LOG - Computational Intelligence

s317661 Tcaciuc Claudiu Constantin

Laboratory

When doing the laboratory I was advised by two other students (320061 - Luca Sturato & 313848 - Gabriele Tomatis) and researched on the web about the topic of the lab using things like wikipedia and stackoverflow.

LAB 01: Set Covering Problem with A* Search

The requirement for this laboratory was to solve the set covering problem using the A* (A-star) Algorithm.

The set covering problem requires finding the smallest number of sets from a given collection such that their union covers all the elements in a universal set. A* is an algorithm for pathfinding and graph traversal, that uses a cost function g(n) for measuring the cost of the path chosen, and a heuristic function f(n) that estimates the distance from the goal.

In my case, the g(n) function gives the number of elements that I have to take from the sets to cover all the elements, and the h(n) function gives us the number of sets that we have to take to cover all the elements in the set.

```
def distance (state):
    return PROBLEM_SIZE - sum(
        reduce(
            np.logical_or,
            [SETS[i] for i in state.taken],
            np.array([False for _ in range(PROBLEM_SIZE)]))
        )
```

- g(n) = distance
- h(n) = len(state.taken)

The A* algorithm will choose the path that has the smallest value of the sum of the two functions.

```
def weight(state):
    return distance(state) + len(state.taken)
```

LAB 02: Define an ES Agent Able to Play Nim

The requirement for the second laboratory was to implement an agent able to play Nim, that uses an Evolutionary Strategy to make decisions.

In the Nim game, players take turns removing objects from distinct heaps based on certain rules, the player forced to take the last objects loses. An optimal Strategies involve creating and maintaining winning positions based on the nim-sum rule.

In the python notebook provided there was already the Nim class and some strategies already implemented:

- pure random: a completely random move;
- gabriele: pick always the maximum possible number of the lowest row;
- optimal: uses the nim-sum rule to play the game, but doesn't follow the mathematical rule to always win.

My implementation for the ES Agent follows the basic rule of an evolutionary strategies:

- define an initial population: I choose to use the optimal strategy to generate the population and to introduce more variability I append some gene generated by the pure random strategies;
- mutate the population: based on the mutation rate mutate the population, the mutation rate is variable to stabilize the population based on performance;

```
def mutate(move, mutation_rate, population):
    if random.random() < mutation_rate:
        return random.choice(population)
    return move
```

- tournament selection: select the top two individuals, in this case just randomly selects two individuals. I should have used a fitness function, like the nim_sum to evaluate every individual in the population and choose the top two;

```
def tournament_selection(population, tournament_size):
    tournament = random.sample(population, tournament_size)
    return max(tournament, key=lambda m: m[1])
```

After the tournament selection to select the move to apply I use the nim_sum() function to evaluate the score of the individuals and I choose the best, then based on the performance the mutation rate is updated.

The parameters choose for the ES Agent are:

```
POPULATION_SIZE = 50
OFFSPRING_SIZE = 30
TOURNAMENT_SIZE = 2
MUTATION_RATE = 0.1
```

They were chosen by me after multiple runs. I took into account the computation time required, and the win rate.

I made my ES agent play 50 games (arbitrary number) against the other implemented strategies (pure random, gabriele, optimal) to evaluate the performance and this are some results:

- 50 games against pure random:
  - 35 wins for ES agent
  - 15 wins for pure random
- 50 games against gabriele:
  - 50 wins for ES agent
  - 0 wins for gabriele
- 50 games against optimal:
  - 24 wins for ES agent
  - 26 wins for optimal

This results shows that ES is at least competitive against a random player and an optimal strategy, although achieving almost 50% win rate against the optimal strategy is the best case scenario.

LAB 09: Local Search able to Solve a Black-Box Problem

The requirement for this laboratory was to implement a local-search algorithm using Evolutionary Algorithm to solve a black-box problem on a 1000-loci genomes.

All we know about this black-box is the fitness function that returns a fitness value based on the 1000-loci genomes.

My implementation of the EA local-search is quite similar to the one implemented in the previous laboratory, the algorithm runs for a specified number of generations, evolving the population to improve the best fitness.

In each generation:

```
def initialize_population(self):
        k = 1000
        return [choices([0, 1], k=k) for _ in range(self.population_size)]
```

- Parents are selected from the current population using the select method

```
def select(self, population):
        return choices(population, k=2, weights=[self.fitness_function(individual) for
individual in population])
```

- Cross-over is applied to the parents to generate the offspring

```
def crossover(self, parent1, parent2):
        k = randint(1, len(parent1) - 1)
        child1 = parent1[:k] + parent2[k:]
        child2 = parent2[:k] + parent1[k:]
        return child1, child2
```

- Mutation is applied to the offspring to generate the mutated offspring
    - mutation strategy: each gene is mutated to 0 or 1 based initially on a 50/50 probability, that is tuned based on the current best fitness (decrease the mutation rate if the fitness increase with 1, otherwise increase the mutation rate)
    - this strategy is applied to give the genome a direction, instead of randomly mutating the genome

```
def mutate(self, individual):
        for gene in range(len(individual)):
            if random() < self.mutation_rate:
                individual[gene] = 1
            else:
                individual[gene] = 0
        return individual
```

- The worst individuals (based on fitness) are replaced by the mutated offspring
- Only the best individual is kept from the previous generation

To reduce the number of fitness calls, the algorithm keeps track of the stagnation of the best fitness. If the best fitness does not improve for a specified number of generations, the algorithm stops. Also, if the best fitness arrives at a certain threshold, the algorithm stops.

The mutation rate was implemented this way because after multiple runs I noted that the fitness improved if the genome had more 1s than 0s, so this solutions actually works only because I exploited the fitness function, but a more accurate function for a EA local-search would be:

```
def mutate(self, individual):
        for gene in range(len(individual)):
            if random() < self.mutation_rate:
                individual[gene] = 1 - individual[gene]
        return individual
```

In this way the mutation would "flip the bit" introducing random mutation to the genome.

The parameters chosen are:

```
MUTATION_RATE = 0.5
NUM_GENERATIONS = 1000
MAX_STAGNATION = 100
```

They were chosen like this after multiple runs where I tested various combinations, and achieved good results.

Also to reduce the number of fitness calls, I implemented the EA local-search with a memory that stores the genome and his fitness, in this way I can store the fitness for the genome instead of computing everytime.

This are some results that I obtained:

```
Local Search evolutionary algorithm with 1 instance(s)
Stopping early after 32 generations due to fitness convergence.
fitness: 1.0,
fitness calls: 1650
----------------------------------------------
Local Search evolutionary algorithm with 1 instance(s) and memoization
Stopping early after 23 generations due to fitness convergence.
fitness: 0.999,
fitness calls: 47
```

```
Local Search evolutionary algorithm with 2 instance(s)
Stopping early after 21 generations due to fitness convergence.
fitness: 1.0,
fitness calls: 801
----------------------------------------------
Local Search evolutionary algorithm with 2 instance(s) and memoization
Stopping early after 23 generations due to fitness convergence.
fitness: 0.998,
fitness calls: 48
```

```
Local Search evolutionary algorithm with 5 instance(s)
Stopping early after 13 generations due to fitness convergence.
fitness: 1.0,
fitness calls: 461
-----------------------------------------------
Local Search evolutionary algorithm with 5 instance(s) and memoization
Stopping early after 19 generations due to fitness convergence.
fitness: 1.0,
fitness calls: 42
```

```
Local Search evolutionary algorithm with 10 instance(s)
Stopping early after 15 generations due to fitness convergence.
fitness: 1.0,
fitness calls: 806
-----------------------------------------------
Local Search evolutionary algorithm with 10 instance(s) and memoization
Stopping early after 13 generations due to fitness convergence.
fitness: 1.0,
fitness calls: 35
```

LAB 10: Reinforcement Learning to devise a tic-tac-toe player

The requirements for this laboratory was to use reinforcement learning to train a player to play tic-tac-toe. The initial python notebook provided a random player and a class for the tic-tac-toe.

To fulfill the requirements I used a q-learning technique to achieve reinforcement learning. Q-Learn uses a dictionary that updated after each match giving a reward if the move gets me to win the game, and three other parameters that are:

- Alpha - Learning rate: determines the extent to which newly acquired information overrides old information (0.1 is a good starting point);
- Gamma - Discount factor: determines the importance of future rewards (0.9 is a good starting point);
- Epsilon - Exploration rate: determines the probability that the agent will explore the environment rather than exploiting it (0.1 is a good starting point);

So basically based on that exploration rate the Player will exploit or explore our solutions space, and then based on the performance the q value is updated.

```
def get_move(self, game):
        if random.uniform(0, 1) < self.epsilon:
            available_moves = [i for i in range(9) if game.board[i] == " "]
            return random.choice(available_moves)
        else:
            current_board_state = tuple(game.board)
            available_moves = [i for i in range(9) if game.board[i] == " "]
            q_values = {move: self.q_values.get((current_board_state, move), 0) for move
in available_moves}
            return max(q_values, key=q_values.get)
```

To correctly use this Player the first thing to do is train it, so that we can correctly exploit or explore the field of solutions. I choose to train this Player against the already implemented random player on 10_000 simulated games, and then test it on 1_000 simulated games.

The number chosen for training and testing are arbitrary, but upon multiple executions I found that these are good to achieve the scope of this laboratory.

The initial parameter I chose were:
- alpha = 0.5
- gamma = 0.9
- epsilon = 0.1

But to verify that these were good parameters I created a vector containing:

```
alpha = np.linspace(0.01, 1, 5)
epsilon = np.linspace(0.01, 1, 5)
gamma = np.linspace(0.01, 1, 5)
```

In this way I can use product function from itertools to try every combination, and see what are the best parameters to use based on the win rate.

```
Best win rate: 87.10%
Best parameters: (0.01, 0.505, 1.0)
```

Is worth noting that these may not be the best parameters but are a good starting point.

Also, Being that I couldn’t try more combinations because the computation with this small number of parameters took too long, I also implemented a multi-thread approach to compute training and testing with different combinations of parameters. I used the concurrent module to parallelize the execution, because multiprocessing (that is what I always do in python, instead of multi-threading) didn’t work on python notebooks.

EXAM

Quixo:

The requirements for the exams were similar to the one of laboratory 10, without the constraints on the reinforcement learning algorithm on a player and on a Quixo game instead of tic-tac-toe.

The provided project contained the Game class with everything already implemented, and a RandomPlayer that makes random moves.

Because debugging and the fact that an Invalid action, for a determined board is useless I send only valid moves every turn.

```
def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        while True:
            from_pos = (random.randint(0, 4), random.randint(0, 4))
            slide = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT])
            if verifie_move(from_pos, slide, game):
                return from_pos, slide
```

Also to not use private method from the game class to test the validity of the action taken in a turn I reimplemented in an utils.py file the validation for the move, and I also added a simulation move that returns how the boards would be after I move was make, in this way to compute the fitness I can simulate the move.

The fitness function computes the next board after the action is taken and then calculates the score for each row, column, and diagonal and then takes the maximum score

```
def fitness(from_pos: tuple[int, int], slide: Move, game: 'Game') -> int:

    copy_board = simulate_move(from_pos, slide, game)
    if copy_board is None:
        return -1

    piece = game.get_current_player()

    count_row = np.max(np.sum(copy_board == piece, axis=1))
    count_col = np.max(np.sum(copy_board == piece, axis=0))
    count_diag = np.max(np.sum(np.diag(copy_board) == piece))
    count_diag_2 = np.max(np.sum(np.diag(np.fliplr(copy_board)) == piece))

    score = max(count_row, count_col, count_diag, count_diag_2)
    return score
```

For the Player I implemented a reinforcement learning strategy (with q-learn like in the previous laboratory) and a genetic strategy.

The Genetic Player is defined like this:

```
class GeneticPlayer(Player):
    def __init__(self, population_size=50, generations=10, memory=1) -> None:
        super().__init__()
        self.population_size = population_size
        self.generations = generations
        self.best_moves = {}
        self.memory = memory
```

The population size and the number of generations were chosen after multiple runs to achieve the best win rate and not having the computation taking too long when playing multiple games.

The strategy follows a basic implementation for a genetic algorithm, for every round:

- The random population is generated, and as stated before only valid actions for the round board are selected;

```
def __generate_random_population(self, game: 'Game') -> list[tuple[tuple[int, int],
Move]]:
        population = []
        for _ in range(self.population_size):
            while True:
                from_pos = (random.randint(0, 4), random.randint(0, 4))
                slide = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT])
                if verifie_move(from_pos, slide, game):
                    population.append((from_pos, slide))
                    break
        return population
```

- Then the population is evolved:
    - Every individual is rated based on their fitness (a score is computed for how they will change the board) and sorted based on that,
    - Then only the elite is chosen (20% of the population).
    - From the Elite offspring are generated, and then they are mutated.
    - The mutation based on the mutation rate, that in this case is fixed at 0.2 (from the results seemed a good value), will mutate one action with another valid action.
- The evolved population is sorted again based on their fitness and the best individual is chosen.

```python
def __evolve_population(self,
                            population: list[tuple[tuple[int, int], Move]],
                            game: 'Game') -> list[tuple[tuple[int, int], Move]]:
        sorted_population = sorted(population,
                                   key=lambda move: fitness(move[0], move[1], game),
                                   reverse=True)
        elite_size = int(self.population_size * 0.2)
        elite = sorted_population[:elite_size]

        offspring = []
        for _ in range(self.population_size - elite_size):
            parent_1, parent_2 = random.sample(elite, 2)
            crossover_point = random.randint(1, len(parent_1) - 1)
            child1 = parent_1[:crossover_point] + parent_2[crossover_point:]
            offspring.append(child1)

        mutated_offspring = [self.__mutate(move, game) for move in offspring]
        return elite + mutated_offspring
```

Then if the Player has memory (self.memory = True) the chosen action is saved inside the memory if that board is not present, otherwise before inserting the fitness for the action already inside the memory and the newly computed one is computed and this one is inserted only if outperforms the previous one.

```python
def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        current_board = tuple(map(tuple, game.get_board()))

        population = self.__generate_random_population(game)

        for _ in range(self.generations):
            population = self.__evolve_population(population, game)

        sorted_population = sorted(population, key=lambda move:
            fitness(move[0], move[1], game), reverse=True)
        best_from_pos, best_slide = sorted_population[0]

        if self.memory:
            if current_board not in self.best_moves:
                self.best_moves[current_board] = (best_from_pos, best_slide)
            else:
                if fitness(best_from_pos, best_slide, game) > fitness(
                    self.best_moves[current_board][0],
                    self.best_moves[current_board][1], game):
                    self.best_moves[current_board] = (best_from_pos,
                                                      best_slide)
                else:
                    best_from_pos, best_slide = self.best_moves[current_board]
```

The Q-Learning Player is defined like this:

```
class QLearningPlayer(Player):
    def __init__(self, learning_rate=0.1, discount_factor=0.9,
exploration_prob=0.1):
        super().__init__()
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob
        self.q_table = {}
        self.history = []
```

The Learning rate, discount factor and exploration probability where chosen upon multiple run, where I tested various parameter, I also did a 4 hour training-test session where I tried different combination, but the I changed the code and ended up not using those value (that were epsilon=0.6, alpha=0.5, gamma=0.4).

This implementation also uses a history list that saves the action taken during a game, in this way when the q table is updated all the actions that lead to a win or lose the game will be updated (reward = 10 for win, -1 for loss).

During a game the player either explores or exploits the solutions space.

```
def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        current_state = tuple(map(tuple, game.get_board()))

        if random.uniform(0, 1) < self.exploration_prob:
            # Explore
            from_pos, move = self.__explore(game)
        else:
            # Exploit
            from_pos, move = self.__exploit(current_state, game)

        self.history.append((current_state, from_pos, move))
        return from_pos, move
```

If the Player explores the solution space then a random, but valid, action is taken.

```
def __explore(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        """ Explore the environment """
        while True:
            from_pos = (random.randint(0, 4), random.randint(0, 4))
            move = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT])
            if verifie_move(from_pos, move, game):
                return from_pos, move
```

If the Player exploits the solution space then firstly the current state of the game is searched inside the q_table, then if that state is still unknown a valid action that tries to maximize the fitness is taken.

```python
def __exploit(self, current_state: tuple, game: 'Game') -> tuple[tuple[int, int], Move]:
        current_state = tuple(map(tuple, current_state))

        state = self.q_table.get(current_state)
        if state is not None:
            from_pos = state["from_pos"]
            move = state["move"]
            if verifie_move(from_pos, move, game):
                return from_pos, move

        while True:
            possible_moves = [Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT]
            from_pos = (random.randint(0, 4), random.randint(0, 4))
            move = max(possible_moves, key=lambda move: fitness(from_pos, move, game))
            if verifie_move(from_pos, move, game):
                return from_pos, move
```

After a match ends the q table is updated giving a positive reward for a win and a negative one for a loss. The q table takes the actions from the history list and inserts them in the q table, if the current state has already an entry in the table the fitness for the action in the table and the action from the game is calculated and the one with the highest score is inserted in the q table.

To test my two players I implemented a play games function that takes two players and plays a determined number of games.

The Q Learning Player was trained for 200 games against the Genetic Player and against the Random Player, and then tested against the same Player for 100 games.

The Genetic Player was tested for 100 games against a Random Player and Q Learning Player.

This are some results from the latest run:

```
---- Training Q-Learning Player against Genetic Player with Memory ----
---- Testing Q-Learning Player against Genetic Player with Memory ----
Q-Learning Player wins: 0
Genetic Player with Memory wins: 100

---- Training Q-Learning Player against Random Player ----
---- Testing Q-Learning Player against Random Player ----
Q-Learning Player wins: 78
Random Player wins: 22
```

```
---- Testing Genetic Player with Memory against Random Player ----
Genetic Player with Memory wins: 99
Random Player wins: 1

---- Testing Genetic Player with Memory against Genetic Player without Memory
----
Genetic Player with Memory wins: 72
Genetic Player without Memory wins: 28

---- Testing Genetic Player without Memory against Random Player ----
Genetic Player without Memory wins: 100
Random Player wins: 0
```

REVIEWS

Regarding the reviews I mainly reviewed the same people, because I wanted to give them some feedback.

LAB 01: Set Covering Problem with A* Search

There was no requirement to make a review for the first laboratory.

LAB 02: Define an ES Agent Able to Play Nim

To: 313848 - Gabriele Tomatis

```
Clarity of the code:

The code is clear and the comments help understanding the different parts of the
code, but the structure of the code is a bit flat. Consider breaking down the
code in function (especially in the Oversimplified match part) for better
readability and reusability.

Consider giving function more suitable name, for better understanding of their
purpose, for example pure_random(state: Nim) and pure_random_2(state: Nim) seems
to be doing the same thing from the name, but the actual implementation differ
in how they handle the situation where num_objects is greater than or equal to
state._k.

Consider printing only the important parts in the final version of the code, for
example when you print the various part of the match if you intend to play
multiple games, you could print only the final result (number of wins for each
player with the strategy they have used)

Evolutionary Strategy Parameters:
```

```
Initial Randomization:

Randomizing alfa, beta and sigma at the beginning is a good practice to explore a wide range of parameters values. Consider adding comments to explain the reasoning behind the selection of the parameters.
N_MATCHES = 30 and N_TRAINS = 50 may be a reasonable number of matches for this problem (nim-sum game), but may be little if the selection behind the best parameters becomes more complex.
Evolutionary Strategy:

The evolutionary strategy loop is well-structured, with clear comments indicating the plus and comma strategies, again here you could have made a function for those strategies and compare their performance.
The evolutionary loop updates the best parameters (alfa_best and beta_best) based on the performance of the current model. The use of a normal distribution for updating parameters (round(np.random.normal(alfa, sigma))) is typical in evolutionary strategies.
The adaptive adjustment of sigma during parameter updates is a good practice. It allows for a dynamic exploration-exploitation trade-off during the optimization process.
Overall Evaluation

Overall the code represents a valid implementation of the Evolutionary Strategy based on comma and plus implementation for playing the Nim game, using a gaussian mutation strategy to improve the parameters of the code.
```

To: 320061 - Luca Sturaro

```
Clarity of the code:

The code is clear and the comments help understanding the different part of the code, the overall structure of the code is good, with just the "main loop" (where you implement the strategy into a game) that could have been more clear with the use of some function to generalize the common part. In this way you would have been able to utilize the same code for both of the parts (Match 1 - Relative position strategy and Match 2 - Ideal Nim Sum strategy)
Consider printing only the important parts in the final version of the code, for example when you print the various part of the match if you intend to play multiple games, you could print only the final result (number of wins for each player with the strategy they have used) or if you intend to show the evolution of your parameter you could print only every 10th matches.
Evolutionary Strategy:

Both evolutionary strategies are implemented clearly and provide a good basis for experimentation. The strategies involve a mutation step based on a Gaussian Distribution, and the parameters are adapted over cycles (every 100th game). You could also try improving the parameter over every game instead of "waiting" 1 cycle, to get to a solution in less iteration.
Initial Parameter:
```

```
The initial parameters for the first loop (Match 1 - Relative position strategy)
are randomly selected at the beginning, this is a good practice to explore a
wide range of parameters values. Consider adding comments to explain the
reasoning behind the selection of the parameters.
For the second loop you are using a single parameter representing the ideal nim
sum, also randomly selected, but this should be the goal of the ES algorithm,
not the starting point.
Evolutionary Strategy:
Mutation Logic:
The mutation logic in evolve_1 and evolve_2 uses a gaussian distribution, which
is common in evolutionary strategy, however the parameter sigma seems to be
hardcoded in both implementations, probably would be better to update even that
parameter.
Strategy evolution loop:
The first loop structure for evolving the strategy is clear, and the mechanism
for choosing the best parameters and evolving them over cycles seems reasonable,
but as I said earlier you could also improve the parameter over every game.
In the case of the second loop, The strategy is similar, the difference is in
how the "choice" is made. In this case instead of choosing based on the
parameters that are modified every cycle, the choice is based on the ideal nim
sum the move wants to achieve.
Overall Evaluation:
Overall the code is well structured and the general purpose of the loops are
clear. This seems to be a good implementation for an Evolutionary Strategy that
uses a gaussian mutation strategy to improve their parameters.
```

LAB 09: Local Search able to Solve a Black-Box Problem

To: 313848 - Gabriele Tomatis

```
The overall structure of the code is good, the only thing I'll point out is to
remove unused code to make the overall code better for readability. The use of
comments help understand the various strategies used in the implementation of
the evolutionary algorithm. I like very much the use of the class for the ea
algorithm, but instead of coding the actual use of the algorithm in the "main",
I would have done a "run" functions that takes the problem and uses the method
implemented in the class, so in this way u only have to call the run method in
the main.
For the actual ea strategy in the parent selection, instead of selecting
randomly the parents I would have ranked them by their fitness and take the two
most fitness parents, to maximize the potential improvements. The mutation
functions appends "1" or "0" based on the mutation rate, in my opinion this is a
good way to go directly to the solution, because this is a black box problem and
the only thing we know is the fitness, but with this strategy you may lose on
variability, because you are forcing the genome to become "1" or "0". Also the
mutation rate is updated if the new solution is better or worse (+10%, -10%)
this is a good way to make your genome go in a "direction" to obtain the best
fitness.
One thing that I like is the use of a stagnation parameter, this not only helps
```

the code stop early if your fitness does not improve over a 100 generation, but also since it stops early reduces the number of fitness calls.
The overall code is well implemented and does what is asked by the constraint.

To: 320061 - Luca Sturaro

The overall structure of the code is good, the comments help understanding the purpose of the various functions, and the name of the functions are suitable for what their purpose is. The only thing I'll point out is the print format for the various generations, with this number of generations, would be better to print stats only after, for example, 10 or 100 iterations to make the output more readable, so that the actual improvements are seen but the readability is improved.
For the local-search algorithm based on evolutionary algorithm, the initial value for generations, population size are overall a good choice, the only thing I'll point out is that with this large number of generations the code will cap very sooner and there will be a lot of time passed to only achieve low improvements, for example for the first iteration of the problem from generation 678 (90% accuracy) to 1713 (95.70% accuracy) we obtain only a 5.7 % increase in accuracy.
In your code you introduce acceptable fitness and a stagnation counter, to check if after the code arrives at a certain threshold the improvements are meaningless so in this way you can stop the code at a lower generation. This is a good way to reduce execution time and obtain faster results.
For the strategy implemented in the ea local-search the mutation function and crossover are a good way to introduce variability in your genome, the only thing I will point out is that the mutation rate seems fixed to 0.5 which may lead to not be able to achieve better fitness value, because the mutation will be unpredictable. A better solution would be starting from a fixed mutation rate and decreasing the mutation gradually if the fitness rate improves.
Overall the code is well structured, clear and does what it should do.

LAB 10: Reinforcement Learning to devise a tic-tac-toe player

To: 313848 - Gabriele Tomatis

The lab 10 required to implement a q-learning strategy to play the tic-tac-toe game.
The overall structure of the code is good, in my opinion it is a bit too much "c-like" style, but that's just preference. The comments provide helpful insights on the function purpose. I would have just created a class for the game and one for the q-learning agent so in this way The code is more clear.
Going into the q-learning strategy the overall parameters used are good in this case, but I would have tried another iteration of these parameters to see if changing them improves the overall win rate. Also using only 2_000 epoch to train and only 10 epoch to test the model seems a bit low, I would have raised them to at least 10_000 for training a 1_000 for testing.
Another thing that I learned is that turning the learning rate off during

testing is a good idea so in this way the model will just use things learned during training, and in your case it doesn't seem that way.
During the print I would have shown more stats, like winning rate during training.
Overall the code seems to resolve the lab requirements, good job!

To: 320061 - Luca Sturaro

Lab 10 required to implement a q-learning strategy for the tic-tac-toe game.
The code seems well structured and the comments provide better explanation on what that function does, also the name of the function is suitable for their purpose.
The Q-Learning class implements a basic q-learning strategy that uses a hashtable to save the various states of the game. For the reward I think it was a good idea to also consider draw games.
The implementation of the training phase is good and a number of epoch = 100_000 seems suitable to train the model. For the testing phase I would only test with the learning rate set to 0 so in this way the test is only done on the result coming from the training phase.
Overall the code is well structured and solves the requirement of the lab, good job!