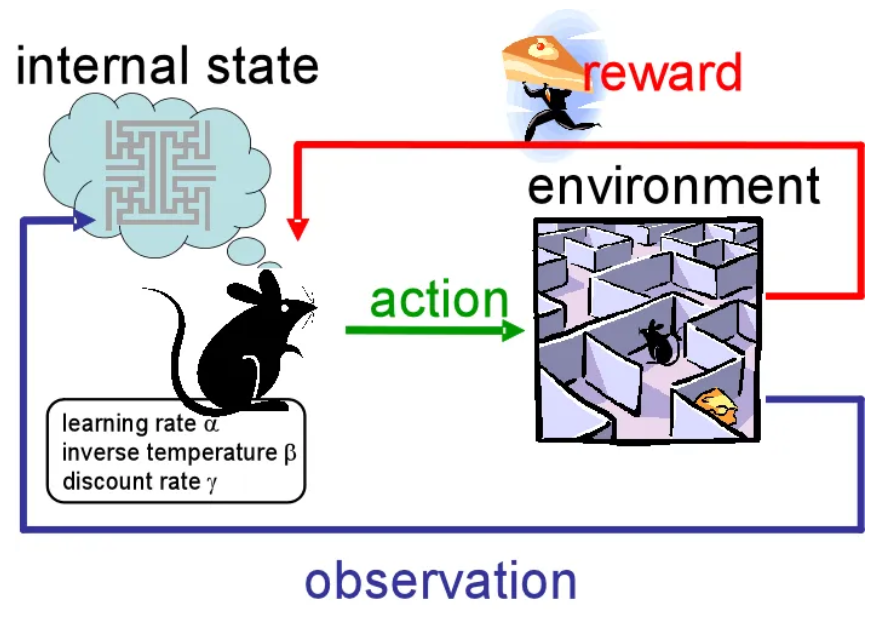
# **2. Reinforcement Learning**

## **2.1 Overview of Reinforcement Learning (RL)**

Reinforcement learning is a branch of machine learning that deals with decision-making and control in dynamic environments. In RL, an agent interacts with an environment, perceiving its state, taking actions, and receiving feedback in the form of rewards or penalties. The objective of RL is to learn an optimal policy that maximizes cumulative rewards over time.

Key components of RL include:

* Agent: The entity that learns and takes actions in the environment.
* Environment: The external context in which the agent operates, providing feedback based on its actions.
* Actions: The possible choices or behaviors available to the agent in each state.
* Rewards: Numeric signals that reflect the desirability of an agent's actions. Rewards are used to guide the learning process.

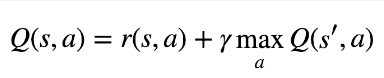


***Figure 2****. Structure of RL system*

## **2.2 What is Q-learning ?**

Q-learning is a popular model-free reinforcement learning algorithm that enables an agent to learn an optimal policy in an environment by iteratively updating an action-value function, known as a Q-function, based on the observed rewards received from the environment.

The Q-function estimates the maximum expected cumulative reward, called the Q-value, that an agent can achieve by taking a particular action in a specific state. The Q-value for a state-action pair (s, a) is defined as the sum of immediate reward (r) received for the current action and the discounted maximum Q-value of the next state (s') achievable by taking the optimal action (a'):

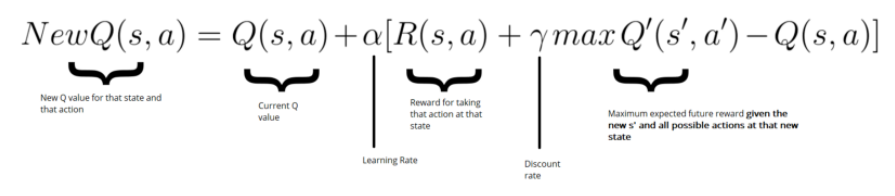


where γ is the discount factor that determines the importance of future rewards and helps to prevent infinite loops in cyclic environments.

Q-learning updates the Q-values for each state-action pair by repeatedly performing the following two steps until convergence:

***Exploration:*** The agent selects an action in a given state based on a policy, which is usually an epsilon-greedy strategy that balances exploration and exploitation. With a probability of ε, the agent chooses a random action, and with a probability of (1-ε), it selects the action that corresponds to the highest Q-value.

***Exploitation:*** The agent observes the reward (r) obtained by taking the selected action in the current state and the resulting next state (s'). It then updates the Q-value for the state-action pair (s, a) using the following update rule:



where α is the learning rate that determines the impact of the new experience on the existing Q-value. It is usually a small value to avoid overfitting and oscillation.

Q-learning has several benefits such as being model-free, computationally efficient, and able to learn optimal policies in environments with discrete state and action spaces. However, it also suffers from several challenges, such as the curse of dimensionality when dealing with continuous state and action spaces and the need to balance exploration and exploitation.

## **2.3 What is deep Q-learning ?**

Deep Q-Learning is a type of reinforcement learning algorithm that combines deep neural networks with the Q-learning algorithm to learn an optimal policy in a given environment. In this algorithm, an agent interacts with the environment by taking actions, and it receives rewards or penalties based on its actions. The agent's goal is to maximize the cumulative reward it receives over time.

The algorithm works by approximating the Q-function, which is the expected total reward of taking a particular action in a particular state. The Q-function is updated using the Bellman equation, which is a recursive equation that expresses the expected total reward in terms of the expected immediate reward and the expected total reward in the next state.

The basic Q-learning algorithm uses a lookup table to store the Q-values for each state-action pair, but this becomes impractical for large state spaces. Deep Q-Learning solves this problem by using a deep neural network to approximate the Q-function. The neural network takes the state as input and outputs Q-values for each possible action. During training, the network is updated using a variant of the backpropagation algorithm called stochastic gradient descent.

The deep Q-learning algorithm has several key components:

***Experience replay:*** The agent stores its experiences (state, action, reward, next state) in a replay buffer and samples from it randomly during training. This helps to prevent the agent from getting stuck in a local minimum by reducing the correlation between consecutive updates.

***Target network:*** The Q-values are estimated using a separate target network that is updated less frequently than the main network. This helps to stabilize the training process by preventing the Q-values from oscillating.

***Epsilon-greedy policy:*** The agent selects actions according to an epsilon-greedy policy, which means that it selects the action with the highest Q-value with probability (1-epsilon) and a random action with probability epsilon. This helps to ensure the exploration of the environment.

***Discount factor:*** The algorithm uses a discount factor (gamma) to give less weight to future rewards. This is necessary to prevent the algorithm from valuing immediate rewards over long-term rewards.

In summary, Deep Q-Learning is a powerful algorithm that can learn optimal policies in complex environments with large state spaces. It combines the Q-learning algorithm with deep neural networks to approximate the Q-function and uses experience replay, a target network, an epsilon-greedy policy, and a discount factor to improve the stability and performance of the algorithm.

## **2.4 Why is Reinforcement Learning used for self-driving cars?**

Reinforcement learning is particularly well-suited for self-driving cars due to the complexity and uncertainty inherent in real-world driving scenarios. Traditional rule-based approaches and deterministic algorithms struggle to handle the vast array of dynamic and unpredictable situations on the road.

Reinforcement learning allows self-driving cars to learn from experience and adapt their behavior based on the feedback received from the environment. By collecting data through interactions with the environment, self-driving cars can develop policies that balance exploration (trying new actions) and exploitation (leveraging learned knowledge) to optimize their driving behavior.

The ability of reinforcement learning to continually learn and adapt makes it a powerful approach for self-driving cars, enabling them to navigate diverse road conditions, handle unforeseen situations, and improve performance over time.

These two sections provide an introduction to self-driving cars and reinforcement learning, setting the stage for the subsequent sections of your project report. You can expand on each subtopic by providing more detailed explanations and supporting your statements with relevant examples or research findings.

***Why do we use Deep Q-Learning instead of the Q-Learning algorithm?***

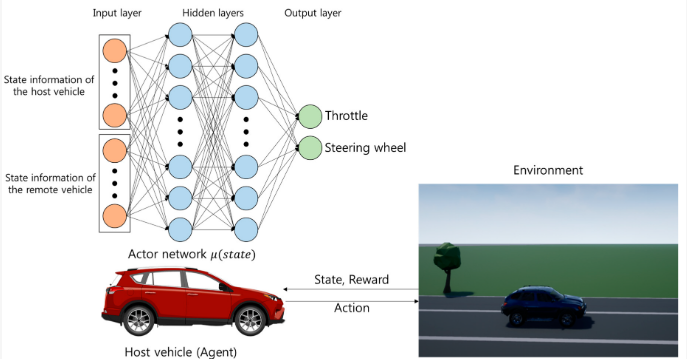
In this project, Deep Q-Learning is used instead of traditional Q-Learning due to the complexity and high-dimensional nature of the self-driving car environment. Q-Learning is a form of reinforcement learning that learns an optimal action-value function by iteratively updating a Q-table. However, Q-Learning faces challenges when applied to environments with large state and action spaces, as the size of the Q-table grows exponentially, making it computationally infeasible.

Deep Q-Learning, on the other hand, addresses these challenges by utilizing deep neural networks to approximate the Q-function instead of maintaining a Q-table. Deep Q-Learning combines Q-Learning with deep neural networks, allowing the agent to learn directly from raw sensory input, such as images from cameras mounted on the car.

The advantages of Deep Q-Learning in the context of self-driving cars are as follows:

* High-Dimensional Inputs: Self-driving car environments often involve high-dimensional inputs, such as images or sensor data. Deep neural networks excel at learning complex representations from raw data, enabling the agent to capture intricate patterns and features in the environment.
* Generalization: Deep Q-Learning enables the agent to generalize its learned knowledge across similar states. By leveraging the power of neural networks, the agent can infer appropriate actions even in states it has not encountered during training. This ability to generalize is crucial in dynamic and varied driving scenarios.
* Efficient Memory: Instead of storing all state-action pairs in a Q-table, Deep Q-Learning only requires storing a set of weights representing the neural network's parameters. This significantly reduces memory requirements and allows the agent to handle large state spaces more efficiently.
* Continuous Action Spaces: Self-driving car control often involves continuous action spaces, such as steering angles and acceleration. Deep Q-Learning can be combined with techniques like Deep Deterministic Policy Gradient (DDPG) to handle continuous actions, enabling the agent to learn smooth and continuous driving behaviors.

By using Deep Q-Learning, the self-driving car agent can learn directly from virtual sensor input, handle high-dimensional state spaces, generalize its knowledge, and efficiently explore and exploit the environment. These advantages make Deep Q-Learning a suitable and effective choice for training self-driving car agents in complex and realistic driving scenarios.



***Figure 3.****The architecture of the proposed decision-making system.*

# **3. Implement Reinforcement Learning in stimulating self-driving car**

## **3.1 Environment Setup**

Define the environment in which our self-driving car will operate. This would be a virtual environment with simulated roads, and obstacles. In this project, we use a simple library Pygame to create a virtual gaming environment that includes simple definitions such as defining **State, Action, and Reward of Agent**. The following steps will explain how the gaming environment is constructed.

***Set up supportive functions and classes:***

## Import needed libraries.

import pygame

import math

from Walls import Wall

from Walls import getWalls

from Goals import Goal

from Goals import getGoals

## Declare THE POINT of reward and punishment.

GOALREWARD = 1

LIFE\_REWARD = 0

PENALTY = -1

## Calculate the distance between 2 points pt1 and pt2.

def distance(pt1, pt2):

    return(((pt1.x - pt2.x)\*\*2 + (pt1.y - pt2.y)\*\*2)\*\*0.5)

## This is a function that rotates a POINT around an ORIGIN by an ANGLE.

def rotate(origin,point,angle):

    qx = origin.x + math.cos(angle) \* (point.x - origin.x) - math.sin(angle) \* (point.y - origin.y)

    qy = origin.y + math.sin(angle) \* (point.x - origin.x) + math.cos(angle) \* (point.y - origin.y)

    q = myPoint(qx, qy)

    return q

## Here is a function that rotates a rectangle with vertices pt1, pt2, pt3, pt4 around a center using def rotate(origin,point,angle)

def rotateRect(pt1, pt2, pt3, pt4, angle):

    pt\_center = myPoint((pt1.x + pt3.x)/2, (pt1.y + pt3.y)/2)

    pt1 = rotate(pt\_center,pt1,angle)

    pt2 = rotate(pt\_center,pt2,angle)

    pt3 = rotate(pt\_center,pt3,angle)

    pt4 = rotate(pt\_center,pt4,angle)

    return pt1, pt2, pt3, pt4

## This is a simple class presenting a point in 2D space. It has 2 properties 'x' and 'y' to save coordination

class myPoint:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

## Class to represent a line segment in 2D space. It has two properties pt1 and pt2 to store the two endpoints of the line segment

class myLine:

    def \_\_init\_\_(self, pt1, pt2):

        self.pt1 = myPoint(pt1.x, pt1.y)

        self.pt2 = myPoint(pt2.x, pt2.y)

## Class representing a ray in 2D space including 3 properties x, y (coordinates of the ray's starting point) and angle (the angle of rotation of the ray).

class Ray:

    def \_\_init\_\_(self,x,y,angle):

        self.x = x

        self.y = y

        self.angle = angle

    """This class provides a cast(wall) method that checks if the ray intersects the wall segment

            and returns the intersection point ( OBJECT 'myPoint') of the ray and the wall segment if so."""

    def cast(self, wall):

        x1 = wall.x1

        y1 = wall.y1

        x2 = wall.x2

        y2 = wall.y2

        vec = rotate(myPoint(0,0), myPoint(0,-1000), self.angle)

        x3 = self.x

        y3 = self.y

        x4 = self.x + vec.x

        y4 = self.y + vec.y

        den = (x1 - x2) \* (y3 - y4) - (y1 - y2) \* (x3 - x4)

        if(den == 0):

            den = 0

        else:

            t = ((x1 - x3) \* (y3 - y4) - (y1 - y3) \* (x3 - x4)) / den

            u = -((x1 - x2) \* (y1 - y3) - (y1 - y2) \* (x1 - x3)) / den

            if t > 0 and t < 1 and u < 1 and u > 0:

                pt = myPoint(math.floor(x1 + t \* (x2 - x1)), math.floor(y1 + t \* (y2 - y1)))

                return(pt)

***Class Car where we construct the car’s methods and properties such as: initial state, update state, action, checking collision, checking reach goal, virtual sensors for the car ( the way car look the environment):***

class Car:

    """ Initialize a new object car and set up its first properties such as coordination,shape,image,angle,velocity """

    def \_\_init\_\_(self, x, y):

        self.pt = myPoint(x, y) ## set up the  coordination and shape of the car

        self.x = x

        self.y = y

        self.width = 14

        self.height = 30

        self.points = 0

        self.original\_image = pygame.image.load("car.png").convert()

        self.image = self.original\_image  ## This will reference the rotated image.

        self.image.set\_colorkey((0,0,0))

        self.rect = self.image.get\_rect().move(self.x, self.y)

        self.angle = math.radians(180) ## set up the angle of the car

        self.soll\_angle = self.angle

        self.dvel = 1

        self.vel = 0

        self.velX = 0

        self.velY = 0

        self.maxvel = 10 # set up the maximum velocity of the car

        self.angle = math.radians(180)

        self.soll\_angle = self.angle

        ## calculate 4 vertices of the car (rectangular shape)

        self.pt1 = myPoint(self.pt.x - self.width / 2, self.pt.y - self.height / 2)

        self.pt2 = myPoint(self.pt.x + self.width / 2, self.pt.y - self.height / 2)

        self.pt3 = myPoint(self.pt.x + self.width / 2, self.pt.y + self.height / 2)

        self.pt4 = myPoint(self.pt.x - self.width / 2, self.pt.y + self.height / 2)

        self.p1 = self.pt1

        self.p2 = self.pt2

        self.p3 = self.pt3

        self.p4 = self.pt4

        self.distances = []

    """ Possible actions of the car """

    def action(self, choice):

        if choice == 0:                 # do nothing

            pass

        elif choice == 1:               # accelerate the car

            self.accelerate(self.dvel)

        elif choice == 8:

            self.accelerate(self.dvel)  # accelerate the car and rotate clockwise 15 degree

            self.turn(1)

        elif choice == 7:

            self.accelerate(self.dvel)  # accelerate the car and rotate counterclockwise 15 degree

            self.turn(-1)

        elif choice == 4:

            self.accelerate(-self.dvel) # decelerate the car

        elif choice == 5:

            self.accelerate(-self.dvel) # decelerate the car and rotate clockwise 15 degree

            self.turn(1)

        elif choice == 6:

            self.accelerate(-self.dvel) # decelerate the car and rotate counterclockwise 15 degree

            self.turn(-1)

        elif choice == 3:               # rotate clockwise 15 degree

            self.turn(1)

        elif choice == 2:               # rotate counterclockwise 15 degree

            self.turn(-1)

        pass

    """ Acceleration and deceleration function """

    def accelerate(self,dvel):

        dvel = dvel \* 2

        self.vel = self.vel + dvel

        if self.vel > self.maxvel:

            self.vel = self.maxvel

        if self.vel < -self.maxvel:

            self.vel = -self.maxvel

    """ Rotation function """

    def turn(self, dir):

        self.soll\_angle = self.soll\_angle + dir \* math.radians(15)

    """ This method updates the vehicle's position and rotation based on the desired velocity and rotation.

        It also updates the image of the car according to the new angle."""

    def update(self):

        # drifting code

        """This code handles reversing the rotation of the vehicle when the target angle ('self.soll\_angle')

            is different from the current angle (self.angle)."""

        if(self.soll\_angle > self.angle):

            if(self.soll\_angle > self.angle + math.radians(10) \* self.maxvel / ((self.velX\*\*2 + self.velY\*\*2)\*\*0.5 + 1)):

                self.angle = self.angle + math.radians(10) \* self.maxvel / ((self.velX\*\*2 + self.velY\*\*2)\*\*0.5 + 1)

            else:

                self.angle = self.soll\_angle

        if(self.soll\_angle < self.angle):

            if(self.soll\_angle < self.angle - math.radians(10) \* self.maxvel / ((self.velX\*\*2 + self.velY\*\*2)\*\*0.5 + 1)):

                self.angle = self.angle - math.radians(10) \* self.maxvel / ((self.velX\*\*2 + self.velY\*\*2)\*\*0.5 + 1)

            else:

                self.angle = self.soll\_angle

        self.angle = self.soll\_angle ## Update current angle by desired angle

        vec\_temp = rotate(myPoint(0,0), myPoint(0,self.vel), self.angle) ## Calculate X-axis and Y-axix velocity

        self.velX, self.velY = vec\_temp.x, vec\_temp.y

        ## Update the coordinate of the car

        self.x = self.x + self.velX

        self.y = self.y + self.velY

        self.rect.center = self.x, self.y ## the center point

        ## Update 4 point vertices

        self.pt1 = myPoint(self.pt1.x + self.velX, self.pt1.y + self.velY)

        self.pt2 = myPoint(self.pt2.x + self.velX, self.pt2.y + self.velY)

        self.pt3 = myPoint(self.pt3.x + self.velX, self.pt3.y + self.velY)

        self.pt4 = myPoint(self.pt4.x + self.velX, self.pt4.y + self.velY)

        self.p1 ,self.p2 ,self.p3 ,self.p4  = rotateRect(self.pt1, self.pt2, self.pt3, self.pt4, self.soll\_angle)

        self.image = pygame.transform.rotate(self.original\_image, 90 - self.soll\_angle \* 180 / math.pi) ## Rotate the image of the car

        x, y = self.rect.center  # Save its current center.

        self.rect = self.image.get\_rect()  # Replace old rectangle with new rectangle.

        self.rect.center = (x, y)

    """This method performs raying from the vehicle to identify obstacles in the environment.

        It returns observations of the distance to the nearest obstructions."""

    def cast(self, walls):

        ray1 = Ray(self.x, self.y, self.soll\_angle)

        ray2 = Ray(self.x, self.y, self.soll\_angle - math.radians(30))

        ray3 = Ray(self.x, self.y, self.soll\_angle + math.radians(30))

        ray4 = Ray(self.x, self.y, self.soll\_angle + math.radians(45))

        ray5 = Ray(self.x, self.y, self.soll\_angle - math.radians(45))

        ray6 = Ray(self.x, self.y, self.soll\_angle + math.radians(90))

        ray7 = Ray(self.x, self.y, self.soll\_angle - math.radians(90))

        ray8 = Ray(self.x, self.y, self.soll\_angle + math.radians(180))

        ray9 = Ray(self.x, self.y, self.soll\_angle + math.radians(10))

        ray10 = Ray(self.x, self.y, self.soll\_angle - math.radians(10))

        ray11 = Ray(self.x, self.y, self.soll\_angle + math.radians(135))

        ray12 = Ray(self.x, self.y, self.soll\_angle - math.radians(135))

        ray13 = Ray(self.x, self.y, self.soll\_angle + math.radians(20))

        ray14 = Ray(self.x, self.y, self.soll\_angle - math.radians(20))

        ray15 = Ray(self.p1.x,self.p1.y, self.soll\_angle + math.radians(90))

        ray16 = Ray(self.p2.x,self.p2.y, self.soll\_angle - math.radians(90))

        ray17 = Ray(self.p1.x,self.p1.y, self.soll\_angle + math.radians(0))

        ray18 = Ray(self.p2.x,self.p2.y, self.soll\_angle - math.radians(0))

        self.rays = []

        self.rays.append(ray1)

        self.rays.append(ray2)

        self.rays.append(ray3)

        self.rays.append(ray4)

        self.rays.append(ray5)

        self.rays.append(ray6)

        self.rays.append(ray7)

        self.rays.append(ray8)

        self.rays.append(ray9)

        self.rays.append(ray10)

        self.rays.append(ray11)

        self.rays.append(ray12)

        self.rays.append(ray13)

        self.rays.append(ray14)

        self.rays.append(ray15)

        self.rays.append(ray16)

        self.rays.append(ray17)

        self.rays.append(ray18)

        observations = [] ## contains observed values, including the distance to the obstacle and the velocity ratio.

        self.closestRays = []

        for ray in self.rays:

            closest = None #myPoint(0,0)

            record = math.inf

            for wall in walls:

                pt = ray.cast(wall)

                if pt: ## if pt (intersection exists),the distance will be calculated

                    dist = distance(myPoint(self.x, self.y),pt)

                    if dist < record:

                        record = dist

                        closest = pt

            if closest:

                #append distance for current ray

                self.closestRays.append(closest)

                observations.append(record)

            else:

                observations.append(1000) ## distance is 1000 to describe infinite distance

        for i in range(len(observations)):

            #invert observation values 0 is far away 1 is close

            observations[i] = ((1000 - observations[i]) / 1000) ## the value is converted to range from 0 (very far) to 1 (very close).

        observations.append(self.vel / self.maxvel) ## The rate velocity is equal the current velocity/ maximum velocity

        return observations

    """The ('collision') method helps to determine if the vehicle has collided with an obstacle based on

        checking the intersection of the lines of the vehicle and the obstacle."""

    def collision(self, wall):

        ## create 4 lines for 4 vertices of the car

        line1 = myLine(self.p1, self.p2)

        line2 = myLine(self.p2, self.p3)

        line3 = myLine(self.p3, self.p4)

        line4 = myLine(self.p4, self.p1)

        ## extract the begging and ending point of wall

        x1 = wall.x1

        y1 = wall.y1

        x2 = wall.x2

        y2 = wall.y2

        lines = []

        lines.append(line1)

        lines.append(line2)

        lines.append(line3)

        lines.append(line4)

        ## check collision

        for li in lines:

            x3 = li.pt1.x

            y3 = li.pt1.y

            x4 = li.pt2.x

            y4 = li.pt2.y

            den = (x1 - x2) \* (y3 - y4) - (y1 - y2) \* (x3 - x4)

            if(den == 0): ## there is no collision

                den = 0

            else:

                t = ((x1 - x3) \* (y3 - y4) - (y1 - y3) \* (x3 - x4)) / den

                u = -((x1 - x2) \* (y1 - y3) - (y1 - y2) \* (x1 - x3)) / den

                if t > 0 and t < 1 and u < 1 and u > 0:

                    return(True) ## collision is found

        return(False)

    """Score(self, goal) method: This method calculates the vehicle's score based on whether the vehicle passes a goal.

            If the vehicle approaches the target close enough, points will be added"""

    def score(self, goal):

        ## create a line from 2 vertices of the car

        line1 = myLine(self.p1, self.p3)

        vec = rotate(myPoint(0,0), myPoint(0,-50), self.angle)

        line1 = myLine(myPoint(self.x,self.y),myPoint(self.x + vec.x, self.y + vec.y))

        ## extract 2 points of the goal

        x1 = goal.x1

        y1 = goal.y1

        x2 = goal.x2

        y2 = goal.y2

        ## extract 2 points in the car's line

        x3 = line1.pt1.x

        y3 = line1.pt1.y

        x4 = line1.pt2.x

        y4 = line1.pt2.y

        den = (x1 - x2) \* (y3 - y4) - (y1 - y2) \* (x3 - x4)

        if(den == 0):  ## car does not touch goal

            den = 0

        else:

            t = ((x1 - x3) \* (y3 - y4) - (y1 - y3) \* (x3 - x4)) / den

            u = -((x1 - x2) \* (y1 - y3) - (y1 - y2) \* (x1 - x3)) / den

            if t > 0 and t < 1 and u < 1 and u > 0:

                pt = math.floor(x1 + t \* (x2 - x1)), math.floor(y1 + t \* (y2 - y1)) ## determine the intersection point

                d = distance(myPoint(self.x, self.y), myPoint(pt[0], pt[1])) ## calculate the distance between car and goal

                if d < 20: ## it means car went through goal

                    self.points += GOALREWARD

                    return(True)

        return(False)

    """This method resets the vehicle properties to their original state. It is used when wanting to start a new round."""

    def reset(self):

        self.x = 50

        self.y = 300

        self.velX = 0

        self.velY = 0

        self.vel = 0

        self.angle = math.radians(180)

        self.soll\_angle = self.angle

        self.points = 0

        self.pt1 = myPoint(self.pt.x - self.width / 2, self.pt.y - self.height / 2)

        self.pt2 = myPoint(self.pt.x + self.width / 2, self.pt.y - self.height / 2)

        self.pt3 = myPoint(self.pt.x + self.width / 2, self.pt.y + self.height / 2)

        self.pt4 = myPoint(self.pt.x - self.width / 2, self.pt.y + self.height / 2)

        self.p1 = self.pt1

        self.p2 = self.pt2

        self.p3 = self.pt3

        self.p4 = self.pt4

    def draw(self, win):

        win.blit(self.image, self.rect)

***The RacingEnv class provides methods for interacting with the racing environment, such as reset, step, and render.***

class RacingEnv:

    """ Initialize the variables and properties of the gaming environment """

    def \_\_init\_\_(self):

        pygame.init()

        self.font = pygame.font.Font(pygame.font.get\_default\_font(), 36)

        self.fps = 120

        self.width = 1000

        self.height = 600

        self.history = []

        self.screen = pygame.display.set\_mode((self.width, self.height))

        pygame.display.set\_caption("RACING DQN")

        self.screen.fill((0,0,0))

        self.back\_image = pygame.image.load("track.png").convert()

        self.back\_rect = self.back\_image.get\_rect().move(0, 0)

        self.action\_space = None

        self.observation\_space = None

        self.game\_reward = 0

        self.score = 0

        self.reset()

    """ Resetting the racing environment. It resets the screen and regenerates the Car object,

        the list of Wall and Goal objects, and sets the game reward to 0."""

    def reset(self):

        self.screen.fill((0, 0, 0))

        self.car = Car(50, 300)

        self.walls = getWalls()

        self.goals = getGoals()

        self.game\_reward = 0

    """ The method that executes a step in a racing environment is based on the action given.

        This method updates the state of the ('Car') object """

    def step(self, action):

        done = False

        self.car.action(action)

        self.car.update()

        reward = LIFE\_REWARD

        ## Check if car passes Goal and calculate the reward point

        index = 1

        for goal in self.goals:

            if index > len(self.goals):

                index = 1

            if goal.isactiv:

                if self.car.score(goal):

                    goal.isactiv = False

                    self.goals[index-2].isactiv = True

                    reward += GOALREWARD

            index = index + 1

        ## check if car crashed in the wall the game will be reset

        for wall in self.walls:

            if self.car.collision(wall):

                reward += PENALTY

                done = True

        new\_state = self.car.cast(self.walls)

        ## normalize states

        if done:

            new\_state = None

        return new\_state, reward, done

    """ Represent the current state of the gaming environment on the screen by using class Car,Wall,Goal,Ray """

    def render(self, action):

        DRAW\_WALLS = False

        DRAW\_GOALS = False

        DRAW\_RAYS = False

        pygame.time.delay(10)

        self.clock = pygame.time.Clock()

        self.screen.fill((0, 0, 0))

        self.screen.blit(self.back\_image, self.back\_rect)

        if DRAW\_WALLS:

            for wall in self.walls:

                wall.draw(self.screen)

        if DRAW\_GOALS:

            for goal in self.goals:

                goal.draw(self.screen)

                if goal.isactiv:

                    goal.draw(self.screen)

        self.car.draw(self.screen)

        if DRAW\_RAYS:

            i = 0

            for pt in self.car.closestRays:

                pygame.draw.circle(self.screen, (0,0,255), (pt.x, pt.y), 5)

                i += 1

                if i < 15:

                    pygame.draw.line(self.screen, (0,255,255), (self.car.x, self.car.y), (pt.x, pt.y), 1)

                elif i >=15 and i < 17:

                    pygame.draw.line(self.screen, (0,255,255), ((self.car.p1.x + self.car.p2.x)/2, (self.car.p1.y + self.car.p2.y)/2), (pt.x, pt.y), 1)

                elif i == 17:

                    pygame.draw.line(self.screen, (0,255,255), (self.car.p1.x , self.car.p1.y ), (pt.x, pt.y), 1)

                else:

                    pygame.draw.line(self.screen, (0,255,255), (self.car.p2.x, self.car.p2.y), (pt.x, pt.y), 1)

        ## Render controll

        pygame.draw.rect(self.screen,(255,255,255),(800, 100, 40, 40),2)

        pygame.draw.rect(self.screen,(255,255,255),(850, 100, 40, 40),2)

        pygame.draw.rect(self.screen,(255,255,255),(900, 100, 40, 40),2)

        pygame.draw.rect(self.screen,(255,255,255),(850, 50, 40, 40),2)

        ## Render the action of the car

        if action == 4:

            pygame.draw.rect(self.screen,(0,255,0),(850, 50, 40, 40))

        elif action == 6:

            pygame.draw.rect(self.screen,(0,255,0),(850, 50, 40, 40))

            pygame.draw.rect(self.screen,(0,255,0),(800, 100, 40, 40))

        elif action == 5:

            pygame.draw.rect(self.screen,(0,255,0),(850, 50, 40, 40))

            pygame.draw.rect(self.screen,(0,255,0),(900, 100, 40, 40))

        elif action == 1:

            pygame.draw.rect(self.screen,(0,255,0),(850, 100, 40, 40))

        elif action == 8:

            pygame.draw.rect(self.screen,(0,255,0),(850, 100, 40, 40))

            pygame.draw.rect(self.screen,(0,255,0),(800, 100, 40, 40))

        elif action == 7:

            pygame.draw.rect(self.screen,(0,255,0),(850, 100, 40, 40))

            pygame.draw.rect(self.screen,(0,255,0),(900, 100, 40, 40))

        elif action == 2:

            pygame.draw.rect(self.screen,(0,255,0),(800, 100, 40, 40))

        elif action == 3:

            pygame.draw.rect(self.screen,(0,255,0),(900, 100, 40, 40))

        # Present the reward score

        text\_surface = self.font.render(f'Points {self.car.points}', True, pygame.Color('green'))

        self.screen.blit(text\_surface, dest=(0, 0))

        # Present the speed

        text\_surface = self.font.render(f'Speed {self.car.vel\*-1}', True, pygame.Color('green'))

        self.screen.blit(text\_surface, dest=(800, 0))

        self.clock.tick(self.fps)

        pygame.display.update()

    def close(self):

        pygame.quit()

## **3.2 Implement Agent and Reinforcement Learning Algorithm**

***The ReplayBuffer class is used to store and sample experiences for training the DDQN agent.***

"""The ReplayBuffer class is used to store and sample experiences for training the DDQN agent."""

class ReplayBuffer(object):

    def \_\_init\_\_(self, max\_size, input\_shape, n\_actions, discrete=False):

        ## max\_size: Maximum size of the replay buffer.

        ## input\_shape: Shape of the input state.

        ## n\_actions: Number of possible actions.

        ## discrete: Boolean value indicating whether the action space is discrete or continuous.

        self.mem\_size = max\_size

        self.mem\_cntr = 0

        self.discrete = discrete

        self.state\_memory = np.zeros((self.mem\_size, input\_shape))

        self.new\_state\_memory = np.zeros((self.mem\_size, input\_shape))

        dtype = np.int8 if self.discrete else np.float32

        self.action\_memory = np.zeros((self.mem\_size, n\_actions), dtype=dtype)

        self.reward\_memory = np.zeros(self.mem\_size)

        self.terminal\_memory = np.zeros(self.mem\_size, dtype=np.float32)

    ## Stores a transition in the replay buffer.

    def store\_transition(self, state, action, reward, state\_, done):

        ## state: Current state

        ## action: Action taken.

        ## reward: Reward received

        ## state\_: Next state

        ## done: Boolean value indicating whether the episode is done.

        index = self.mem\_cntr % self.mem\_size

        self.state\_memory[index] = state

        self.new\_state\_memory[index] = state\_

        ## store one hot encoding of actions, if appropriate

        if self.discrete:

            actions = np.zeros(self.action\_memory.shape[1])

            actions[action] = 1.0

            self.action\_memory[index] = actions

        ## store value of actions , but not one hot encoding form

        else:

            self.action\_memory[index] = action

        self.reward\_memory[index] = reward

        self.terminal\_memory[index] = 1 - done

        self.mem\_cntr += 1

    """ Take a transition batch from replay buffer. """

    def sample\_buffer(self, batch\_size):

        ## batch\_size: size of batch

        max\_mem = min(self.mem\_cntr, self.mem\_size)

        batch = np.random.choice(max\_mem, batch\_size)

        states = self.state\_memory[batch]

        actions = self.action\_memory[batch]

        rewards = self.reward\_memory[batch]

        states\_ = self.new\_state\_memory[batch]

        terminal = self.terminal\_memory[batch]

        return states, actions, rewards, states\_, terminal

***DDQNAgent class presents all of task the Agent need to do during training time:***

class DDQNAgent(object): ## Q(s, a) = Q(s, a) + α \* [r + γ \* max\_a' Q(s', a') - Q(s, a)]

    def \_\_init\_\_(self, alpha, gamma, n\_actions, epsilon, batch\_size,

                 input\_dims, epsilon\_dec=0.999995,  epsilon\_end=0.10,

                 mem\_size=25000, fname='ddqn\_model.h5', replace\_target=25):

        self.action\_space = [i for i in range(n\_actions)]

        self.n\_actions = n\_actions

        self.gamma = gamma

        self.epsilon = epsilon

        self.epsilon\_dec = epsilon\_dec

        self.epsilon\_min = epsilon\_end

        self.batch\_size = batch\_size

        self.model\_file = fname

        self.replace\_target = replace\_target

        self.memory = ReplayBuffer(mem\_size, input\_dims, n\_actions, discrete=True)

        self.brain\_eval = Brain(input\_dims, n\_actions, batch\_size)

        self.brain\_target = Brain(input\_dims, n\_actions, batch\_size)

    ## Stores a transition in the replay buffer.

    def remember(self, state, action, reward, new\_state, done):

        self.memory.store\_transition(state, action, reward, new\_state, done)

    ## Chooses an action based on the epsilon-greedy policy.

    def choose\_action(self, state):

        state = np.array(state)

        state = state[np.newaxis, :]

        rand = np.random.random()

        if rand < self.epsilon:

            action = np.random.choice(self.action\_space)

        else:

            actions = self.brain\_eval.predict(state)

            action = np.argmax(actions)

        return action

    """ Performs the Q-learning update using a batch of samples from the replay buffer."""

    def learn(self):

        ## checks if the replay buffer has enough data to perform the learning process. If not, the method stops without doing anything.

        if self.memory.mem\_cntr > self.batch\_size:

            ##  randomly samples a batch of data from the replay buffer

            state, action, reward, new\_state, done = self.memory.sample\_buffer(self.batch\_size)

            ##  creates an array containing the possible action values.

            action\_values = np.array(self.action\_space, dtype=np.int8)

            action\_indices = np.dot(action, action\_values)

            ## predicts the Q-values for the new states using the target network (brain\_target).

            q\_next = self.brain\_target.predict(new\_state)

            ##  predicts the Q-values for the new states using the evaluation network (brain\_eval).

            q\_eval = self.brain\_eval.predict(new\_state)

            ##  predicts the Q-values for the current states using the evaluation network (brain\_eval).

            q\_pred = self.brain\_eval.predict(state)

            ##  finds the indices of the actions with the highest Q-values in the predicted Q-values for the new states.

            max\_actions = np.argmax(q\_eval, axis=1)

            ## copies the predicted Q-values for the current states to use as the target for the update.

            q\_target = q\_pred

            batch\_index = np.arange(self.batch\_size, dtype=np.int32)

            ## updates the target Q-values (q\_target) for the selected state-action pairs.

            # Q(s, a) = r + γ \* max\_a' Q(s', a')

            q\_target[batch\_index, action\_indices] = reward + self.gamma\*q\_next[batch\_index, max\_actions.astype(int)]\*done

            ## trains the evaluation network (brain\_eval) using the current states and the updated target Q-values.

            \_ = self.brain\_eval.train(state, q\_target)

            """ updates the exploration rate (epsilon) by decreasing it gradually over time,

                ensuring a balance between exploration and exploitation during training."""

            self.epsilon = self.epsilon\*self.epsilon\_dec if self.epsilon > self.epsilon\_min else self.epsilon\_min

    ## Updates the target network with the evaluation network's weights.

    def update\_network\_parameters(self):

        self.brain\_target.copy\_weights(self.brain\_eval)

    ## Saves the trained model

    def save\_model(self):

        self.brain\_eval.model.save(self.model\_file)

    ## Loads a trained model from a file

    def load\_model(self):

        self.brain\_eval.model = load\_model(self.model\_file)

        self.brain\_target.model = load\_model(self.model\_file)

        if self.epsilon == 0.0:

            self.update\_network\_parameters()

***Explain more the most significant parameter:   
ddqn\_agent:*** The DDQNAgent object that represents the agent. You can modify the parameters when creating the agent:

* ***alpha:*** The learning rate for the agent. You can adjust this parameter to control the rate at which the agent learns from new experiences.
* ***gamma:*** The discount factor for future rewards. You can modify this parameter to adjust the importance of future rewards in the agent's decision-making process.
* ***n\_actions:*** The number of possible actions in the environment. Modify this parameter according to the specific environment you are using.
* ***epsilon:*** The exploration rate of the agent. You can adjust this parameter to control the balance between exploration and exploitation.
* ***epsilon\_end:*** The minimum exploration rate. You can change this parameter to determine the lowest exploration rate the agent can reach.
* ***epsilon\_dec:*** The rate at which the exploration rate is decayed over time. You can modify this parameter to adjust the speed of exploration rate decay.
* ***batch\_size:*** The batch size used for training the agent's neural network. Modify this parameter according to your available computational resources.
* ***input\_dims:*** The dimensions of the input state. Modify this parameter according to the specific environment you are using

***The Brain class represents the neural network model used in the DQN agent. Let's go through the different components and methods of the class:***

""" Representing the neural network model used in the DQN agent."""

class Brain:

    def \_\_init\_\_(self, NbrStates, NbrActions, batch\_size = 256):

        self.NbrStates = NbrStates

        self.NbrActions = NbrActions

        self.batch\_size = batch\_size

        self.model = self.createModel()

    ## model i have designed for train agent

    def createModel(self):

        model = tf.keras.Sequential()

        model.add(tf.keras.layers.Dense(256, activation=tf.nn.relu)) #prev 256

        model.add(tf.keras.layers.Dense(self.NbrActions, activation=tf.nn.softmax))

        model.compile(loss = "mse", optimizer="adam")

        return model

    ## x is agent's states and y is predicted Q-values

    def train(self, x, y, epoch = 1, verbose = 0):

        self.model.fit(x, y, batch\_size = self.batch\_size , verbose = verbose)

    """ This method takes a state (s) as input and returns the predicted Q-values for all possible actions.

        It uses the neural network model to perform the prediction."""

    def predict(self, s):

        return self.model.predict(s)

    """ This method is similar to 'predict', but it takes a single state (s) as input and returns the predicted Q-values as a flattened array.

        It reshapes the input state to match the expected shape of the model input."""

    def predictOne(self, s):

        return self.model.predict(tf.reshape(s, [1, self.NbrStates])).flatten()

    """This method copies the weights of the trainable variables from another 'Brain' object (TrainNet) to the current 'Brain' object.

It iterates over the trainable variables of both models and assigns the values from TrainNet to the corresponding variables in the current model.

    This is typically used to update the target network with the weights of the evaluation network in the DQN algorithm."""

    def copy\_weights(self, TrainNet):

        variables1 = self.model.trainable\_variables

        variables2 = TrainNet.model.trainable\_variables

        for v1, v2 in zip(variables1, variables2):

            v1.assign(v2.numpy())

## **3.3 Train Agent**

***Initialize the needed parameters and variables:***

## Import needed libraries

import GameEnv

import pygame

import numpy as np

from ddqn\_keras import DDQNAgent

from collections import deque

import random, math

TOTAL\_GAMETIME = 10000 # Max game time for one episode

N\_EPISODES = 10000 # Number of training episodes

REPLACE\_TARGET = 50  # Number of episodes to update target model

game = GameEnv.RacingEnv() # Initialize game environment

game.fps = 60  # Set frame rate

GameTime = 0  # Game time counter

GameHistory = [] # Store game state history

renderFlag = False # # Flag to render the game

## ## Initializes a DDQNAgent object. The arguments of DDQNAgent determine important training parameters

ddqn\_agent = DDQNAgent(alpha=0.0005, gamma=0.99, n\_actions=8, epsilon=1.00, epsilon\_end=0.10, epsilon\_dec=0.9995, replace\_target= REPLACE\_TARGET, batch\_size=512, input\_dims=19)

ddqn\_scores = [] # A list to store the scores achieved in each simulation episode.

eps\_history = [] # A list to store the epsilon (exploration rate) values during

***Define the run() function to run the training process:***

a. Iterate through episodes:

b. Reset the game environment (game.reset()), set done flag, and initialize the score and counter.

c. Set the initial observation state (observation) and game time (gtime).

d. Loop until the game is done (done = True):

* Get an action from the model (ddqn\_agent.choose\_action(observation)).
* Perform the action and update the environment state (game.step(action)).
* Check the game end condition and calculate the reward.
* Store the current observation, action, reward, and next observation in the memory buffer (ddqn\_agent.remember()).
* Update the observation and train the model with the new state (observation = observation\_ and ddqn\_agent.learn()).
* Increment the game time.
* Render the game state if the renderFlag is set.

e. Append the current epsilon value to eps\_history and the score to ddqn\_scores.

f. Calculate the average score over the last 100 episodes (avg\_score).

g. Update the target network (ddqn\_agent.update\_network\_parameters()) if the specified number of episodes has passed.

h. Save the model and print relevant training information at regular intervals.

Call the run() function to start the training process.

During training, the model interacts with the game environment through selected actions and observed states. The model is trained through memory replay (remember()) and updating the neural network (learn()). Information about scores and epsilon values is stored to monitor and evaluate

def run():

    for e in range(N\_EPISODES):

        game.reset() #reset environment to start a new episode

        done = False

        score = 0

        counter = 0

        # initialize observation list

        observation\_, reward, done = game.step(0)

        observation = np.array(observation\_)

        gtime = 0 # set game time back to 0

        renderFlag = True # if you want to render every episode set to true

        if e % 10 == 0 and e > 0: # render every 10 episodes

            renderFlag = True

        while not done:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    return

            action = ddqn\_agent.choose\_action(observation) # Uses the DQN model to choose an action based on the current state (observation).

            ##  Performs the chosen action in the game environment and receives the next state (observation\_), reward, and done flag.

            observation\_, reward, done = game.step(action)

            observation\_ = np.array(observation\_)

            # This is a countdown if no reward is collected the car will be done within 100 ticks

            if reward == 0:

                counter += 1

                if counter > 100:

                    done = True

            else:

                counter = 0

            score += reward ## update reward

            ddqn\_agent.remember(observation, action, reward, observation\_, int(done)) ## Stores a transition in the replay buffer.

            observation = observation\_

            ddqn\_agent.learn() ## Update the neural network for predicting Q-values

            # Update simulation time if TOTAL\_GAMETIME = 10000 we end simlation

            # Increase TOTAL\_GAMETIME if we want to increase training time

            gtime += 1

            if gtime >= TOTAL\_GAMETIME:

                done = True

            if renderFlag:

                game.render(action)

        eps\_history.append(ddqn\_agent.epsilon) ## Append the current 'epsilon' value (exploration rate) to the 'eps\_history' list.

        ddqn\_scores.append(score) ## Append the current episode's score to the 'ddqn\_scores' list, which stores the scores achieved during training.

        """ Calculate the average score over the last 100 episodes using the ddqn\_scores list.

            This provides an indication of the agent's overall performance."""

        avg\_score = np.mean(ddqn\_scores[max(0, e-100):(e+1)])

        """ Check if the current episode number is divisible by REPLACE\_TARGET and greater than REPLACE\_TARGET.

            This condition is used to determine when to update the target network.

            The target network is a separate copy of the main network that is periodically updated to stabilize the training process."""

        if e % REPLACE\_TARGET == 0 and e > REPLACE\_TARGET:

            ddqn\_agent.update\_network\_parameters()

        """ After every 10 episodes save model once

            The model is saved periodically to capture the progress and allow for resuming training or inference later."""

        if e % 10 == 0 and e > 10:

            ddqn\_agent.save\_model()

            print("save model")

        ##  Print the training progress information

        print('episode: ', e,'score: %.2f' % score,

              ' average score %.2f' % avg\_score,

              ' epsolon: ', ddqn\_agent.epsilon,

              ' memory size', ddqn\_agent.memory.mem\_cntr % ddqn\_agent.memory.mem\_size)

run()

## **3.4 Test trained Agent on the environment**

***After training the model to predict Q-values for states and saving it in the 'ddqn\_model.h5' file, we use this trained model to control a car in a game environment. Here's a breakdown of how the model is used to control the car:***

## Import needed libraries

import GameEnv

import pygame

import numpy as np

from ddqn\_keras import DDQNAgent

from collections import deque

import random, math

## Set up parameters

TOTAL\_GAMETIME = 10000 # Max game time for one episode

N\_EPISODES = 10000 # Number of training episodes

REPLACE\_TARGET = 50  # Number of episodes to update target model

game = GameEnv.RacingEnv() # Initialize game environment

game.fps = 60  # Set frame rate

GameTime = 0  # Game time counter

GameHistory = [] # Store game state history

renderFlag = False # # Flag to render the game

## ## Initializes a DDQNAgent object. The arguments of DDQNAgent determine important training parameters

ddqn\_agent = DDQNAgent(alpha=0.0005, gamma=0.99, n\_actions=8, epsilon=1.00, epsilon\_end=0.10, epsilon\_dec=0.9995, replace\_target= REPLACE\_TARGET, batch\_size=512, input\_dims=19)

ddqn\_scores = [] # A list to store the scores achieved in each simulation episode.

eps\_history = [] # A list to store the epsilon (exploration rate) values during

***The ‘run()’ function:***

Inside the for loop, iterating over the simulation episodes from 0 to N\_EPISODES:

* game.reset(): Resets the game environment.
* Sets counters and initializes the score.
* Within the while loop, until the game state is done (done = True):
* Checks for pygame events.
* ddqn\_agent.choose\_action(observation): Uses the DQN model to choose an action based on the current state (observation).
* game.step(action): Performs the chosen action in the game environment and receives the next state (observation\_), reward, and done flag.
* Handles game-specific time and reward characteristics.
* Updates the current state (observation) with the next state (observation\_).
* Checks for game end conditions.
* Renders the current state of the game (if enabled).

This ‘run()’ function runs the game by utilizing the trained DQN model to control the car in the game environment.

def run():

    for e in range(N\_EPISODES):

        game.reset() #reset environment to start a new episode

        done = False

        score = 0

        counter = 0

        # initialize observation list

        observation\_, reward, done = game.step(0)

        observation = np.array(observation\_)

        gtime = 0 # set game time back to 0

        renderFlag = True # if you want to render every episode set to true

        if e % 10 == 0 and e > 0: # render every 10 episodes

            renderFlag = True

        while not done:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    return

            action = ddqn\_agent.choose\_action(observation) # Uses the DQN model to choose an action based on the current state (observation).

            ##  Performs the chosen action in the game environment and receives the next state (observation\_), reward, and done flag.

            observation\_, reward, done = game.step(action)

            observation\_ = np.array(observation\_)

            # This is a countdown if no reward is collected the car will be done within 100 ticks

            if reward == 0:

                counter += 1

                if counter > 100:

                    done = True

            else:

                counter = 0

            score += reward ## update reward

            ddqn\_agent.remember(observation, action, reward, observation\_, int(done)) ## Stores a transition in the replay buffer.

            observation = observation\_

            ddqn\_agent.learn() ## Update the neural network for predicting Q-values

            # Update simulation time if TOTAL\_GAMETIME = 10000 we end simlation

            # Increase TOTAL\_GAMETIME if we want to increase training time

            gtime += 1

            if gtime >= TOTAL\_GAMETIME:

                done = True

            if renderFlag:

                game.render(action)

        eps\_history.append(ddqn\_agent.epsilon) ## Append the current 'epsilon' value (exploration rate) to the 'eps\_history' list.

        ddqn\_scores.append(score) ## Append the current episode's score to the 'ddqn\_scores' list, which stores the scores achieved during training.

        """ Calculate the average score over the last 100 episodes using the ddqn\_scores list.

            This provides an indication of the agent's overall performance."""

        avg\_score = np.mean(ddqn\_scores[max(0, e-100):(e+1)])

        """ Check if the current episode number is divisible by REPLACE\_TARGET and greater than REPLACE\_TARGET.

            This condition is used to determine when to update the target network.

            The target network is a separate copy of the main network that is periodically updated to stabilize the training process."""

        if e % REPLACE\_TARGET == 0 and e > REPLACE\_TARGET:

            ddqn\_agent.update\_network\_parameters()

        """ After every 10 episodes save model once

            The model is saved periodically to capture the progress and allow for resuming training or inference later."""

        if e % 10 == 0 and e > 10:

            ddqn\_agent.save\_model()

            print("save model")

        ##  Print the training progress information

        print('episode: ', e,'score: %.2f' % score,

              ' average score %.2f' % avg\_score,

              ' epsolon: ', ddqn\_agent.epsilon,

              ' memory size', ddqn\_agent.memory.mem\_cntr % ddqn\_agent.memory.mem\_size)

run()