Computational Intelligence 23/24

Index

Set covering A* - Lab 1

Description

Code

Import

```
import numpy as np
import random as rd
from math import ceil
from functools import reduce
from queue import SimpleQueue, PriorityQueue
from collections import namedtuple
import time
```

Variables

```
PROBLEM_SIZE = 30
NUM_SETS = 50
SETS = tuple([np.array([rd.random() < 0.15 for _ in range(PROBLEM_SIZE)]) for _ in
range(NUM_SETS)])

initial_state = (set(), set(range(NUM_SETS)))
```

Cost functions

```
def covered(state):
    return reduce(np.logical_or, [SETS[i] for i in state[0]], np.array([False for
_ in range(PROBLEM_SIZE)]))

def goal_check(state):
    return np.all(covered(state))

def distance(state):
    return PROBLEM_SIZE - np.sum(covered(state))

def h1(state):
    largest_set_size = max(sum(s) for s in SETS)
    missing_size = PROBLEM_SIZE - sum(covered(state))
    optimistic_estimate = ceil(missing_size / largest_set_size)
    return optimistic_estimate

def h2(state):
    already_covered = covered(state)
    if np.all(already_covered):
```

```
        return 0
    largest_set_size = max(sum(np.logical_and(s, np.logical_not(already_covered)))
for s in SETS)
    missing_size = PROBLEM_SIZE - sum(already_covered)
    optimistic_estimate = ceil(missing_size / largest_set_size)
    return optimistic_estimate

def h3(state):
    already_covered = covered(state)
    if np.all(already_covered):
        return 0
    missing_size = PROBLEM_SIZE - sum(already_covered)
    candidates = sorted((sum(np.logical_and(s, np.logical_not(already_covered)))
for s in SETS), reverse=True)
    taken = 1
    while sum(candidates[:taken]) < missing_size:
        taken += 1
    return taken

def h4(state): #not worth compared to h3
    filtered_sets = [SETS[i] for i in state[1]]
    covered_elements = covered(state)
    missing_elements = PROBLEM_SIZE - sum(covered_elements)
    steps = 0

    while missing_elements > 0 and filtered_sets:
        filtered_sets = sorted([np.logical_and(filtered_sets[i],
np.logical_not(covered_elements)) for i in range(len(filtered_sets))], key=lambda
x: sum(x))
        best_set = filtered_sets.pop()

        missing_elements -= sum(best_set)
        steps += 1
        covered_elemts = best_set

    return steps if missing_elements <= 0 else PROBLEM_SIZE

def gready_cost(state):
    return distance(state) + len(state[0])

def A_cost1(state):
    return h1(state) + len(state[0])

def A_cost2(state):
    return h2(state) + len(state[0])

def A_cost3(state):
    return h3(state) + len(state[0])

def A_cost4(state):
    return h4(state) + len(state[0])

def breath_cost(state):
    return len(state[0])
```

Search function

```
def search(initial_state, cost_function):
    print(cost_function.__name__)
    frontier = PriorityQueue()
    state = initial_state
    counter = 0
    if(not goal_check((state[1], state[0]))):
        print("\tNo solution found")
    else:
        start = time.time()
        while state[1] and not goal_check(state):
            counter += 1
            for action in state[1]:
                if action not in state[0]:
                    new_state = (state[0] ^ {action}, state[1] ^ {action})
                    frontier.put((cost_function(new_state), new_state))
            _, state = frontier.get()
        end = time.time()

        print("\tSolution:", state[0], _)
        print("\tSteps: ", counter)
        print(f'\tTime for step: {((end - start)/counter):.2e}')
```

Main

```
for cost_fun in [gready_cost, A_cost3, A_cost4]:
    search(initial_state, cost_fun)
```

Nim - Lab 2

Description

Code

Import

```
import logging
from itertools import product
from pprint import pprint, pformat
from collections import namedtuple
import random
from copy import deepcopy
import numpy as np
import time
```

Nim class

```
Nimply = namedtuple("Nimply", "row, num_objects")

class Nim:
    def __init__(self, num_rows: int) -> None:
        self._rows = [i * 2 + 1 for i in range(num_rows)]

    def __bool__(self):
        return sum(self._rows) > 0

    def __str__(self):
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

    @property
    def rows(self) -> tuple:
        return tuple(self._rows)

    def nimming(self, ply: Nimply) -> None:
        row, num_objects = ply
        assert self._rows[row] >= num_objects
        self._rows[row] -= num_objects
```

Random strategy

```
def pure_random(state: Nim) -> Nimply:
    """A completely random move"""
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)
```

Gabriel strategy

```
def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c
+ 1)]
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))
```

Nim sum stretegy

```
def nim_sum(state: Nim) -> int:
    tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in state.rows])
    xor = tmp.sum(axis=0) % 2
    return int("".join(str(_) for _ in xor), base=2)

def analize(raw: Nim) -> dict:
    cooked = dict()
    cooked["possible_moves"] = dict()
    for ply in (Nimply(r, o) for r, c in enumerate(raw.rows) for o in range(1, c +
1)):
        tmp = deepcopy(raw)
        tmp.nimming(ply)
        cooked["possible_moves"][ply] = nim_sum(tmp)
    return cooked

def optimal(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    good_moves = [ply for ply, ns in analysis["possible_moves"].items() if ns ==
0]
    if not good_moves:
        good_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(good_moves)
    return ply

def spicy(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    spicy_moves = [ply for ply, ns in analysis["possible_moves"].items() if ns !=
0]
    if not spicy_moves:
        spicy_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(spicy_moves)
    return ply
```

Human strategy

```
def me(state: Nim) -> Nimply:
    row = input("Row: ")
    num_objects = input("Num objects: ")
    return Nimply(int(row) -1, int(num_objects))
```

Adaptive strategy

The genotype

The adaptive strategy genotype is based on a multidimensional array that rappresents all the possbile states of the game, where each dimension rapresents a row of the game. In each cell of the array there is the next

move that the strategy will do in that state.

Example:

`GAME_SIZE = 2`: A Game with 2 rows, one with 1 elements and one with 3 elements The array is a 2x4 matrix where the first dimension rappresent the first row that can have from 0 up to 1 elements, and the second dimension rappresent the second row that can have from 0 up to 3 elements.

One possibile istance of the moves array is:

None	Nimply(row=1, num_objects=1)	Nimply(row=1, num_objects=2)	Nimply(row=1, num_objects=3)
Nimply(row=0, num_objects=1)	Nimply(row=1, num_objects=1)	Nimply(row=1, num_objects=1)	Nimply(row=0, num_objects=1)

In this case the move that the strategy will do in the state (1, 3) is the cell `moves[1][3]` and it is `Nimply(row=0, num_objects=1)`. The next state after this move will be (0, 3), the other player do his move and then the adaptive strategy gives its next move based on the new state in the same way.

The fitness function

The fitness function is really simple and it is 1 if the strategy wins the game and 0 if it loses the game.

```
class Adaptive:
    """A strategy that can adapt its parameters"""
    def __init__(self, dim: int, name: str = "") -> None:
        self.dim = dim
        self.name = name
        self.records = []

        loaded_moves, tries = self.load_moves()

        self.moves = Adaptive.init_moves(dim)

        if loaded_moves.size:
            self.moves[..., *[0 for _ in range(tries)]] = loaded_moves

    def load_moves(self):
        """Load moves from the file if one strategy has already been trained, if
no file is found, tries to load the moves for dim - 1"""
        loaded_moves = np.array([])
        tries = 0
        while not loaded_moves.size and tries < self.dim:
            try:
                loaded_moves = np.load(f"{self.name}_adaptive_{self.dim -
tries}.npy", allow_pickle=True)
            except FileNotFoundError:
                tries += 1
        return loaded_moves, tries
```

```
    @staticmethod
    def init_moves(dim: int):
        """Init the moves array with random moves"""
        moves = np.empty(tuple([2*i+2 for i in range(dim)]), dtype=Nimply)

        for i in tuple(product(*[range(2*i+2)  for i in range(dim)])):
            if sum(i) > 0:
                row = random.choice([r for r, c in enumerate(i) if c > 0])
                num_objects = random.randint(1, i[row])

                moves[i] = Nimply(row, num_objects)

        return moves


    def move(self, state: Nim) -> Nimply:
        """Used to get the move from the strategy"""
        self.records.append(state.rows)
        return self.moves[state.rows]

    def clean_records(self):
        self.records = []

    def mutation(self, ply: Nimply) -> None:
        """Mutate one move of the strategy randomly"""
        possible_rows = [r for r, c in enumerate(ply) if c > 0]
        row = random.choice(possible_rows)
        num_objects = random.randint(1, ply[row])
        self.moves[ply] = Nimply(row, num_objects)

    @staticmethod
    def get_moves(strategies: dict) -> tuple:
        """Get the moves of the strategies that won and the moves of the
strategies that lost"""
        good_moves = {}
        bad_moves = set()

        for strategy, result in strategies.items():
            for rd in strategy.records:
                if result:
                    if rd in good_moves:
                        good_moves[rd].append(strategy.moves[rd])
                    else :
                        good_moves[rd] = [strategy.moves[rd]]
                else:
                    bad_moves.add(rd)

        bad_moves = bad_moves - good_moves.keys()

        return good_moves, bad_moves

    @staticmethod
    def get_candidates(strategies: dict, n_sample: int) -> dict:
        """Extract n_sample candidates from the strategies"""
```

```
        candidates = list(strategies.items())
        extracted = {deepcopy(random.choice([strat for strat, _ in candidates])):
0 for _ in range(n_sample)}
        return dict(extracted)

    @staticmethod
    def fake_get_candidates(strategies: dict) -> dict:
        """
        Does not really extract candidates, just returns a dict with the same keys
and reset the win flag.
        Really faster than get_candidates
        """
        return {strat: 0 for strat in strategies.keys()}

    @staticmethod
    def next_epoch(strategies: dict, n_sample: int) -> list:
        """Get the next epoch of the strategies, selcting the best moves and
mutating the bad ones from the previous epoch"""

        good_moves, bad_moves = Adaptive.get_moves(strategies)

        new_strategies = Adaptive.fake_get_candidates(strategies)

        for strat in new_strategies.keys():
            strat.clean_records()
            for pos, moves in good_moves.items():
                strat.moves[pos] = random.choice(moves)
            for pos in bad_moves:
                strat.mutation(pos)

        return new_strategies

    def save(self):
        """Save the trained moves of the strategy in a file"""
        np.save(f"{self.name}_adaptive_{self.dim}.npy", self.moves)
```

Match function

```
def match(nim: Nim, strategies: dict, start: bool = 0, verbose: bool = True) ->
bool:
    """Play a match of nim between two strategies"""
    player = 1 - start
    if verbose:
            print(f"\tstatus: {nim}")
    while nim:
        player = 1 - player
        ply = strategies[player](nim)
        nim.nimming(ply)
        if verbose:
            print(f"\tply: player {player} plays {ply}")
            print(f"\tstatus: {nim}")
```

```
    return player
```

Main

```
GAME_SIZE = 6
N_EPOCH = 2000
N_POPULATION = 300
"""The strategies are stored in a dict, the key is the strategy and the value is 0
if the strategy lost and 1 if it won"""
apt_strategies = {Adaptive(GAME_SIZE, "1"): 0 for _ in range(N_POPULATION)}
apt_strategies2 = {Adaptive(GAME_SIZE, "2"): 0 for _ in range(N_POPULATION)}
win = 0

for i in range(N_EPOCH):
    for apt, apt2 in zip(apt_strategies.keys(), apt_strategies2.keys()):
        nim = Nim(GAME_SIZE)
        strategy = (apt.move, apt2.move)

        winner = match(nim, strategy, start=i%2, verbose=False)
        if winner == 0:
            apt_strategies[apt] += 1
            win += 1
        else:
            apt_strategies2[apt2] += 1

    if (i+1) % 10 == 0:
        print(f"\tEpoch: {i+1}/{N_EPOCH} Avg win rate: {(win*100/(N_POPULATION *
(i+1))):.3f} %  ", end="\r")

    apt_strategies = Adaptive.next_epoch(apt_strategies, n_sample=N_POPULATION)
    apt_strategies2 = Adaptive.next_epoch(apt_strategies2, n_sample=N_POPULATION)

print(f"win rate: {(win*100/(N_EPOCH*N_POPULATION)):.3f} %" + " "*50)

final_apt = Adaptive.get_candidates(apt_strategies, n_sample=1).popitem()[0]
final_apt2 = Adaptive.get_candidates(apt_strategies2, n_sample=1).popitem()[0]

final_apt.save()
final_apt2.save()
```

Review

- Vincenzo Micciche' s313592
    - https://github.com/vinz321/computational_intelligence_23_24
- Gabriele Ferro s308552
    - https://github.com/Gabbo62/ComputationalIntelligence

Review to Vincenzo Micciche'

The code is in general well written and does what it was designed for, but there are some points that has caught my attention:

The genotype idea is missing the goal of ES, it can't really explore new strategies only chose which is the best from given options. So if we don't know the optimal solution will be not able to find.

The individual class implement also all the pipeline to evaluate and make the next generation, interesting idea but I personally prefer to divide the individual (and his methods to mutate) from playing a game, because in general it's more clear which role have each part, but it works so it's okay.

The code doesn't consider the possibility to use population number greater than one and so the crossover isn't implemented, loosing all its benefit. Probably with few strategies is not so use full but maybe with a huge number of it will converge a in a good solution before.

Running the code it seems that something doesn't work as expected because the solution does not really chose the best strategy, so probably there is a small error somewhere. This comment is based on the result of evolve_first_improv function after 150 epoch: pure_random, vinzgorithm, optimal, gabriele -> 0.39130279, 0.1710468, 0.25711428, 0.18053614

One little mistake is that consider only games where one player start always as first.

Nice idea of static strategy vinzgorithm.

Review to Gabriele Ferro

The code is really well written and also if there aren't comment and there are some complex part and idea, all it's clear and tidy so great work!

The main observation that I have for this project is about the goal, because it doesn't try to find the best strategy but instead the best move at each step of the game. So, probably it isn't what is required from the lab but move on and lets analyse it.

Logical issues:

With this kind of goal probably an ES loses a bit of sense, in fact the solution is some similar to hill climber strategy where the fitness function is the nim sum done separately for each game.

The use of the nim sum as a fitness function does not represent a good way to explore the problem, because if we know that there is a condition (the nim sum) that ensure the victory why we would use a ES to find the best solution "randomly". It more useful find a move with nim sum zero with a path search and use it. So one possible fitness function that does not count the nim sum could be if the strategy win or lose the match.

Implementation issues:

There are only two little implementation issue that not compromise the project.

The optimal function proposed by the teacher is not really optimal and fixing it the result are worse for the fixed rule strategy. But the evolved one still has good result.

The starting player is always the same.

Although all the criticity I really appreciated the logic behind the fixed rule and the way the evolved rule works.

Black box - Lab 9

Description

Code

Import

```
import random
from abc import abstractmethod
import numpy as np
import lab9_lib
```

Individual class

```
LOCI_NUMBER = 1000
MAX_ITERATIONS = 1500

class Individual:
    def __init__(self, genotype=None):
        self.genotype = genotype if genotype else random.choices([0, 1],
k=LOCI_NUMBER)

    @staticmethod
    def evaluate_population(population, fitness_fuction) -> dict['Individual',
float]:
        return {ind: fitness_fuction(ind.genotype) for ind in population}

    @staticmethod
    def difference(ind1, ind2) -> float:
        return sum(np.logical_xor(ind1.genotype, ind2.genotype)) / LOCI_NUMBER if
ind1 and ind2 else 1

    @staticmethod
    @abstractmethod
    def population(size) -> list['Individual']:
        pass

    @staticmethod
    @abstractmethod
    def algorithm() :
        pass
```

Hill climbing

```
class HillIndividual(Individual):
    POPULATION_NUMBER = 1
    OFFSPRING_NUMBER = 10

    def __init__(self, genotype=None):
        super().__init__(genotype)

    def population() -> list['HillIndividual']:
        return [HillIndividual() for _ in range(HillIndividual.POPULATION_NUMBER)]

    def tweak(self, index=-1) -> 'Individual':
        chunks_number = random.choice([2 ** i for i in
range(np.log2(LOCI_NUMBER).astype(int))])
        chunks = np.array_split(self.genotype, chunks_number)
        random.shuffle(chunks)
        genotype = np.concatenate(chunks)
        return HillIndividual(genotype.tolist())

    @staticmethod
    def algorithm(population, fitness_function, max_iterations=MAX_ITERATIONS) ->
tuple[tuple['HillIndividual', float], int]:
        population = HillIndividual.evaluate_population(population,
fitness_function)

        for i in range(max_iterations):
            new_population = {}
            for individual, evaluation in population.items():
                new_individuals = [individual.tweak() for _ in
range(HillIndividual.OFFSPRING_NUMBER)]
                evalueated_individuals =
HillIndividual.evaluate_population(new_individuals, fitness_function)
                best_individual, best_evaluation =
max(evalueated_individuals.items(), key=lambda x: x[1])
                print(f'{i}/{max_iterations} - {best_evaluation:.2%}:
{''.join(str(g) for g in best_individual.genotype)}', end='\r')
                if best_evaluation > evaluation:
                    new_population[best_individual] = best_evaluation
                    if best_evaluation == 1:
                        return (best_individual, best_evaluation),
fitness_function.calls
                else:
                    new_population[individual] = evaluation
            population = new_population

        return max(population.items(), key=lambda x: x[1]), fitness_function.calls
```

Genetic algorithm

```
class GeneticIndividual(Individual):
```

```
    POPULATION_NUMBER = 100
    POPULATION_INCREASE = 0.01
    OFFSPRING_NUMBER = 10
    OFFSPRING_INCREASE = 0.001
    MUTATION_PROBABILITY = 0.4
    MUTATION_PROBABILITY_INCREASE = 0
    MUTATION_METHOD = 2
    CROSSOVER_METHOD = 0
    SELECTIVE_METHOD = 1

    def __init__(self, genotype=None):
        super().__init__(genotype)

    @staticmethod
    def single_mutation(ind) -> 'GeneticIndividual':
        genotype = ind.genotype.copy()
        index = random.choice(range(LOCI_NUMBER))
        genotype[index] = 1 - genotype[index]
        return GeneticIndividual(genotype)

    def multi_mutation(ind, probabilty=1/LOCI_NUMBER) -> 'GeneticIndividual':
        genotype = [1 - g if random.random() < probabilty else g for g in
ind.genotype]
        return GeneticIndividual(genotype)

    def chunk_mutation(ind) -> 'GeneticIndividual':
        chunks_number = random.choice([2 ** i for i in
range(np.log2(LOCI_NUMBER).astype(int))])
        chunks = np.array_split(ind.genotype, chunks_number)
        random.shuffle(chunks)
        genotype = np.concatenate(chunks)
        return GeneticIndividual(genotype.tolist())

    @staticmethod
    def scrumble_crossover(ind1, ind2) -> 'GeneticIndividual':
        genotype = [random.choice([g1, g2]) for g1, g2 in zip(ind1.genotype,
ind2.genotype)]
        return GeneticIndividual(genotype)

    @staticmethod
    def cut_crossover(ind1, ind2) -> 'GeneticIndividual':
        index = random.choice(range(LOCI_NUMBER))
        genotype = ind1.genotype[:index] + ind2.genotype[index:]
        return GeneticIndividual(genotype)

    @staticmethod
    def chunck_crossover(ind1, ind2) -> 'GeneticIndividual':
        chunks_number = random.choice([2 ** i for i in
range(np.log2(LOCI_NUMBER).astype(int))])
        chunks1 = np.array_split(ind1.genotype, chunks_number)
        chunks2 = np.array_split(ind2.genotype, chunks_number)
        genotype = np.concatenate([random.choice([chunk1, chunk2]) for chunk1,
chunk2 in zip(chunks1, chunks2)])
        return GeneticIndividual(genotype.tolist())
```

```
    @staticmethod
    def roulette_selection(population, iteration=0) -> list['GeneticIndividual']:
        new_population = random.choices(list(population.keys()),
weights=population.values(), k=round(GeneticIndividual.OFFSPRING_NUMBER +
GeneticIndividual.OFFSPRING_INCREASE*iteration*iteration))
        return new_population

    @staticmethod
    def tournament_selection(population, tournament_size=2, iteration=0) ->
list['GeneticIndividual']:
        new_population = []
        for _ in range(round(GeneticIndividual.OFFSPRING_NUMBER +
GeneticIndividual.OFFSPRING_INCREASE*iteration*iteration)):
            tournament = random.choices(list(population.keys()),
k=tournament_size)
            new_population.append(max(tournament, key=lambda x: population[x]))

        return new_population

    @staticmethod
    def difference_selection(population, iteration=0) ->
list['GeneticIndividual']:
        new_population = []
        new_individual = None
        for _ in range(round(GeneticIndividual.OFFSPRING_NUMBER +
GeneticIndividual.OFFSPRING_INCREASE*iteration*iteration)):
            scaled_population = {ind: value*(1 + Individual.difference(ind,
new_individual)) for ind, value in population.items()}
            new = random.choices(list(scaled_population.keys()),
weights=scaled_population.values(), k=1)[0]
            print(GeneticIndividual.difference(new, new_individual), end='\r')
            new_individual = new
            new_population.append(new_individual)

        return new_population

    @staticmethod
    def new_generation(population,iteration=0) -> list['GeneticIndividual']:
        new_population = []
        for _ in range(round(GeneticIndividual.POPULATION_NUMBER +
GeneticIndividual.POPULATION_INCREASE*iteration*iteration)):
            if random.random() < (GeneticIndividual.MUTATION_PROBABILITY +
GeneticIndividual.MUTATION_PROBABILITY_INCREASE*iteration) % 1:
                ind = random.choice(population)
                if GeneticIndividual.MUTATION_METHOD == 0:
                    new_population.append(GeneticIndividual.single_mutation(ind))
                elif GeneticIndividual.MUTATION_METHOD == 1:
                    new_population.append(GeneticIndividual.multi_mutation(ind,
(5*(iteration + 1)%LOCI_NUMBER)/LOCI_NUMBER))
                elif GeneticIndividual.MUTATION_METHOD == 2:
                    new_population.append(GeneticIndividual.chunk_mutation(ind))
                else:
                    if random.random() < 0.5:
```

```
new_population.append(GeneticIndividual.chunk_mutation(ind))
                    else:

new_population.append(GeneticIndividual.single_mutation(ind))

            else:
                ind1, ind2 = random.choices(population, k=2)
                if GeneticIndividual.CROSSOVER_METHOD == 0:

new_population.append(GeneticIndividual.scrumble_crossover(ind1, ind2))
                elif GeneticIndividual.CROSSOVER_METHOD == 1:
                    new_population.append(GeneticIndividual.cut_crossover(ind1,
ind2))
                else:
                    new_population.append(GeneticIndividual.chunck_crossover(ind1,
ind2))
        return new_population

    @staticmethod
    def population() -> list['GeneticIndividual']:
        return [GeneticIndividual() for _ in
range(GeneticIndividual.POPULATION_NUMBER)]

    @staticmethod
    def epoch(population, fitness_function, iteration=0) ->
dict['GeneticIndividual', float]:
        if GeneticIndividual.SELECTIVE_METHOD == 0:
            offspring = GeneticIndividual.roulette_selection(population,
iteration)
        elif GeneticIndividual.SELECTIVE_METHOD == 1:
            offspring = GeneticIndividual.tournament_selection(population, 100,
iteration)
        else:
            offspring = GeneticIndividual.difference_selection(population,
iteration)
        new_population = GeneticIndividual.new_generation(offspring, iteration)
        return GeneticIndividual.evaluate_population(new_population,
fitness_function)

    @staticmethod
    def algorithm(population, fitness_function, max_iterations=MAX_ITERATIONS):
        population = GeneticIndividual.evaluate_population(population,
fitness_function)
        best = (None, None)
        for i in range(max_iterations):
            population = GeneticIndividual.epoch(population, fitness_function, i)
            last = best
            best = max(population.items(), key=lambda x: x[1])
            print(f'{i}/{max_iterations} -
{round(GeneticIndividual.difference(last[0], best[0])*LOCI_NUMBER)} -
{best[1]:.2%}: {''.join(str(g) for g in best[0].genotype)}', end='\r')
            if max(population.values()) >= 1:
                break
```

```
        return max(population.items(), key=lambda x: x[1]), fitness_function.calls
```

Genetic algorithm with isolation

```
class IsolationIndividual(GeneticIndividual):
    ISLAND_NUMBER = 5
    ISLAND_ITERATIONS = 10

    def __init__(self, genotype=None):
        super().__init__(genotype)

    @staticmethod
    def algorithm(population, fitness_function, max_iterations=MAX_ITERATIONS):
        population = GeneticIndividual.evaluate_population(population,
fitness_function)

        islands = np.array_split(list(population.items()),
IsolationIndividual.ISLAND_NUMBER)
        last = [None]*IsolationIndividual.ISLAND_NUMBER
        best = (None, None)
        for i in range(max_iterations):
            for j, island in enumerate(islands):
                last[j] = best
                best = max(island, key=lambda x: x[1])
                island = GeneticIndividual.epoch(dict(island),
fitness_function).items()
                print(f'{i}/{max_iterations} -
{j}/{IsolationIndividual.ISLAND_NUMBER} -
{round(GeneticIndividual.difference(last[j][0], best[0])*LOCI_NUMBER)} -
{best[1]:.2%}: {''.join(str(g) for g in best[0].genotype)}', end='\r')
            population = np.concatenate(islands)

            if max(population, key=lambda x: x[1])[1] == 1:
                break

            if (i+1) % IsolationIndividual.ISLAND_ITERATIONS == 0:
                random.shuffle(population)
                islands = np.array_split(population,
IsolationIndividual.ISLAND_NUMBER)

        return max(population.items(), key=lambda x: x[1]), fitness_function.calls
```

Main

```
types = [HillIndividual, GeneticIndividual]

for problem in [1, 2, 5, 10]:
```

```
    print(f'\nProblem {problem}')
    for t in types:
        fitness = lab9_lib.make_problem(problem)

        best, calls = t.algorithm(t.population(), fitness)
        print(f'\t{t.__name__}' + ' '*1100)
        #print(f'\t\tBest individual: {''.join(str(g) for g in
best[0].genotype)}')
        print(f'\t\tBest evaluation: {best[1]:.2%}')
        print(f'\t\tNumber of calls: {round(calls/1000)} K')
```

Review

- Luca Barbato s320213
    - https://github.com/lucabubi/Computational-Intelligence
- Andrea Galella s310166
    - https://github.com/andrea-ga/computational-intelligence

Review to Luca Barbato

The project is well written, the code is clear and well structured, the documentations is really complete and almost perfect, explains all the ideas behind and how they are implemented.

The most relevant problem of this project is the lack of new and original ideas to solve it with different points of view. The main goal it's to try to discover various ways to implement possible solutions but only one idea is developed. Also a simple variation of the crossover, mutation or selection method could be a good starting point to explore better alterative.

Review to Andrea Galella

The project is really complete and thorough, the code is structured well and the documentation shows clearly the results and the settings with graphs.

The implementation of the Evolutionary Algorithm tests many possible variation of the canonical methods such as mutation and crossover methods. There is also different ways to generate the new population using the elitism concept.

The idea of evaluate the population beyond their fitness with a sort of distance metric could be a usefull to promote heterogeneity between individual and to try to don't get stuck in a local maximum.

Summing up, the project hits the goal to discover and test multiple options to find the most suitable for this problem.

Tic tac toe - Lab 10

Description

Code

Import

```
from itertools import combinations
from random import choice, choices
import numpy as np
import pickle
import time
```

TicTacToe class

The game is implemented as a sum of 15 game, so the goal is pick three number from 1 to 9 that sum to 15. If the numbers are displayed in a 3x3 grid as below the goal is to pick three numbers that are in a straight line (horizontal, vertical or diagonal), as tic tac toe game.

2	**7**	**6**
9	**5**	**1**
4	**3**	**8**

```
SEQUENCE = [2, 7, 6, 9, 5, 1, 4, 3, 8]

class TicTacToe():
    def  __init__(self, board=None, x=None, o=None):
        self.board = frozenset(board) if type(board) == set else
frozenset(SEQUENCE)
        self.x = frozenset(x) if x else frozenset()
        self.o = frozenset(o) if o else frozenset()

    def __str__(self):
        return str(self.board) + " " + str(self.x) + " " + str(self.o)

    def __key(self):
        return (self.board, self.x, self.o)

    def __hash__(self):
        return hash(self.__key())

    def __eq__(self, other):
        if isinstance(other, TicTacToe):
            return self.board == other.board and self.x == other.x and self.o ==
other.o
        return NotImplemented

    """
    Show the board in a human readable format
    """
    def show (self):
        for i, move in enumerate(SEQUENCE):
            print(" ", end="")
```

```
                if move in self.x:
                    print("X", end="")
                elif move in self.o:
                    print("O", end="")
                else:
                    print(".", end="")
                if i % 3 == 2:
                    print()
            print()

        """
        Change the board using a possible moves for a player
        """
        def move(self, player, pos):
            if pos in self.board:
                if player == 0:
                    self.x = self.x.union({pos})
                else:
                    self.o = self.o.union({pos})
                self.board = self.board.difference({pos})
                return True
            else:
                return False

        """
        Check if a player has won or if the game is a draw
        """
        def check_win(self):
            if TicTacToe.check(self.x):
                if TicTacToe.check(self.o):
                    raise ValueError("Both players have won")
                return 0
            elif TicTacToe.check(self.o):
                return 1
            elif len(self.board) == 0:
                return -1
            else:
                return None

        def copy(self):
            return TicTacToe(set(self.board), set(self.x), set(self.o))

        """
        Check all the possible triplets of moves to see if a player has won
        """
        @staticmethod
        def check(moves):
            if any([sum(triple) == 15 for triple in combinations(moves, 3)]):
                return True
            else:
                return False

        """
        Flip the board along the diagonal
```

```
        """
        def flip_function (x):
            if x > 6:
                return x - 6
            elif x > 3:
                return x
            else:
                return x + 6

        """
        Rotate the board 90 degrees clockwise
        """
        def rotate_function (x):
            return (10 - 2*x) % 10 if x % 2 == 0 else (5 - 2*x) % 10

        """
        Transform the board using a function above
        """
        def transform(self, function):
            return TicTacToe(set([function(i) for i in self.board]), set([function(i)
    for i in self.x]), set([function(i) for i in self.o]))

        """
        Get the inverse transformations of a list of transformations
        """
        def get_inverse_transformations(transformations):
            inverse_transformations = []
            for transformation in transformations:
                if transformation == TicTacToe.flip_function:
                    inverse_transformations.append(TicTacToe.flip_function)
                else:
                    for _ in range(3):
                        inverse_transformations.append(TicTacToe.rotate_function)
            inverse_transformations.reverse()
            return inverse_transformations

        """
        Get the transformation of the board to another board
        """
        def get_transformation(self, other):
            equilvalent_game = other
            transformation = []
            for _ in range(2):
                for _ in range(4):
                    if self == equilvalent_game:
                        return transformation
                    equilvalent_game =
    equilvalent_game.transform(TicTacToe.rotate_function)
                    transformation.append(TicTacToe.rotate_function)
                equilvalent_game = other.transform(TicTacToe.flip_function)
                transformation = [TicTacToe.flip_function]

            return []
```

```
    """
    Check if two boards are equivalent by checking if one is a transformation of
the other
    """
    def equivalent(self, other):
        return self.get_transformation(other) != []

    """
    Get an equivalent board and the transformations to get to it from a list of
possible equivalent boards
    """
    def get_equivalent(self, possible_equivalents):
        for equivalent in possible_equivalents:
            transformations = equivalent.get_transformation(self)
            if transformations != []:
                return equivalent, transformations
        return self , []
```

Environment

The environment gives reward, next state and if the game is finished given the current state and the action. The state is represented as a TicTacToe object, the action is represented as a number from 1 to 9, the reward is a number different for each situation (win, lose, draw, invalid move). The next state is a TicTacToe object, the game is finished if the game is won, lost, draw or if the action is invalid.

The environment implements dome strategies to play the game as the oppoent of the agent. The strategies are:

- random: pick a random action
- win move: if there is a move that wins the game pick it
- win loss move: if there is a move that does not let win the agent pick it
- me: let the human play

The states are saved in a file, so they can be reused in the next run of the program. All the states are not equivalent to each other, so they are minimized using the symmetries of the game.

```
class Environment():
    INVALID_MOVE_REWARD = -1
    MOVE_REWARD = 0.02
    WIN_REWARD = 1
    LOSE_REWARD = -1
    DRAW_REWARD = 0

    def __init__(self, player = 0, strategy=None) -> None:
        self.states = Environment.get_states()
        self.player = player
        self.strategy = strategy if strategy else Environment.random_strategy

    """
```

```
    Get all the possible states of the game
    """
    def get_states():
        try:
            with open('states.npy', 'rb') as f:
                states = [np.load(f, allow_pickle=True) for _ in
range(len(SEQUENCE) + 1)]
        except FileNotFoundError:
            states = Environment.generate_states()
            with open('states.npy', 'wb') as f:
                for state in states:
                    np.save(f, state)

        return states

    """
    Generate all the possible states of the game
    """
    def generate_states():
        states = []

        for depth in range(len(SEQUENCE) + 1):
            boards = [set(e) for e in combinations(SEQUENCE, depth)]
            good_games = []

            for board in boards:
                x_and_o = set(SEQUENCE) - board
                o = [set(e) for e in combinations(x_and_o, len(x_and_o) // 2)]
                x = [x_and_o - x_moves for x_moves in o]

                for x_moves, o_moves in zip(x, o):
                    game = TicTacToe(board, x_moves, o_moves)
                    equivalent = [game.equivalent(good) for good in good_games]

                    if not any(equivalent):
                        try:
                            game.check_win()
                            good_games.append(game)
                        except ValueError:
                            pass

            states.append(good_games)

        states.reverse()

        return states

    def random_strategy(self, actions):
        return choice(list(actions))

    def win_move_strategy(self, actions):
        for action in actions:
            game = self.current_state.copy()
            game.move(1 - self.player, action)
```

```
                if game.check_win() == 1 - self.player:
                    return action
            return Environment.random_strategy(self, actions)

        def win_loss_move_strategy(self, actions):
            for action in actions:
                game = self.current_state.copy()
                game.move(1 - self.player, action)
                if game.check_win() == 1 - self.player:
                    return action
            for agent_action in actions:
                agent_game = game.copy()
                agent_game.move(self.player, agent_action)
                if agent_game.check_win() == self.player:
                    return agent_action
            return Environment.random_strategy(self, actions)

        def me_strategy(self, actions):
            print("Current state:")
            show_state = self.transform_inv_state(self.current_state)
            show_state.show()
            time.sleep(0.2)
            index = int(input("Enter move: "))
            action = SEQUENCE[index - 1]
            for transformation in self.transformations:
                action = transformation(action)
            return action

        """
        Reset the environment to initial state
        """
        def reset(self):
            self.transformations = []
            self.current_state = choice(self.states[0])
            if self.player == 1:
                action = self.strategy(self, self.current_state.board)
                self.current_state.move(1 - self.player, action)

                self.update_transformations()

            return self.current_state, False

        """
        Transform the equivalent state back to the original state
        """
        def transform_inv_state(self, state):
            for transformation in
    TicTacToe.get_inverse_transformations(self.transformations):
                state = state.transform(transformation)
            return state

        """
        Transform the equivalent action back to the original action
        """
```

```
    def transform_inv_action(self, action):
        for transformation in
TicTacToe.get_inverse_transformations(self.transformations):
            action = transformation(action)
        return action

    """
    Update the current state to an equivalent state that is know from the
environment
    """
    def update_transformations(self):
        possible_equivalents = self.states[len(SEQUENCE) -
len(self.current_state.board)]
        self.current_state, transformation =
self.current_state.get_equivalent(possible_equivalents)
        self.transformations += transformation

    """
    Make a move in the environment
    """
    def step(self, action):

        if not self.current_state.move(self.player, action):
            return self.current_state, Environment.INVALID_MOVE_REWARD, True

        self.update_transformations()

        win = self.current_state.check_win()

        if win == self.player:
            return self.current_state, Environment.MOVE_REWARD +
Environment.WIN_REWARD, True
        elif win == -1:
            return self.current_state, Environment.MOVE_REWARD +
Environment.DRAW_REWARD, True

        env_action = self.strategy(self, self.current_state.board)
        self.current_state.move(1 - self.player, env_action)

        self.update_transformations()

        win = self.current_state.check_win()

        if win == 1 - self.player:
            return self.current_state, Environment.MOVE_REWARD +
Environment.LOSE_REWARD, True
        elif win == -1:
            return self.current_state, Environment.MOVE_REWARD +
Environment.DRAW_REWARD, True

        return self.current_state, Environment.MOVE_REWARD, False
```

Agent

The agent is a Q-learning agent that use a Monte Carlo aproach, so from each game it updates the Q value function with the rewards of the environment. The Q values are an estimation of thw expected reward of each action in each state. The Q values are updated using the formula:

`Q(s, a) = Q(s, a) + (reward - Q(s, a)) / N(s, a)`

where `N(s, a)` is the number of times the agent has visited the state `s` and has taken the action `a`. The reward is the reward of the environment.

The agent has a policy that is epsilon greedy, so it picks a random action with probability `epsilon/number of moves` and the best action with probability `epsilon/number of moves + (1 - epsilon)`, where the best action is the action with the highest Q value. The epsilon is decreased at each game, so the agent starts with a random policy and then it starts to exploit the Q values.

The Q values is saved in a file, so it can be reused in the next run of the program. ALso the number of games played is saved in a file.

```
class Agent():
    """
    Good value for greedy_exp:
    - 0.1 or 0.2 for promote exploration
    - 0.5 for balance exploration and exploitation
    - 1 for promote exploitation
    - 10 for optimal policy
    """
    def __init__(self, greedy_exp=0.5) -> None:
        self.q_values = Agent.get_q_values()
        self.episodes = Agent.get_episodes()
        self.greedy_exp = greedy_exp

    """
    Define the policy of the agent
    """
    def e_greedy_policy(self, state):
        probability = [1/(len(SEQUENCE)*self.episodes**self.greedy_exp) + (1 -
1/(self.episodes**self.greedy_exp)) if action == np.argmax(self.q_values[state]
[0]) else 1/(len(SEQUENCE)*self.episodes**self.greedy_exp) for action in
range(len(SEQUENCE))]
        action = choices(SEQUENCE, probability, k=1).pop()

        return action

    def optimal_policy(self, state):
        return SEQUENCE[np.argmax(self.q_values[state][0])]

    """
    Get the q values from a file or generate them
    """
    def get_q_values():
        try:
```

```
            with open('q_values.pkl', 'rb') as fp:
                q_values = pickle.load(fp)
        except FileNotFoundError:
            q_values = Agent.generate_q_values()

        return q_values

    """
    Generate the q values
    """
    def generate_q_values():
        q_values = {state: np.zeros((2, len(SEQUENCE))) for state in
np.concatenate(Environment.get_states())}

        return q_values
    """
    Get the number of episodes from a file or generate them
    """
    def get_episodes():
        try:
            with open('episodes.pkl', 'rb') as fp:
                episodes = pickle.load(fp)
        except FileNotFoundError:
            episodes = 1

        return episodes
    """
    Update the q values using the policy improvment algorithm
    """
    def policy_improvment(self, episode_states, episode_actions, episode_rewards):
        for index, (state, action) in enumerate(zip(episode_states,
episode_actions)):
            index_action = SEQUENCE.index(action)
            # print("State: ", state)
            # state.show()
            # print("Action: ", action)

            cumulative_reward = sum(episode_rewards[index:])

            # print("Cumulative reward: ", cumulative_reward)
            # print("Q value: ", self.q_values[state][0])
            # print("Q value count: ", self.q_values[state][1])

            self.q_values[state][1][index_action] += 1
            self.q_values[state][0][index_action] += (cumulative_reward -
self.q_values[state][0][index_action]) / self.q_values[state][1][index_action]

            # print("Q value update: ", self.q_values[state][0])
            # print("Q value count update: ", self.q_values[state][1])
        self.episodes += 1

    """
    Save the q values and the number of episodes to files
    """
```

```
    def save(self):
        with open('q_values.pkl', 'wb') as fp:
            pickle.dump(self.q_values, fp)
        with open('episodes.pkl', 'wb') as fp:
            pickle.dump(self.episodes, fp)
```

RL algorithm

The reinforcement learning algorithm is implemented as a function that run an episode of the game. The function takes as input the agent and the strategy of the opponent. The function returns three list of rewards, states and actions. Each episode is runned until the game is finished. At each step the agent picks an action using the policy, then the environment gives the reward, the next state and if the game is finished. At the end of the episode the Q values are updated using the rewards, states and actions.

If you want to train from zero delete the files `q_values.pkl` and `episodes.pkl` and run the code.

```
def episode(agent, player, env_strategy=Environment.random_strategy,
verbose=True):
    env = Environment(player, env_strategy)
    state, end = env.reset()

    if verbose:
        print("Agent play as player ", player)

    states = [state.copy()]
    actions = []
    rewards = []

    while not end:
        if verbose:
            value = agent.q_values[state]
            show_state = env.transform_inv_state(state)
            show_state.show()
            print("Moves:\t\t\t", end="")
            for i in SEQUENCE:
                print("{}   \t".format(env.transform_inv_action(i)), end="")
            print("\nExpected reward:\t", end="")
            for j in value[0]:
                print("{:.2f}\t".format(j), end="")
            print()

        action = agent.optimal_policy(state)
        if verbose:
            print("Agent action: ", env.transform_inv_action(action))
            print()
        state, reward, end = env.step(action)


        states.append(state.copy())
        actions.append(action)
        rewards.append(reward)
```

```
    if verbose:
        print("Final state: ")
        show_state = env.transform_inv_state(state)
        show_state.show()

    return states, actions, rewards


iterations = 1000
agent = Agent(0.2)
win = 0
loss = 0

print(f"Exploaration rate: {1/(agent.episodes**agent.greedy_exp):.4f}")

for i in range(iterations):

    states, actions, rewards = episode(agent, i%2,
env_strategy=Environment.win_loss_move_strategy, verbose=False)
    print(f"Game: {i} Reward: {sum(rewards):.2f}", end="\r")
    if sum(rewards) > 0.5:
        win += 1
    elif sum(rewards) < -0.5:
        loss += 1

    agent.policy_improvment(states, actions, rewards)

agent.save()

print("Game won: ", win, " "*50)
print("Game lost: ", loss)
```

Review

- Michelangelo Caretto s310178
    - Repository: https://github.com/rasenqt/computational_intelligence23_24
- Luca Pastore s288976
    - Repository: https://github.com/s288976/computational_intelligence_23_24

Review to Michelangelo Caretto

The implementation of the RL algorithm is simple but effective. The code is well written even if a little bit articulated and complex to understand the ideas are clear also thanks to the documentation.

In my opinion, the real value part it is the benchmark part because show what the agent learnt and how well it performs against different strategies. The strategies are different, maybe some more complex strategies could be implemented to see how it will performs in a more difficult scenarios.

The RL algorithm is a little bit poor of ideas because the agent learn only from static random games that it didn't actively play. A probably more effective option could be make the agent learn from games played with its own policy (as epsilon greedy policy) improving it after each match.

An other suggestion for the agent training is to play against the different strategies that you implemented, in this way it will probably learn a better and more general strategies against every one.

In conclusion the work done is good and well structured but it could be expand more on the reinforcement learning part that lacks of experiments and tries to explore more possibilities.

Review to Luca Pastore

The project is well develop, the code is well articulated and structured but it's hard to understand the flow of the ideas and which components does what. A more complete doc file could be helpful to indicate the reason of the architecture choices, instead it does only briefly introduction.

The most interesting part of this project in my opinion is the choice to follow original intuition instead of a canonical approach. Probably the performance of the agent aren't the best archivable but I found stimulating a different point of view of a possible solution.

The main original proposal are:

An alternative form of the reward function that is not -1 for loss, 0 for draw and 1 for win The policy used by the player that balance random and elaborated moves without using epsilon greedy approach, but instead a sort of flag The implementation is structured to train against other RL agents instead of static ones The points that could be improved are mainly two:

The class structure seems to be overcomplicated and some structure are probably useless with some little changes, in particular the variables relate to the states and moves tracking that is used in the player class and in the game class The unconventional methods stimulate ideas but probably lead to under perform. in this case is not really necessary but I think that also know and implement some canonical approach could be useful.

Quixo

Description

Quixo is a board game for two players where the goal is to be the first player to arrange five of their pieces in a row, either horizontally, vertically, or diagonally. The board is a 5x5 grid of squares and starts with all 25 squares empty. On each player turn, they choose one of the emtpy squares or one of their own pieces, turn it to the player's symbol and move it on one of the four sides of the board, pushing all the pieces in that row one square.

DQN player

The player is implemented as a DQN agent, so it approximates the Q function using a neural network and the Q values are used to select the best action to take in a given state of the game.

Neural network

The neural network is implemented as a fully connected network with 3/4 hidden layers (one is optional) and ReLU activation functions. The input space is the `N*N` board, where each square has the possible values `EMPTY`, `X` and `O`. The output space is dependent on the `ACTION_SPACE` that can include all the possible actions, also the invalid ones e.g. `(0, 0) TOP` or can include only the valid ones e.g. `(0, 0) BOTTOM`.

Training

The network is trained using the Adam optimizer and the loss function is the mean squared error between the target Q values and the predicted Q values. To reduce variance in the training are used two networks, one for the target Q values: `target_net` and one for the predicted Q values: `policy_net`. The `target_net` is updated after each optimization step with the weights of the `policy_net` rescaled by a factor `TAU`.

Policy

During training, the agent is using an epsilon-greedy policy to select the next action, where the epsilon cloud be a costant value `EPSILON` or a decaying value as `EPSILON_B` / (`EPSILON_B` + n. of steps). The espilon action is chosen as a random action belonging to the `ACTION_SPACE` without the best action, where each action is weighted by the q values of the `policy_net` and normalised with softmax. Instead, during testing, the agent uses a greedy policy to select the next action, where the greedy action is the best action belonging to the `ACTION_SPACE`.

Game batch

The agent samples a batch of `BATCH_SIZE` games with their trajectories (state, action, reward) and then calculates the mean squared error between the expected Q values and the predicted Q values. The expected Q values are calculated using the `target_net` and the Bellman equation.

Invalid moves

The player implements a trace of the last moves done in a specific state to avoid repeating the same move, that could be invalid, chosing the second, third, etc. best action.

Board normalization

The board of each state is normalized to a canonical form that exploits simmetries if `TRANSFORMATION` is set to `True`. The board is transformed in all the possible ways and the one with the lowest hash value is chosen as the canonical form and also the moves are tranformed coerently with the board. To speed up the process the most used transformations are stored in a dinamic dictionary used as a cache.

Environment

The main goal of the environment is to simulate a match between the agent and a environment player that can be chosen e.g. the `RandomPlayer`, The game is suddivided in steps, where each step is a turn of both player. The environment takes the action of the agent and if it's valid return the next state, the reward and if the game is ended. The next state is the game board after the agent and the environment player have done their moves.

Last move player

An other type of player is the `LastMovePlayer` that is based on any other player's strategy but it analyses the board to find if there is a move that can win the game or if the opponent can win the game in the next move, in this case it blocks the opponent if it's possible. In the other cases it plays as the base player.

Masks

To find which states of the game are winning or losing in an efficient way it checks if the mask of the board is in the precomputed dictionary of the terminal masks. The mask is board where only the squares of the potential winning lines are set to 1, the other squares are set to 0, in this way many games are equivalent and can be grouped in the same mask. From the dictionary it gets the winning or losing square (to archive a winning line) that is used to evalute the move to win according with specific board. To avoid losing the player uses the base player to find the best move and then it checks if the opponent can win in the next move, if it's the case an other move is chosen.

Role

The `LastMovePlayer` can be used as agent or as environment player, in the first case the base player can be trained as usual, in the second case the agent it's trained and tested against a more challenging player.

Training

During the training many combinations and hyperparameters have been tested that can be summerized as follows:

- `N` - The board size, to understand how the agent performs with different complexity of the game.
- `ITERATIONS` - The number of iterations, that reach performance plateau around 1000K iterations.
- `VERSION` - The version of the agent, the first version is trained against only the `RandomPlayer`, from the second version it's trained against `RandomPlayer` and all previous version of the `DQNPlayer` with same hyperparameters.
- `TRANSFORMATION` - The board is normalized to a canonical form after each move to reduce the state space, the process is slow and don't improve the performance.
- `INVALID_SPACE` - The action space is expanded with all possible actions, also the invalid ones that can be done in every state. In this way the agent can learn that some actions are invalid and avoid them but increase the state space and the training complexity.
- `INVALID_MOVES` - The environment notify the agent when it does an invalid move instead of terminating the game, so the agent can do another one according to the policy.
- `LAST_MOVE` - Both the agent and the environment player use the `LastMovePlayer` and the base player is chosen according to other parameters.
- `LOAD` - The environment players are loaded from two different setups, the first is `simple` and it's the standard one, so the agent is trained/tested against the `RandomPlayer` and it's previous versions. The second is `mix` and the agent is trained/tested against the `RandomPlayer` and each player already trained among the 100K and 1000K iterations.

Network parameters

- Rewards of each action: the rewards of each action are tested between [0, 1] and [-1, 1], with different values for `MOVE_REWARD` in range [0, 0.05]

- Epsilon mode: `ESPILON_MODE` chose the epsilon value between `EPSILON` and `EPSILON_B` / (`EPSILON_B` + n. of steps).
- Layers: the number of layers of the network can be 3 or 4 and the number of neurons per layer are tested in the range [128, 1024].
- Batch size: `BATCH_SIZE` is tested in the range [2, 64].
- Gamma: `GAMMA` is the discount factor and it's tested in the range [0.5, 0.9]
- Tau: `TAU` is the factor used to update the `target_net` and it's tested in the range [0.01, 0.05]

Results

The agent is trained to win against as many players as possible, so it tries to find the best and more general policy to win the game.

Usage

To use the agent the following code can be used:

```
from player import DQNPlayer, LastMovePlayer
player = LastMovePlayer(DQNPlayer(mode='test', path'path/to/agent_model.pth'))
```

The path of the most successful agents are:

1. `models/model_5_2000K_MIX_1024S.pth`
2. . . .

Extra - Human player

The `HumanPlayer` is a player that can be used to play against the agent, it's a simple player that asks the user to insert the action to do and it's used to understand how good is the agent policy.

Code