Project Report

On

"Self-Driving AI Car Simulation"

Submitted in partial fulfilment of requirements for the award of the degree

Bachelor of technology
in
Information Technology

to
IKG Punjab Technical University, Jalandhar

Submitted By:

SHUBH SARPAL 2003368

RAVINDER BISHNOI 2003361

ADITYA PANDEY 2002661

Submitted to:

Department of Information Technology

Chandigarh Engineering College

Landran, Mohali, Punjab, 140307

ACKNOWLEDGEMENT

Any achievement big or small should have a catalyst and constant encouragement and advice of valuable and noble minds. The satisfaction and euphoria that accompanies the successful completion of any task would be incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crowned our efforts with success.

We would like to thank **Mrs. Harpreet Kaur**
for constantly motivating us and guiding us.

And **Cheesy AI** for giving us this excellent idea of self driving cars simulation

Shubh Sarpal
Ravinder Bishnoi
Aditya Pandey

ABSTRACT

For the past decade, there has been a surge of interest in self-driving cars. This is due to breakthroughs in the field of deep learning where deep neural networks are trained to perform tasks that typically require human intervention. These cars learn driving using NEAT, which is a reinforcement learning technique.

This simulator helps us to create an easy model to act as a pivot for upcoming AI based solutions for Self Driving cars.

CONTENTS

LANGUAGE

Python Basic Theory

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

Python is one of those rare languages which can claim to be both simple and powerful. You will find yourself pleasantly surprised to see how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.

The official introduction to Python is:

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Features in Python

There are many features in Python, some of which are discussed below as follows:

1. Easy to code

Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in the Python language and anybody can learn Python basics in a few hours or days. It is also a developer-friendly language.

2. Easy to read

As you will see, learning Python is quite simple. As was already established, Python's syntax is really straightforward. The code block is defined by the indentations rather than by semicolons or brackets.

3. Object-Oriented Language

One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.

4. GUI Programming Support

Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python. PyQt5 is the most popular option for creating graphical apps with Python.

5. High-Level Language

Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.

6. Extensible feature

Python is an **Extensible** language. We can write some Python code into C or C++ language and also we can compile that code in C/C++ language.

7. Easy to debug

Excellent information for mistake tracing. You will be able to quickly identify and correct the majority of your program's issues once you understand how to interpret Python's error traces. Simply by glancing at the code, you can determine what it is designed to perform.

8. Python is a Portable language

Python language is also a portable language. For example, if we have Python code for windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

9. Python is an Integrated language

Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.

10. Interpreted Language:

Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code this makes it easier to debug our code. The source code of Python is converted into an immediate form called **bytecode**.

11. Large Standard Library

Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing. There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.

12. Dynamically Typed Language

Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

13. Frontend and backend development

With a new project py script, you can run and write Python codes in HTML with the help of some simple tags <py-script>, <py-env>, etc. This will help you do frontend development work in Python like javascript. Backend is the strong forte of Python it's extensively used for this work cause of its frameworks like Django and Flask.

14. Allocating Memory Dynamically

In Python, the variable data type does not need to be specified. The memory is automatically allocated to a variable at runtime when it is given a value. Developers do not need to write int y = 18 if the integer value 15 is set to y. You may just type y=18.

Artificial Intelligence (AI)

According to the father of Artificial Intelligence, John McCarthy, it is "The science and engineering of making intelligent machines, especially intelligent computer programs".

Artificial intelligence (AI) refers to the simulation of human intelligence in machines that are programmed to think like humans and mimic their actions. The term may also be applied to any machine that exhibits traits associated with a human mind such as learning and problem-solving.

When most people hear the term artificial intelligence, the first thing they usually think of is robots. That's because big-budget films and novels weave stories about human-like machines that wreak havoc on Earth. But nothing could be further from the truth.

Artificial intelligence is based on the principle that human intelligence can be defined in a way that a machine can easily mimic it and execute tasks, from the most simple to those that are even more complex. The goals of artificial intelligence include mimicking human cognitive activity. Researchers and developers in the field are making surprisingly rapid strides in mimicking activities such as learning, reasoning, and perception, to the extent that these can be concretely defined. Some believe that innovators may soon be able to develop systems that exceed the capacity of humans to learn or reason out any subject. But others remain skeptical because all cognitive activity is laced with value judgments that are subject to human experience.

The Necessity of AI

As we know that AI pursues creating the machines as intelligent as human beings. There are numerous reasons for us to study AI. The reasons are as follows −

AI can learn through data

In our daily life, we deal with huge amount of data and human brain cannot keep track of so much data. That is why we need to automate the things. For doing automation, we need to study AI because it can learn from data and can do the repetitive tasks with accuracy and without tiredness.

AI can teach itself

It is very necessary that a system should teach itself because the data itself keeps changing and the knowledge which is derived from such data must be updated constantly. We can use AI to fulfill this purpose because an AI enabled system can teach itself.

AI can respond in real time

Artificial intelligence with the help of neural networks can analyze the data more deeply. Due to this capability, AI can think and respond to the situations which are based on the conditions in real time.

AI achieves accuracy

With the help of deep neural networks, AI can achieve tremendous accuracy. AI helps in the field of medicine to diagnose diseases such as cancer from the MRIs of patients.

AI can organize data to get most out of it

The data is an intellectual property for the systems which are using self-learning algorithms. We need AI to index and organize the data in a way that it always gives the best results.

Understanding Intelligence

With AI, smart systems can be built. We need to understand the concept of intelligence so that our brain can construct another intelligence system like itself.

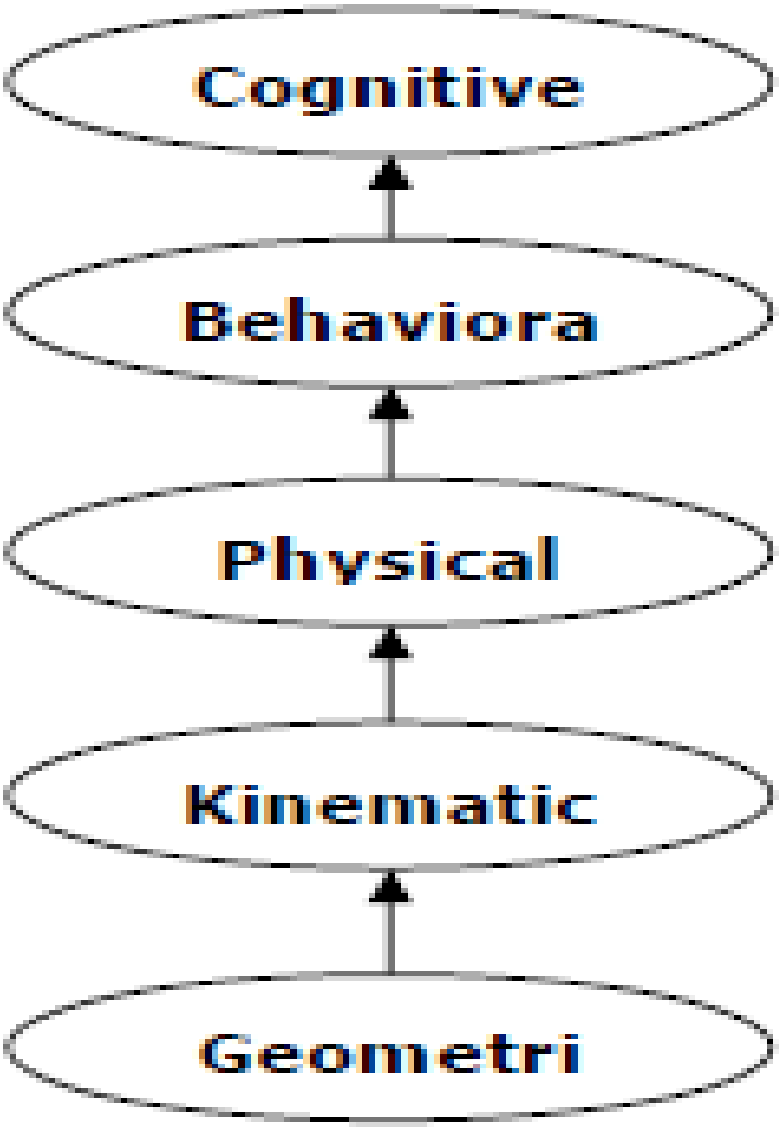
Applications of Artificial Intelligence:

The applications for artificial intelligence are endless. The technology can be applied to many different sectors and industries. AI is being tested and used in the healthcare industry for dosing drugs and doling out different treatments tailored to specific patients, and for aiding in surgical procedures in the operating room.

Artificial intelligence also has applications in the financial industry, where it is used to detect and flag activity in banking and finance such as unusual debit card usage and large account deposits—all of which help a bank's fraud department. Applications for AI are also being used to help streamline and make trading easier. This is done by making supply, demand, and pricing of securities easier to estimate.

Cognitive Modeling: Simulating Human Thinking Procedure

Cognitive modeling is basically the field of study within computer science that deals with the study and simulating the thinking process of human beings. The main task of AI is to make machine think like human. The most important feature of human thinking process is problem solving. That is why more or less cognitive modeling tries to understand how humans can solve the problems. After that this model can be used for various AI applications such as machine learning, robotics, natural language processing, etc. Following is the diagram of different thinking levels of human brain −


Cognitive
Behaviora
Physical
Kinematic
Geometri

Project Overview

Machine learning is awesome and one of the most exciting things that we can do with artificial intelligence on our computers is running AI simulations. We are going to simulate the evolution of self-driving two dimensional cars in python. For this we will use a technique called neural evolution of augmenting topologies (NEAT). The basic maps are used which can be drawn with simple tools like paint. To challenge our models we can draw some crazy maps. Our cars have five sensors that look out for the borders those are the five input neurons of our neural network. This is the only thing that our AI actually sees. Also our model has four output neurons that represent the four actions it can take. Those are:

- Steering left
- Steering right
- Speeding up
- Slowing down

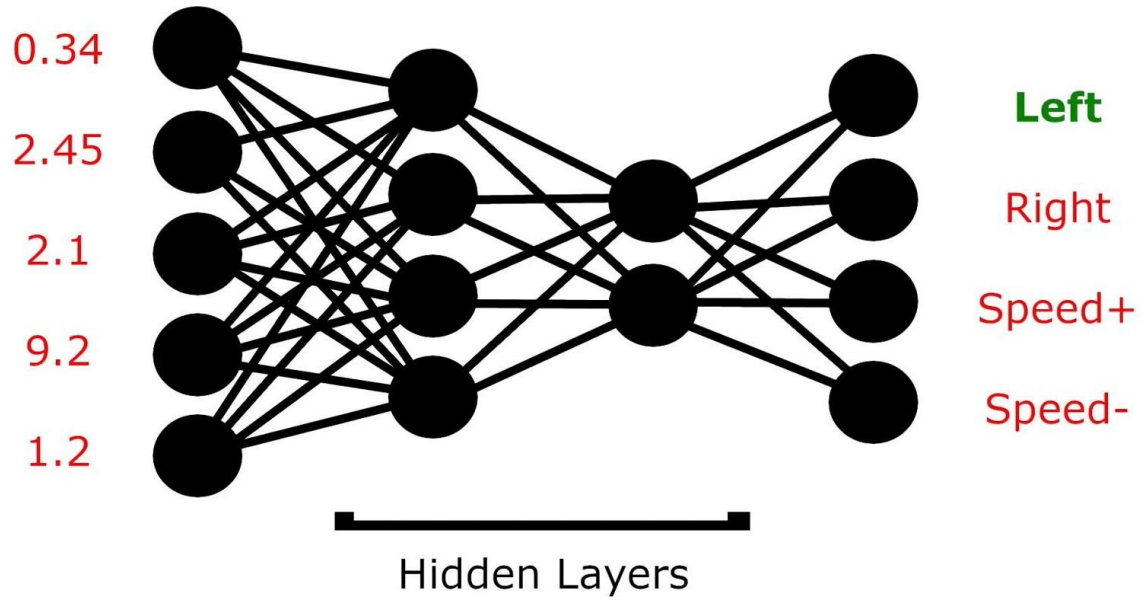
The project consists of two major parts:

1. Python file
2. Neat config file

In need config file we can specify all sorts of parameters which will influence the things like reproduction, mutations, amount of species etc

Our model is neural network with five input neurons, the sensors and four output neurons the actions it can take.

In between those there can optionally have some hidden layers with additional neurons. Those layers increase the complexity and sophistication of four model but they also increase the training time and the likelihood of overfitting. All of those neurons are interconnected and those connections have certain weights. Depending on all those values our model will react ina certain way based on the input. In beginning all those reactions will be totally random, there will be zero intelligence behind what our cars are doing. However for each action our cars will either receive a reward or a penality.



Implementation

To implement this fitness metric is used. In our simple simulation the fitness of a car increases depending on the distance it covers without crashing.

After each generation evolution of cars is done, the car with the highest fitness value will probably survive and reproduce whereas the cars that didn't perform so well will go extinct after a while.

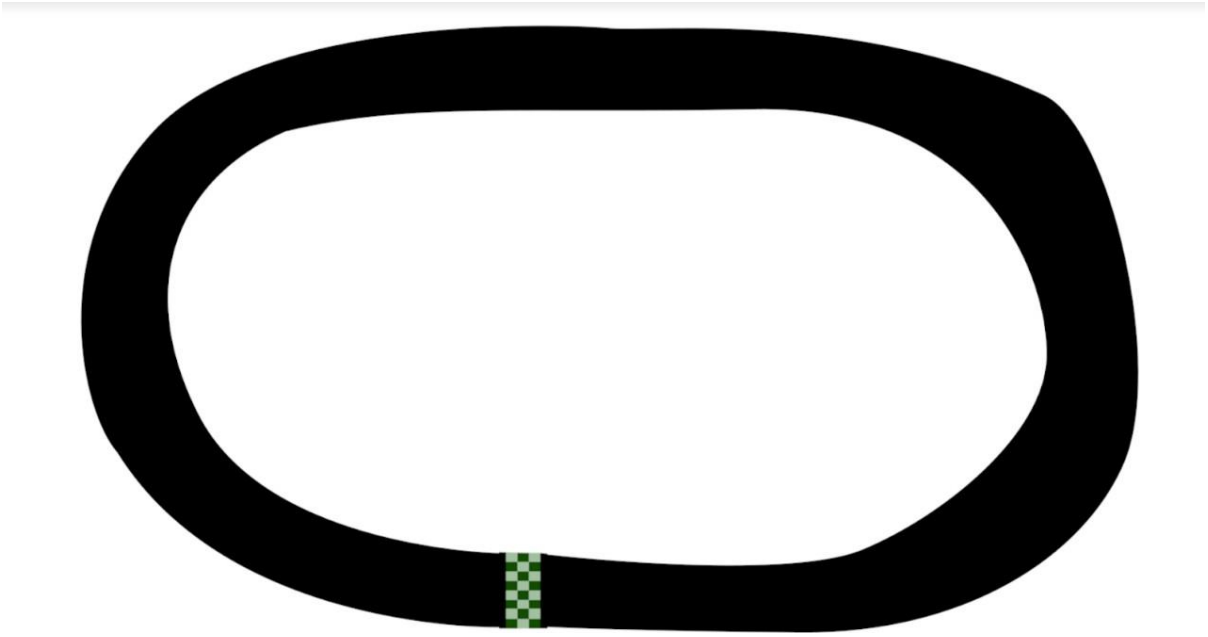
When a car reproduces it will not just duplicate the child car and will be quite similar to its parent but not the same. Therefore it has a chance to become better. Cars that are very similar to each other form an owned species. If a species doesn't see any improvements for a fixed number of generations it will go extinct.

Working

- **MAP # 1**

  Level : Easy

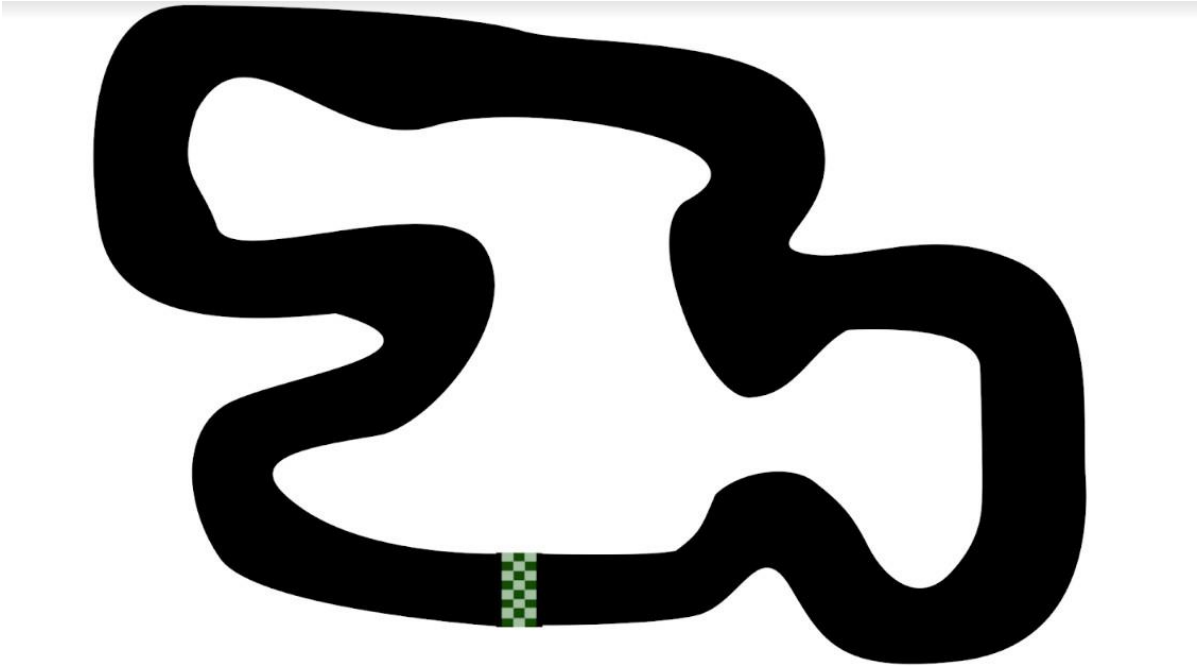
The basic map possible is just a ring going full circle.



Let the population size is of 30 and by mere chance we already have a card that drives quite well in the first generation. The only thing the car has to do is steer left in the right moment and then it survives. In the next generation the car is reproduced and there are two cars that were able to drive and not crash for few seconds. One of the two cars is also way faster than the other one. The fitness value increases harder the more distance a car covers in those few seconds. The cars have the ability to accelerate, therefore if a car figures out that increasing the speed in the right moment leads to better results we might see a new alpha species here. In generation four let there are four of those fast driving cars . By generation five most of the cars driving fast and good attempt will be seen by speeding the racing course.

- **MAP # 2**

  Level: Less Easy



In this case every car crashes right in the beginning of generation one. This is little bit harder than the map one. In generation two there are promising attempts but still all the cars crash and even though not a single car succeeds in the next couple of generations. The quote unquote best cars reproduce and there are good attempts over time. By generation seven few of the cars succeeding at going full circle, one of the cars goes fast and the other tries carefully through the racetrack. By this guessing can be done which one will persist and reproduce over time. By the twelfth generations there are not too many differences between the two types of cars. Even though the fast cars have a better fitness value, the likelihood of a mutated version surviving at that speed is less likely than that of slower cars. This might definitely change after a couple of more generations.

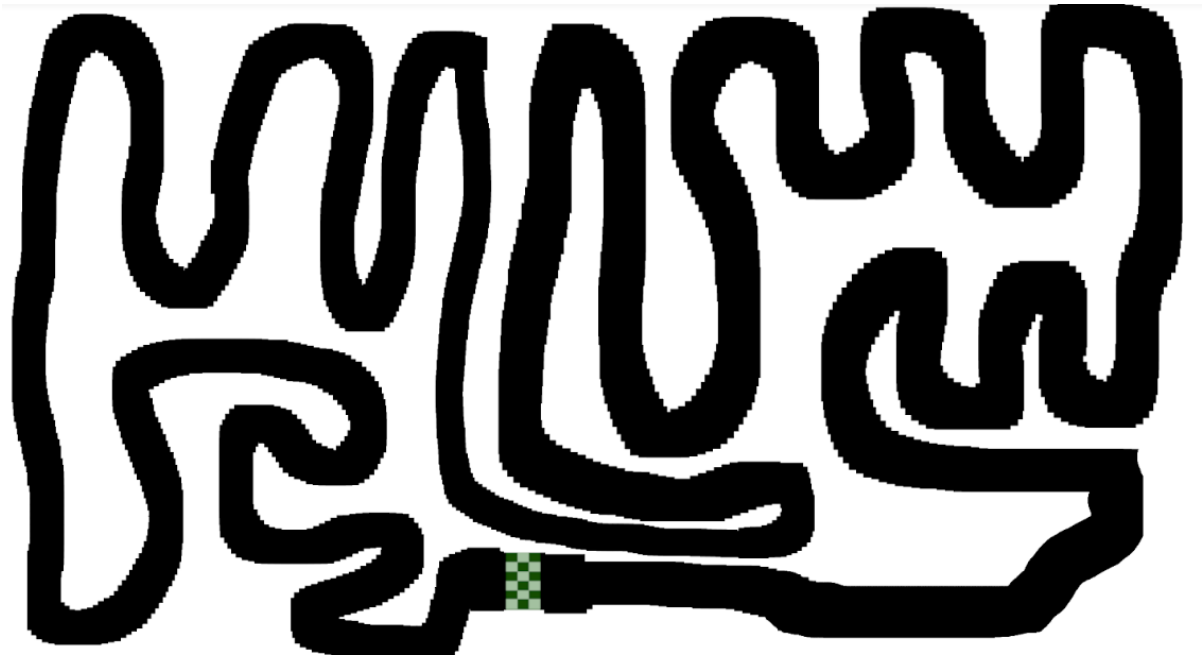
- **MAP # 3**

  Level: Medium

The given map is more challenging than the others. There is no any meaningful improvements until generations. It can crash soon after passing the starting curves. When it tries to reproduce even slight changes in the offspring lead to crash. By generations super child car is seen and this car does not only overcome the u-turn, its parent failed add multiple times. It can even pass the full circuit in the first try. By the generations the number of cars can be increased or decreased. This is done by a parameter called elitism. This parameter decides how many of the best cars will survive 100%. In simulation this number is set to be two, so it guarantees that the best two cars will survive for sure and other cars might die without reproducing even if their results are quite good. By choosing that settings decision can be taken that only the best can survive and reproduce.

- **MAP # 4**

  Level: Crazy Hard



In this map the thing is there is also some additional pretty narrow passages. This simulation is started with the same parameters and settings as before and it failed miserably. The simulation is run hours and hours but there didn't seem to manage to pass through the whole track. There is significant advancement and there is even passed the narrow parts but in the end they all crashed before reaching the finishing line. In this hidden layer is added to increase the complexity. The performance is worse in this case with a hidden layer. Instead of increasing the complexity of the neural network, choices are simplified. So the hidden layer is removed with two of the output neurons. And the two output neurons were responsible for increasing and decreasing the speed. Now the car will decide to drive at a constant speed. So the manual choice is made to focus on the steering

System Requirements:

- The user should have the appropriate version of windows.
- The system should have up to 4 GB ram minimum requirement for the application.
- Internet Connectivity is a must for this purpose.
- CPU with a speed of 1.3Ghz is a good choice for normal usage.

Hardware Requirements:

Hardware is the physical part of the computer system like mouse, keyboard, monitor etc.

- 400 MB Hard disk space.
- 1280 x 800 minimum screen resolution.
- Keyboard and mouse.
- Intel i3/i5 processor.

Software requirements:

Software is a set of applications which is used to run the operating system.

- Operating System : Windows 10

Tools & Techniques used:

❖ **Visual Studio Code:** Visual Studio Code is a source-code editor made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add additional functionality. Visual Studio Code was first announced on April 29, 2015, by Microsoft at the 2015 Build conference. A preview build was released shortly thereafter. On November 18, 2015, the source of Visual Studio Code was released under the MIT License, and made available on GitHub. Extension support was also announced. On April 14, 2016, Visual Studio Code graduated from the public preview stage and was released to the Web. Microsoft has released most of Visual Studio Code's source code on GitHub under the permissive MIT License, while the releases by Microsoft are proprietary freeware.

Implementation

Program

Below is the code for the config text

```
[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
```

Below is the python code for the newcar:

```
import math
import random
import sys
import os
```

```
import neat
import pygame

# Constants
# WIDTH = 1600
# HEIGHT = 880

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 = [690, 740] # Starting Position
        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 We Don't Hit BORDER_COLOR AND length < 300 (just a max) -> go
further and further
        while not game_map.get_at((x, y)) == BORDER_COLOR and length < 300:
            length = 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)

        # Calculate Distance To Border And Append To Radars List
        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):
        # Set The Speed To 20 For The First Time
        # Only When Having 4 Output Nodes With Speed Up and Down
        if not self.speed_set:
            self.speed = 20
            self.speed_set = True

        # Get Rotated Sprite And Move Into The Right X-Direction
        # Don't Let The Car Go Closer Than 20px To The Edge
```

```
        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)

        # Increase Distance and Time
        self.distance += self.speed
        self.time += 1

        # Same For Y-Position
        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], WIDTH - 120)

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

        # Calculate Four Corners
        # Length Is Half The Side
        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]

        # Check Collisions And Clear Radars
        self.check_collision(game_map)
        self.radars.clear()

        # From -90 To 120 With Step-Size 45 Check Radar
        for d in range(-90, 120, 45):
            self.check_radar(d, game_map)

    def get_data(self):
        # Get Distances To Border
        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):
        # Basic Alive Function
        return self.alive

    def get_reward(self):
        # Calculate Reward (Maybe Change?)
        # return self.distance / 50.0
        return self.distance / (CAR_SIZE_X / 2)

    def rotate_center(self, image, angle):
        # Rotate The Rectangle
        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 = []

    # 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 Settings
    # Font Settings & Loading Map
    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() # Convert Speeds Up A
Lot

    global current_generation
    current_generation += 1

    # Simple Counter To Roughly Limit Time (Not Good Practice)
    counter = 0

    while True:
        # Exit On Quit Event
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit(0)

        # For Each Car Get The Acton It Takes
        for i, car in enumerate(cars):
            output = nets[i].activate(car.get_data())
            choice = output.index(max(output))
            if choice == 0:
                car.angle += 10 # Left
            elif choice == 1:
                car.angle -= 10 # Right
            elif choice == 2:
                if(car.speed - 2 >= 12):
                    car.speed -= 2 # Slow Down
            else:
                car.speed += 2 # Speed Up

        # Check If Car Is Still Alive
        # Increase Fitness If Yes And Break Loop If Not
        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: # Stop After About 20 Seconds
            break

        # Draw Map And All Cars That Are Alive
        screen.blit(game_map, (0, 0))
        for car in cars:
```

```
            if car.is_alive():
                car.draw(screen)

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

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

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

if__name__== "__main__":

    # Load Config
    config_path = "./config.txt"
    config = neat.config.Config(neat.DefaultGenome,
                                neat.DefaultReproduction,
                                neat.DefaultSpeciesSet,
                                neat.DefaultStagnation,
                                config_path)

    # Create Population And Add Reporters
    population = neat.Population(config)
    population.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    population.add_reporter(stats)

    # Run Simulation For A Maximum of 1000 Generations
    population.run(run_simulation, 1000)
```

Conclusion

We could go on and on and create more ridiculous maps over time and get the most efficient ones to reproduce more generations over time.