F1 - Reinforcement Learning Algorithms for Self Driving Cars

Table of Contents:

1. Project Idea & Game Engine

The goal of this project is to build a simulated race track where an AI learns to drive itself. The AI will need to figure out steering, speeding and how not to crash into walls (by that in itself we are proposing an agent with better driving accuracy than most of the bucureșteni)

Under the Hood (The Game Engine)

- **The Looks (visible only if the trainer wants them):** The game engine will draw the track, the car, and update everything you'd see on the screen.
- **The Physics:** It'll handle how the car moves – speeding up, slowing down, and turning (at the same implementation logic level as Super Mario.) Plus, figuring out when it bumps into things! Which will definitely steer us clear from lots of headaches! 🙂
- **The Feedback Loop:** This tells us how well the AI is doing by taking into consideration lots of observational factors such as the speed, steering angle, distance from walls etc.

The AI Drivers

- **Q-learning:** This one learns by building a map of what actions work best in each situation.
- **DQNs:** Q-learning but on steroids using Neural Nets.
- **NEAT:** An evolutionary algorithm that optimizes both the weights and topology of neural networks.

Environment 1 (DQN):

<u>Action Space:</u>

0: Do Nothing - The agent does not change its current speed or direction.
1: Accelerate - Increases the agent's speed, moving faster forward.
2: Decelerate - Decreases the agent's speed, slowing it down.
3: Turn Right - Adjusts the agent's direction to the right without altering its speed.
4: Turn Left - Adjusts the agent's direction to the left without altering its speed.

<u>Observation Space:</u>

- Position (x, y): Coordinates on the track. Range varies with track dimensions.
- Angle: Orientation in degrees. Range: [-360, 360].
- Speed: Current velocity. Range: [0, ∞).
- Distance from Edges: Distances to the nearest track edge in four directions (front, back, left, right). Range: [0, ∞) for each direction.

<u>Reward Structure:</u>

Total Reward Calculation: The total reward is a sum of being alive, speed, and distance from edges. Specifically:

- Being Alive Reward (alive_reward):
    - +100 if the car does not hit the walls (i.e., no collision).
    - -100 if the car hits the walls (i.e., a collision occurs).

- Speed Reward (speed_reward):
    - Calculated as 5 times the current speed of the car. If the car's speed is at or below 0, a penalty is applied instead of a reward.

- Distance from Edges Reward (distance_reward):
  - The mean distance from the walls in all four directions (left, right, front, back) is used. This encourages the car to maintain a safe distance from the edges.

Penalties:

- Stationary Penalty (stationary_penalty): A penalty of -50 is applied if the car's speed is at or below 0, encouraging the car to keep moving.
- Wrong Direction Penalty (direction_penalty): A penalty of -100 is applied if the car is on the wrong side of the start, ensuring the car is moving in the intended direction on the track.

<u>Starting State</u>

At the start of an episode, the car is placed at a random position near the starting line with a speed of 0 and an angle of 0 (facing along the track).

<u>Episode Termination</u>

An episode ends if the car collides with the track boundary or if a predefined time limit is exceeded.

<u>Rendering</u>

The environment uses Pygame for rendering:

The track is displayed with distinct colors for the terrain, start/finish line, and track boundaries. The car's position, orientation, and movement are updated in real-time based on the agent's actions.

Environment 2 (Q learning + NEAT):

Neat

Action and observation spaces:

a) Actions
   - Same actions as in section 1 of DQN Environment

b) Observations
   - 7 sensors between -90 and 90 degrees, with a 30 degree step. The role of sensors is to return the distance from the closest wall.
   - 1 extra sensor for the 0 degree angle to go more in depth

c) Rewards

- total reward = total distance achieved + 10k bonus for reaching finish line

Q learning

Action and observation spaces:

a)Actions

- Same actions as in section 1 of DQN Environment

b) Observations

- 2 sensors oriented at -90 and 90 degrees checking distance from left and right walls

c)Types of reward

- |right sensor distance-left sensor distance| negative reward meant to ensure that agent doesn't touch left and right walls
- -10000 if agent has touched finish going backwards
- 10000 if agent has touched finish by going forward
- -10 if agent is not moving
- -10 if agent collides with walls
- positive reward for distance reached in current lap

Implementation details

- Rendering was implemented the same as for DQN environment (pygame). As for MDP modeling, no custom library was used. Both NEAT and Q learning agents inherited the same abstract class with custom methods that arose from the differences in actions, observations and rewards, as well for implementation details of the algorithms which shall be explained in the next sections.

2. Algorithms

2.1. NEAT (NeuroEvolution of Augmenting Topologies)

2.1.1. Theory

According to Stanley et al. in Efficient Evolution of Neural Network Topologies (https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf) genetic algorithms used in the artificial evolution of neural networks show a great improvement over traditional reinforcement learning tasks in some areas but face 3 challenges: the genetic representation that allows disparate topologies to crossover in a meaningful way, the protection of topological innovation and the minimization of topologies throughout evolution (Stanley et al., Efficient Evolution of Neural Network Topologies).

In order to address these problems, NEAT was introduced and uses the following concepts (https://www.youtube.com/watch?v=lAjcH-hCusg):

- Phenotype = neural network
- Genotype = the information for constructing the neural network

- Crossover = combination of individual properties of the neurons and synapses
- Mutations
  - Structural: adding link between neurons (with cycle checking), remove a link between two neurons, adding a hidden neuron (by splitting a link), removing a hidden neuron (and all its links)
  - Non-structural: they mutate an existing neuron or a link (values are capped to avoid overfitting)

2.1.2. Configuration

In the file *neat.config* there are the configuration values. In this paper the most important ones will be explained.

The **fitness criterion** is maximum, therefore the maximum reward will be used to ascertain the fitness.

The **population size** is 30 individuals.

The **bias** mutations are configured as:

bias_mutate_power	= 0.5
bias_mutate_rate	= 0.7
bias_replace_rate	= 0.1

The **genome compatibility** is calculated as:

compatibility_disjoint_coefficient	= 1.0
compatibility_weight_coefficient	= 0.5

This means that a higher value is placed on dissimilar genes rather than the average weight difference of matching genes, which will involve the fact that genomes with more different genes will not be likely to be selected for crossover.

The **connection add/delete probability** is set to:

conn_add_prob	= 0.5
conn_delete_prob	= 0.5

This means that there is a 50% chance of adding or deleting a connection.

The **initial setup** of the neural network:

feed_forward	= True
initial_connection	= full

This represents that the initial network is feedforward and that the connectivity is full.

The **node add/delete probability** is set to:

node_add_prob	= 0.2
node_delete_prob	= 0.2

The **input/output/hidden nodes** are set to:

num_hidden = 0
num_inputs = 8
num_outputs = 5

This is to ensure a mapping of all of the 8 sensors as input and 5 possible moves as output. It is also a very simple network with no hidden layers because there is no need for complexity at this task.

The **species compatibility** threshold is:

compatibility_threshold = 2.0

The **species stagnation** is set as:

species_fitness_func = max
max_stagnation = 20
species_elitism = 2

This evaluates the fittest member of that species over 20 generations for stagnation. It also preserves the 2 most fitted individuals.

The **reproduction** is set as:

elitism = 3
survival_threshold = 0.2

This means that top 3 fitted members will survive without any modifications and carried in the next generation and that only 20% of the individuals are selected for reproduction.

In order to provide data to the input layer, the following radars are being used:

```
for d in range(-90, 120, 30):
            self.check_radar(d)
```

This means that there is a radar from -90 to 90 degrees at 30 degrees stop. Each radar will check for collision until a MAX_LENGTH distance

```
while not self.is_collision_points(x, y) and length < NeatCar.MAX_LENGTH:
            length += 1
            x = int(self.center[0] + length * math.cos(math.radians(360 -
(self.angle + degree))))
            y = int(self.center[1] + length * math.sin(math.radians(360 -
(self.angle + degree))))
```

Moreover, for the 0 degree angle, an in-depth radar was added that checks until MAX_LENGTH + 100 distance:

```
if degree == 0:
            length = 0
            x = int(self.center[0] + length * math.cos(math.radians(360 -
(self.angle + degree))))
```

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

            while not self.is_collision_points(x, y) and length <
NeatCar.MAX_LENGTH + 100:
                length += 1
                x = int(self.center[0] + length * math.cos(math.radians(360
- (self.angle + degree))))
                y = int(self.center[1] + length * math.sin(math.radians(360
- (self.angle + degree))))
```

All the radars ar drawn on screen.

```
def draw(self, screen):
        super().draw(screen)
        for r in self.radars:
            pg.draw.line(screen, (0, 255, 0), self.center, r[0], 1)
            pg.draw.circle(screen, (0, 255, 0), r[0], 5)
```

The reward is tied to the distance (amended by the length of the car for mathematical purposes) and the passing of the finish line from the right distance (otherwise a penalty will be incurred).

```
def get_reward(self):
        return self.distance / (Car.CAR_SIZE_X / 2) +
self.has_touched_finish() * 10000
```

A simulation can be run using pygame and the fitness of each individual assessed on screen

```
def run_simulation(genomes, config, speed = 20, full_screen = True):
   networks = []
   cars = []

   pg.init()

   global current_map_path
   my_track = Track(current_map['file'])

   screen = pg.display.set_mode((my_track.map_width, my_track.map_height),
(SCALED | RESIZABLE) if not full_screen else FULLSCREEN)
   my_track.load_game_map()
```

```
    top_start_line, bottom_start_line = my_track.get_start_line_points()

    clock = pg.time.Clock()

    start_pos_x, start_pos_y, angle = my_track.start_pos

    start_pos_y -= Car.CAR_SIZE_Y // 2

    for _, g in genomes:
        net = neat.nn.FeedForwardNetwork.create(g, config)
        networks.append(net)
        g.fitness = 0
        cars.append(
            NeatCar('cars/car2d.png', start_pos_x, start_pos_y, angle,
speed, my_track.game_map, my_track.border_color,
                    my_track.width, my_track.height, top_start_line,
bottom_start_line))

    font_generation = pg.font.SysFont("Arial", 30)
    font_alive = pg.font.SysFont("Arial", 20)

    global current_generation
    current_generation += 1

    timer = time.time()
    MAX_TIME = 60

    while True:
        for event in pg.event.get():
            if event.type == QUIT:
                pg.quit()

        no_still_alive = 0

        for i, car in enumerate(cars):
            output = networks[i].activate(car.get_neat_data())
            choice = output.index(max(output))

            car.move(choice)
            if car.is_alive():
                no_still_alive += 1
                car.update()
                genomes[i][1].fitness = car.get_reward()

        if no_still_alive == 0 or time.time() - timer > MAX_TIME:
            break

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

        text_gen = font_generation.render("Generation: " +
str(current_generation), True, (0, 0, 0))
        text_gen_rect = text_gen.get_rect()
        text_gen_rect.center = (current_map['x_text'],
current_map['y_text'])
        screen.blit(text_gen, text_gen_rect)

        text_alive = font_alive.render("Alive: " + str(no_still_alive),
True, (0, 0, 0))
        text_alive_rect = text_alive.get_rect()
        text_alive_rect.center = (current_map['x_text'],
current_map['y_text'] + current_map['distance_text'])
        screen.blit(text_alive, text_alive_rect)

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

    # pg.quit()

max_fitness = []
```

In order to have good results, 30 generations were used for the first track and 60 for the second.

2.1.3. Results

For the first track, the first generation looks as follows:

```
Population's average fitness: 17.13222 stdev: 15.76406
Best fitness: 53.90000 - size: (5, 40) - species 1 - id 6
Average adjusted fitness: 0.293
Mean genetic distance 1.078, standard deviation 0.280
Population of 30 members in 1 species:
  ID   age  size  fitness  adj fit  stag
 ====  ===  ====  =======  =======  ====
   1    0    30     53.9    0.293     0
Total extinctions: 0
Generation time: 4.557 sec
```

This means that the average fitness is low and that the population is heterogenous. The best individual has 5 nodes and 40 connections in total. The mean genetic distance is 1.078 with a little standard deviation. The population had no extinctions (didn't fail to produce offspring) and it took 4.557 seconds to evolve.

For the first track, the last generation looks as follows:

```
****** Running generation 29 ******

Population's average fitness: 107.26556 stdev: 225.44513
Best fitness: 931.50000 - size: (5, 36) - species 1 - id 524
Average adjusted fitness: 0.112
Mean genetic distance 1.069, standard deviation 0.318
Population of 30 members in 1 species:
  ID   age  size  fitness  adj fit  stag
 ====  ===  ====  =======  =======  ====
   1   29    30    931.5    0.112    10
Total extinctions: 0
Generation time: 52.959 sec (52.687 average)
```

This means that the fitness has improved on average more than 5 times but the species also has heterogenous members. This may be because of the fact that diversification inside the population is encouraged. Also the number of connections decreased which is in line with the research paper's ambitions. The genetic distance has decreased meaning that an optimum is close to being reached. The generation time was 52.959 seconds.

For the second track, the first generation looks as follows:

```
Population's average fitness: 15.24444 stdev: 15.70978
Best fitness: 39.83333 - size: (5, 40) - species 1 - id 10
Average adjusted fitness: 0.344
Mean genetic distance 1.224, standard deviation 0.197
Population of 30 members in 1 species:
  ID   age  size  fitness  adj fit  stag
```

```
 ==== === ==== ======= ======= ====
        1       0       30      39.8     0.344   0
Total extinctions: 0
Generation time: 4.635 sec
```

Judging by the first track, the fitness seems to increase, but that may be due to the track. The size of the neural network is still full but the population is more diverse than in the first generation of the first track.

For the second track, the second generation looks as follows:

```
Population's average fitness: 477.17111 stdev: 951.18665
Best fitness: 2844.53333 - size: (6, 35) - species 2 - id 823
Average adjusted fitness: 0.157
Mean genetic distance 1.503, standard deviation 0.642
Population of 30 members in 2 species:
  ID   age  size  fitness  adj fit  stag
 ==== === ==== ======= ======= ====
        2   25    19   2844.5      0.182   24
        3        4        11   2844.0      0.131   2
Total extinctions: 0
Generation time: 62.603 sec (92.954 average)
```

We can see a great improvement in population fitness but also a huge disparity with the elite. We can also see that the differences are so big that there exist two species. Also the size of the network decreased.

Checking visually, both best fitted individuals seem to have found the perfect route.

Trying to use the first individual on the second track produces a distance of 1552 and the second individual on the first track a distance of 1278. This fails to be comparable to 27945 on the first track by the first individual and 111359 on the second track by the second individual. This means that the model has adapted to a specific track and confirms the perfect route that was visually observed.

2.2. DQN

2.2.1 Theory

An extension of Q-Learning, DQN comes to solve a whole lot of problems, like handling higher-dimensionality problems, generalization and stability.

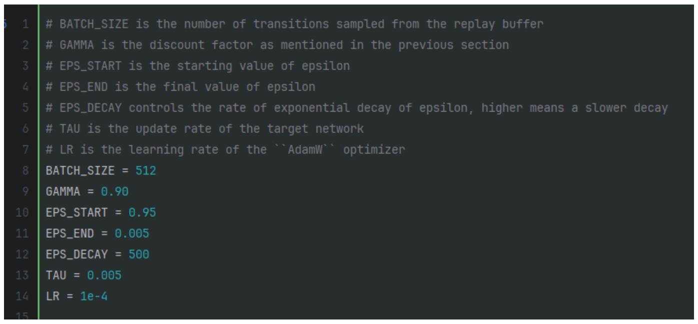
It does it by replacing the Q-table with a NN, making it a Q-Network. It is used to make learn Q(s,a) by iteratively updating the Bellman equations.

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

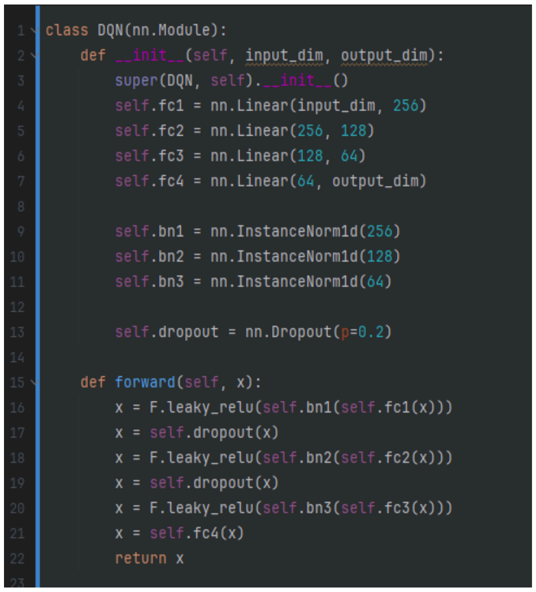
The following pseudocode description resembles our whole implementation scheme of the DQN:

```
**Algorithm 1** Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

2.3.2. Configuration



```python
# BATCH_SIZE is the number of transitions sampled from the replay buffer
# GAMMA is the discount factor as mentioned in the previous section
# EPS_START is the starting value of epsilon
# EPS_END is the final value of epsilon
# EPS_DECAY controls the rate of exponential decay of epsilon, higher means a slower decay
# TAU is the update rate of the target network
# LR is the learning rate of the ``AdamW`` optimizer
BATCH_SIZE = 512
GAMMA = 0.90
EPS_START = 0.95
EPS_END = 0.005
EPS_DECAY = 500
TAU = 0.005
LR = 1e-4
```



```python
class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, output_dim)

        self.bn1 = nn.InstanceNorm1d(256)
        self.bn2 = nn.InstanceNorm1d(128)
        self.bn3 = nn.InstanceNorm1d(64)

        self.dropout = nn.Dropout(p=0.2)

    def forward(self, x):
        x = F.leaky_relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.leaky_relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = F.leaky_relu(self.bn3(self.fc3(x)))
        x = self.fc4(x)
        return x
```

2.2.3. Results

Median ones

For track01 (size: 1920x1080):

- Distance traveled in 30 seconds or until first crash: ~80000
- Average speed: 3.52 units/sec
- First episode when finish line is reached: -

For track02 (size: 1920x1080):

- Distance traveled in 30 seconds or until first crash: ~3000
- Average speed: ~1.25 units/sec
- First episode when finish line is reached: -

2.3. Q-Learning

2.3.1. Theory

Q-learning is a model-free reinforcement learning algorithm that learns the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.

After Δt steps into the future the agent will decide some next step. The weight for this step is calculated as $\gamma \Delta t$, where γ (the discount factor) is a number between 0 and 1 ($0 \leq \gamma \leq 1$). Assuming $\gamma < 1$, it has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). Gamma (γ) may also be interpreted as the probability to succeed (or survive) at every step Δt. The algorithm, therefore, has a function that calculates the quality of a state–action combination: $Q : S \times A \rightarrow R$.

2.3.2. Configuration

My agent is an instance of the class CarAgent which is derived from class Car2. In addition to Car2, our agent has the following:

- Hyperparameters for the Q Function (alpha - learning rate, gamma - discount factor, epsilon)
- The number of actions that can be taken by the agent, two radars that find the distance from the car to the nearest left and right wall (-90° and 90°), a dictionary that stores the values from each state when taking every action
- A function that selects an action: at the beggining it will choose random actions more often, but, with every episode, epsilon decreases and the action with the best reward will be chosen more frequently by the end of the training
- The function updateExperience uses the q function formula and updates the value for an action for the current state
- A get_reward function that returns the reward, based on the distance travelled and on how centered is the car on the track
- The function check_radar creates a radar at a given degree related to the car
- The function draw, draws the car and its radars on the screen
- The update function updates the car's position and its radars' position
- The run_simulation function trains the agent when is_training flag is True and saves the q dictionary, otherwise it loads the pickle files with the dictionary corresponding to the q function and the car takes the action with the best reward at each step.

2.3.3. Results

For track01 (size: 482x270):

- Distance traveled in 30 seconds or until first crash: 1228
- Average speed: 779.65 (1228 / 1.57 sec)

- First episode when finish line is reached: 1382

For track02 (size: 480x270):

- Distance traveled in 30 seconds or until first crash: 1121
- Average speed: 582.86 (1121 / 1.92 sec)
- First episode when finish line is reached: 913

3. Comparison

	DQN	**Q-learning**	**NEAT**	**Tracks**
Distance in 30 seconds / until first accident	~80000	1228	27945	**Track01**
	~3000	1121	111359	**Track02**
Average speed	~3.52	779.65	9.43	**Track01**
	~1.25	582.86	29.84	**Track02**
Episodes/Generations	240	1382	30	**Track01**
	80	913	60	**Track02**

We see that for the first track the best distance is produced by DQN and the least by Q-learning. For the second, however, DQN produces a lower distance than NEAT. We also see DQN preferring lower speeds than the other 2. However, the training process is quite fast for both DQN and NEAT.

In conclusion, for this specific problem, DQN and NEAT do a better job than Q-Learning and are both suited for the solution.

4. Bibliography

1. DQN for PyTorch (reinforcement_q_learning.ipynb - Colaboratory (google.com))
2. CONTINUOUS CONTROL WITH DL (1509.02971.pdf (arxiv.org))
3. DQN Explained (DQN Explained | Papers With Code)
4. Car Env Gym (Car Racing - Gymnasium Documentation (farama.org))
5. AI car simulation: (AI Car Simulator - Github)

6. Efficient Evolution of Neural Network Topologies (Paper)
7. NEAT Algorithm from Scratch (NEAT algorithm from scratch (it was hard))
8. Cursurile si laboratoarele de RL ❤️😉