PROJECT: RACE CAR

Name: Khevna Vadaliya

Banner ID: 001395699

Course: Artificial Intelligence - 01 (CSCI - 4740-01)

Goal: *To build an intelligent agent that finishes the race on 8 different types of race tracks without crashing, and by taking the least time and collecting maximum rewards*

INITIAL SETUP

Date: June 5, 2025

```
import random

class Agent:

    def chooseAction(self, observations, possibleActions):
        far_left, near_left, center, near_right, far_right =
observations['lidar']
        vel = observations['velocity']

        # STEERING LOGIC
        error_combined = (far_left + near_left) - (far_right + near_right)
        error_near = near_left - near_right

        # Weighted sum: a for combined, b for near
        a = 0.3
        b = 0.7
        error = a * error_combined + b * error_near

        # Dead-zone Threshold (Smaller Threshold)
        c = 0.1

        if error > c:
```

```
            direction = 'left'
        elif error < -c:
            direction = 'right'
        else:
            direction = 'straight'

        # SPEED LOGIC
        if vel < 0.2:
            # Starting Acceleration
            action_speed = 'accelerate'
        else:
            # Dynamic Brake Thresholds
            base_brake = 4.0
            vel_factor = 8.0
            brake_threshold = base_brake + vel * vel_factor
            side_threshold = brake_threshold / 2.0

            # BRACKING AND ACCELERATING LOGIC
            if center < brake_threshold or near_left < side_threshold or
near_right < side_threshold:
                action_speed = 'brake'
            else:
                if vel < 1.0:
                    action_speed = 'accelerate'
                else:
                    action_speed = 'coast'

        return (direction, action_speed)

    def load(self, data=None):
        pass
```

Logic Behind The Code: This is my initial code. This code tells the agent where to go (that is, left, right or straight) and how fast to move (that is to accelerate, brake or coast).

The car has five LIDAR sensors for different directions which are far left, near left, center, near right, and far right.

The car will compare the left and right lidar, if there is more space on the right, it will turn right and vice versa. If the value returns 0, that is, left space is equal to the right space, it will go straight. I have implemented the error formula to ignore the small difference and to stabilize it on

the track. It helps the agent to avoid wiggling too much on the track and make sure that it stays at the center most of the time.

If the agent is going at a velocity less than 0.2, as per the condition, it will start accelerating. Now, once it accelerates:

- If the far left and far right lidar returns a smaller value, it will hit the brakes.
- If the track is clear and it is moving with a velocity of less than 1, it will accelerate.
- If the track is clear and it is moving with a velocity of more than 1, it will coast.

Thus, I decided to go with this simple logic, which overall performs pretty well.

Readings:

Track	Rewards
1	125.15
2	125.65
3	124.93
4	124.85
5	124.37
6	122.86
7	129.06
8	124.00

The only mistake I made was not taking an average of at least 100 iterations for each track. The above reading was the first output I got and I uploaded the code to my GitHub repo.

ITERATION LOG 1

Date: June 6, 2025

```
import random

class Agent:

    def chooseAction(self, observations, possibleActions):
        lidar = observations['lidar']
        velocity = observations['velocity']

        lidar = [d if d != float('inf') else 100.0 for d in lidar]
        far_left, left, center, right, far_right = lidar

        # Steering Logic
        side_diff = (far_left + left) - (far_right + right)
        near_diff = left - right

        # Weighted sum
        a = 0.6 # for side diff
        b = 0.4 # for near diff
        error = 0.6 * side_diff + 0.4 * near_diff

        # Small Threshold
        c = 0.1

        if error > c:
            direction = 'left'
        elif error < -c:
            direction = 'right'
        else:
            direction = 'straight'

        min_front = min(left, center, right)
        min_side = min(far_left, far_right)
        curvature = (left + right) - (far_left + far_right)

        # Apply brake only if car is very close to crashing into side
lanes
        if center < 0.06 or min_front < 0.05:
```

```
            speed_action = 'brake'

        # If the curve is sharp and the car is fast => only coast
        elif (min_front < 0.2 or abs(error) > 1.8) and velocity > 0.12:
            speed_action = 'coast'

        # If the car is slow => accelerate
        elif velocity < 0.2:
            speed_action = 'accelerate'

        # Always accelerate when there is clear straight track
        else:
            speed_action = 'accelerate'

        return (direction, speed_action)
```

Comparison between Log 1 and Initial Code:

1. **Lidar:** The only difference between the Log 1 code and the initial code is the addition of lidar = [d if d != float('inf') else 100.0 for d in lidar] to convert the infinite floating values to 100 to avoid calculation mishaps.
2. **Steering Logic:** I tried many different values for a and b. At the end, a=0.6 and b=0.4 gave the best rewards as they emphasized the far left and far right lidar sensors.
3. **Speeding Logic:** I removed the dynamic thresholds and added the curvature variable.
4. **Curvature:** This variable checks for the sharp turns and the side lanes.
5. **Coasting Conditions:** Added abs(error) threshold (> 1.8) to identify quick turnings, it will return coast condition and car will not accelerate further.
6. **Braking Logic:** Agent will focus on center lidar, if it is too close to lanes, it will apply emergency brake and quickly turns left or right.

What Issues did I face?:

While trying to fine-tune things like the speed and steering weights, I kept changing values expecting the car's behavior to improve—but nothing worked. Turns out, the real issue wasn't the numbers at all. I had placed my `if-elif-else` conditions in the wrong order, so the code was skipping the important checks. Once I fixed the sequence, everything finally started working.

Initially, there was no success in race track 8. I spent about 4 to 5 hours working on numbers that I can increase or decrease and what is the exact highest value that would give me the best rewards.

Readings:

Track	Rewards
1	118.92
2	118.38
3	117.23
4	118.56
5	116.99
6	116.23
7	102.34
8	103.78

ITERATION LOG 2

Date: June 7, 2025 and June 8, 2025

```
 import random

class Agent:
    def chooseAction(self, observations, possibleActions):
        lidar = observations['lidar']
        velocity = observations['velocity']

        # Replacing infinity distance factor with a large static number
that is 100 for easier calculations
        lidar = [d if d != float('inf') else 100.0 for d in lidar]
        far_left, left, center, right, far_right = lidar

        side_diff = (far_left + left) - (far_right + right)
        near_diff = left - right

        # Weight Errors for side diff and near diff
        a = 0.7
        b = 0.3
        error = a * side_diff + b * near_diff

        # Controls for deciding which action to take
        if error > 0.05:
            direction = 'left'
        elif error < -0.05:
            direction = 'right'
        else:
            direction = 'straight'

        # Speeding Logic with breakdows of path
        min_front = min(left, center, right)
        min_side = min(far_left, far_right)
        curvature = (left + right) - (far_left + far_right)

        safe_velocity = 0.12

        # Condition for braking
        if center < 0.12 or min_front < 0.12 or min_side < 0.12:
```

```
            speed_action = 'brake'

        # Sharp Turn within safe velocity => coast
        elif (min_front < 0.25 or abs(error) > 2.2) and velocity > 0.08:
            speed_action = 'coast'

        # If car is very slow than the safe velocity => accelerate
        elif velocity < 0.05 and center > 0.2 and min_side > 0.2:
            speed_action = 'accelerate'

        # If car is moving above the safe velocity => coast
        elif velocity >= safe_velocity:
            speed_action = 'coast'

        # Condition other than that, agent always needs to accelerate
        else:
            speed_action = 'accelerate'

        return (direction, speed_action)
```

Comparison between Log 2 and Log 1:

1. **Braking logic:** For this Log 2 code, I included Lidar values for braking logic to avoid crashing into side lanes.
2. **Sharper Curve Detection:** After lot of modifications in values, I finally increased the error threshold to 2.2. It will help to detect the sharp turnings.
3. **Defined Safe Velocity Threshold:** I decided to introduce the concept of a safe velocity variable to improve clarity and to give better control over acceleration and coasting decisions.
4. **More Conservative Acceleration:** Log 2 accelerates only when the path is safe (center and sides > 0.2), avoiding crashes.
5. **Refined Steering Sensitivity:** With a smaller steering threshold (b) and a higher weight on side diff (a), Log 2 responds faster to path shifts.

What Issues did I face?:

This code was far better than Log 1. However, I was still struggling with low reward values for tracks 7 and 8. For the initial code and log 1 code, I completely forgot to take an average value for 100 runs which I rectified in this version. It showed me relatively poor performance in tracks 7 and 8 in comparison to the circular tracks.

Readings:

Track Number	Average Score (out of 100 runs)
Track 1	118.958152712433
Track 2	119.021203468410
Track 3	120.682765918334
Track 4	120.628004340022
Track 5	120.590710126357
Track 6	119.480346559646
Track 7	102.510962912634
Track 8	104.916742863920

ITERATION LOG 3 - FINAL VERSION

Date: June 9, 2025

```
import random

class Agent:

    def chooseAction(self, observations, possibleActions):
        far_left, left, center, right, far_right = observations['lidar']
        vel = observations['velocity']

        # Steering Logic
        error_combined = (far_left + left) - (far_right + right)
        error_near = left - right

        a = 0.8
        b = 0.15
        error = a * error_combined + b * error_near

        c = 0.05
        if error > c:
            direction = 'left'
        elif error < -c:
            direction = 'right'
        else:
            direction = 'straight'

        # For Track 8
        spiral_turn = (center < 0.3 and left < 0.4 and right < 0.4) or
(center < 0.25)
        distance = min(center, left, right) < 0.15

        # Speeding Logic
        if vel < 0.15:
            action_speed = 'accelerate'
        elif distance:
            action_speed = 'brake'
        elif spiral_turn:
            if vel > 0.17:
                action_speed = 'brake'
```

```
            else:
                action_speed = 'coast'
        else:
            base_brake = 3.0
            vel_factor = 7.5
            brake_threshold = base_brake + vel * vel_factor
            side_threshold = brake_threshold / 2.2

            if center < brake_threshold or left < side_threshold or right
< side_threshold:
                action_speed = 'brake'
            elif vel < 1.0:
                action_speed = 'accelerate'
            else:
                action_speed = 'coast'

        return (direction, action_speed)

    def load(self, data=None):
        pass
```

Logic Behind this Code:
I was getting more and more confused with the different versions of the same code. I had made it more complex and it was getting difficult to get more ideas on how to improve the performance of my agent. So, I went back to the original code, took the average for 100 runs and started again with different ideas. This code is very similar to the initial code. However, I decided to apply a separate logic for track 8 so that agent can detect the spiral track and controls its speed accordingly. Secondly, I heavily relied on the if-elif-else loop for this code. I jot down too many conditions so that car avoids crashing every time but sometimes, it leads to car being stationary at a random coordinate.

- How it handles spiral curves: The agent will recognize spirals by noticing the closeness of center and left & right sensors. If the velocity exceeds 0.17 on sharp turns on spiral tracks, the agent will apply brakes in order to avoid crashing.
- How it controls speed: If the velocity is low, it will speed up. If sensors detect lanes too close, it will hit the brake. If the agent is going in track 8, it will either coast or apply brake, depending on its speed. Otherwise, it calculates a safe distance based on its current speed: If it's too close to side lanes, it will apply brake. If path is clear and velocity is less than 0.15, it will speed up. Lastly, If it's already moving with the velocity greater than 0.15, it just coasts.

Readings:

Track Number	Average Score (out of 100 runs)
Track 1	143.465959871967
Track 2	143.536384620165
Track 3	123.529267901901
Track 4	123.612639582729
Track 5	118.407345080306
Track 6	118.520195926021
Track 7	133.407045631926
Track 8	128.132588920589

CONCLUSION:

By this iterative development of my Race Car Project, I explored many learning inputs in the field of AI. I made logic structures for steering control, speed controls, brake and coast mechanisms to achieve the goal of building an intelligent agent which performs well across all the 8 tracks. Starting with a basic rule-based approach, I gradually introduced curvature detection, dynamic braking thresholds, and spiral-specific handling to fine-tune the agent's response to different track challenges. Each iteration taught me the importance of sequence in conditional logic structures, balancing parameters to the best numeric value possible and to test across multiple runs for stable reward patterns. I was able to achieve a stable balance between speen and steering control which I consider a huge milestone for myself. The final results, averaging over 100 runs per track, clearly show consistent improvement, with Tracks 1, 2, and 7 performing exceptionally well. This project helped me to understand and deepened my skills in iterative AI development and debugging strategies.

FUTURE ENHANCEMENTS:

I believe I can make further iterations using learningAgent.py and by implementing the Q table and epsilon greedy method. As of now, with the limited time constraints, I have not exceeded what I have mentioned in this log.