“Racecar Evolution”

A MINI PROJECT REPORT

18CSC305J - ARTIFICIAL INTELLIGENCE

Submitted by

SHUBHAM GUSAIN [RA2011027010048]

Under the guidance of

Dr. Mercy Theresa

Assistant Professor, Department of Computer Science and Engineering

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur, Chengalpattu District

MAY 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that Mini project report titled **"Racecar Evolution"** is the bona fide work of **Shubham Gusain (RA2011027010048)** who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Dr. Mercy Theresa
GUIDE
Assistant Professor
Department of Data Science and Business Systems

SIGNATURE

Dr. M Lakshmi
HEAD OF THE DEPARTMENT
Professor & Head
Department of Data Science and Business Systems

ABSTRACT

This project report presents an AI program in python that simulates how a racecar finds the optimal speed and path to complete a given course. The program is designed to use computer vision techniques and machine learning algorithms to detect the course layout, predict the optimal path, and calculate the optimal speed for the racecar to follow and avoid crashing.

To achieve accurate course detection, I used a simple computer vision technique, which uses the track colour, differentiates it from the background on the basis of contrast. It then lets the racecars know about the track edges and boundaries. If the racecars contact the boundary, they crash (get eliminated from the map) and will not spawn (be loaded again) until the next generation. To find the optimal speed and path for the racecar to follow, I used an optimization algorithm – "Reinforcement Algorithm". In the first generations of the racecars the cars may not actually be able to complete the course as they are not yet optimized. But as more generations' data is used to reinforce the optimal path for the racecars to follow and the speeds that should be kept, the cars start to efficiently complete the course while considering the boundaries and obstacles.

The program provides real-time feedback to the user, including a visual display of the racecar's current speed, planned path. The program is capable of making decisions in real-time, and it can adjust its speed and path based on the feedback it receives from the environment.

Overall, the program shows promising results in accurately detecting the course layout and finding the optimal speed and path for the racecar to follow. The program has several potential applications, including autonomous racing technologies, racing simulators, and driver assistance systems.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Autonomous driving technology has rapidly advanced in recent years, with self-driving cars becoming a reality in many parts of the world. One of the key challenges in autonomous driving is to develop algorithms that can safely navigate vehicles through complex environments. This involves not only detecting obstacles but also planning optimal paths and speeds to reach a destination.

In this context, simulating how a racecar finds the optimal speed and path to complete a course is a relevant and challenging problem. The racecar must be able to detect the course layout, identify obstacles, and calculate the optimal path and speed to complete the course in the shortest time possible.

To address this problem, I propose an AI program that uses computer vision techniques and machine learning algorithms to simulate how a racecar can find the optimal speed and path to complete a course. The program uses computer vision to detect the course layout, and a combination of search and optimization algorithms to plan the optimal path and speed.

The program has several potential applications, including autonomous racing technologies, crash avoidance technologies, racing simulators, and driver assistance systems. By simulating how a racecar can navigate a course, I can develop algorithms that can be applied to real-world scenarios, such as self-driving cars, where safety and efficiency are critical factors.

CHAPTER 2

LITERATURE SURVEY

[1] Q. Ye and D. Doermann, "Text Detection and Recognition in Imagery: A Survey," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 37, no. 7, pp. 1480-1500, 1 July 2015, doi: 10.1109/TPAMI.2014.2366765.

[2] Long, S., He, X. & Yao, C. Scene Text Detection and Recognition: The Deep Learning Era. *Int J Comput Vis* **129**, 161–184 (2021). https://doi.org/10.1007/s11263-020-01369-0 Above techniques were considered to do path detection, ultimately a much basic version of them was used (based on contrast difference). This change was made as the path detection happens in real time and thus has to be easily calculable by the processor to avoid lag.

[3] Wiering, Marco A., and Martijn Van Otterlo. "Reinforcement learning." *Adaptation, learning, and optimization* 12.3 (2012): 729.

[4] Peter Dayan, Yael Niv, Reinforcement learning: The Good, The Bad and The Ugly, Current Opinion in Neurobiology, Volume 18, Issue 2, 2008, Pages 185-196, ISSN 0959-4388, https://doi.org/10.1016/j.conb.2008.08.003.

[5] Barto, Andrew G., and Richard S. Sutton. "Reinforcement learning." *Handbook of brain theory and neural networks* (1995): 804-809.

[6] François-Lavet, Vincent, et al. "An introduction to deep reinforcement learning." *Foundations and Trends® in Machine Learning* 11.3-4 (2018): 219-354.

CHAPTER 3

METHODOLOGY

Step 1. When a track is loaded, the program checks if it is of correct size to ensure the car will not spawn outside the track. This is done as the track “start” coordinates are pre-specified. The cars should only be spawned on the track itself.

Step 2. The first-generation cars are spawned in. For the sake of processor, each generation spawns with not more than 30 cars (although more can be spawned in with a faster processor or less with a slower processor). The cars start off at the same coordinates, but each have slightly different spawning parameters which are randomized for the sake of “evolution”. Each of the 30 cars will be given a slightly different angle (30 degrees deviation in either clockwise or anti-clockwise direction) when spawning to increase randomness. It should be noted that the number of generations that evolve is also dependent on the number of cars per generation. The more cars in each generation, the less total number of generations are needed.

Step 3. Each car has its own sense of “radar”. In order to decide if the car should go left, right, fast or slow etc., 5 different “radars” are thrown by the car. With the help of basic trigonometry, their distances from edges and boundaries are contextualized.

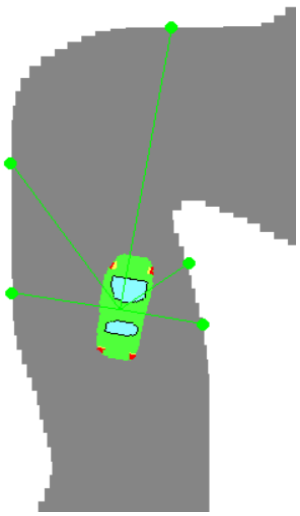

Fig. 3.1 – Radar Positioning

3.1. If the forward-facing radar does not collide with boundary, The car will speed up. If collision happens, the car will slow down to avoid collision.

3.2. If the left-facing radar does collide with a boundary, but its adjacent radar does not collide, it pushes a possibility of upcoming turn, hence car will turn left. Similarly right-side also works. If both the side radar and its adjacent radars collide, the car will not turn in that direction. This detection happens in real-time. It can be noted that this can also cause car to crash if the turn was false or a decoy.

Step 4. In case of collision, the car will get destroyed which means it will be removed from the map. Since multiple cars are spawned at once, the cars themselves do not collide and can overlap. But any overlap with boundary denotes a crash. The car's crash data will be used to reinforce the cars of the new next generation

Step 5. In case that all of the cars of a generation get destroyed, a new generation will be spawned. This generation will use the data gathered from the previous generations to better maneuver the cars and avoid crashes. Technically, a generation will continue until the last car is crashed, and the program will keep running infinitely. Hence if a car can complete the course in one generation more than 50 times then the program will exit. Often in the later generations, a single car will survive the course and keep running. This shows a mostly optimized racecar which has basic knowledge of where to speed-up, slow-down, or take a turn without crashing.

CHAPTER 4

CODING AND TESTING

The coding of this program is done using python. This is done to utilize the neural networking library: NEAT. The python script renders the track and cars using a library: pygame. This renders the frame with the track, the cars, and also indicates the Generation number, the number of surviving cars. This display updates multiple times every second (as much as processor can handle).

The parameters are stored in a ".txt" file, which gets accessed by the python script to work. It also stores the car's parameters which include number of radars, number of cars that will spawn, angle of deviation of spawned cars, etc.

Here is the Python code used to run the program:

```
import pygame
import os
import math
import sys
import random
import neat

screen_width = 1500
screen_height = 800
generation = 0

class Car:
    def __init__(self):
        self.surface = pygame.image.load("car.png")
        self.surface =
pygame.transform.scale(self.surface, (100, 100))
        self.rotate_surface = self.surface
        self.pos = [700, 650]
        self.angle = 0
        self.speed = 0
        self.center = [self.pos[0] + 50, self.pos[1] +
50]
        self.radars = []
        self.radars_for_draw = []
        self.is_alive = True
        self.goal = False
        self.distance = 0
        self.time_spent = 0
```

```
    def draw(self, screen):
        screen.blit(self.rotate_surface, self.pos)
        self.draw_radar(screen)

    def draw_radar(self, screen):
        for r in self.radars:
            pos, dist = r
            pygame.draw.line(screen, (0, 255, 0),
self.center, pos, 1)
            pygame.draw.circle(screen, (0, 255, 0), pos,
5)

    def check_collision(self, map):
        self.is_alive = True
        for p in self.four_points:
            if map.get_at((int(p[0]), int(p[1]))) ==
(255, 255, 255, 255):
                self.is_alive = False
                break

    def check_radar(self, degree, map):
        len = 0
        x = int(self.center[0] +
math.cos(math.radians(360 - (self.angle + degree))) *
len)
        y = int(self.center[1] +
math.sin(math.radians(360 - (self.angle + degree))) *
len)

        while not map.get_at((x, y)) == (255, 255, 255,
255) and len < 300:
            len = len + 1
            x = int(self.center[0] +
math.cos(math.radians(360 - (self.angle + degree))) *
len)
            y = int(self.center[1] +
math.sin(math.radians(360 - (self.angle + degree))) *
len)

        dist = int(math.sqrt(math.pow(x - self.center[0],
2) + math.pow(y - self.center[1], 2)))
        self.radars.append([(x, y), dist])

    def update(self, map):
        #speed check
```

```
        self.speed = 15
        #position check
        self.rotate_surface =
self.rot_center(self.surface, self.angle)
        self.pos[0] += math.cos(math.radians(360 -
self.angle)) * self.speed
        if self.pos[0] < 20:
            self.pos[0] = 20
        elif self.pos[0] > screen_width - 120:
            self.pos[0] = screen_width - 120

        self.distance += self.speed
        self.time_spent += 1
        self.pos[1] += math.sin(math.radians(360 -
self.angle)) * self.speed
        if self.pos[1] < 20:
            self.pos[1] = 20
        elif self.pos[1] > screen_height - 120:
            self.pos[1] = screen_height - 120

        # calculate 4 collision points
        self.center = [int(self.pos[0]) + 50,
int(self.pos[1]) + 50]
        len = 40
        left_top = [self.center[0] +
math.cos(math.radians(360 - (self.angle + 30))) * len,
self.center[1] + math.sin(math.radians(360 - (self.angle
+ 30))) * len]
        right_top = [self.center[0] +
math.cos(math.radians(360 - (self.angle + 150))) * len,
self.center[1] + math.sin(math.radians(360 - (self.angle
+ 150))) * len]
        left_bottom = [self.center[0] +
math.cos(math.radians(360 - (self.angle + 210))) * len,
self.center[1] + math.sin(math.radians(360 - (self.angle
+ 210))) * len]
        right_bottom = [self.center[0] +
math.cos(math.radians(360 - (self.angle + 330))) * len,
self.center[1] + math.sin(math.radians(360 - (self.angle
+ 330))) * len]
        self.four_points = [left_top, right_top,
left_bottom, right_bottom]

        self.check_collision(map)
        self.radars.clear()
        for d in range(-90, 120, 45):
```

```
            self.check_radar(d, map)

    def get_data(self):
        radars = self.radars
        ret = [0, 0, 0, 0, 0]
        for i, r in enumerate(radars):
            ret[i] = int(r[1] /5)

        return ret

    def get_alive(self):
        return self.is_alive

    def get_reward(self):
        return self.distance / 50.0

    def rot_center(self, image, angle):
        orig_rect = image.get_rect()
        rot_image = pygame.transform.rotate(image, angle)
        rot_rect = orig_rect.copy()
        rot_rect.center = rot_image.get_rect().center
        rot_image = rot_image.subsurface(rot_rect).copy()
        return rot_image

def run_car(genomes, config):

    # Init NEAT
    nets = []
    cars = []

    for id, g in genomes:
        net = neat.nn.FeedForwardNetwork.create(g,
config)
        nets.append(net)
        g.fitness = 0
        cars.append(Car())

    pygame.init()
    screen = pygame.display.set_mode((screen_width,
screen_height))
    clock = pygame.time.Clock()
    generation_font = pygame.font.SysFont("Fixedsys", 70)
    font = pygame.font.SysFont("Fixedsys", 30)
    map = pygame.image.load('map2.png')
```

```
    global generation
    generation += 1
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit(0)


        for index, car in enumerate(cars):
            output = nets[index].activate(car.get_data())
            i = output.index(max(output))
            if i == 0:
                car.angle += 10
            else:
                car.angle -= 10

        remain_cars = 0
        for i, car in enumerate(cars):
            if car.get_alive():
                remain_cars += 1
                car.update(map)
                genomes[i][1].fitness += car.get_reward()

        if remain_cars == 0:
            break

        screen.blit(map, (0, 0))
        for car in cars:
            if car.get_alive():
                car.draw(screen)

        text = generation_font.render("Generation : " +
str(generation), True, (0, 255, 0))
        text_rect = text.get_rect()
        text_rect.center = (screen_width/2, 100)
        screen.blit(text, text_rect)

        text = font.render("Remaining cars : " +
str(remain_cars), True, (0, 0, 0))
        text_rect = text.get_rect()
        text_rect.center = (screen_width/2, 200)
        screen.blit(text, text_rect)

        pygame.display.flip()
        clock.tick(0)
```

```
if __name__ == "__main__":
    config_path = "./config-feedforward.txt"
    config = neat.config.Config(neat.DefaultGenome,
neat.DefaultReproduction,
                                neat.DefaultSpeciesSet,
neat.DefaultStagnation, config_path)
    p = neat.Population(config)
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.run(run_car, 1000)
```

Here is the text file which stores the parameters used by the network and the program:

[NEAT]
fitness_criterion   = max
fitness_threshold   = 100000
pop_size   = 5
reset_on_extinction   = True

[DefaultGenome]
node activation options
activation_default   = tanh
activation_mutate_rate = 0.01
activation_options   = tanh

node aggregation options
aggregation_default   = sum
aggregation_mutate_rate = 0.01
aggregation_options   = sum

node bias options
bias_init_mean   = 0.0
bias_init_stdev   = 1.0
bias_max_value   = 30.0

```
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 5
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
```

```
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2

[DefaultReproduction]
elitism            = 3
survival_threshold = 0.2
```

CHAPTER 5

SCREENSHOTS AND RESULTS

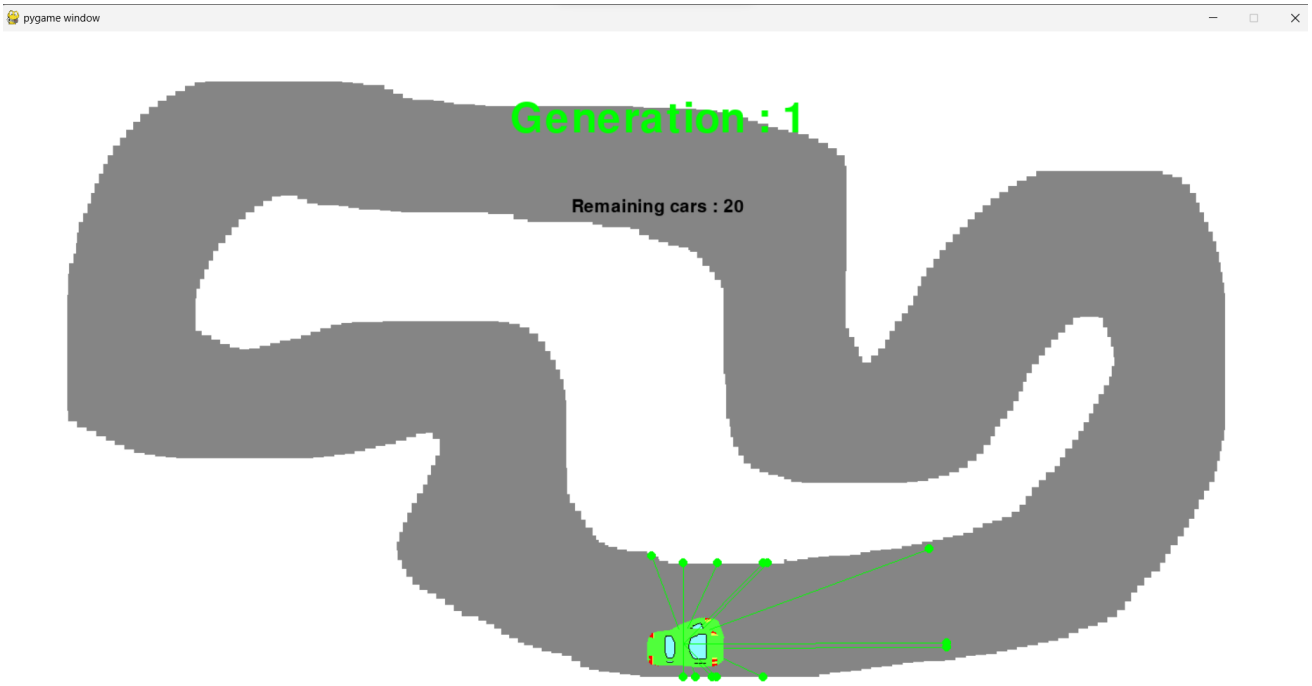

Fig. 5.1 - Start of course

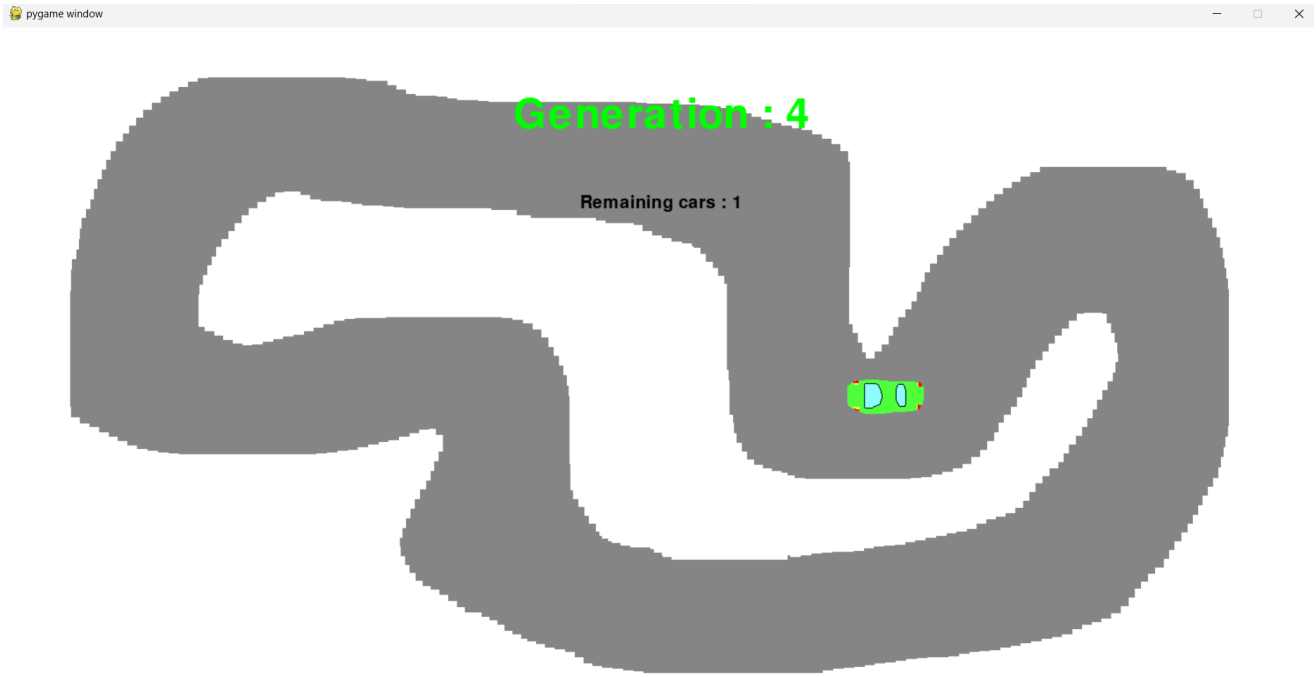

Fig. 5.2 – For Image clarity, the radars are not being rendered on screen

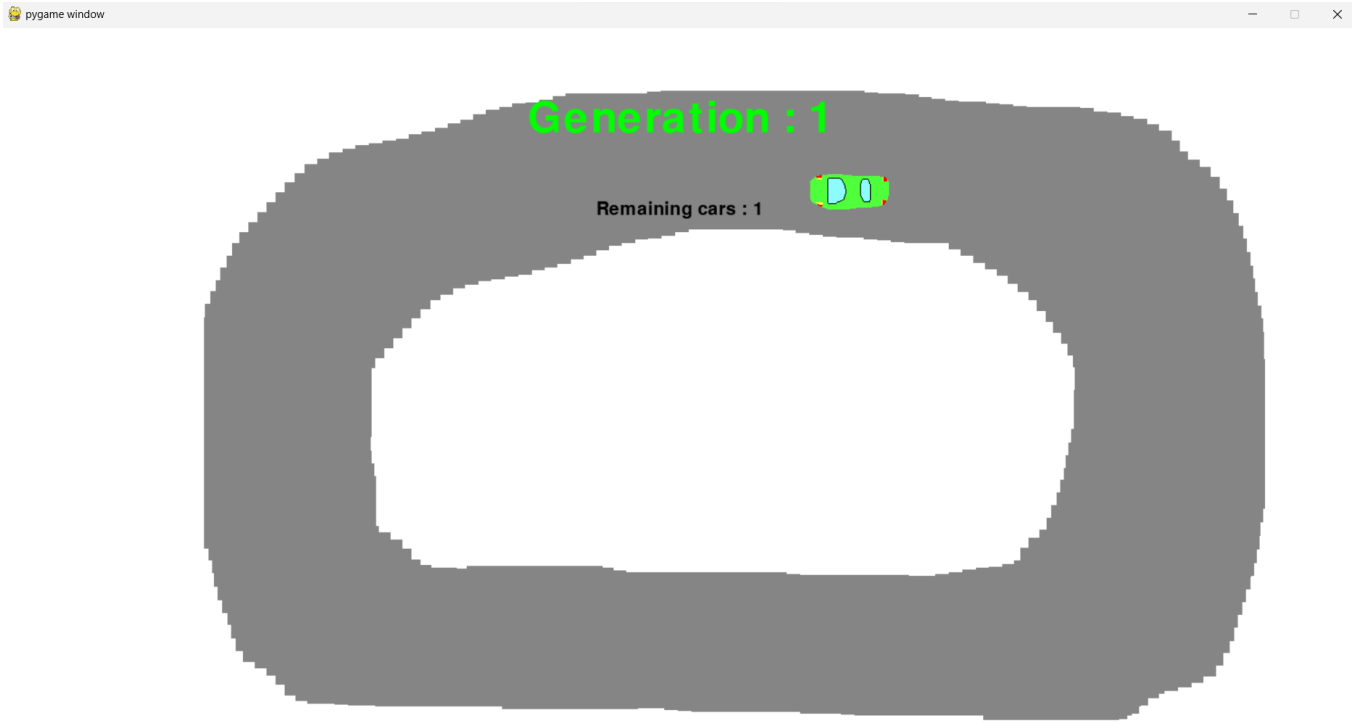

Fig. 5.3 – In simpler tracks, lesser generations are needed.

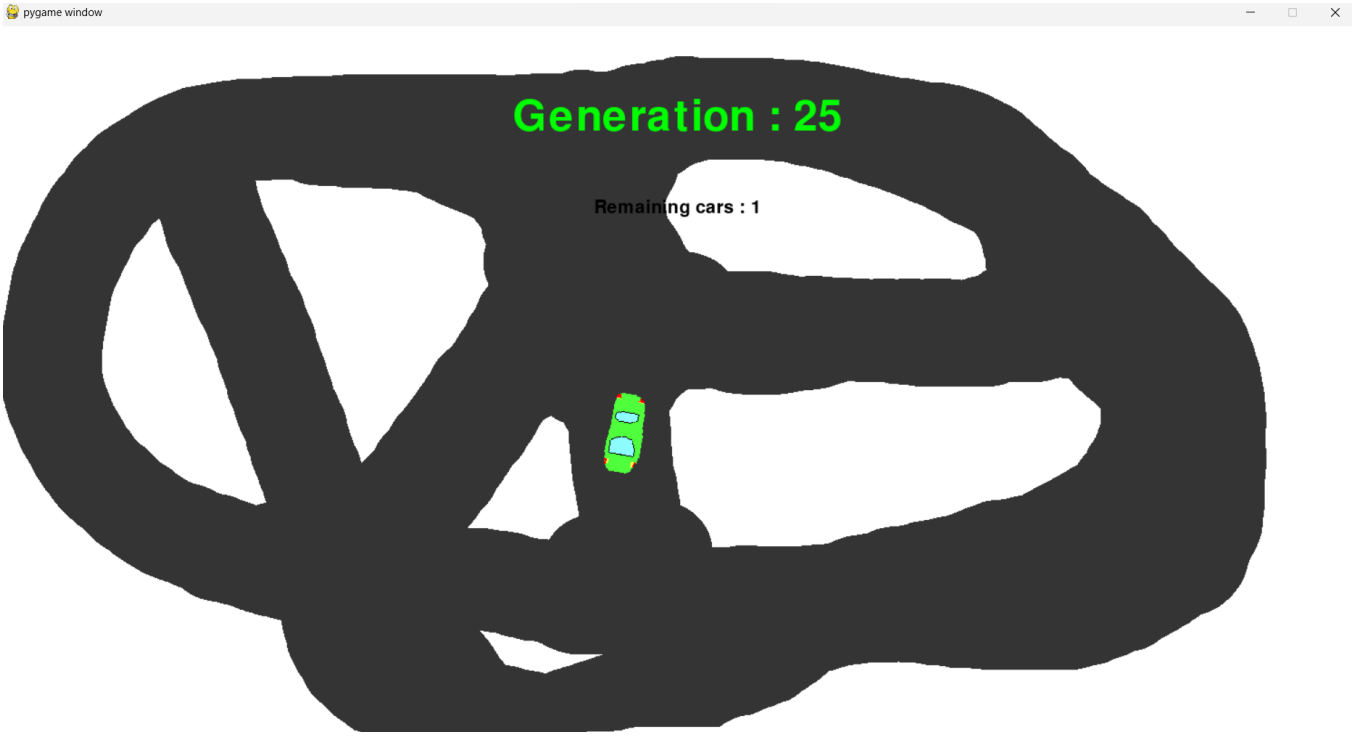

Fig. 5.4 – In more complex tracks, more generations are required.

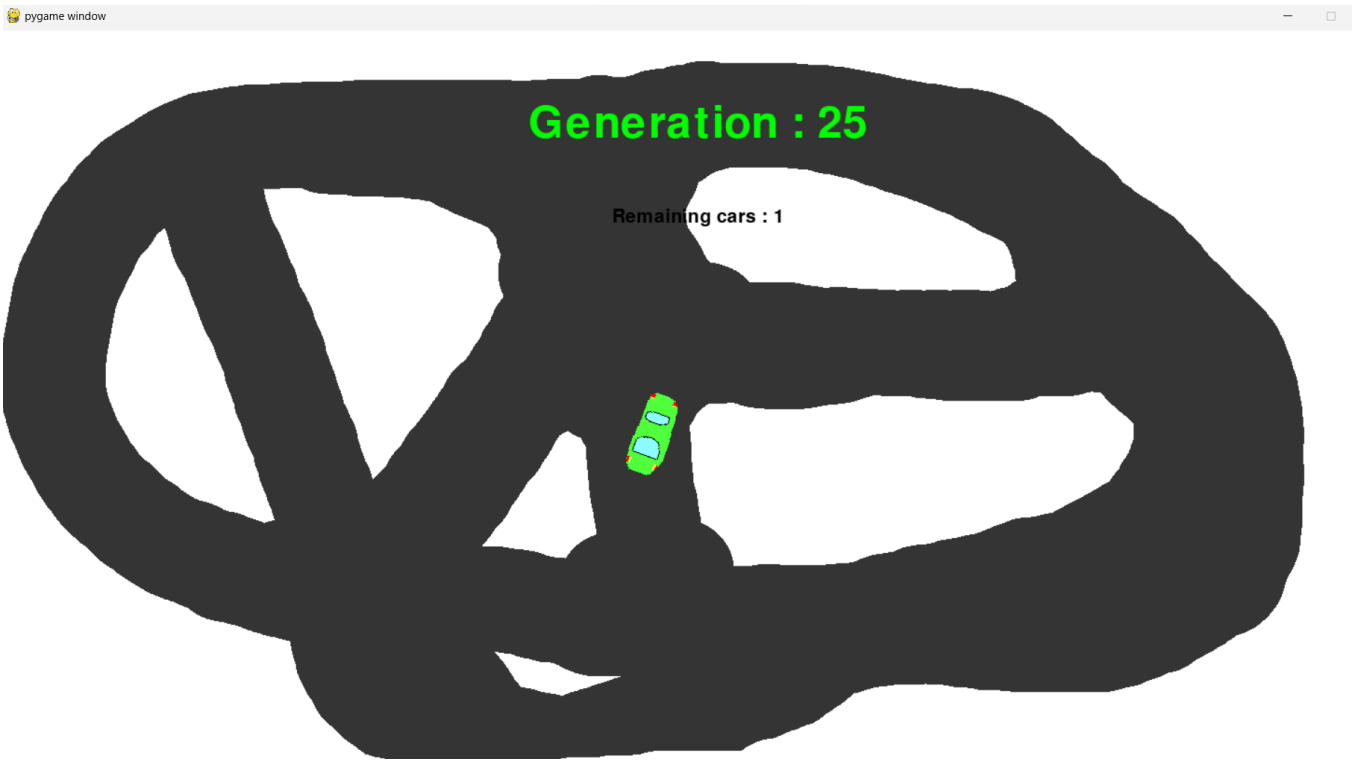

Fig. 5.5 – In such tracks with multiple pathways, the car can get “confused” and keep circling the same area in the track without doing a proper lap.

CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENTS

6.1 Conclusion

In this project report, I have presented an AI program that simulates how a racecar can find the optimal speed and path to complete a course. The program uses computer vision techniques and machine learning algorithms to detect the course layout, identify obstacles, and calculate the optimal path and speed.

The program has shown promising results in accurately detecting the course layout and finding the optimal speed and path for the racecar to follow. The real-time feedback provided by the program allows the user to monitor the racecar's progress and adjust its path and speed if necessary. The program has several potential applications, including autonomous racing technologies, racing simulators, and driver assistance systems. The simulation of a racecar navigating a course provides valuable insights into how autonomous vehicles can safely and efficiently navigate complex environments.

6.2 Future Enhancements

Future work can focus on improving the program's accuracy and performance, as well as expanding its capabilities to handle more complex courses and obstacles. In addition, integrating the program with real-world autonomous driving technologies can lead to significant advancements in the field of self-driving cars.

In conclusion, the AI program presented in this project report demonstrates the potential of machine learning and computer vision in simulating how a racecar can find the optimal speed and path to complete a course, which can be applied to real-world scenarios and benefit society as a whole.

REFERENCES

[1] Q. Ye and D. Doermann, "Text Detection and Recognition in Imagery: A Survey," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 37, no. 7, pp. 1480-1500, 1 July 2015, doi: 10.1109/TPAMI.2014.2366765.

[2] Long, S., He, X. & Yao, C. Scene Text Detection and Recognition: The Deep Learning Era. *Int J Comput Vis* **129**, 161–184 (2021). https://doi.org/10.1007/s11263-020-01369-0

[3] Wiering, Marco A., and Martijn Van Otterlo. "Reinforcement learning." *Adaptation, learning, and optimization* 12.3 (2012): 729.

[4] Peter Dayan, Yael Niv, Reinforcement learning: The Good, The Bad and The Ugly, Current Opinion in Neurobiology, Volume 18, Issue 2, 2008, Pages 185-196, ISSN 0959-4388, https://doi.org/10.1016/j.conb.2008.08.003.

[5] Barto, Andrew G., and Richard S. Sutton. "Reinforcement learning." *Handbook of brain theory and neural networks* (1995): 804-809.

[6] François-Lavet, Vincent, et al. "An introduction to deep reinforcement learning." *Foundations and Trends® in Machine Learning* 11.3-4 (2018): 219-354.

[7] www.youtube.com/@CheesyAI

[8] www.youtube.com/@NeuralNine

[9] https://www.sciencebuddies.org/science-fair-projects/project-ideas/CompSci_p068/computer-science/machine-learning-AWS-DeepRacer

[10] https://en.wikipedia.org/wiki/Self-driving_car