Code TIPE

Marilou Bernard de Courville,

30 juin 2024

Table des matières

1 Configuration

```
DEBUG = False
ORIGINAL_SIZE_THREE = True
DISPLAY_ALL_POPULATION = True

DISPLAY_LARGEST_SNAKE = False

DISPLAY_GRAPHICS = False

# number of cells for the snake to move in each game
WIDTH = 10
HEIGHT = 10

BOARD_SIDE = 880 # indication of largest board side (for max of WIDTH and HEIGHT)
POPULATION = 22**2 # 484 population of snakes or number of games in the collection
ZOOM_FACTOR = 2 # zoom factor for the longest snake

# game strategy, 1:24,18,18,4; 2:9,10,10,4
GAME_STRATEGY = 5
FITNESS_STRATEGY = 3

MAX_ITERATION = 256 # number of iterations before stopping the program
SAVE = True # save the game brains to a file
RESTORE = False # restore the game brains from a file
BRAINS_FILE = 'saved_brains' + '-' + str(POPULATION) + '-' + str(GAME_STRATEGY) +
↪ str(FITNESS_STRATEGY) + '.pickle' # name of the file to save the brains
```

```
CURVES_FILES = 'saved_curves' + '-' + str(POPULATION) + '-' + str(GAME_STRATEGY) + str(FITNESS_STRATEGY) + '.pickle' # name of the file to save the curves

NUMBER_CROSSOVER_POINTS = 2 # number of crossover points for the genetic algorithm
MUTATION_CHANCE = 0.4 # chance of mutation for the genetic algorithm
MUTATION_COEFF = 0.4 # coefficient for the mutation
PORTION_BESTS = 20 # percentage of bests brains to keep for the genetic algorithm

# k=1 KPointsCrossover
#NUMBER_GAMES: u32 = 2_000; WIDTH: u32 = 30; HEIGHT: u32 = 30;
#MUTATION_CHANCE: f64 = 0.5; MUTATION_COEFF: f32 = 0.5; SAVE_BESTS: usize = 100; MAX_AGE: u32 = 500; APPLE_LIFETIME_GAIN: i32 = 50;

LIFE_TIME = True # apply life time constraint to the snake to avoid infinite loops
MAX_LIFE_POINTS = 50 # maximum number of life points for the snake
APPLE_LIFETIME_GAIN = 20 # number of life points gained when eating an apple
RESET_LIFETIME = True # reset life points when eating an apple
NORMALIZE_BOARD = False

SINGLE_SNAKE_BRAIN = 1 # number of snakes in the single snake game

PLAY_SNAKE_ITERATIONS = 10  # number of iterations for the play snake game

up = (0, 1)
down = (0, -1)
left = (-1, 0)
right = (1, 0)
up_right = (1, 1)
up_left = (-1, 1)
down_left = (-1, -1)
down_right = (1, -1)
eight_directions = [right, up_right, up, up_left, left, down_left, down, down_right]
four_directions = [right, up, left, down]
```

2 Main

```
import pygame
import os
import signal
import sys
from game_collection import GameCollection
import math
import matplotlib.pyplot as plt
import numpy as np
import config as c
from scipy.interpolate import make_interp_spline
import pickle
import sys

game_collection = GameCollection(c.POPULATION, c.WIDTH, c.HEIGHT)

if c.RESTORE and os.path.exists(c.BRAINS_FILE):
    game_collection.restore_brains(c.BRAINS_FILE)
```

```
# board with all populations has games_per_side games per side
# each game has WIDTH x HEIGHT cells

if c.DISPLAY_ALL_POPULATION:
    games_per_side = math.ceil(math.sqrt(c.POPULATION))
else:
    games_per_side = 1

CELL_SIDE = (c.BOARD_SIDE // games_per_side) // max(c.WIDTH, c.HEIGHT)
GAME_WIDTH = CELL_SIDE * c.WIDTH
GAME_HEIGHT = CELL_SIDE * c.HEIGHT
BOARD_WIDTH = games_per_side * GAME_WIDTH
BOARD_HEIGHT = games_per_side * GAME_HEIGHT

print(f"CELL_SIDE: {CELL_SIDE}, GAME_WIDTH: {GAME_WIDTH}, GAME_HEIGHT: {GAME_HEIGHT}, BOARD_WIDTH: {BOARD_WIDTH}, BOARD_HEIGHT: {BOARD_HEIGHT}")

if c.DISPLAY_GRAPHICS:
    # pygame setup
    pygame.init()
    screen = pygame.display.set_mode((BOARD_WIDTH, BOARD_HEIGHT))
    clock = pygame.time.Clock()

running = True
dt = 0

iteration = 0

max_fitness = []
min_fitness = []
avg_fitness = []
max_apple_eaten = []
min_apple_eaten = []
avg_apple_eaten = []
max_snake_length = 0

def save_curves(filename):
    with open(filename, 'wb') as f:
        pickle.dump((max_fitness, min_fitness, avg_fitness, max_apple_eaten, min_apple_eaten, avg_apple_eaten, max_snake_length), f)

def restore_curves(filename):
    with open(filename, 'rb') as f:
        data = pickle.load(f)
    return data

def save_and_exit(signal, frame):
    if c.SAVE:
        game_collection.save_brains(c.BRAINS_FILE)
        save_curves(c.CURVES_FILES)
    sys.exit(0)

# save program state in case of interruption
```

```python
signal.signal(signal.SIGINT, save_and_exit)

while running:

    cur_max_fitness = game_collection.best_fitness()
    cur_min_fitness = game_collection.worst_fitness()
    cur_avg_fitness = game_collection.average_fitness()
    cur_max_apple_eaten = game_collection.max_apple_eaten()
    cur_min_apple_eaten = game_collection.min_apple_eaten()
    cur_avg_apple_eaten = game_collection.average_apple_eaten()

    if cur_max_apple_eaten >= max_snake_length:
        max_snake_length = cur_max_apple_eaten + 1

    # retrieve the new game
    if c.DISPLAY_LARGEST_SNAKE:
        game, current_snake = game_collection.longest_snake() # to see the longest snake
    else:
        game, current_snake = game_collection.snake_to_display()

    # display game iteration and fitness of the game (generation) as window title
    #info = f"Gen {game_collection.generation} - Iter {game_collection.iteration} - Fitness {game.fitness():.2e} - Max fitness {cur_max_fitness:.2e} - Avg fitness {round(cur_avg_fitness, 2):.2e} - Max eaten {cur_max_apple_eaten} - Longest ever {max_snake_length}"
    info = f"Gen {game_collection.generation} - Iter {game_collection.iteration} - Fitness ({cur_min_fitness:.1e}:{cur_avg_fitness:.1e}:{cur_max_fitness:.1e}) - Apple ({cur_min_apple_eaten}:{round(cur_avg_apple_eaten, 1)}:{cur_max_apple_eaten}) - Best snake {max_snake_length}"

    if c.DISPLAY_GRAPHICS:
        # poll for events
        # pygame.QUIT event means the user clicked X to close your window
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        # fill the screen with a color to wipe away anything from last frame
        screen.fill("white")

        pygame.display.set_caption(info)

        if not c.DISPLAY_ALL_POPULATION:
            for (x, y) in game.snake_body:
                pygame.draw.circle(screen, "darkolivegreen3", (x * CELL_SIDE + CELL_SIDE / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
            (x, y) = game.snake_body[0] # head of the snake
            pygame.draw.circle(screen, "black", (x * CELL_SIDE + CELL_SIDE / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 4)
            (x, y) = game.apple
            pygame.draw.circle(screen, "brown3", (x * CELL_SIDE + CELL_SIDE / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
            # surround the current game with a black rectangle
```

```
        pygame.draw.rect(screen, "black", (BOARD_WIDTH, BOARD_HEIGHT, BOARD_WIDTH, BOARD_HEIGHT), 1)
    else:
        # draw all games of the game collection in one big table and each game has coordinate and use a square matrix of sqrt(POPULATION) x sqrt(POPULATION)
        # Iterate over each game in the collection
        for i, game in enumerate(game_collection.games):
            # Calculate the row and column of the current game in the table
            row = i // games_per_side
            col = i % games_per_side

            # if game is lost change the color of the rectangle to red
            if game.lost:
                pygame.draw.rect(screen, "red", (col * GAME_WIDTH, row * GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))

            # do a case switch to change the color of the rectangle depending on the death reason
            if game.death_reason == "Wall":
                pygame.draw.rect(screen, "orange", (col * GAME_WIDTH, row * GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))
            elif game.death_reason == "Body":
                pygame.draw.rect(screen, "blue", (col * GAME_WIDTH, row * GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))
            elif game.death_reason == "Life":
                pygame.draw.rect(screen, "green", (col * GAME_WIDTH, row * GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))

            # surround the current game with a black rectangle
            pygame.draw.rect(screen, "black", (col * GAME_WIDTH, row * GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT), 1)

            # Calculate the position of the game cell on the screen
            cell_x = col * GAME_WIDTH
            cell_y = row * GAME_HEIGHT

            # Draw the game on the screen at the calculated position
            for (x, y) in game.snake_body:
                pygame.draw.circle(screen, "darkolivegreen3", (cell_x + x * CELL_SIDE + CELL_SIDE / 2, cell_y + y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
            (x, y) = game.snake_body[0]
            pygame.draw.circle(screen, "black", (cell_x + x * CELL_SIDE + CELL_SIDE / 2, cell_y + y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 4)
            (x, y) = game.apple
            pygame.draw.circle(screen, "brown3", (cell_x + x * CELL_SIDE + CELL_SIDE / 2, cell_y + y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)

        # zoom on longest snake
        game, current_snake = game_collection.longest_snake() # to see the longest snake
```

```
            row = current_snake // games_per_side
            col = current_snake % games_per_side
            cell_x = col * GAME_WIDTH
            cell_y = row * GAME_HEIGHT
            # draw a white rectangle centred on (cell_x, cell_y) with a width of c.ZOOM_FACTOR * WIDTH + CELL_SIDE and a height of c.ZOOM_FACTOR * HEIGHT + CELL_SIDE
            pygame.draw.rect(screen, "yellow", (cell_x, cell_y, c.ZOOM_FACTOR * GAME_WIDTH, c.ZOOM_FACTOR * GAME_HEIGHT))
            for (x, y) in game.snake_body:
                pygame.draw.circle(screen, "darkolivegreen3", (cell_x + c.ZOOM_FACTOR * (x + CELL_SIDE + CELL_SIDE / 2), cell_y + c.ZOOM_FACTOR * (y * CELL_SIDE + CELL_SIDE / 2)), c.ZOOM_FACTOR * CELL_SIDE / 2)
            (x, y) = game.snake_body[0]
            pygame.draw.circle(screen, "black", (cell_x + c.ZOOM_FACTOR * (x + CELL_SIDE + CELL_SIDE / 2), cell_y + c.ZOOM_FACTOR * (y * CELL_SIDE + CELL_SIDE / 2)), c.ZOOM_FACTOR * CELL_SIDE / 4)
            (x, y) = game.apple
            pygame.draw.circle(screen, "brown3", (cell_x + c.ZOOM_FACTOR * (x + CELL_SIDE + CELL_SIDE / 2), cell_y + c.ZOOM_FACTOR * (y * CELL_SIDE + CELL_SIDE / 2)), c.ZOOM_FACTOR * CELL_SIDE / 2)
    else:
        print(info)


    # update your game state here
    if not game_collection.step(c.LIFE_TIME): # all sakes in collection dead go next iteration
        max_fitness.append(cur_max_fitness)
        min_fitness.append(cur_min_fitness)
        avg_fitness.append(cur_avg_fitness)
        max_apple_eaten.append(cur_max_apple_eaten)
        min_apple_eaten.append(cur_min_apple_eaten)
        avg_apple_eaten.append(cur_avg_apple_eaten)
        # plot max_fitness as function of 0:iteration
        iteration += 1
        if iteration >= c.MAX_ITERATION:
            break

    if c.DISPLAY_GRAPHICS:
        # flip() the display to put your work on screen
        pygame.display.flip()

        clock.tick(500)

if c.SAVE:
    game_collection.save_brains(c.BRAINS_FILE)
    save_curves(c.CURVES_FILES)

print(max_fitness)

fig, ax1 = plt.subplots()

```

```
color1 = 'tab:blue'
color2 = 'tab:red'
color3 = 'tab:green'
color4 = 'tab:orange'

ax1.set_xlabel('Génération')
ax1.set_ylabel('Fitness maximum', color=color1)
ax1.set_yscale('log')

# Key change: Use iterations as the x-axis data
ax1.plot(range(len(max_fitness)), max_fitness, color=color2, label='Fitness max')
ax1.plot(range(len(avg_fitness)), avg_fitness, color=color1, label='Fitness avg')
ax1.tick_params(axis='y', labelcolor=color1)

ax1.legend(loc='upper left')  # Add a legend for clarity

color3 = 'tab:green'
ax2 = ax1.twinx()
ax2.set_ylabel('Pommes mangées maximum', color=color3)
# Key change: Use iterations as the x-axis data
ax2.plot(range(len(max_apple_eaten)), max_apple_eaten, color=color4, label='Pommes')
ax2.tick_params(axis='y', labelcolor=color3)

ax2.legend(loc='lower right')

# Add Vertical Gridlines (The Key Change)
ax1.grid(axis='x', linestyle='--')  # Gridlines on the x-axis (iterations)
ax2.grid(axis='y', linestyle='--')  # You need to add it for the second axis too

# Additional styling improvement
plt.title('Fitness et pommes mangées fct. nombre de générations')
fig.tight_layout()

plt.show()

if c.DISPLAY_GRAPHICS:
    pygame.quit()

```

3 Game

```
# un jeu = un seul serpent

from random import randrange
from neural_network import NeuralNetwork
from numpy import argmax
import collections
import config as c
from typing import Tuple, List
import math

class Game:

```

```
    vision = []

    def __init__(self, width: int = 10, height: int = 10, max_life_points: int = 50, apple_lifetime_gain: int = 500, strategy: int = 2, num_fitness: int = 1) -> None:
        self.width = width
        self.height = height
        self.max_life_points = max_life_points
        self.apple_lifetime_gain = apple_lifetime_gain
        self.strategy = strategy
        self.last_space = 0
        self.last_visited = set()

        """
        Various rules to create a neural network:
        * The number of hidden neurons should be between the size of the input layer and the size of the output layer.
        * The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
        * The number of hidden neurons should be less than twice the size of the input layer.
        * The number of hidden neurons should be between the size of the input layer and the output layer.
        * The most appropriate number of hidden neurons is sqrt(input layer nodes * output layer nodes)
        """

        if strategy == 1:
            # Neural network composed of 4 layers, input layer has 24 neurons, 2 hidden layers each with 18 neurons, output layer has 4 neurons (4 directions)
            # in total it has 24 + 18 + 18 + 4 = 64 neurons.
            self.brain = NeuralNetwork([24, 18, 18, 4])
            self.vision_strategy = self.process_vision
        elif strategy == 2:
            self.brain = NeuralNetwork([9, 10, 10, 4])
            self.vision_strategy = self.process_vision2
        elif strategy == 3:
            self.brain = NeuralNetwork([13, 12, 12, 4])
            self.vision_strategy = self.process_vision3
        elif strategy == 4:
            self.brain = NeuralNetwork([25, 18, 18, 4])
            self.vision_strategy = self.process_vision4
        elif strategy == 5:
            self.brain = NeuralNetwork([13, 12, 12, 4])
            self.vision_strategy = self.process_vision5

        self.age = 0
        self.lost = False
        self.apples_eaten = 0
        #self.direction = (-1, 0) # default direction is left for first move
        self.direction = (randrange(-1, 2), randrange(-1, 2)) # make first move random
```

```python
        self.snake_body = [ # snake starts at the center and has 3 bits
            (int(width / 2), int(height / 2))
            ]
        if c.ORIGINAL_SIZE_THREE:
            self.snake_body.append((int(width / 2) + 1, int(height / 2)))
            self.snake_body.append((int(width / 2) + 2, int(height / 2))
            )
        self.original_size = len(self.snake_body)
        self.seed_new_apple()
        self.life_points = self.max_life_points
        self.died_bc_no_apple = 0
        self.death_reason = "None"
        if c.NORMALIZE_BOARD:
            self.norm_constant_diag = math.sqrt(width ** 2 + height ** 2)
            self.norm_constant_board = width * height / 10.0
        else:
            self.norm_constant_diag = 1
            self.norm_constant_board = 20.0

        if num_fitness == 1:
            self.fitness = self.fitness1
        elif num_fitness == 2:
            self.fitness = self.fitness2
        elif num_fitness == 3:
            self.fitness = self.fitness3
        elif num_fitness == 4:
            self.fitness = self.fitness4
        elif num_fitness == 5:
            self.fitness = self.fitness5

    def seed_new_apple(self):
        self.apple = (randrange(0, self.width), randrange(0, self.height))
        while self.apple in self.snake_body:
            self.apple = (randrange(0, self.width), randrange(0, self.height))

    def step(self, life_time: bool) -> bool:
        # process the vision output through the neural network and output activation
        activation = self.brain.feedforward(self.vision_strategy())
        # take the highest activation index for the direction to take
        index = argmax(activation)

        match index:
            case 0:
                self.direction = c.right
            case 1:
                self.direction = c.up
            case 2:
                self.direction = c.left
            case 3:
                self.direction = c.down

        return self.move_snake(self.direction, life_time)

```

```python
    def move_snake(self, incrementer: Tuple[int, int], life_time: bool) -> bool:
        moved_head = (self.snake_body[0][0] + incrementer[0], self.snake_body[0][1] + incrementer[1])

        # vérification de la présence de la tête dans la grille
        if not (0 <= moved_head[0] < self.width and 0 <= moved_head[1] < self.height):
            self.death_reason = "Wall"
            self.lost = True
            return False

        # sauvegarde de la fin de la queue
        end_tail = self.snake_body[-1]

        # déplacement du serpent
        for i in reversed(range(1, len(self.snake_body))):
            self.snake_body[i] = self.snake_body[i - 1]

        self.snake_body[0] = moved_head

        #collisions avec le corps
        for bit in self.snake_body[1:]:
            if bit == self.snake_body[0]:
                self.lost = True
                self.death_reason = "Body"
                return False

        self.age += 1
        self.life_points -= 1

        #collisions avec la pomme
        if self.snake_body[0] == self.apple:
            self.snake_body.append(end_tail) # agrandir le serpent avec la queue précédente
            self.seed_new_apple()
            self.apples_eaten += 1
            if c.RESET_LIFETIME:
                self.life_points = self.max_life_points # on réinitialise la durée de vie au max
            else:
                self.life_points += self.apple_lifetime_gain # on réinitialise la durée de vie conformément au commentaire en dessous:
            # optimize not to recalculate last_visited and last_space for strategy 2
            # if moved_head is in last_visited it needs to be removed since the snake has its head there now
            if self.strategy == 2  or self.strategy == 5: # update last_visited and last_space
                if moved_head in self.last_visited: # adapt last_visited and last_space
                    self.last_visited.remove(moved_head) # only head is to be removed since tail not moved with apple eaten
                    self.last_space -= 1
                else: # reset last_visited and last_space
```

```
                    self.last_space = 0
                    self.last_visited = set()
        else:
            # optimize not to recalculate last_visited and last_space for strategy 2
            if self.strategy == 2 or self.strategy == 5: # update last_visited and last_space
                if moved_head in self.last_visited: # adapt last_visited and last_space
                    self.last_visited.remove(moved_head) # only head is to be removed since tail not moved with apple eaten
                    self.last_space -= 1
                    # check if end_tail is connected to last_visited elements (can be visited) since it has moved and leaves an empty space
                    if any(abs(end_tail[0] - x) == 1 ^ abs(end_tail[1] - y) == 1 for (x, y) in self.last_visited):
                        self.last_visited.add((end_tail[0], end_tail[1]))
                        self.last_space += 1
                else: # reset last_visited and last_space
                    self.last_space = 0
                    self.last_visited = set()

        # vérification de la durée de vie
        if life_time and self.life_points <= 0:
            self.death_reason = "Life"
            self.lost = True
            self.died_bc_no_apple = 1
            return False

        return True

    # vision strategy: 8 directions, 3 informations per direction
    # (1D distance to apple in direction of move, 1 / wall_distance in direction of move, tail_distance in direction of move) + apples_eaten + original_size
    def process_vision(self) -> List[float]:
        vision = [0 for _ in range(3*8)]

        for (i, incrementer) in enumerate(c.eight_directions):
            apple_distance = -1
            wall_distance = -1
            tail_distance = -1

            (x, y) = self.snake_body[0]
            distance = 0

            while True:
                x += incrementer[0]
                y += incrementer[1]
                distance += 1

                # sortie de grille
                if not self.is_on_board(x, y):
                    wall_distance = distance
                    break
```

```

                # sur la pomme
                if (x, y) == self.apple and apple_distance == -1:
                    apple_distance = distance

                # sur la queue
                if (x, y) in self.snake_body and tail_distance == -1:
                    tail_distance = distance

            vision[3*i] = 0 if apple_distance == -1 else 1
            vision[3*i + 1] = 1 / wall_distance
            vision[3*i + 2] = tail_distance if tail_distance != -1 else 0

        self.vision = vision
        return vision

    # vision strategy: 4 directions, 3 informations per direction
    # (manhattan distance to apple, 1 / wall_distance in direction of move,
    ↪  tail_distance in direction of move) + apples_eaten + original_size
    def process_vision3(self) -> List[float]:
        vision = []

        for (i, incrementer) in enumerate(c.four_directions):
            apple_distance = -1
            wall_distance = -1
            tail_distance = -1

            (x, y) = self.snake_body[0]
            distance = 0

            # try to get inputs between [0,1] for the neural network

            distance_apple = self.manhattan_distance_to_apple((x + incrementer[0], y
            ↪  + incrementer[1]))

            vision.append(1.0 / distance_apple if distance_apple != 0 else 1)

            while True:
                x += incrementer[0]
                y += incrementer[1]
                distance += 1

                # sortie de grille
                if not self.is_on_board(x, y):
                    wall_distance = distance
                    break

                # sur la queue
                if (x, y) in self.snake_body and tail_distance == -1:
                    tail_distance = distance

            vision.append(1.0 / wall_distance)
            vision.append(1.0 / tail_distance if tail_distance != -1 else 1)
```

```python

        vision.append(1 / (self.apples_eaten + self.original_size))

        self.vision = vision
        return vision

    # vision strategy: 4 directions, 3 informations per direction
    # (1 if direction is the closest to the apple, 1 / wall_distance in direction of move, tail_distance in direction of move) + apples_eaten + original_size
    def process_vision4(self) -> List[float]:
        vision = []

        min_distance_index = min(range(len(c.eight_directions)), key=lambda i: self.manhattan_distance_to_apple((self.snake_body[0][0] + c.eight_directions[i][0], self.snake_body[0][1] + c.eight_directions[i][1])))

        for (i, incrementer) in enumerate(c.eight_directions):
            apple_distance = -1
            wall_distance = -1
            tail_distance = -1

            (x, y) = self.snake_body[0]
            distance = 0

            while True:
                x += incrementer[0]
                y += incrementer[1]
                distance += 1

                # sortie de grille
                if not self.is_on_board(x, y):
                    wall_distance = distance
                    break

                # sur la queue
                if (x, y) in self.snake_body and tail_distance == -1:
                    tail_distance = distance

            vision.append(1 if i == min_distance_index else 0)
            vision.append(1.0 / wall_distance)
            vision.append(tail_distance if tail_distance != -1 else 0)

        vision.append(self.apples_eaten + self.original_size)
        self.vision = vision
        return vision

    #? weights 8 bits vs. float? normalization?

    # vision strategy: 4 directions, 3 informations per direction
    # (free spaces in direction of move, manhattan distance to apple in direction of move, apple is in the free space in this direction) + apples_eaten + original_size
```

```
    def process_vision5(self) -> List[float]:
        # neural network input contains free space in all directions, distance to apple in all directions, and number of apples eaten (size of snake)
        # 9 inputs in total
        neural_network_input = []
        (hx, hy) = self.snake_body[0] # head of the snake body
        for direction in c.four_directions:
            (dx, dy) = direction
            (cnx, cny) = (hx + dx, hy + dy)
            #metric = self.count_free_moving_spaces(cnx, cny)
            #neural_network_input.append(1.0 / metric if metric != 0 else 1)
            #metric = self.manhattan_distance_to_apple((cnx, cny))
            #neural_network_input.append(1.0 / metric if metric != 0 else 1)
            neural_network_input.append(self.count_free_moving_spaces(cnx, cny) / self.norm_constant_board)
            neural_network_input.append(self.manhattan_distance_to_apple((cnx, cny)) / self.norm_constant_diag)
            neural_network_input.append(1 if self.apple in self.last_visited else 0) # apple can be reached going in this direction
        #neural_network_input.append(1.0 / (self.apples_eaten + self.original_size))
        neural_network_input.append(self.apples_eaten + self.original_size)
        self.vision = neural_network_input
        return neural_network_input

    # vision strategy: 4 directions, 2 informations per direction
    # (free spaces in direction of move, manhattan distance to apple in direction of move) + apples_eaten + original_size
    def process_vision2(self) -> List[float]:
        # neural network input contains free space in all directions, distance to apple in all directions, and number of apples eaten (size of snake)
        # 9 inputs in total
        neural_network_input = []
        (hx, hy) = self.snake_body[0] # head of the snake body
        for direction in c.four_directions:
            (dx, dy) = direction
            (cnx, cny) = (hx + dx, hy + dy)
            #metric = self.count_free_moving_spaces(cnx, cny)
            #neural_network_input.append(1.0 / metric if metric != 0 else 1)
            #metric = self.manhattan_distance_to_apple((cnx, cny))
            #neural_network_input.append(1.0 / metric if metric != 0 else 1)
            neural_network_input.append(self.count_free_moving_spaces(cnx, cny) / self.norm_constant_board)
            neural_network_input.append(self.manhattan_distance_to_apple((cnx, cny)) / self.norm_constant_diag)
        #neural_network_input.append(1.0 / (self.apples_eaten + self.original_size))
        neural_network_input.append(self.apples_eaten + self.original_size)
        self.vision = neural_network_input
        return neural_network_input

    def is_on_board(self, x, y) -> bool:
        return 0 <= x < self.width and 0 <= y < self.height

    def is_possible_move(self, x, y) -> bool:
```

```python
        # check if the move is on the board and not on the snake body except for the tail (since it has moved)
        return self.is_on_board(x, y) and (x, y) not in self.snake_body[:-1]

    def get_possible_moves(self, cur):
        (x, y) = cur
        moves = []
        for direction in c.eight_directions:
            (i, j) = direction
            if self.is_possible_move(x + i, y + j):
                moves.append(direction)
        return moves

    def count_free_moving_spaces(self, x, y) -> int:
        # Breadth-First Search, BFS, snake heads moves to (x, y) and tail's end is no more
        if not self.is_possible_move(x, y): # does not check snake's tail
            return 0
        if (x, y) in self.last_visited:
            return self.last_space
        space = 0
        visited = set([(x, y)])
        queue = collections.deque([(x, y)]) # efficient for pop(0) and append
        while (len(queue) > 0):
            cur = queue.popleft()
            space += 1
            for direction in self.get_possible_moves(cur):
                (i, j) = direction
                (cx, cy) = cur
                cn = (cx + i, cy + j)
                (cnx, cny) = cn
                if cn not in visited and self.is_possible_move(cnx, cny): # does not check snake's tail
                    queue.append(cn)
                    visited.add(cn)
        self.last_visited = visited
        self.last_space = space
        return space

    def manhattan_distance_to_apple(self, head):
        return abs(self.apple[0] - head[0]) + abs(self.apple[1] - head[1])

    def fitness1(self):
        return pow(3, self.apples_eaten) * (self.age - c.MAX_LIFE_POINTS * self.died_bc_no_apple)

    def fitness2(self):
        return (self.apples_eaten ** 3) * (self.age - c.MAX_LIFE_POINTS * self.died_bc_no_apple)

    def fitness3(self):
        return ((self.apples_eaten * 2) ** 2) * ((self.age - c.MAX_LIFE_POINTS * self.died_bc_no_apple) ** 1.5)
```

```python

    def fitness4(self):
        return (self.age * self.age) * pow(2, self.apples_eaten) * (100 *
         ↪  self.apples_eaten + 1)

    def fitness5(self):
        return (self.age * self.age * self.age * self.age) * pow(2,
         ↪  self.apples_eaten) * (500 * self.apples_eaten + 1)

    # age^2*2^apple*(coeff*apple+1)
    # age^2*2^10*(apple-9)*(coeff*10)

    # score = self.apples_eaten, frame_score = self.age
    # ((score^3)*(frame_score)
    # ((score*2)^2)*(frame_score^1.5)

    # remarks
    # * 3^apple*(age): pow(3, self.apples_eaten) * (self.age - 50 *
     ↪  self.died_bc_no_apple) trains faster

```

4 Game Collection

```python
from game import Game
from genetic_algorithm import GeneticAlgorithm
import pickle
import config as c
import math
from typing import List, Tuple

class GameCollection:
    games = []
    ga = GeneticAlgorithm(math.ceil(c.PORTION_BESTS * c.POPULATION / 100),
     ↪  c.NUMBER_CROSSOVER_POINTS, c.MUTATION_CHANCE, c.MUTATION_COEFF)
    iteration = 0
    generation = 1

    def __init__(self, number_games:int, width:int, height:int) -> None:
        self.games = [Game(width, height, c.MAX_LIFE_POINTS, c.APPLE_LIFETIME_GAIN,
         ↪  c.GAME_STRATEGY, c.FITNESS_STRATEGY) for _ in range(number_games)]

    def snake_to_display(self) -> Tuple[Game, int]:
        for i in range(len(self.games)):
            if not self.games[i].lost:
                return self.games[i], i
        return self.games[0], 0

    def longest_snake(self) -> Tuple[Game, int]:
        longest = 0
        index = 0
        for i in range(len(self.games)):
            if len(self.games[i].snake_body) > longest:
                longest = len(self.games[i].snake_body)
```

```
                index = i
        return self.games[index], index

    def step(self, life_time: bool) -> bool:

        self.iteration += 1

        one_game_not_lost = False

        for game in self.games:
            if not game.lost:
                one_game_not_lost = True
                game.step(life_time)

        # if all games are lost, evolve
        if not one_game_not_lost:
            self.evolve()
        return one_game_not_lost

    def evolve(self):

        new_population = self.ga.evolve([
            (game.brain, game.fitness())
            for game in self.games
        ])

        width, height = self.games[0].width, self.games[0].height

        for i in range(len(new_population)):
            g = Game(width, height, c.MAX_LIFE_POINTS, c.APPLE_LIFETIME_GAIN,
            ↪  c.GAME_STRATEGY, c.FITNESS_STRATEGY) # create new game
            g.brain = new_population[i] # inject brain in game
            self.games[i] = g # replace current game with new one

        self.iteration = 0
        self.generation += 1

    def save_brains(self, filename):
        # save the game collection and all the games in the game collection to a file
        #for game in self.games:
        #    print(game.brain.layers_sizes)
        new_games = sorted(self.games, key=lambda game: game.fitness())
        game_brains = [game.brain for game in new_games]
        if c.DEBUG:
            for brain in game_brains:
                print(brain.weights, end=' ')
            print()
        print("save_brains: len(game_brains): ", len(game_brains))
        with open(filename, 'wb') as f:
            pickle.dump(game_brains, f)

    def restore_brains(self, filename):
        with open(filename, 'rb') as f:
```

```
            game_brains = pickle.load(f)
            print("restore_brains: len(game_brains): ", len(game_brains))
            for i in range(len(self.games)):
                self.games[i].brain = game_brains[i]
            if c.DEBUG:
                for brain in game_brains:
                    print(brain.weights, end=' ')
                print()

    def save_to_file(self, filename):
        with open(filename, 'wb') as f:
            pickle.dump(self, f)

    @classmethod
    def load_from_file(cls, filename):
        with open(filename, 'rb') as f:
            return pickle.load(f)

    def best_fitness(self):
        return max(game.fitness() for game in self.games)

    def worst_fitness(self):
        return min(game.fitness() for game in self.games)

    def average_fitness(self):
        return sum(game.fitness() for game in self.games) / len(self.games)

    def max_apple_eaten(self):
        return max(game.apples_eaten for game in self.games)

    def min_apple_eaten(self):
        return min(game.apples_eaten for game in self.games)

    def average_apple_eaten(self):
        return sum(game.apples_eaten for game in self.games) / len(self.games)

```

5 Genetic Algorithm

```
import numpy as np
from neural_network import NeuralNetwork
from typing import List, Tuple
import copy

class GeneticAlgorithm:

    def __init__(self, save_bests: int = 10, k: int = 5, mut_chance: float = 0.5,
    ↪   coeff: float = 0.5) -> None:
        self.save_bests = save_bests
        self.k = k
        self.mut_chance = mut_chance
        self.coeff = coeff

```

```python
    def select_parent(self, population: List[Tuple[NeuralNetwork, int]]) -> Tuple[NeuralNetwork, NeuralNetwork]:
        # Roulette-wheel selection: numpy.random.choice
        maxi = sum([x[1] for x in population])
        selection_probability = [x[1] / maxi for x in population]
        parent1, parent2 = np.random.choice(len(population), p=selection_probability), np.random.choice(len(population), p=selection_probability)
        return population[parent1][0], population[parent2][0]

    def crossover(self, parent_a: List[float], parent_b: List[float]) -> List[float]:
        """
        K-point crossover cf Wikipedia:
        - select k random points in range(len(parent_a))
        - create a new array which alternate between coefficients of parent_a and parent_b
        """
        n = len(parent_a)
        # list of crossover points
        l = sorted([np.random.randint(0, n) for _ in range(self.k)]) # to avoid having two times the same index
        l.append(-1) # to avoid index out of range but never ued
        child = []
        current_parent = 0
        current_index = 0
        for i in range(n):
            if i == l[current_index]:
                current_parent = 1 - current_parent
                current_index += 1
            if current_parent == 0:
                child.append(parent_a[i])
            else:
                child.append(parent_b[i])
        return child

    def mutate(self, genome: List[float]) -> None:
        """
        Gaussian mutation:
        - for each coefficient:
            - if random() <=  mutation chance (paramètre réglé):
                - generate a sign at random
                - generate an amplitude (between 0 and 1)
                - add sign * amplitude * coeff to the coefficient (coeff is a parameter)
        """
        for i in range(len(genome)):
            if np.random.random() <= self.mut_chance:
                sign = 1 if np.random.random() <= 0.5 else -1
                amplitude = np.random.random()
                genome[i] += sign * amplitude * self.coeff

    def evolve(self, population: Tuple[NeuralNetwork, int]) -> list:
```

```python
        assert(len(population) != 0)
        new_population = []
        # sélection des meilleurs
        population.sort(key=lambda x : x[1], reverse=True)
        for i in range(len(population)):
            if i < self.save_bests:
                new_population.append(copy.deepcopy(population[i][0])) # to avoid reference
            else:
                parent_a, parent_b = self.select_parent(population)
                child = self.crossover(parent_a.to_genome(), parent_b.to_genome())
                self.mutate(child)
                new_population.append(NeuralNetwork.from_genome(child, population[i][0].layers_sizes))
        return new_population

```

6 Neural Network

```python
import numpy as np
from typing import List

def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

class NeuralNetwork:

    layers_sizes = []
    weights = []
    biases = []
    activation_function = None

    def __init__(self, layers_sizes:List[int]) -> None:
        self.biases = [np.random.randn(i, 1) for i in layers_sizes[1:]]
        self.weights = [np.random.randn(i, j) for (i, j) in zip(layers_sizes[1:], layers_sizes[:-1])]
        self.activation_function = sigmoid
        self.layers_sizes = layers_sizes

    def feedforward(self, activation):
        for w, b in zip(self.weights, self.biases):
            activation = self.activation_function(np.dot(w, activation) + b)
        return activation

    """
    def to_genome(self) -> List[float]:
        genome = []
        for w in self.weights:
            for line in w:
                for c in line:
                    genome.append(c)
        for b in self.biases:
            for c in b:
```

```
                genome.append(c)
        return genome
    """

    def to_genome(self) -> List[float]:
        genome = np.concatenate([w.flatten() for w in self.weights] + [b.flatten() for b in self.biases])
        return genome.tolist()

    @classmethod
    def from_genome(cls, genome: List[float], layers: List[int]):
        assert len(layers) > 0
        nn = cls(layers)
        # this code is more efficient than the commented code below because it avoids the list inversions
        offset = 0
        for i, (j, k) in enumerate(zip(layers[:-1], layers[1:])):
            nn.weights[i] = np.reshape(genome[offset:offset + j * k], (k, j))
            offset += j * k
        for i, k in enumerate(layers[1:]):
            nn.biases[i] = np.reshape(genome[offset:offset + k], (k, 1))
            offset += k
        """
        genome = list(reversed(genome))
        nn.weights = [np.array([[genome.pop() for _ in range(j)] for _ in range(i)]) for (i, j) in zip(nn.layers_sizes[1:], nn.layers_sizes[:-1])]
        nn.biases = [np.array([genome.pop() for _ in range(i)]) for i in nn.layers_sizes[1:]]
        """
        return nn

```

7 Courbes

```
import os
import pickle
from game import Game
import config as c
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.gridspec as gridspec
import matplotlib.animation as animation

def restore_brain(brain_number: int) -> Game:
    # restore brain from file and inject it into the snake
    with open("brains_53.pickle", 'rb') as f:
        game_brains = pickle.load(f)
        brain = game_brains[brain_number]
        if c.DEBUG:
            print(game.brain, end=' ')
            print()
    return brain

```

```
def visualize_neural_network(brain):
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 8))
    fig.suptitle("Visualisation réseau de neurones", fontsize=16)
    for i in range(3):
        visualize_matrix(brain.weights[i], f"Poids synaptiques - Couche {i+1}", axes[0, i])
        visualize_matrix(brain.biases[i], f"Biais - Couche {i+1}", axes[1, i])
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.savefig("brain_matrix.svg")
    plt.savefig("brain_matrix.eps")
    plt.savefig("brain_matrix.pdf")
    plt.savefig("brain_matrix.png")
    plt.show()

def visualize_matrix(matrix, title, ax=None):
    if ax is None:
        ax = plt.gca() # Get the current axes if not provided
    im = ax.imshow(matrix, cmap='viridis', interpolation='nearest')
    plt.colorbar(im, ax=ax, label='Valeur du poids synaptique')
    ax.set_xlabel('Index du neurone en entrée')
    ax.set_ylabel('Index du neurone en sortie')
    ax.set_title(title)
    # Setting tick parameters
    ax.tick_params(axis='both', which='major', labelsize=6)
    ax.set_xticks(range(matrix.shape[1]))
    ax.set_yticks(range(matrix.shape[0]))

def visualize_neural_network2(brain, fig, axes):
    fig.suptitle("Visualisation réseau de neurones", fontsize=16)
    for i in range(3):
        visualize_matrix(brain.weights[i], f"Poids synaptiques - Couche {i+1}", axes[0, i])
        visualize_matrix(brain.biases[i], f"Biais - Couche {i+1}", axes[1, i])

def update_visualization(i):
    brain = restore_brain(i)
    visualize_neural_network2(brain, fig, axes)

brain = restore_brain(c.SINGLE_SNAKE_BRAIN)
# brain has layers_sizes = [] weights = [] biases = []

for i, (w, b) in enumerate(zip(brain.weights, brain.biases)):
    print(f"Layer {i+1}:")
    print(f"  Weights: {w.shape}")
    print(f"  Biases: {b.shape}")

visualize_neural_network(brain)

# do an animation of the brain matrices

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 8))
fig.suptitle("Visualisation réseau de neurones", fontsize=16)
```

```
ims = []  # List to store the animation frames (heatmaps)
for i in range(256):
    brain = restore_brain(i)
    frames = []
    for j in range(3):
        frame1 = axes[0, j].imshow(brain.weights[j], cmap='viridis', interpolation='nearest', animated=True)
        frame2 = axes[1, j].imshow(brain.biases[j], cmap='viridis', interpolation='nearest', animated=True)
        frames.extend([frame1, frame2])
    ims.append(frames)  # Add frames for the current brain to the list

ani = animation.ArtistAnimation(fig, ims, interval=500, blit=True, repeat_delay=1000)
plt.show()
```

```
import pickle
import matplotlib.pyplot as plt

def restore_curves(filename):
    with open(filename, 'rb') as f:
        data = pickle.load(f)
    return data

max_iterations = 256

(max_fitness5, min_fitness5, avg_fitness5, max_apple_eaten5, min_apple_eaten5, avg_apple_eaten5, max_snake_length5) = restore_curves("curve_53.pickle")
(max_fitness1, min_fitness1, avg_fitness1, max_apple_eaten1, min_apple_eaten1, avg_apple_eaten1, max_snake_length1) = restore_curves("curve_13.pickle")

fig, ax1 = plt.subplots()

color1 = 'tab:blue'
color2 = 'tab:red'
color3 = 'tab:green'
color4 = 'tab:orange'

ax1.set_xlabel('Génération')
ax1.set_ylabel('Fitness maximum', color=color1)
ax1.set_yscale('log')
# Key change: Use iterations as the x-axis data
ax1.plot(range(1, max_iterations + 1), max_fitness1[1:max_iterations + 1], color=color2, label='Fitness strat 1')
ax1.plot(range(1, max_iterations + 1), max_fitness5[1:max_iterations + 1], color=color1, label='Fitness strat 2')
ax1.tick_params(axis='y', labelcolor=color1)

ax1.legend(loc='upper left')  # Add a legend for clarity

color3 = 'tab:green'
ax2 = ax1.twinx()
ax2.set_ylabel('Pommes mangées maximum', color=color3)
# Key change: Use iterations as the x-axis data
```

```python
ax2.plot(range(1, max_iterations + 1), max_apple_eaten1[1:max_iterations + 1], color=color4, label='Pommes strat 1')
ax2.plot(range(1, max_iterations + 1), max_apple_eaten5[1:max_iterations + 1], color=color3, label='Pommes strat 2')
ax2.tick_params(axis='y', labelcolor=color3)

ax2.legend(loc='lower right')

# Add Vertical Gridlines (The Key Change)
ax1.grid(axis='x', linestyle='--')  # Gridlines on the x-axis (iterations)
ax2.grid(axis='y', linestyle='--')  # You need to add it for the second axis too

# Additional styling improvement
plt.title('Fitness et pommes mangées fct. nombre de générations')
fig.tight_layout()
plt.savefig("curve_compare_cv.svg")
plt.savefig("curve_compare_cv.eps")
plt.savefig("curve_compare_cv.pdf")
plt.savefig("curve_compare_cv.png")
plt.show()
```

```python
import pickle
import matplotlib.pyplot as plt
import config as c

max_fitness = []
min_fitness = []
avg_fitness = []
max_apple_eaten = []
min_apple_eaten = []
avg_apple_eaten = []
max_snake_length = 0

def restore_curves(filename):
    with open(filename, 'rb') as f:
        data = pickle.load(f)
    return data

(max_fitness, min_fitness, avg_fitness, max_apple_eaten, min_apple_eaten, avg_apple_eaten, max_snake_length) = restore_curves("curve.pickle")

fig, ax1 = plt.subplots()

color1 = 'tab:blue'
ax1.set_xlabel('Itération')
ax1.set_ylabel('Fitness maximum', color=color1)
ax1.set_yscale('log')
ax1.plot(range(len(max_fitness)), max_fitness, color=color1)
ax1.tick_params(axis='y', labelcolor=color1)

color3 = 'tab:green'
ax2 = ax1.twinx()
ax2.set_ylabel('Pommes mangées maximum', color=color3)
ax2.plot(range(len(max_apple_eaten)), max_apple_eaten, color=color3)
```

```
ax2.tick_params(axis='y', labelcolor=color3)

plt.title('Fitness vs Iteration')
# Add Vertical Gridlines (The Key Change)
ax1.grid(axis='x', linestyle='--')  # Gridlines on the x-axis (iterations)
ax2.grid(axis='y', linestyle='--')  # You need to add it for the second axis too
fig.tight_layout()
plt.savefig("curve.svg")
plt.savefig("curve.eps")
plt.savefig("curve.pdf")
plt.show()
```

8 Faire jouer le mmeilleur serpent

```
import pygame
import os
import pickle
from game import Game
import config as c
from PIL import Image

def restore_snake(brain_number: int) -> Game:
    # restore brain from file and inject it into the snake
    assert(os.path.exists(c.BRAINS_FILE))
    game = Game(c.WIDTH, c.HEIGHT, c.MAX_LIFE_POINTS, c.APPLE_LIFETIME_GAIN,
    ↪  c.GAME_STRATEGY, c.FITNESS_STRATEGY)
    with open(c.BRAINS_FILE, 'rb') as f:
        game_brains = pickle.load(f)
        game.brain = game_brains[brain_number]
        if c.DEBUG:
            print(game.brain, end=' ')
            print()
    return game

game = restore_snake(c.SINGLE_SNAKE_BRAIN)

frames = []

# pygame setup
pygame.init()

# board contains one game/snake

#CELL_SIDE = c.BOARD_SIDE // max(c.WIDTH, c.HEIGHT)
CELL_SIDE = 10
GAME_WIDTH = CELL_SIDE * c.WIDTH
GAME_HEIGHT = CELL_SIDE * c.HEIGHT

screen = pygame.display.set_mode((GAME_WIDTH, GAME_HEIGHT))

clock = pygame.time.Clock()
running = True
dt = 0
```

```python

iteration = 0

max_snake_length = 0

for n in range(c.PLAY_SNAKE_ITERATIONS):
    while running:

        iteration += 1

        cur_fitness = game.fitness()
        cur_apple_eaten = game.apples_eaten
        if cur_apple_eaten >= max_snake_length:
            max_snake_length = cur_apple_eaten + 1

        # display game iteration and fitness of the game (generation) as window title
        info = f"Iter {iteration} - Fitness {cur_fitness:.2e} - Eaten {cur_apple_eaten} - Longest ever {max_snake_length}"

        # poll for events
        # pygame.QUIT event means the user clicked X to close your window
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        # fill the screen with a color to wipe away anything from last frame
        screen.fill("white")
        # draw grid
        for x in range(0, GAME_WIDTH, CELL_SIDE):
            pygame.draw.line(screen, "gray", (x, 0), (x, GAME_HEIGHT))
        for y in range(0, GAME_HEIGHT, CELL_SIDE):
            pygame.draw.line(screen, "gray", (0, y), (GAME_WIDTH, y))

        pygame.display.set_caption(info)

        for (x, y) in game.snake_body:
            pygame.draw.circle(screen, "darkolivegreen3", (x * CELL_SIDE + CELL_SIDE / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
        (x, y) = game.snake_body[0] # head of the snake
        pygame.draw.circle(screen, "black", (x * CELL_SIDE + CELL_SIDE / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 4)
        (x, y) = game.apple
        pygame.draw.circle(screen, "brown3", (x * CELL_SIDE + CELL_SIDE / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
        # suround the current game with a black rectangle
        pygame.draw.rect(screen, "black", (GAME_WIDTH, GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT), 1)

        # update your game state here (do not constrain snake life time)
        if not game.step(False): # snake is dead
            break;
        # flip() the display to put your work on screen
        pygame.display.flip()
        frame_str = pygame.image.tostring(screen, "RGB")
```

```
        frame_image = Image.frombytes("RGB", (GAME_WIDTH, GAME_HEIGHT), frame_str)
        frames.append(frame_image)
        clock.tick(25)
    iteration = 0;
    game = restore_snake(c.SINGLE_SNAKE_BRAIN)

frames[0].save("game_animation.gif", save_all=True, append_images=frames[1:],
↪  duration=100, loop=0)
pygame.quit()
```