NeuroEvolutionary Car Racing Simulator

The project implements a neural network-based car racing simulation using the NEAT (NeuroEvolution of Augmenting Topologies) algorithm and Pygame for visualization.

Explanation of the key components and algorithms used:

. Libraries and Modules:

- `math`, `random`, `sys`, `os`: Standard libraries for math operations, random number generation, system functions, and file manipulation.
- `pickle`: Used to save and load the best neural network models (genomes) for later use.
- `pygame`: Provides a graphical interface and handles game elements like displaying images, updating screens, and managing events.
- `neat-python`: A neural evolution library implementing NEAT (NeuroEvolution of Augmenting Topologies), which evolves neural networks based on genetic algorithms.

NEAT Algorithm Implementation

The simulation utilizes the NEAT algorithm, which evolves artificial neural networks to control the cars. NEAT is an evolutionary algorithm that simultaneously optimizes both the topology and weights of neural networks.

A **genome** represents the "blueprint" of an individual neural network within the population.

Genome Structure

- The genome is configured in the `config.txt` file, specifying parameters for node activation, aggregation, bias, and connection properties.
- The network has 5 input nodes (corresponding to radar readings) and 4 output nodes (for car control decisions).

Evolutionary Process

1. A population of genomes is initialized based on the configuration.
2. Each genome is evaluated in the simulation environment.
3. Fitness is calculated based on the distance traveled by each car.
4. The population evolves through selection, crossover, and mutation.
5. This process repeats for multiple generations (up to 1000 in this implementation).

Car Simulation

Car Class

The `Car` class represents individual cars in the simulation:

- **Initialization**: Cars are initialized with a sprite, position, angle, and speed.
- **Sensors**: Each car has 5 radar sensors to detect distances to obstacles.
- **Movement**: Cars update their position based on speed and angle, with boundary checks.
- **Collision Detection**: Checks for collisions with the track boundaries.

Physics Simulation

- The car's movement is simulated using basic trigonometry:

    - Position update: $x'=x+\cos(360-\theta)*speed$x′=x+cos(360−θ)∗speed
    - $y'=y+\sin(360-\theta)*speed$y′=y+sin(360−θ)∗speed
- Rotation is handled by Pygame's transform functions.

Neural Network Control

Each car is controlled by a feedforward neural network:

1. **Inputs**: The network receives 5 inputs from the car's radar sensors.
2. **Processing**: The inputs are processed through the evolved network structure.
3. **Outputs**: The network produces 4 outputs, determining the car's action:
    - Turn left
    - Turn right
    - Decrease speed
    - Increase speed

Simulation Loop

The main simulation loop (`run_simulation` function) handles:

1. Initialization of cars and neural networks for each genome.
2. Updating car positions and checking for collisions.
3. Feeding sensor data to neural networks and executing their decisions.
4. Calculating fitness based on distance traveled.
5. Rendering the simulation using Pygame.

Fitness Evaluation

Fitness is calculated as:

fitness=distance/(CARSIZEX/2)
This rewards cars that travel further distances.

Model Persistence

The best genome from each generation is saved using Python's `pickle` module, allowing for later analysis or continued training.

Visualization

Pygame is used to render the simulation:

- The track is loaded from an image file.
- Cars are represented by sprites that rotate based on their angle.
- Radar lines are drawn to visualize the car's sensors.
- Generation and alive car count are displayed on screen.

This implementation combines evolutionary computation, neural networks, and game development techniques to create a self-learning car racing simulation. The NEAT algorithm allows for the evolution of both the structure and parameters of the neural networks, potentially leading to more efficient and adaptable solutions compared to fixed-topology approaches

Python Code:

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

# Constants
WIDTH = 1920
HEIGHT = 1080

CAR_SIZE_X = 60
CAR_SIZE_Y = 60

BORDER_COLOR = (255, 255, 255, 255) # Color To Crash on Hit

current_generation = 0 # Generation counter

class Car:
   def __init__(self):
       # Load Car Sprite and Rotate
       self.sprite = pygame.image.load('car.png').convert() # Convert Speeds Up A Lot
       self.sprite = pygame.transform.scale(self.sprite, (CAR_SIZE_X, CAR_SIZE_Y))
       self.rotated_sprite = self.sprite

       self.position = [830, 920] # Starting Position
       self.angle = 0
       self.speed = 0

       self.speed_set = False # Flag For Default Speed Later on

       self.center = [self.position[0] + CAR_SIZE_X / 2, self.position[1] + CAR_SIZE_Y / 2] # Calculate Center
```

```
        self.radars = [] # List For Sensors / Radars
        self.drawing_radars = [] # Radars To Be Drawn

        self.alive = True # Boolean To Check If Car is Crashed

        self.distance = 0 # Distance Driven
        self.time = 0 # Time Passed

    def draw(self, screen):
        screen.blit(self.rotated_sprite, self.position) # Draw Sprite
        self.draw_radar(screen) # OPTIONAL FOR SENSORS

    def draw_radar(self, screen):
        # Optionally Draw All Sensors / Radars
        for radar in self.radars:
            position = radar[0]
            pygame.draw.line(screen, (0, 255, 0), self.center, position, 1)
            pygame.draw.circle(screen, (0, 255, 0), position, 5)

    def check_collision(self, game_map):
        self.alive = True
        for point in self.corners:
            # If Any Corner Touches Border Color -> Crash
            # Assumes Rectangle
            if game_map.get_at((int(point[0]), int(point[1]))) ==
BORDER_COLOR:
                self.alive = False
                break

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

        while not game_map.get_at((x, y)) == BORDER_COLOR and length < 300:
            length += 1
            x = int(self.center[0] + math.cos(math.radians(360 -
(self.angle + degree))) * length)
```

```
            y = int(self.center[1] + math.sin(math.radians(360 -
(self.angle + degree))) * length)

        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, game_map):
        if not self.speed_set:
            self.speed = 20
            self.speed_set = True

        self.rotated_sprite = self.rotate_center(self.sprite, self.angle)
        self.position[0] += math.cos(math.radians(360 - self.angle)) *
self.speed
        self.position[0] = max(self.position[0], 20)
        self.position[0] = min(self.position[0], WIDTH - 120)

        self.distance += self.speed
        self.time += 1

        self.position[1] += math.sin(math.radians(360 - self.angle)) *
self.speed
        self.position[1] = max(self.position[1], 20)
        self.position[1] = min(self.position[1], HEIGHT - 120)

        self.center = [int(self.position[0]) + CAR_SIZE_X / 2,
int(self.position[1]) + CAR_SIZE_Y / 2]

        length = 0.5 * CAR_SIZE_X
        left_top = [self.center[0] + math.cos(math.radians(360 -
(self.angle + 30))) * length,
                    self.center[1] + math.sin(math.radians(360 -
(self.angle + 30))) * length]
        right_top = [self.center[0] + math.cos(math.radians(360 -
(self.angle + 150))) * length,
                     self.center[1] + math.sin(math.radians(360 -
(self.angle + 150))) * length]
        left_bottom = [self.center[0] + math.cos(math.radians(360 -
(self.angle + 210))) * length,
```

```
                      self.center[1] + math.sin(math.radians(360 -
(self.angle + 210))) * length]
       right_bottom = [self.center[0] + math.cos(math.radians(360 -
(self.angle + 330))) * length,
                       self.center[1] + math.sin(math.radians(360 -
(self.angle + 330))) * length]
       self.corners = [left_top, right_top, left_bottom, right_bottom]

       self.check_collision(game_map)
       self.radars.clear()

       for d in range(-90, 120, 45):
           self.check_radar(d, game_map)

   def get_data(self):
       radars = self.radars
       return_values = [0, 0, 0, 0, 0]
       for i, radar in enumerate(radars):
           return_values[i] = int(radar[1] / 30)
       return return_values

   def is_alive(self):
       return self.alive

   def get_reward(self):
       return self.distance / (CAR_SIZE_X / 2)

   def rotate_center(self, image, angle):
       rectangle = image.get_rect()
       rotated_image = pygame.transform.rotate(image, angle)
       rotated_rectangle = rectangle.copy()
       rotated_rectangle.center = rotated_image.get_rect().center
       rotated_image = rotated_image.subsurface(rotated_rectangle).copy()
       return rotated_image


def run_simulation(genomes, config):
   # Empty Collections For Nets and Cars
   nets = []
   cars = []
```

```python
# Initialize PyGame And The Display
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT), pygame.FULLSCREEN)

# For All Genomes Passed Create A New Neural Network
for i, g in genomes:
    net = neat.nn.FeedForwardNetwork.create(g, config)
    nets.append(net)
    g.fitness = 0
    cars.append(Car())

clock = pygame.time.Clock()
generation_font = pygame.font.SysFont("Arial", 30)
alive_font = pygame.font.SysFont("Arial", 20)
game_map = pygame.image.load('map.png').convert()

global current_generation
current_generation += 1

counter = 0
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit(0)

    for i, car in enumerate(cars):
        output = nets[i].activate(car.get_data())
        choice = output.index(max(output))
        if choice == 0:
            car.angle += 10
        elif choice == 1:
            car.angle -= 10
        elif choice == 2:
            if car.speed - 2 >= 12:
                car.speed -= 2
        else:
            car.speed += 2

    still_alive = 0
```

```
        for i, car in enumerate(cars):
            if car.is_alive():
                still_alive += 1
                car.update(game_map)
                genomes[i][1].fitness += car.get_reward()

        if still_alive == 0:
            break

        counter += 1
        if counter == 30 * 40:
            break

        screen.blit(game_map, (0, 0))
        for car in cars:
            if car.is_alive():
                car.draw(screen)

        text = generation_font.render("Generation: " +
str(current_generation), True, (0,0,0))
        screen.blit(text, text.get_rect(center=(900, 450)))

        text = alive_font.render("Still Alive: " + str(still_alive), True,
(0, 0, 0))
        screen.blit(text, text.get_rect(center=(900, 490)))

        pygame.display.flip()
        clock.tick(60)

    best_genome = max(genomes, key=lambda g: g[1].fitness)
    with open(f"best_genome_gen_{current_generation}.pkl", "wb") as f:
        pickle.dump(best_genome[1], f)


if __name__ == "__main__":
    config_path = "./config.txt"
    config = neat.config.Config(neat.DefaultGenome,
neat.DefaultReproduction,
                                neat.DefaultSpeciesSet,
neat.DefaultStagnation, config_path)
```

```
population = neat.Population(config)
population.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
population.add_reporter(stats)

population.run(run_simulation, 1000)
```

Config.txt Code:

```
[NEAT]
fitness_criterion   = max
fitness_threshold         = 100000000
pop_size            = 30
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   = 4

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 = 2.0

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

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

Sample Map 1: