

Code TIPE

Marilou Bernard de Courville,
30 juin 2024

Table des matières

1	Configuration	1
2	Main	2
3	Game	7
4	Game Collection	16
5	Genetic Algorithm	18
6	Neural Network	20
7	Courbes	21
8	Faire jouer le mmeilleur serpent	25

1 Configuration

```
1  DEBUG = False
2  ORIGINAL_SIZE_THREE = True
3  DISPLAY_ALL_POPULATION = True
4
5  DISPLAY_LARGEST_SNAKE = False
6
7  DISPLAY_GRAPHICS = False
8
9  # number of cells for the snake to move in each game
10 WIDTH = 10
11 HEIGHT = 10
12
13 BOARD_SIDE = 880 # indication of largest board side (for max of WIDTH and HEIGHT)
14 POPULATION = 22*2 # 44 population of snakes or number of games in the collection
15 ZOOM_FACTOR = 2 # zoom factor for the longest snake
16
17 # game strategy, 1:24,18,18,4; 2:9,10,10,4
18 GAME_STRATEGY = 5
19 FITNESS_STRATEGY = 3
20
21 MAX_ITERATION = 256 # number of iterations before stopping the program
22 SAVE = True # save the game brains to a file
23 RESTORE = False # restore the game brains from a file
24 BRAINS_FILE = 'saved_brains' + '-' + str(POPULATION) + '-' + str(GAME_STRATEGY) +
    ↪ str(FITNESS_STRATEGY) + '.pickle' # name of the file to save the brains
```

```
25 CURVES_FILES = 'saved_curves' + '-' + str(POPULATION) + '-' + str(GAME_STRATEGY) +  
    ↪ str(FITNESS_STRATEGY) + '.pickle' # name of the file to save the curves  
26  
27 NUMBER_CROSSOVER_POINTS = 2 # number of crossover points for the genetic algorithm  
28 MUTATION_CHANCE = 0.4 # chance of mutation for the genetic algorithm  
29 MUTATION_COEFF = 0.4 # coefficient for the mutation  
30 PORTION_BESTS = 20 # percentage of bests brains to keep for the genetic algorithm  
31  
32 # k=1 KPointsCrossover  
33 #NUMBER_GAMES: u32 = 2_000; WIDTH: u32 = 30; HEIGHT: u32 = 30;  
34 #MUTATION_CHANCE: f64 = 0.5; MUTATION_COEFF: f32 = 0.5; SAVE_BESTS: usize = 100;  
    ↪ MAX_AGE: u32 = 500; APPLE_LIFETIME_GAIN: i32 = 50;  
35  
36 LIFE_TIME = True # apply life time constraint to the snake to avoid infinite loops  
37 MAX_LIFE_POINTS = 50 # maximum number of life points for the snake  
38 APPLE_LIFETIME_GAIN = 20 # number of life points gained when eating an apple  
39 RESET_LIFETIME = True # reset life points when eating an apple  
40 NORMALIZE_BOARD = False  
41  
42 SINGLE_SNAKE_BRAIN = 1 # number of snakes in the single snake game  
43  
44 PLAY_SNAKE_ITERATIONS = 10 # number of iterations for the play snake game  
45  
46 up = (0, 1)  
47 down = (0, -1)  
48 left = (-1, 0)  
49 right = (1, 0)  
50 up_right = (1, 1)  
51 up_left = (-1, 1)  
52 down_left = (-1, -1)  
53 down_right = (1, -1)  
54 eight_directions = [right, up_right, up, up_left, left, down_left, down, down_right]  
55 four_directions = [right, up, left, down]
```

2 Main

```
1 import pygame  
2 import os  
3 import signal  
4 import sys  
5 from game_collection import GameCollection  
6 import math  
7 import matplotlib.pyplot as plt  
8 import numpy as np  
9 import config as c  
10 from scipy.interpolate import make_interp_spline  
11 import pickle  
12 import sys  
13  
14 game_collection = GameCollection(c.POPULATION, c.WIDTH, c.HEIGHT)  
15  
16 if c.RESTORE and os.path.exists(c.BRAINS_FILE):  
17     game_collection.restore_brains(c.BRAINS_FILE)
```

```
18 # board with all populations has games_per_side games per side
19 # each game has WIDTH x HEIGHT cells
20
21 if c.DISPLAY_ALL_POPULATION:
22     games_per_side = math.ceil(math.sqrt(c.POPULATION))
23 else:
24     games_per_side = 1
25
26 CELL_SIDE = (c.BOARD_SIDE // games_per_side) // max(c.WIDTH, c.HEIGHT)
27 GAME_WIDTH = CELL_SIDE * c.WIDTH
28 GAME_HEIGHT = CELL_SIDE * c.HEIGHT
29 BOARD_WIDTH = games_per_side * GAME_WIDTH
30 BOARD_HEIGHT = games_per_side * GAME_HEIGHT
31
32 print(f"CELL_SIDE: {CELL_SIDE}, GAME_WIDTH: {GAME_WIDTH}, GAME_HEIGHT: {GAME_HEIGHT},
↪ BOARD_WIDTH: {BOARD_WIDTH}, BOARD_HEIGHT: {BOARD_HEIGHT}")
33
34 if c.DISPLAY_GRAPHICS:
35     # pygame setup
36     pygame.init()
37     screen = pygame.display.set_mode((BOARD_WIDTH, BOARD_HEIGHT))
38     clock = pygame.time.Clock()
39
40 running = True
41 dt = 0
42
43 iteration = 0
44
45 max_fitness = []
46 min_fitness = []
47 avg_fitness = []
48 max_apple_eaten = []
49 min_apple_eaten = []
50 avg_apple_eaten = []
51 max_snake_length = 0
52
53 def save_curves(filename):
54     with open(filename, 'wb') as f:
55         pickle.dump((max_fitness, min_fitness, avg_fitness, max_apple_eaten,
↪ min_apple_eaten, avg_apple_eaten, max_snake_length), f)
56
57 def restore_curves(filename):
58     with open(filename, 'rb') as f:
59         data = pickle.load(f)
60     return data
61
62 def save_and_exit(signal, frame):
63     if c.SAVE:
64         game_collection.save_brains(c.BRAINS_FILE)
65         save_curves(c.CURVES_FILES)
66     sys.exit(0)
67
68 # save program state in case of interruption
```

```
69 signal.signal(signal.SIGINT, save_and_exit)
70
71 while running:
72
73     cur_max_fitness = game_collection.best_fitness()
74     cur_min_fitness = game_collection.worst_fitness()
75     cur_avg_fitness = game_collection.average_fitness()
76     cur_max_apple_eaten = game_collection.max_apple_eaten()
77     cur_min_apple_eaten = game_collection.min_apple_eaten()
78     cur_avg_apple_eaten = game_collection.average_apple_eaten()
79
80     if cur_max_apple_eaten >= max_snake_length:
81         max_snake_length = cur_max_apple_eaten + 1
82
83     # retrieve the new game
84     if c.DISPLAY_LARGEST_SNAKE:
85         game, current_snake = game_collection.longest_snake() # to see the longest
86         ↪ snake
87     else:
88         game, current_snake = game_collection.snake_to_display()
89
90     # display game iteration and fitness of the game (generation) as window title
91     #info = f"Gen {game_collection.generation} - Iter {game_collection.iteration} -
92     ↪ Fitness {game.fitness():.2e} - Max fitness {cur_max_fitness:.2e} - Avg
93     ↪ fitness {round(cur_avg_fitness, 2):.2e} - Max eaten {cur_max_apple_eaten} -
94     ↪ Longest ever {max_snake_length}"
95     info = f"Gen {game_collection.generation} - Iter {game_collection.iteration} -
96     ↪ Fitness ({cur_min_fitness:.1e}:{cur_avg_fitness:.1e}:{cur_max_fitness:.1e}) -
97     ↪ Apple ({cur_min_apple_eaten}:{round(cur_avg_apple_eaten,
98     ↪ 1)}:{cur_max_apple_eaten}) - Best snake {max_snake_length}"
99
100     if c.DISPLAY_GRAPHICS:
101         # poll for events
102         # pygame.QUIT event means the user clicked X to close your window
103         for event in pygame.event.get():
104             if event.type == pygame.QUIT:
105                 running = False
106         # fill the screen with a color to wipe away anything from last frame
107         screen.fill("white")
108
109         pygame.display.set_caption(info)
110
111         if not c.DISPLAY_ALL_POPULATION:
112             for (x, y) in game.snake_body:
113                 pygame.draw.circle(screen, "darkolivegreen3", (x * CELL_SIDE +
114                 ↪ CELL_SIDE / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
115             (x, y) = game.snake_body[0] # head of the snake
116             pygame.draw.circle(screen, "black", (x * CELL_SIDE + CELL_SIDE / 2, y *
117             ↪ CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 4)
118             (x, y) = game.apple
119             pygame.draw.circle(screen, "brown3", (x * CELL_SIDE + CELL_SIDE / 2, y *
120             ↪ CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
121             # surround the current game with a black rectangle
```

```
112     pygame.draw.rect(screen, "black", (BOARD_WIDTH, BOARD_HEIGHT,
    ↪   BOARD_WIDTH, BOARD_HEIGHT), 1)
113 else:
114     # 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)
115     # Iterate over each game in the collection
116     for i, game in enumerate(game_collection.games):
117         # Calculate the row and column of the current game in the table
118         row = i // games_per_side
119         col = i % games_per_side
120
121         # if game is lost change the color of the rectangle to red
122         if game.lost:
123             pygame.draw.rect(screen, "red", (col * GAME_WIDTH, row *
    ↪   GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))
124
125         # do a case switch to change the color of the rectangle depending on
    ↪   the death reason
126         if game.death_reason == "Wall":
127             pygame.draw.rect(screen, "orange", (col * GAME_WIDTH, row *
    ↪   GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))
128         elif game.death_reason == "Body":
129             pygame.draw.rect(screen, "blue", (col * GAME_WIDTH, row *
    ↪   GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))
130         elif game.death_reason == "Life":
131             pygame.draw.rect(screen, "green", (col * GAME_WIDTH, row *
    ↪   GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT))
132
133         # surround the current game with a black rectangle
134         pygame.draw.rect(screen, "black", (col * GAME_WIDTH, row *
    ↪   GAME_HEIGHT, GAME_WIDTH, GAME_HEIGHT), 1)
135
136         # Calculate the position of the game cell on the screen
137         cell_x = col * GAME_WIDTH
138         cell_y = row * GAME_HEIGHT
139
140         # Draw the game on the screen at the calculated position
141         for (x, y) in game.snake_body:
142             pygame.draw.circle(screen, "darkolivegreen3", (cell_x + x *
    ↪   CELL_SIDE + CELL_SIDE / 2, cell_y + y * CELL_SIDE + CELL_SIDE
    ↪   / 2), CELL_SIDE / 2)
143         (x, y) = game.snake_body[0]
144         pygame.draw.circle(screen, "black", (cell_x + x * CELL_SIDE +
    ↪   CELL_SIDE / 2, cell_y + y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE
    ↪   / 4)
145         (x, y) = game.apple
146         pygame.draw.circle(screen, "brown3", (cell_x + x * CELL_SIDE +
    ↪   CELL_SIDE / 2, cell_y + y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE
    ↪   / 2)
147
148         # zoom on longest snake
149         game, current_snake = game_collection.longest_snake() # to see the
    ↪   longest snake
```

```
150     row = current_snake // games_per_side
151     col = current_snake % games_per_side
152     cell_x = col * GAME_WIDTH
153     cell_y = row * GAME_HEIGHT
154     # 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
155     pygame.draw.rect(screen, "yellow", (cell_x, cell_y, c.ZOOM_FACTOR *
    ↪ GAME_WIDTH, c.ZOOM_FACTOR * GAME_HEIGHT))
156     for (x, y) in game.snake_body:
157         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)
158     (x, y) = game.snake_body[0]
159     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)
160     (x, y) = game.apple
161     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)
162 else:
163     print(info)
164
165
166 # update your game state here
167 if not game_collection.step(c.LIFE_TIME): # all sakes in collection dead go next
    ↪ iteration
168     max_fitness.append(cur_max_fitness)
169     min_fitness.append(cur_min_fitness)
170     avg_fitness.append(cur_avg_fitness)
171     max_apple_eaten.append(cur_max_apple_eaten)
172     min_apple_eaten.append(cur_min_apple_eaten)
173     avg_apple_eaten.append(cur_avg_apple_eaten)
174     # plot max_fitness as function of 0:iteration
175     iteration += 1
176     if iteration >= c.MAX_ITERATION:
177         break
178
179 if c.DISPLAY_GRAPHICS:
180     # flip() the display to put your work on screen
181     pygame.display.flip()
182
183     clock.tick(500)
184
185 if c.SAVE:
186     game_collection.save_brains(c.BRAINS_FILE)
187     save_curves(c.CURVES_FILES)
188
189 print(max_fitness)
190
191 fig, ax1 = plt.subplots()
192
```

```
193 color1 = 'tab:blue'
194 color2 = 'tab:red'
195 color3 = 'tab:green'
196 color4 = 'tab:orange'
197
198 ax1.set_xlabel('Génération')
199 ax1.set_ylabel('Fitness maximum', color=color1)
200 ax1.set_yscale('log')
201
202 # Key change: Use iterations as the x-axis data
203 ax1.plot(range(len(max_fitness)), max_fitness, color=color2, label='Fitness max')
204 ax1.plot(range(len(avg_fitness)), avg_fitness, color=color1, label='Fitness avg')
205 ax1.tick_params(axis='y', labelcolor=color1)
206
207 ax1.legend(loc='upper left') # Add a legend for clarity
208
209 color3 = 'tab:green'
210 ax2 = ax1.twinx()
211 ax2.set_ylabel('Pommes mangées maximum', color=color3)
212 # Key change: Use iterations as the x-axis data
213 ax2.plot(range(len(max_apple_eaten)), max_apple_eaten, color=color4, label='Pommes')
214 ax2.tick_params(axis='y', labelcolor=color3)
215
216 ax2.legend(loc='lower right')
217
218 # Add Vertical Gridlines (The Key Change)
219 ax1.grid(axis='x', linestyle='--') # Gridlines on the x-axis (iterations)
220 ax2.grid(axis='y', linestyle='--') # You need to add it for the second axis too
221
222 # Additional styling improvement
223 plt.title('Fitness et pommes mangées fct. nombre de générations')
224 fig.tight_layout()
225
226 plt.show()
227
228 if c.DISPLAY_GRAPHICS:
229     pygame.quit()
230
```

3 Game

```
1 # un jeu = un seul serpent
2
3 from random import randrange
4 from neural_network import NeuralNetwork
5 from numpy import argmax
6 import collections
7 import config as c
8 from typing import Tuple, List
9 import math
10
11 class Game:
12
```

```
13 vision = []
14
15 def __init__(self, width: int = 10, height: int = 10, max_life_points: int = 50,
16 ↪ apple_lifetime_gain: int = 500, strategy: int = 2, num_fitness: int = 1) ->
17 ↪ None:
18     self.width = width
19     self.height = height
20     self.max_life_points = max_life_points
21     self.apple_lifetime_gain = apple_lifetime_gain
22     self.strategy = strategy
23     self.last_space = 0
24     self.last_visited = set()
25
26     """
27     Various rules to create a neural network:
28     * The number of hidden neurons should be between the size of the input layer
29 ↪ and the size of the output layer.
30     * The number of hidden neurons should be 2/3 the size of the input layer,
31 ↪ plus the size of the output layer.
32     * The number of hidden neurons should be less than twice the size of the
33 ↪ input layer.
34     * The number of hidden neurons should be between the size of the input layer
35 ↪ and the output layer.
36     * The most appropriate number of hidden neurons is sqrt(input layer nodes *
37 ↪ output layer nodes)
38     """
39
40     if strategy == 1:
41         # Neural network composed of 4 layers, input layer has 24 neurons, 2
42         ↪ hidden layers each with 18 neurons, output layer has 4 neurons (4
43         ↪ directions)
44         # in total it has 24 + 18 + 18 + 4 = 64 neurons.
45         self.brain = NeuralNetwork([24, 18, 18, 4])
46         self.vision_strategy = self.process_vision
47     elif strategy == 2:
48         self.brain = NeuralNetwork([9, 10, 10, 4])
49         self.vision_strategy = self.process_vision2
50     elif strategy == 3:
51         self.brain = NeuralNetwork([13, 12, 12, 4])
52         self.vision_strategy = self.process_vision3
53     elif strategy == 4:
54         self.brain = NeuralNetwork([25, 18, 18, 4])
55         self.vision_strategy = self.process_vision4
56     elif strategy == 5:
57         self.brain = NeuralNetwork([13, 12, 12, 4])
58         self.vision_strategy = self.process_vision5
59
60     self.age = 0
61     self.lost = False
62     self.apples_eaten = 0
63     #self.direction = (-1, 0) # default direction is left for first move
64     self.direction = (randrange(-1, 2), randrange(-1, 2)) # make first move
65     ↪ random
```



```
56     self.snake_body = [ # snake starts at the center and has 3 bits
57         (int(width / 2), int(height / 2))
58     ]
59     if c.ORIGINAL_SIZE_THREE:
60         self.snake_body.append((int(width / 2) + 1, int(height / 2)))
61         self.snake_body.append((int(width / 2) + 2, int(height / 2))
62     )
63     self.original_size = len(self.snake_body)
64     self.seed_new_apple()
65     self.life_points = self.max_life_points
66     self.died_bc_no_apple = 0
67     self.death_reason = "None"
68     if c.NORMALIZE_BOARD:
69         self.norm_constant_diag = math.sqrt(width ** 2 + height ** 2)
70         self.norm_constant_board = width * height / 10.0
71     else:
72         self.norm_constant_diag = 1
73         self.norm_constant_board = 20.0
74
75     if num_fitness == 1:
76         self.fitness = self.fitness1
77     elif num_fitness == 2:
78         self.fitness = self.fitness2
79     elif num_fitness == 3:
80         self.fitness = self.fitness3
81     elif num_fitness == 4:
82         self.fitness = self.fitness4
83     elif num_fitness == 5:
84         self.fitness = self.fitness5
85
86     def seed_new_apple(self):
87         self.apple = (randrange(0, self.width), randrange(0, self.height))
88         while self.apple in self.snake_body:
89             self.apple = (randrange(0, self.width), randrange(0, self.height))
90
91     def step(self, life_time: bool) -> bool:
92         # process the vision output through the neural network and output activation
93         activation = self.brain.feedforward(self.vision_strategy())
94         # take the highest activation index for the direction to take
95         index = argmax(activation)
96
97         match index:
98             case 0:
99                 self.direction = c.right
100             case 1:
101                 self.direction = c.up
102             case 2:
103                 self.direction = c.left
104             case 3:
105                 self.direction = c.down
106
107         return self.move_snake(self.direction, life_time)
108
```

```
109     def move_snake(self, incrementer: Tuple[int, int], life_time: bool) -> bool:
110         moved_head = (self.snake_body[0][0] + incrementer[0], self.snake_body[0][1] +
111             ↪ incrementer[1])
112
113         # vérification de la présence de la tête dans la grille
114         if not (0 <= moved_head[0] < self.width and 0 <= moved_head[1] <
115             ↪ self.height):
116             self.death_reason = "Wall"
117             self.lost = True
118             return False
119
120         # sauvegarde de la fin de la queue
121         end_tail = self.snake_body[-1]
122
123         # déplacement du serpent
124         for i in reversed(range(1, len(self.snake_body))):
125             self.snake_body[i] = self.snake_body[i - 1]
126
127         self.snake_body[0] = moved_head
128
129         # collisions avec le corps
130         for bit in self.snake_body[1:]:
131             if bit == self.snake_body[0]:
132                 self.lost = True
133                 self.death_reason = "Body"
134                 return False
135
136         self.age += 1
137         self.life_points -= 1
138
139         # collisions avec la pomme
140         if self.snake_body[0] == self.apple:
141             self.snake_body.append(end_tail) # agrandir le serpent avec la queue
142             ↪ précédente
143             self.seed_new_apple()
144             self.apples_eaten += 1
145             if c.RESET_LIFETIME:
146                 self.life_points = self.max_life_points # on réinitialise la durée de
147                 ↪ vie au max
148             else:
149                 self.life_points += self.apple_lifetime_gain # on réinitialise la
150                 ↪ durée de vie conformément au commentaire en dessous:
151             # optimize not to recalculate last_visited and last_space for strategy 2
152             # if moved_head is in last_visited it needs to be removed since the snake
153             ↪ has its head there now
154             if self.strategy == 2 or self.strategy == 5: # update last_visited and
155             ↪ last_space
156                 if moved_head in self.last_visited: # adapt last_visited and
157                 ↪ last_space
158                     self.last_visited.remove(moved_head) # only head is to be removed
159                     ↪ since tail not moved with apple eaten
160                     self.last_space -= 1
161                 else: # reset last_visited and last_space
```

```
153         self.last_space = 0
154         self.last_visited = set()
155     else:
156         # optimize not to recalculate last_visited and last_space for strategy 2
157         if self.strategy == 2 or self.strategy == 5: # update last_visited and
158             ↪ last_space
159             if moved_head in self.last_visited: # adapt last_visited and
160                 ↪ last_space
161                 self.last_visited.remove(moved_head) # only head is to be removed
162                 ↪ since tail not moved with apple eaten
163                 self.last_space -= 1
164                 # check if end_tail is connected to last_visited elements (can be
165                 ↪ visited) since it has moved and leaves an empty space
166                 if any(abs(end_tail[0] - x) == 1 ^ abs(end_tail[1] - y) == 1 for
167                     ↪ (x, y) in self.last_visited):
168                     self.last_visited.add((end_tail[0], end_tail[1]))
169                     self.last_space += 1
170             else: # reset last_visited and last_space
171                 self.last_space = 0
172                 self.last_visited = set()
173
174     # vérification de la durée de vie
175     if life_time and self.life_points <= 0:
176         self.death_reason = "Life"
177         self.lost = True
178         self.died_bc_no_apple = 1
179         return False
180
181     return True
182
183     # vision strategy: 8 directions, 3 informations per direction
184     # (1D distance to apple in direction of move, 1 / wall_distance in direction of
185     ↪ move, tail_distance in direction of move) + apples_eaten + original_size
186     def process_vision(self) -> List[float]:
187         vision = [0 for _ in range(3*8)]
188
189         for (i, incrementer) in enumerate(c.eight_directions):
190             apple_distance = -1
191             wall_distance = -1
192             tail_distance = -1
193
194             (x, y) = self.snake_body[0]
195             distance = 0
196
197             while True:
198                 x += incrementer[0]
199                 y += incrementer[1]
200                 distance += 1
201
202                 # sortie de grille
203                 if not self.is_on_board(x, y):
204                     wall_distance = distance
205                     break
```

```
200
201     # sur la pomme
202     if (x, y) == self.apple and apple_distance == -1:
203         apple_distance = distance
204
205     # sur la queue
206     if (x, y) in self.snake_body and tail_distance == -1:
207         tail_distance = distance
208
209     vision[3*i] = 0 if apple_distance == -1 else 1
210     vision[3*i + 1] = 1 / wall_distance
211     vision[3*i + 2] = tail_distance if tail_distance != -1 else 0
212
213     self.vision = vision
214     return vision
215
216 # vision strategy: 4 directions, 3 informations per direction
217 # (manhattan distance to apple, 1 / wall_distance in direction of move,
218   ↪ tail_distance in direction of move) + apples_eaten + original_size
219 def process_vision3(self) -> List[float]:
220     vision = []
221
222     for (i, incrementer) in enumerate(c.four_directions):
223         apple_distance = -1
224         wall_distance = -1
225         tail_distance = -1
226
227         (x, y) = self.snake_body[0]
228         distance = 0
229
230         # try to get inputs between [0,1] for the neural network
231
232         distance_apple = self.manhattan_distance_to_apple((x + incrementer[0], y
233   ↪ + incrementer[1]))
234
235         vision.append(1.0 / distance_apple if distance_apple != 0 else 1)
236
237     while True:
238         x += incrementer[0]
239         y += incrementer[1]
240         distance += 1
241
242         # sortie de grille
243         if not self.is_on_board(x, y):
244             wall_distance = distance
245             break
246
247         # sur la queue
248         if (x, y) in self.snake_body and tail_distance == -1:
249             tail_distance = distance
250
251     vision.append(1.0 / wall_distance)
252     vision.append(1.0 / tail_distance if tail_distance != -1 else 1)
```

```
251
252     vision.append(1 / (self.apples_eaten + self.original_size))
253
254     self.vision = vision
255     return vision
256
257 # vision strategy: 4 directions, 3 informations per direction
258 # (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
259 def process_vision4(self) -> List[float]:
260     vision = []
261
262     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]))))
263
264     for (i, incrementer) in enumerate(c.eight_directions):
265         apple_distance = -1
266         wall_distance = -1
267         tail_distance = -1
268
269         (x, y) = self.snake_body[0]
270         distance = 0
271
272         while True:
273             x += incrementer[0]
274             y += incrementer[1]
275             distance += 1
276
277             # sortie de grille
278             if not self.is_on_board(x, y):
279                 wall_distance = distance
280                 break
281
282             # sur la queue
283             if (x, y) in self.snake_body and tail_distance == -1:
284                 tail_distance = distance
285
286             vision.append(1 if i == min_distance_index else 0)
287             vision.append(1.0 / wall_distance)
288             vision.append(tail_distance if tail_distance != -1 else 0)
289
290         vision.append(self.apples_eaten + self.original_size)
291         self.vision = vision
292         return vision
293
294     #? weights 8 bits vs. float? normalization?
295
296 # vision strategy: 4 directions, 3 informations per direction
297 # (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
```

```
298 def process_vision5(self) -> List[float]:
299     # neural network input contains free space in all directions, distance to
    ↪ apple in all directions, and number of apples eaten (size of snake)
300     # 9 inputs in total
301     neural_network_input = []
302     (hx, hy) = self.snake_body[0] # head of the snake body
303     for direction in c.four_directions:
304         (dx, dy) = direction
305         (cnx, cny) = (hx + dx, hy + dy)
306         #metric = self.count_free_moving_spaces(cnx, cny)
307         #neural_network_input.append(1.0 / metric if metric != 0 else 1)
308         #metric = self.manhattan_distance_to_apple((cnx, cny))
309         #neural_network_input.append(1.0 / metric if metric != 0 else 1)
310         neural_network_input.append(self.count_free_moving_spaces(cnx, cny) /
    ↪ self.norm_constant_board)
311         neural_network_input.append(self.manhattan_distance_to_apple((cnx, cny))
    ↪ / self.norm_constant_diag)
312         neural_network_input.append(1 if self.apple in self.last_visited else 0)
    ↪ # apple can be reached going in this direction
313     #neural_network_input.append(1.0 / (self.apples_eaten + self.original_size))
314     neural_network_input.append(self.apples_eaten + self.original_size)
315     self.vision = neural_network_input
316     return neural_network_input
317
318 # vision strategy: 4 directions, 2 informations per direction
319 # (free spaces in direction of move, manhattan distance to apple in direction of
    ↪ move) + apples_eaten + original_size
320 def process_vision2(self) -> List[float]:
321     # neural network input contains free space in all directions, distance to
    ↪ apple in all directions, and number of apples eaten (size of snake)
322     # 9 inputs in total
323     neural_network_input = []
324     (hx, hy) = self.snake_body[0] # head of the snake body
325     for direction in c.four_directions:
326         (dx, dy) = direction
327         (cnx, cny) = (hx + dx, hy + dy)
328         #metric = self.count_free_moving_spaces(cnx, cny)
329         #neural_network_input.append(1.0 / metric if metric != 0 else 1)
330         #metric = self.manhattan_distance_to_apple((cnx, cny))
331         #neural_network_input.append(1.0 / metric if metric != 0 else 1)
332         neural_network_input.append(self.count_free_moving_spaces(cnx, cny) /
    ↪ self.norm_constant_board)
333         neural_network_input.append(self.manhattan_distance_to_apple((cnx, cny))
    ↪ / self.norm_constant_diag)
334     #neural_network_input.append(1.0 / (self.apples_eaten + self.original_size))
335     neural_network_input.append(self.apples_eaten + self.original_size)
336     self.vision = neural_network_input
337     return neural_network_input
338
339 def is_on_board(self, x, y) -> bool:
340     return 0 <= x < self.width and 0 <= y < self.height
341
342 def is_possible_move(self, x, y) -> bool:
```

```
343         # check if the move is on the board and not on the snake body except for the
344         ↪ tail (since it has moved)
345         return self.is_on_board(x, y) and (x, y) not in self.snake_body[:-1]
346
347     def get_possible_moves(self, cur):
348         (x, y) = cur
349         moves = []
350         for direction in c.eight_directions:
351             (i, j) = direction
352             if self.is_possible_move(x + i, y + j):
353                 moves.append(direction)
354         return moves
355
356     def count_free_moving_spaces(self, x, y) -> int:
357         # Breadth-First Search, BFS, snake heads moves to (x, y) and tail's end is no
358         ↪ more
359         if not self.is_possible_move(x, y): # does not check snake's tail
360             return 0
361         if (x, y) in self.last_visited:
362             return self.last_space
363         space = 0
364         visited = set([(x, y)])
365         queue = collections.deque([(x, y)]) # efficient for pop(0) and append
366         while (len(queue) > 0):
367             cur = queue.popleft()
368             space += 1
369             for direction in self.get_possible_moves(cur):
370                 (i, j) = direction
371                 (cx, cy) = cur
372                 cn = (cx + i, cy + j)
373                 (cnx, cny) = cn
374                 if cn not in visited and self.is_possible_move(cnx, cny): # does not
375                     ↪ check snake's tail
376                     queue.append(cn)
377                     visited.add(cn)
378             self.last_visited = visited
379             self.last_space = space
380         return space
381
382     def manhattan_distance_to_apple(self, head):
383         return abs(self.apple[0] - head[0]) + abs(self.apple[1] - head[1])
384
385     def fitness1(self):
386         return pow(3, self.apples_eaten) * (self.age - c.MAX_LIFE_POINTS *
387             ↪ self.died_bc_no_apple)
388
389     def fitness2(self):
390         return (self.apples_eaten ** 3) * (self.age - c.MAX_LIFE_POINTS *
391             ↪ self.died_bc_no_apple)
392
393     def fitness3(self):
394         return ((self.apples_eaten * 2) ** 2) * ((self.age - c.MAX_LIFE_POINTS *
395             ↪ self.died_bc_no_apple) ** 1.5)
```

```
390
391 def fitness4(self):
392     return (self.age * self.age) * pow(2, self.apples_eaten) * (100 *
        ↪ self.apples_eaten + 1)
393
394 def fitness5(self):
395     return (self.age * self.age * self.age * self.age) * pow(2,
        ↪ self.apples_eaten) * (500 * self.apples_eaten + 1)
396
397 # age^2*2^apple*(coeff*apple+1)
398 # age^2*2^10*(apple-9)*(coeff*10)
399
400 # score = self.apples_eaten, frame_score = self.age
401 # ((score^3)*(frame_score)
402 # ((score*2)^2)*(frame_score^1.5)
403
404 # remarks
405 # * 3^apple*(age): pow(3, self.apples_eaten) * (self.age - 50 *
        ↪ self.died_bc_no_apple) trains faster
406
```

4 Game Collection

```
1 from game import Game
2 from genetic_algorithm import GeneticAlgorithm
3 import pickle
4 import config as c
5 import math
6 from typing import List, Tuple
7
8 class GameCollection:
9     games = []
10    ga = GeneticAlgorithm(math.ceil(c.PORION_BESTS * c.POPULATION / 100),
        ↪ c.NUMBER_CROSSOVER_POINTS, c.MUTATION_CHANCE, c.MUTATION_COEFF)
11    iteration = 0
12    generation = 1
13
14    def __init__(self, number_games:int, width:int, height:int) -> None:
15        self.games = [Game(width, height, c.MAX_LIFE_POINTS, c.APPLE_LIFETIME_GAIN,
        ↪ c.GAME_STRATEGY, c.FITNESS_STRATEGY) for _ in range(number_games)]
16
17    def snake_to_display(self) -> Tuple[Game, int]:
18        for i in range(len(self.games)):
19            if not self.games[i].lost:
20                return self.games[i], i
21        return self.games[0], 0
22
23    def longest_snake(self) -> Tuple[Game, int]:
24        longest = 0
25        index = 0
26        for i in range(len(self.games)):
27            if len(self.games[i].snake_body) > longest:
28                longest = len(self.games[i].snake_body)
```



```
29         index = i
30         return self.games[index], index
31
32     def step(self, life_time: bool) -> bool:
33
34         self.iteration += 1
35
36         one_game_not_lost = False
37
38         for game in self.games:
39             if not game.lost:
40                 one_game_not_lost = True
41                 game.step(life_time)
42
43         # if all games are lost, evolve
44         if not one_game_not_lost:
45             self.evolve()
46         return one_game_not_lost
47
48     def evolve(self):
49
50         new_population = self.ga.evolve([
51             (game.brain, game.fitness())
52             for game in self.games
53         ])
54
55         width, height = self.games[0].width, self.games[0].height
56
57         for i in range(len(new_population)):
58             g = Game(width, height, c.MAX_LIFE_POINTS, c.APPLE_LIFETIME_GAIN,
59                 ↪ c.GAME_STRATEGY, c.FITNESS_STRATEGY) # create new game
60             g.brain = new_population[i] # inject brain in game
61             self.games[i] = g # replace current game with new one
62
63         self.iteration = 0
64         self.generation += 1
65
66     def save_brains(self, filename):
67         # save the game collection and all the games in the game collection to a file
68         #for game in self.games:
69         #    print(game.brain.layers_sizes)
70         new_games = sorted(self.games, key=lambda game: game.fitness())
71         game_brains = [game.brain for game in new_games]
72         if c.DEBUG:
73             for brain in game_brains:
74                 print(brain.weights, end=' ')
75                 print()
76         print("save_brains: len(game_brains): ", len(game_brains))
77         with open(filename, 'wb') as f:
78             pickle.dump(game_brains, f)
79
80     def restore_brains(self, filename):
81         with open(filename, 'rb') as f:
```

```
81         game_brains = pickle.load(f)
82         print("restore_brains: len(game_brains): ", len(game_brains))
83         for i in range(len(self.games)):
84             self.games[i].brain = game_brains[i]
85         if c.DEBUG:
86             for brain in game_brains:
87                 print(brain.weights, end=' ')
88             print()
89
90     def save_to_file(self, filename):
91         with open(filename, 'wb') as f:
92             pickle.dump(self, f)
93
94     @classmethod
95     def load_from_file(cls, filename):
96         with open(filename, 'rb') as f:
97             return pickle.load(f)
98
99     def best_fitness(self):
100         return max(game.fitness() for game in self.games)
101
102     def worst_fitness(self):
103         return min(game.fitness() for game in self.games)
104
105     def average_fitness(self):
106         return sum(game.fitness() for game in self.games) / len(self.games)
107
108     def max_apple_eaten(self):
109         return max(game.apples_eaten for game in self.games)
110
111     def min_apple_eaten(self):
112         return min(game.apples_eaten for game in self.games)
113
114     def average_apple_eaten(self):
115         return sum(game.apples_eaten for game in self.games) / len(self.games)
116
```

5 Genetic Algorithm

```
1  import numpy as np
2  from neural_network import NeuralNetwork
3  from typing import List, Tuple
4  import copy
5
6  class GeneticAlgorithm:
7
8      def __init__(self, save_bests: int = 10, k: int = 5, mut_chance: float = 0.5,
9          ↪ coeff: float = 0.5) -> None:
10         self.save_bests = save_bests
11         self.k = k
12         self.mut_chance = mut_chance
13         self.coeff = coeff
```

```
14     def select_parent(self, population: List[Tuple[NeuralNetwork, int]]) ->
    ↪ Tuple[NeuralNetwork, NeuralNetwork]:
15         # Roulette-wheel selection: numpy.random.choice
16         maxi = sum([x[1] for x in population])
17         selection_probability = [x[1] / maxi for x in population]
18         parent1, parent2 = np.random.choice(len(population),
    ↪ p=selection_probability), np.random.choice(len(population),
    ↪ p=selection_probability)
19         return population[parent1][0], population[parent2][0]
20
21     def crossover(self, parent_a: List[float], parent_b: List[float]) ->
    ↪ List[float]:
22         """
23         K-point crossover cf Wikipedia:
24         - select k random points in range(len(parent_a))
25         - create a new array which alternate between coefficients of parent_a and
    ↪ parent_b
26         """
27         n = len(parent_a)
28         # list of crossover points
29         l = sorted([np.random.randint(0, n) for _ in range(self.k)]) # to avoid
    ↪ having two times the same index
30         l.append(-1) # to avoid index out of range but never used
31         child = []
32         current_parent = 0
33         current_index = 0
34         for i in range(n):
35             if i == l[current_index]:
36                 current_parent = 1 - current_parent
37                 current_index += 1
38             if current_parent == 0:
39                 child.append(parent_a[i])
40             else:
41                 child.append(parent_b[i])
42         return child
43
44     def mutate(self, genome: List[float]) -> None:
45         """
46         Gaussian mutation:
47         - for each coefficient:
48             - if random() <= mutation chance (paramètre réglé):
49                 - generate a sign at random
50                 - generate an amplitude (between 0 and 1)
51                 - add sign * amplitude * coeff to the coefficient (coeff is a
    ↪ parameter)
52         """
53         for i in range(len(genome)):
54             if np.random.random() <= self.mut_chance:
55                 sign = 1 if np.random.random() <= 0.5 else -1
56                 amplitude = np.random.random()
57                 genome[i] += sign * amplitude * self.coeff
58
59     def evolve(self, population: Tuple[NeuralNetwork, int]) -> list:
```

```
60     assert(len(population) != 0)
61     new_population = []
62     # sélection des meilleurs
63     population.sort(key=lambda x : x[1], reverse=True)
64     for i in range(len(population)):
65         if i < self.save_bests:
66             new_population.append(copy.deepcopy(population[i][0])) # to avoid
67             ↪ reference
68         else:
69             parent_a, parent_b = self.select_parent(population)
70             child = self.crossover(parent_a.to_genome(), parent_b.to_genome())
71             self.mutate(child)
72             new_population.append(NeuralNetwork.from_genome(child,
73             ↪ population[i][0].layers_sizes))
74     return new_population
```

6 Neural Network

```
1  import numpy as np
2  from typing import List
3
4  def sigmoid(x):
5      return 1.0/(1.0 + np.exp(-x))
6
7  class NeuralNetwork:
8
9      layers_sizes = []
10     weights = []
11     biases = []
12     activation_function = None
13
14     def __init__(self, layers_sizes:List[int]) -> None:
15         self.biases = [np.random.randn(i, 1) for i in layers_sizes[1:]]
16         self.weights = [np.random.randn(i, j) for (i, j) in zip(layers_sizes[1:],
17         ↪ layers_sizes[:-1])]
18         self.activation_function = sigmoid
19         self.layers_sizes = layers_sizes
20
21     def feedforward(self, activation):
22         for w, b in zip(self.weights, self.biases):
23             activation = self.activation_function(np.dot(w, activation) + b)
24         return activation
25
26     """
27     def to_genome(self) -> List[float]:
28         genome = []
29         for w in self.weights:
30             for line in w:
31                 for c in line:
32                     genome.append(c)
33         for b in self.biases:
34             for c in b:
```

```
34         genome.append(c)
35     return genome
36     """
37
38     def to_genome(self) -> List[float]:
39         genome = np.concatenate([w.flatten() for w in self.weights] + [b.flatten()
40             ↪ for b in self.biases])
41         return genome.tolist()
42
43     @classmethod
44     def from_genome(cls, genome: List[float], layers: List[int]):
45         assert len(layers) > 0
46         nn = cls(layers)
47         # this code is more efficient than the commented code below because it avoids
48         ↪ the list inversions
49         offset = 0
50         for i, (j, k) in enumerate(zip(layers[:-1], layers[1:])):
51             nn.weights[i] = np.reshape(genome[offset:offset + j * k], (k, j))
52             offset += j * k
53         for i, k in enumerate(layers[1:]):
54             nn.biases[i] = np.reshape(genome[offset:offset + k], (k, 1))
55             offset += k
56         """
57         genome = list(reversed(genome))
58         nn.weights = [np.array([genome.pop() for _ in range(j)] for _ in range(i))]
59         ↪ for (i, j) in zip(nn.layers_sizes[1:], nn.layers_sizes[:-1])]
60         nn.biases = [np.array([genome.pop() for _ in range(i)]) for i in
61             ↪ nn.layers_sizes[1:]]
62         """
63         return nn
```

7 Courbes

```
1  import os
2  import pickle
3  from game import Game
4  import config as c
5  import matplotlib.pyplot as plt
6  import numpy as np
7  import matplotlib.gridspec as gridspec
8  import matplotlib.animation as animation
9
10 def restore_brain(brain_number: int) -> Game:
11     # restore brain from file and inject it into the snake
12     with open("brains_53.pickle", 'rb') as f:
13         game_brains = pickle.load(f)
14         brain = game_brains[brain_number]
15         if c.DEBUG:
16             print(game.brain, end=' ')
17             print()
18     return brain
19
```

```
20 def visualize_neural_network(brain):
21     fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 8))
22     fig.suptitle("Visualisation réseau de neurones", fontsize=16)
23     for i in range(3):
24         visualize_matrix(brain.weights[i], f"Poids synaptiques - Couche {i+1}",
25                          ↪ axes[0, i])
26         visualize_matrix(brain.biases[i], f"Biais - Couche {i+1}", axes[1, i])
27     plt.tight_layout(rect=[0, 0.03, 1, 0.95])
28     plt.savefig("brain_matrix.svg")
29     plt.savefig("brain_matrix.eps")
30     plt.savefig("brain_matrix.pdf")
31     plt.savefig("brain_matrix.png")
32     plt.show()
33
34 def visualize_matrix(matrix, title, ax=None):
35     if ax is None:
36         ax = plt.gca() # Get the current axes if not provided
37     im = ax.imshow(matrix, cmap='viridis', interpolation='nearest')
38     plt.colorbar(im, ax=ax, label='Valeur du poids synaptique')
39     ax.set_xlabel('Index du neurone en entrée')
40     ax.set_ylabel('Index du neurone en sortie')
41     ax.set_title(title)
42     # Setting tick parameters
43     ax.tick_params(axis='both', which='major', labelsize=6)
44     ax.set_xticks(range(matrix.shape[1]))
45     ax.set_yticks(range(matrix.shape[0]))
46
47 def visualize_neural_network2(brain, fig, axes):
48     fig.suptitle("Visualisation réseau de neurones", fontsize=16)
49     for i in range(3):
50         visualize_matrix(brain.weights[i], f"Poids synaptiques - Couche {i+1}",
51                          ↪ axes[0, i])
52         visualize_matrix(brain.biases[i], f"Biais - Couche {i+1}", axes[1, i])
53
54 def update_visualization(i):
55     brain = restore_brain(i)
56     visualize_neural_network2(brain, fig, axes)
57
58 brain = restore_brain(c.SINGLE_SNAKE_BRAIN)
59 # brain has layers_sizes = [] weights = [] biases = []
60
61 for i, (w, b) in enumerate(zip(brain.weights, brain.biases)):
62     print(f"Layer {i+1}:")
63     print(f"  Weights: {w.shape}")
64     print(f"  Biases: {b.shape}")
65
66 visualize_neural_network(brain)
67
68 # do an animation of the brain matrices
69
70 fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 8))
71 fig.suptitle("Visualisation réseau de neurones", fontsize=16)
```

```
71 ims = [] # List to store the animation frames (heatmaps)
72 for i in range(256):
73     brain = restore_brain(i)
74     frames = []
75     for j in range(3):
76         frame1 = axes[0, j].imshow(brain.weights[j], cmap='viridis',
77                                     ↪ interpolation='nearest', animated=True)
77         frame2 = axes[1, j].imshow(brain.biases[j], cmap='viridis',
78                                     ↪ interpolation='nearest', animated=True)
78         frames.extend([frame1, frame2])
79     ims.append(frames) # Add frames for the current brain to the list
80
81 ani = animation.ArtistAnimation(fig, ims, interval=500, blit=True, repeat_delay=1000)
82 plt.show()

1 import pickle
2 import matplotlib.pyplot as plt
3
4 def restore_curves(filename):
5     with open(filename, 'rb') as f:
6         data = pickle.load(f)
7     return data
8
9 max_iterations = 256
10
11 (max_fitness5, min_fitness5, avg_fitness5, max_apple_eaten5, min_apple_eaten5,
12  ↪ avg_apple_eaten5, max_snake_length5) = restore_curves("curve_53.pickle")
13 (max_fitness1, min_fitness1, avg_fitness1, max_apple_eaten1, min_apple_eaten1,
14  ↪ avg_apple_eaten1, max_snake_length1) = restore_curves("curve_13.pickle")
15
16 fig, ax1 = plt.subplots()
17
18 color1 = 'tab:blue'
19 color2 = 'tab:red'
20 color3 = 'tab:green'
21 color4 = 'tab:orange'
22
23 ax1.set_xlabel('Génération')
24 ax1.set_ylabel('Fitness maximum', color=color1)
25 ax1.set_yscale('log')
26 # Key change: Use iterations as the x-axis data
27 ax1.plot(range(1, max_iterations + 1), max_fitness1[1:max_iterations + 1],
28         ↪ color=color2, label='Fitness strat 1')
29 ax1.plot(range(1, max_iterations + 1), max_fitness5[1:max_iterations + 1],
30         ↪ color=color1, label='Fitness strat 2')
31 ax1.tick_params(axis='y', labelcolor=color1)
32
33 ax1.legend(loc='upper left') # Add a legend for clarity
34
35 color3 = 'tab:green'
36 ax2 = ax1.twinx()
37 ax2.set_ylabel('Pommes mangées maximum', color=color3)
38 # Key change: Use iterations as the x-axis data
```

```
35 ax2.plot(range(1, max_iterations + 1), max_apple_eaten1[1:max_iterations + 1],
    ↪ color=color4, label='Pommes strat 1')
36 ax2.plot(range(1, max_iterations + 1), max_apple_eaten5[1:max_iterations + 1],
    ↪ color=color3, label='Pommes strat 2')
37 ax2.tick_params(axis='y', labelcolor=color3)
38
39 ax2.legend(loc='lower right')
40
41 # Add Vertical Gridlines (The Key Change)
42 ax1.grid(axis='x', linestyle='--') # Gridlines on the x-axis (iterations)
43 ax2.grid(axis='y', linestyle='--') # You need to add it for the second axis too
44
45 # Additional styling improvement
46 plt.title('Fitness et pommes mangées fct. nombre de générations')
47 fig.tight_layout()
48 plt.savefig("curve_compare_cv.svg")
49 plt.savefig("curve_compare_cv.eps")
50 plt.savefig("curve_compare_cv.pdf")
51 plt.savefig("curve_compare_cv.png")
52 plt.show()

1 import pickle
2 import matplotlib.pyplot as plt
3 import config as c
4
5 max_fitness = []
6 min_fitness = []
7 avg_fitness = []
8 max_apple_eaten = []
9 min_apple_eaten = []
10 avg_apple_eaten = []
11 max_snake_length = 0
12
13 def restore_curves(filename):
14     with open(filename, 'rb') as f:
15         data = pickle.load(f)
16     return data
17
18 (max_fitness, min_fitness, avg_fitness, max_apple_eaten, min_apple_eaten,
    ↪ avg_apple_eaten, max_snake_length) = restore_curves("curve.pickle")
19
20 fig, ax1 = plt.subplots()
21
22 color1 = 'tab:blue'
23 ax1.set_xlabel('Itération')
24 ax1.set_ylabel('Fitness maximum', color=color1)
25 ax1.set_yscale('log')
26 ax1.plot(range(len(max_fitness)), max_fitness, color=color1)
27 ax1.tick_params(axis='y', labelcolor=color1)
28
29 color3 = 'tab:green'
30 ax2 = ax1.twinx()
31 ax2.set_ylabel('Pommes mangées maximum', color=color3)
32 ax2.plot(range(len(max_apple_eaten)), max_apple_eaten, color=color3)
```



```
33 ax2.tick_params(axis='y', labelcolor=color3)
34
35 plt.title('Fitness vs Iteration')
36 # Add Vertical Gridlines (The Key Change)
37 ax1.grid(axis='x', linestyle='--') # Gridlines on the x-axis (iterations)
38 ax2.grid(axis='y', linestyle='--') # You need to add it for the second axis too
39 fig.tight_layout()
40 plt.savefig("curve.svg")
41 plt.savefig("curve.eps")
42 plt.savefig("curve.pdf")
43 plt.show()
```

8 Faire jouer le mmeilleur serpent

```
1 import pygame
2 import os
3 import pickle
4 from game import Game
5 import config as c
6 from PIL import Image
7
8 def restore_snake(brain_number: int) -> Game:
9     # restore brain from file and inject it into the snake
10    assert(os.path.exists(c.BRAINS_FILE))
11    game = Game(c.WIDTH, c.HEIGHT, c.MAX_LIFE_POINTS, c.APPLE_LIFETIME_GAIN,
12    ↪ c.GAME_STRATEGY, c.FITNESS_STRATEGY)
13    with open(c.BRAINS_FILE, 'rb') as f:
14        game_brains = pickle.load(f)
15        game.brain = game_brains[brain_number]
16        if c.DEBUG:
17            print(game.brain, end=' ')
18            print()
19    return game
20
21 game = restore_snake(c.SINGLE_SNAKE_BRAIN)
22
23 frames = []
24
25 # pygame setup
26 pygame.init()
27
28 # board contains one game/snake
29
30 #CELL_SIDE = c.BOARD_SIDE // max(c.WIDTH, c.HEIGHT)
31 CELL_SIDE = 10
32 GAME_WIDTH = CELL_SIDE * c.WIDTH
33 GAME_HEIGHT = CELL_SIDE * c.HEIGHT
34
35 screen = pygame.display.set_mode((GAME_WIDTH, GAME_HEIGHT))
36
37 clock = pygame.time.Clock()
38 running = True
39 dt = 0
```

```
39
40 iteration = 0
41
42 max_snake_length = 0
43
44 for n in range(c.PLAY_SNAKE_ITERATIONS):
45     while running:
46
47         iteration += 1
48
49         cur_fitness = game.fitness()
50         cur_apple_eaten = game.apples_eaten
51         if cur_apple_eaten >= max_snake_length:
52             max_snake_length = cur_apple_eaten + 1
53
54         # display game iteration and fitness of the game (generation) as window title
55         info = f"Iter {iteration} - Fitness {cur_fitness:.2e} - Eaten
56             ↪ {cur_apple_eaten} - Longest ever {max_snake_length}"
57
58         # poll for events
59         # pygame.QUIT event means the user clicked X to close your window
60         for event in pygame.event.get():
61             if event.type == pygame.QUIT:
62                 running = False
63
64         # fill the screen with a color to wipe away anything from last frame
65         screen.fill("white")
66
67         # draw grid
68         for x in range(0, GAME_WIDTH, CELL_SIDE):
69             pygame.draw.line(screen, "gray", (x, 0), (x, GAME_HEIGHT))
70         for y in range(0, GAME_HEIGHT, CELL_SIDE):
71             pygame.draw.line(screen, "gray", (0, y), (GAME_WIDTH, y))
72
73         pygame.display.set_caption(info)
74
75         for (x, y) in game.snake_body:
76             pygame.draw.circle(screen, "darkolivegreen3", (x * CELL_SIDE + CELL_SIDE
77                 ↪ / 2, y * CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
78         (x, y) = game.snake_body[0] # head of the snake
79         pygame.draw.circle(screen, "black", (x * CELL_SIDE + CELL_SIDE / 2, y *
80             ↪ CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 4)
81         (x, y) = game.apple
82         pygame.draw.circle(screen, "brown3", (x * CELL_SIDE + CELL_SIDE / 2, y *
83             ↪ CELL_SIDE + CELL_SIDE / 2), CELL_SIDE / 2)
84         # surround the current game with a black rectangle
85         pygame.draw.rect(screen, "black", (GAME_WIDTH, GAME_HEIGHT, GAME_WIDTH,
86             ↪ GAME_HEIGHT), 1)
87
88         # update your game state here (do not constrain snake life time)
89         if not game.step(False): # snake is dead
90             break;
91
92         # flip() the display to put your work on screen
93         pygame.display.flip()
94         frame_str = pygame.image.tostring(screen, "RGB")
```

```
87         frame_image = Image.frombytes("RGB", (GAME_WIDTH, GAME_HEIGHT), frame_str)
88         frames.append(frame_image)
89         clock.tick(25)
90     iteration = 0;
91     game = restore_snake(c.SINGLE_SNAKE_BRAIN)
92
93 frames[0].save("game_animation.gif", save_all=True, append_images=frames[1:],
94               ↪ duration=100, loop=0)
94 pygame.quit()
95
```