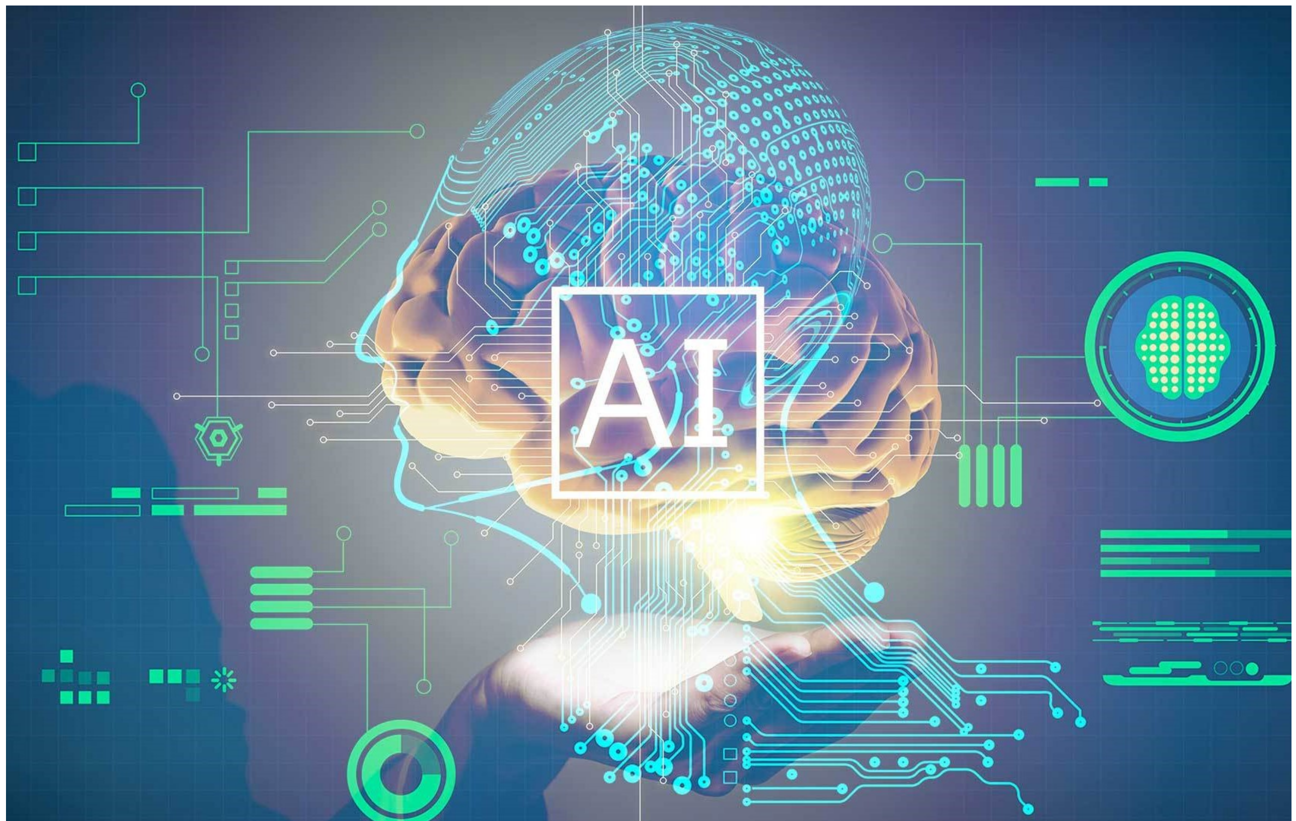
Gymnase de Burier

Programmation d'une intelligence artificielle qui apprend à jouer à un jeu vidéo



Lucas JUNG

Octobre 2019

Table des matières

Introduction **1**

1 Dots Ai **2**
1.1 Objectif . . . 3
1.2 Concept . . . 3
1.3 Apprentissage . . . 3
1.4 Concrètement dans le code . . . 4
1.4.1 Dot . . . 4
Initialisation . . . 4
Déplacement . . . 5
Calcul de la réussite . . . 6
Clone . . . 7
1.4.2 Population . . . 8
Initialisation . . . 8
Actions communes . . . 8
Le meilleur point . . . 9
Tous morts . . . 9
La sélection des candidats . . . 10
La sélection naturelle . . . 11
1.4.3 Brain . . . 12
Initialisation . . . 12
Randomize . . . 13
Mutation . . . 13
1.5 Affichage . . . 14
1.6 Initialisation du programme . . . 14
1.7 Boucle principale . . . 15
1.8 Conclusion . . . 16

2 Snake Ai **18**
2.1 Objectif . . . 19

2.2 Programmer le jeu 19
2.3 Fonctionnement d'un réseau neuronal artificiel 19
2.3.1 Concept et explications relatifs au réseau utilisé 19
2.3.2 Mathématiques 21
2.4 Apprentissage 22
2.5 Entrées du réseau de neurones 22
2.6 Affichage 24
2.7 Concrètement dans le code 25
2.7.1 Initialisation 25
Importation des modules 25
Variables globales 26
2.7.2 Classes mineures 26
Affichage 27
Carrés 28
2.7.3 Serpent 28
Initialisation 28
Déplacement 30
Changement de direction 31
Vérification 32
Nouveau fruit 33
Calcul des entrées 33
Calcul de la fitness 36
Sauvegarde d'un serpent 37
Suppression 38
2.7.4 Fruit 38
Initialisation 39
Génération des coordonnées du fruit 39
Suppression 39
2.7.5 Cerveau 40
Initialisation 40
Tangente hyperbolique et sigmoïde 41
Sortie 42
Mutation d'une matrice 43
Mutations 44
2.7.6 Population 44
Initialisation 44
Déplacements 45
Sont-ils tous morts ? 46

Affichage 46
Calcul des fitness 46
Croisement 47
Sélection naturelle 48
Suppression 50
2.7.7 Rejouer 50
2.7.8 Boucle principale 52
2.8 Choix de la forme du réseau 53
2.9 Programme de récupération des serpents 55

Conclusion **56**

Annexes **57**

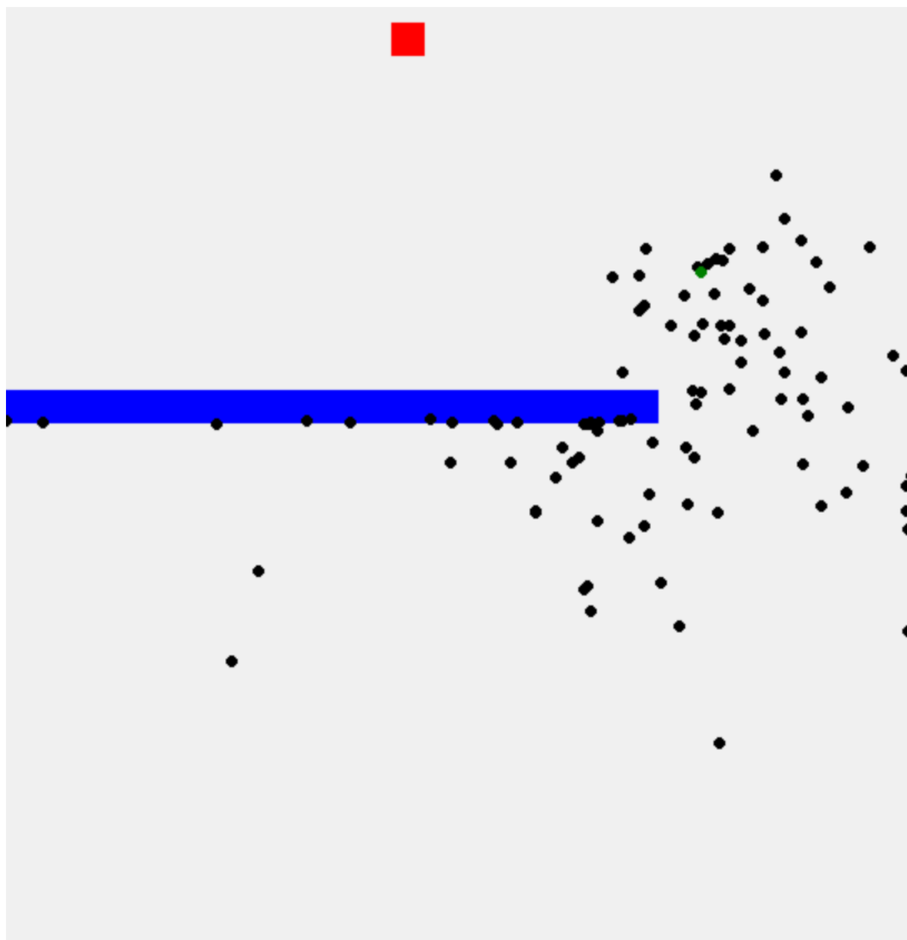
Bibliographie **58**

Introduction

Ce travail de maturité porte sur le sujet de l'intelligence artificielle et plus précisément sur la capacité d'un ordinateur à apprendre. Pour ce faire, j'ai créé deux programmes. Le premier m'a servi à me familiariser avec la programmation d'algorithmes de « machine learning » ; la signification de ces noms est détaillée plus tard dans ce dossier. Le second programme est celui au centre de ce travail de maturité : il s'agit de recréer le fameux jeu intitulé « Snake » et de laisser l'ordinateur apprendre à jouer par lui même. Ces deux programmes sont codés en Python (version 3.7) qui est un langage de programmation que je connais déjà. Ce dossier, quant à lui, est écrit en LaTeX qui est un langage qui m'était encore inconnu mais que j'ai appris au fil de ce travail. J'ajouterai enfin que pour comprendre ce travail dans son intégralité, la connaissance préalable du langage de programmation utilisé est nécessaire ; de même en ce qui concerne la multiplication de 2 matrices.

Chapitre 1

Dots Ai



1.1 Objectif

L'objectif que je souhaitais atteindre avec ce premier programme est le suivant : faire en sorte qu'un point apprenne à contourner un obstacle pour atteindre un but. C'est une idée de programme que j'ai reprise d'un youtubeur surnommé CodeBullet qui a réalisé ce programme en Java. J'ai repris cette idée car je n'avais jamais encore programmé de « machine learning » auparavant et que je souhaitais me familiariser avec cela avant d'attaquer mon véritable programme. C'est un algorithme assez simple qui est employé : l'algorithme génétique. Cet algorithme vise à reproduire l'évolution naturelle des êtres vivants. J'ai donc souhaité, avec ce programme intitulé dots_ai, faire évoluer une population de points à l'aide de cet algorithme programmé en Python.

1.2 Concept

Une population de points va être libérée depuis un lieu de départ défini à l'avance. Ces points devront se déplacer vers un objectif matérialisé à l'aide d'un carré rouge situé en haut de la fenêtre. Ils devront tout d'abord contourner un obstacle rectangulaire bleu pour arriver à leur but. S'ils touchent le bord de la fenêtre ou l'obstacle, ils « meurent » : on les voit s'arrêter.

1.3 Apprentissage

Lorsqu'on génère des points pour la première fois, une liste de vecteurs d'accélération, propre à chacun, leur est donnée de manière aléatoire. Les points ne suivront donc aucune logique dans leur déplacement. Une fois tous les points morts, l'algorithme génétique entre en jeu : chacun des points se verra attribuer un nombre qui définira à quel point il a atteint l'objectif. On appellera ça la « fitness ». Après cela viendra une phase de sélection naturelle durant laquelle les points seront sélectionnés pour former la génération suivante. Lors de cette phase un point ayant bien réussi a plus de chances de se retrouver dans la nouvelle génération qu'un point ayant moins bien réussi. Néanmoins, chaque point garde une chance non nulle d'être repris, ceci dans le but de garder une certaine mixité. Après cela viendra une phase de mutation : chaque point subira quelques légères mutations dans la liste de ses vecteurs d'accélération. On créera ainsi une génération entièrement nouvelle

ayant plus de chance de réussir que celle d'avant. Le processus se répétera autant de fois que voulu, chaque cycle représentant une nouvelle génération.

1.4 Concrètement dans le code

Concrètement, cela se matérialise dans le code sous forme de 3 classes : `Brain`, utilisée par `Dot`, ainsi que `Population`.

1.4.1 Dot

Cette classe est appelée lorsque l'on veut créer un nouveau point.

Initialisation

Au moment de la création d'un nouveau point, une série d'étapes d'initialisation s'exécutent.

```
1  def __init__(self):
2    self.pos = vector(300, 550)   # pops up at the bottom of the window
3    self.vel = vector(0, 0)    # starts velocity = 0
4    self.acc = vector(0, 0)    # starts acceleration = 0
5    self.brain = Brain(400)    # has a brain containing 400 acceleration vectors
6    self.canv = canv_elipse_create(self.pos[0], self.pos[1], 'black')  # shows it on the
   ↪  window
7    self.dead = False  # turns to True if it dies
8    self.reachgoal = False  # turns to True if it reaches the goal
9    self.isbest = False    # turns to True if it is the best of the generation
10   self.fitness = 0  # how good it is (how far he is from the goal)
```

On peut dès lors remarquer plusieurs points importants, notamment le fait qu'un point « naît » en bas de la fenêtre avec une vitesse et une accélération nulles (lignes 2 à 4). On peut aussi relever que dans sa séquence d'initialisation, le point fera appel à une autre classe : `Brain` (ligne 5). Cela lui fournira un « cerveau » lui permettant alors de se déplacer en créant une liste de 400 vecteurs d'accélération. C'est cette partie là de l'objet `Dot` qui évoluera dans le but de rendre ses déplacements plus efficaces ; on y reviendra plus loin. On définit également 4 variables importantes répondant aux questions suivantes :

1. Est-ce qu'il est mort ? (ligne 7)
2. Est-ce qu'il a atteint l'objectif ? (ligne 8)

3. Est-ce le meilleur de la population ? (ligne 9)
4. A quel point est-il fort ? (ligne 10)

La variable d'instance `canv` sert simplement à afficher le point (ligne 6).

Déplacement

Une fois cette phase d'initialisation terminée, le point est apte à se déplacer. Pour cela il dispose d'une méthode appelée : `move`.

```
1  def move(self):
2    """ Update its position. """
3    if self.dead == False and self.reachgoal == False:  # if not dead and not reached
     ↪  the goal
4      if len(self.brain.directions) > self.brain.step:  # if there are still
       ↪  acceleration vectors
5        self.acc = self.brain.directions[self.brain.step]
6        self.brain.step += 1
7        self.vel = [self.vel[0] + self.acc[0], self.vel[1] + self.acc[1]]  # add the
         ↪  acceleration to the velocity
8        limit(self.vel,4)  # limit the velocity
9        self.pos = [self.pos[0] + self.vel[0], self.pos[1] + self.vel[1]]  # add the
         ↪  velocity to the position
10     else:
11       self.dead = True
12     [...]
```

On trouve ici une fonction un peu plus complexe permettant au point de se déplacer dans la fenêtre. Tout d'abord, la fonction ne déplace le point que s'il est encore en vie et qu'il n'a pas atteint l'objectif (ligne 3). Une fois ces conditions vérifiées, on s'assure qu'il reste bien au point des « pas », soit des vecteurs d'accélération (ligne 4). Dans le cas où il n'en aurait plus, on considère simplement qu'il « meurt » (lignes 10 et 11). Par contre, s'il lui en reste, toute une suite d'actions vont s'effectuer pour faire avancer le point d'un « pas » : dans un premier temps, on ajoute à son vecteur vitesse le vecteur d'accélération suivant de sa liste (lignes 5 à 7). Après cela, on limite la norme de ce vecteur vitesse pour ne pas que le point se déplace trop vite dans la fenêtre ou qu'il ne « saute » au-dessus d'un obstacle ou de l'objectif (ligne 8). Une fois cela fait, on ajoute le nouveau vecteur vitesse à sa position précédente faisant ainsi avancer le point (ligne 9). Un petit schéma pour illustrer cela se trouve à la page suivante.

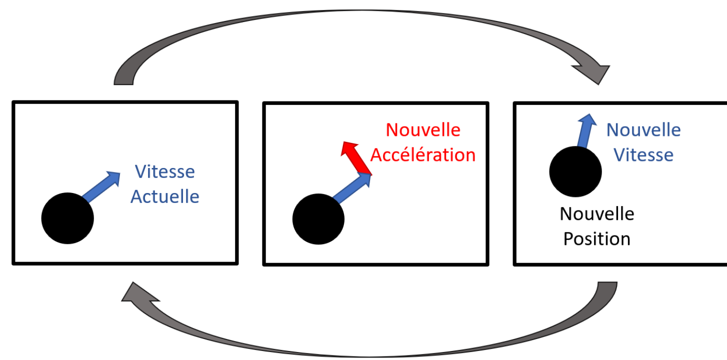


Figure 1.1 – Déplacement d'un point

Après avoir bougé, il faut bien évidemment regarder si le point se trouve dans un cas particulier :

1. en dehors de la fenêtre ;
2. dans un obstacle ;
3. ou sur l'objectif.

C'est de cela que s'occupe la seconde partie de la fonction dont le code est donné ci-dessous.

```
[...]
if self.pos[0] > 598 or self.pos[1] > 598 or self.pos[0] < 4 or self.pos[1] < 4:
↪  # if it tries to go out of the window
  self.dead = True
if self.pos[0] >= goal_pos[0][0] and self.pos[0] <= goal_pos[1][0] and self.pos[1]
↪  >= goal_pos[0][1] and self.pos[1] <= goal_pos[1][1]:  # if it touched the goal
  self.reachgoal = True
if self.pos[0] >= obstacle_pos[0][0] and self.pos[0] <= obstacle_pos[1][0] and
↪  self.pos[1] >= obstacle_pos[0][1] and self.pos[1] <= obstacle_pos[1][1]:  # if
↪  it hits an obstacle
  self.dead = True
```

Si le point sort de la fenêtre ou s'il touche un obstacle, il sera alors considéré comme « mort » (lignes 2,3 et 6,7). Par contre, s'il touche l'objectif, sa variable `reachgoal` deviendra vraie (lignes 4 et 5). Donc si le point remplit l'une de ces conditions, il ne se déplacera plus. Car, pour que la fonction le fasse bouger, il faut qu'il soit « vivant » et qu'il n'ait pas encore atteint l'objectif.

Calcul de la réussite

On passe maintenant à l'une des fonctions qui permet au programme d'apprendre : `calcul_fitness`. C'est cette fonction qui va vérifier si le point

a bien atteint l'objectif ou si ce n'est pas le cas, dans le but de recalculer la valeur de l'attribut `fitness`. Cela nous sera très utile par la suite pour la sélection des futurs points.

```
def calcul_fitness(self):
  """ Calculate how good it is. """
  if self.reachgoal == True:  #if it reached the goal
    self.fitness = 9/9 + 20/self.brain.step
  else:  #else the fitness is 1/distance to the goal
    self.fitness = 9/distance(self.pos,300,20)
```

La façon dont le calcul est effectué dépend du fait que le point a atteint l'objectif ou non. En effet, si le point a atteint l'objectif, le nombre de « pas » qu'il aura mis pour l'atteindre nous intéressera. C'est pour cela que l'on commence par regarder s'il a atteint l'objectif (ligne 3). Si oui, sa `fitness` vaudra :

$$\frac{9}{9} + \frac{20}{\text{nombre de pas}} \quad \text{(ligne 4)}$$

Si non, elle vaudra simplement :

$$\frac{9}{\text{sa distance par rapport à l'objectif}} \quad \text{(ligne 6)}$$

J'utilise la valeur 9 car c'est la distance du point au centre de l'objectif lorsqu'il touche l'objectif. Cela permet d'obtenir une fitness entre 0 et 1 tant qu'il n'a pas atteint l'objectif. J'emploie la valeur 20 car cette valeur permet d'ajouter le paramètre du nombre de « pas » à la fitness sans qu'elle devienne disproportionnée par rapport au reste de la population.

Clone

La dernière méthode que je vais décrire est `clone`.

```
def clone(self):
  """ Return a child dot with the same brain as the current dot. """
  cloned = Dot()
  cloned.brain.directions = self.brain.directions
  return cloned
```

Cette méthode nous sera très utile par la suite car elle permet de créer un nouveau point (ligne 3) qui aura exactement les mêmes vecteurs d'accélération qu'un autre (ligne 4). Elle retournera ce nouveau point (ligne 5).

1.4.2 Population

Cette classe est appelée lorsque l'on veut créer une nouvelle population contenant plusieurs points. Elle est très pratique car elle nous permettra d'effectuer des actions sur tous les points facilement.

Initialisation

Au moment de la création d'une population, une série d'étapes d'initialisation s'exécutent.

```
1 def __init__(self,nb_of_dots):
2   """ Create a population of dots. """
3   self.nb_of_dots = nb_of_dots
4   self.dots_list = []
5   for i in range(nb_of_dots):
6     self.dots_list.append(Dot())
7   self.fitness_sum = 0
```

On remarque tout d'abord que cette classe nécessite un argument : `nb_of_dots` (ligne 1). Cet argument sert simplement à définir combien de points on désire dans cette population. On définit ensuite une nouvelle liste appelée `dots_list` (ligne 4). Une fois cela fait, on y met autant de points que l'on a demandé (ligne 6). Ainsi, on pourra contrôler chaque point individuellement en utilisant cette liste avec l'indice du point souhaité mais on pourra surtout contrôler tous les points en même temps en exécutant des actions pour chaque élément de la liste. On termine en définissant l'attribut `fitness_sum` qui sera simplement la somme de la fitness de tous les points présents dans la population (ligne 7).

Actions communes

Dans cette classe, il y a plusieurs méthodes qui exécutent des actions sur tous les points de la population. On va prendre l'exemple de la méthode `move` qui a pour but de faire bouger chaque point de la population.

```
1 def move(self):
2   """ Move all the dots. """
3   for i in range(len(self.dots_list)):
4     self.dots_list[i].move()
```

On remarque ici une boucle qui va prendre chaque point de la population et lui faire exécuter sa méthode `move` expliquée auparavant (ligne 3 et 4). Les autres méthodes de `Population` similaires à celle-ci sont : `show`, `calcul_fitness`, `mutate` et `__del__`.

Le meilleur point

Regardons maintenant la méthode `best_dot`. Son but est de définir quel point était le meilleur de la population.

```
1 def best_dot(self):
2   """ Sort the dots list and find the best dot and show its step if it reached the
    ↪ goal. """
3   self.sum_fitness()
4   self.dots_list.sort(key = self.return_fitness)    #sort the list (from the lowest
    ↪ fitness first to the highest at the end)
5   if self.dots_list[len(self.dots_list) - 1].reachgoal == True:
6     canv_texte_modify(show_steps, "Steps: " + str(self.dots_list[len(self.dots_list) -
      ↪ 1].brain.step))
```

On commence par calculer la somme de la fitness de tous les points (ligne 3). On trie ensuite la population dans un ordre croissant de fitness (ligne 4). Ainsi, on sait que le meilleur point sera le dernier élément de la liste. Si ce point a atteint l'objectif, on affiche son nombre de « pas » en haut à gauche de la fenêtre (ligne 5 et 6).

Tous morts

On va maintenant se pencher sur la méthode `all_dead` qui va retourner une valeur booléenne : « Faux » tant qu'il reste des points « vivants » et « Vrai » une fois qu'ils sont tous « morts ».

```
1 def all_dead(self):
2   """ Return True if all the dots are dead. """
3   res = 0
4   for i in range(len(self.dots_list)):
5     if self.dots_list[i].dead == True:
6       res += 1
7     if self.dots_list[i].reachgoal == True:
8       if self.dots_list[i].isbest == False:
9         res  += 1
10      else:
```

```
11        res = len(self.dots_list)
12  return res == len(self.dots_list)
```

Pour ce faire, la fonction va définir une variable locale `res` qui vaudra d'abord 0 (ligne 3). On regardera ensuite pour chaque point de la population (ligne 4) si son attribut `dead` est vrai ou faux (ligne 5). Si la valeur de cet attribut s'avère être « vraie », on ajoute 1 à `res`. À la dernière ligne, on compare la variable `res` avec la longueur de la population (ligne 12). Ainsi, si tous les points sont « mort », `res` sera égal au nombre de points dans la population et la fonction renverra « Vrai ». Mais quand est-il des lignes 7 à 11 ? Comme le but du programme est de faire progresser les points, une fois que le meilleur point de la population d'avant a atteint l'objectif, il ne sert à rien de continuer car tous les points arrivant après seront de toute façon moins bons. On regarde donc pour chaque point qui a atteint l'objectif (ligne 7), s'il est le meilleur de la génération précédente. Si ce n'est pas le cas, on ajoute simplement 1 à `res` ; mais si c'est le cas, on considère que tous les points sont « morts » de sorte à terminer la génération pour passer à la suivante.

La sélection des candidats

Avant de passer à la grande étape de la sélection naturelle, il est important d'expliquer la sélection des candidats. Cette sélection est matérialisée dans le code par la méthode `select_individual`. Cette méthode va permettre de choisir un point en prenant en compte sa fitness. Elle sera utilisée plus tard pour remplir la nouvelle population avec des points provenant de l'ancienne.

```
1 def select_individual(self):
2   """ Select a dot in the dots list. """
3   rand = random.uniform(0,self.fitness_sum)
4   rsum = 0
5   for i in range(len(self.dots_list)):  #if a dot has a high fitness it has more
    ↪  chance to get chosen
6     rsum += self.dots_list[i].fitness
7     if rsum >= rand:
8       return self.dots_list[i]
```

Un petit schéma explicatif montrant le fonctionnement de la sélection se trouve à la page suivante.

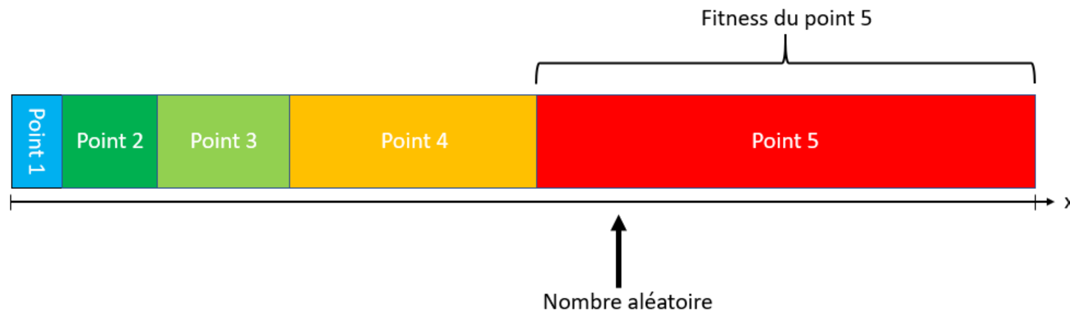


FIGURE 1.2 – Sélection d'un candidat

On peut voir sur ce schéma un exemple illustrant une population contenant 5 points. Ils ont tous obtenu des fitness différentes. Ils ont déjà été classés dans un ordre croissant grâce à la méthode `best_dot`. On génère alors un nombre aléatoire entre 0 et la somme de toutes les fitness (ligne 3). Il suffit maintenant de regarder dans quelle plage est tombé le nombre aléatoire (lignes 4 à 7), puis de retourner le point correspondant (ligne 8). Dans notre exemple ça serait le Point numéro 5. De cette manière, un point ayant bien réussi a plus de chance d'être sélectionné sans forcément condamner toute possibilité à un autre point d'être choisi.

La sélection naturelle

On passe maintenant à la méthode maîtresse de l'apprentissage des points : `natural_selection`. C'est cette méthode qui va sélectionner les points de la génération actuelle et les placer dans la génération suivante.

```
1  def natural_selection(self):
2      """ Select the dots for the next generation. """
3      newpop = Population(self.nb_of_dots)
4      self.best_dot()
5      for i in range(len(self.dots_list) - 1):
6          individual = self.select_individual()
7          newpop.dots_list[i] = individual.clone()
8      newpop.dots_list[len(self.dots_list) - 1] = self.dots_list[len(self.dots_list) -
       ↪ 1].clone()  #the last dot in the newpop list is the best of the previous
       ↪ generation
9      newpop.dots_list[len(self.dots_list) - 1].isbest = True
10     canv.itemconfig(newpop.dots_list[len(self.dots_list) - 1].canv, fill="green") #it
       ↪ shows it as green
11     return newpop
```

On commence par créer une nouvelle population contenant le même nombre de point que la population actuelle (ligne 3). On exécute la méthode `best_dot` vue plus haut pour trier la population et définir le meilleur point (ligne 4). Dès lors, on va remplir cette nouvelle population en y ajoutant petit à petit des points de l'ancienne. Chaque candidat est choisi par la méthode `select_individual` décrite plus haut (ligne 6). Il est ensuite cloné et mis dans la nouvelle population. Pour finir, on ajoute également le meilleur point de la génération précédente de sorte à avoir un référentiel à battre (ligne 8 et 9). On le colore d'ailleurs en vert pour qu'il soit facilement reconnaissable dans la fenêtre (ligne 10). Enfin, on retourne cette nouvelle population (ligne 11).

1.4.3 Brain

Nous allons maintenant revenir sur les points et parler de cette fameuse liste de vecteurs d'accélérations et de la manière de l'obtenir. La classe `Brain` se charge justement de cela. Cette classe est utilisée par la classe `Dot` et lui fournira les vecteurs dont elle a besoin tout au long du programme.

Initialisation

Cette classe possède elle aussi une méthode d'initialisation que voici :

```
1 def __init__(self,size):
2   self.size = size  #the amount of vectors in the directions list
3   self.directions = []  #the list which contains all the acceleration vectors
4   self.step = 0  #where we are in the list
5   self.randomize()  #at its creation randomize all the vectors in the list
```

On peut dès lors remarquer que lorsque l'on veut appeler cette classe, il est nécessaire de préciser la taille de la liste des vecteurs d'accélération (ligne 1 et 2). On définit ensuite cette liste appelée `directions` qui est tout d'abord vide (ligne 3) mais qui sera remplie plus tard. On définit également une variable nommée `step` qui vaut 0. Elle sera incrémentée de 1 chaque fois qu'un nouveau vecteur de la liste sera utilisé, représentant ainsi un nombre de « pas ». Pour finir on fait appel à la méthode `randomize` détaillée ci-dessous.

Randomize

La méthode `randomize` est utilisée par `Brain` lorsque l'on veut que la liste de vecteurs soit complètement aléatoire. On va donc l'employer à l'initialisation de la classe comme on a pu le voir plus tôt.

```
def randomize(self):
  """ For each element of the list randomize a vector. """
  for i in range(self.size):
    random_angle = random.uniform(0,2*math.pi)  #generate a random angle
    self.directions.append([math.cos(random_angle),math.sin(random_angle)])  #take the
    ↪ vector from this angle
```

On commence par entrer dans une boucle de type `for` qui exécutera le code en dessous autant de fois que l'on veut d'éléments dans la liste (ligne 3). On génère ensuite un angle aléatoire en radians, donc un nombre aléatoire entre 0 et 2π. On stocke ensuite ce nombre dans une variable appelée `random_angle` (ligne 4). Après cela, on crée un vecteur en prenant le cosinus et le sinus de l'angle obtenu juste avant. On stocke ensuite ce vecteur unitaire dans la liste `directions`. On obtient ainsi une liste de vecteurs aléatoires de la taille souhaitée.

Mutation

Dans le programme, il y a un moment où l'on a besoin d'effectuer une mutation de cette liste de vecteurs dans le but de ne pas reproduire des clones identiques. C'est la méthode `mutate` qui s'en chargera.

```
def mutate(self):
  """ Randomize several vectors in the list. """
  mutation_chance = 0.03
  new_directions = []
  for i in range(self.size):  #randomize all the vectors in the new list
    random_angle = random.uniform(0,2*math.pi)
    new_directions.append([math.cos(random_angle),math.sin(random_angle)])
  for i in range(self.size):
    r = random.uniform(0,1)
    if r > mutation_chance:
      new_directions[i] = self.directions[i]  #put the same vector from the old list
      ↪ into the new list if the mutation chance wasn't big enough
  return new_directions
```

On commence par définir le taux de mutation (ligne 3). Il est de 0.03 ici mais on peut très bien le changer selon les différents parcours ou les résultats voulus. Il aura pour seul but d'augmenter ou de réduire la chance que les vecteurs subissent une mutation. Plus il est élevé, plus il y aura de vecteurs modifiés et plus il est faible, moins il y aura de vecteurs modifiés. On définit ensuite une nouvelle liste de vecteurs, vide dans un premier temps (ligne 4). On la remplit de vecteurs aléatoires de la même manière que la méthode `randomize` vue plus haut (ligne 5 à 7). On entre ensuite dans une boucle `for` qui va exécuter le code en dessous pour chaque élément de la liste `new_directions` (ligne 8). On génère donc un nombre aléatoire entre 0 et 1 que l'on met dans une variable nommée `r` (ligne 9). Après cela, on compare cette variable `r` au taux de mutation défini plus tôt (ligne 10). Si la valeur de `r` est plus grande que le taux de mutation, ce qui se passe la grande majorité du temps dans cet exemple, on remplace le vecteur de la liste `new_directions` par l'ancien vecteur de `directions` correspondant, le laissant donc inchangé dans la nouvelle liste (ligne 11). On obtient donc une liste très semblable à l'ancienne mais dont quelques vecteurs sont redéfinis aléatoirement. Pour finir, on retourne cette liste (ligne 12).

1.5 Affichage

Lors du lancement du programme, une fenêtre s'ouvrira. On y retrouvera l'objectif, l'obstacle et la position en temps réel des points. Il y aura aussi les informations utiles en haut à gauche comme le nombre de « pas » du meilleur point ou le nombre de générations testées. Cela se matérialise dans le code sous la forme de quelques fonctions basiques que je ne vais pas décrire en détail. J'ai utilisé le module python `tkinter` pour créer la fenêtre et y afficher tous les éléments. On actualisera la fenêtre à une fréquence stockée dans une variable appelée `frame_rate`. De ce fait on peut augmenter ou diminuer cette variable pour que le programme se déroule plus ou moins vite.

1.6 Initialisation du programme

Avant de lancer la boucle principale du programme, quelques petites étapes d'initialisation doivent s'exécuter. On compte parmi elles :

1. la définition de la fenêtre,

2. la définition de la variable `frame_rate`,
3. le placement de l'objectif et de l'obstacle,
4. le placement des informations en haut à gauche énumérées plus tôt,
5. et la création de la première population contenant autant de points que souhaité (100 dans le programme).

Une fois que toutes ces étapes ont été exécutées, on peut entrer dans la boucle principale.

1.7 Boucle principale

On va maintenant discuter de la boucle principale du programme qui utilise toutes les fonctions, méthodes et variables que nous avons étudiées jusqu'à présent.

```
while True:
  if pop.all_dead() == False:
    pop.move()
    pop.show()
  else:
    pop.calcul_fitness()
    newpop = pop.natural_selection()
    newpop.mutate()
    del pop
    pop = newpop
    gen()
  try:
    canv.update()
  except:
    pass
  time.sleep(frame_rate)
```

On remarque tout d'abord que cette boucle est divisée en deux parties par une condition. Cette condition est : si tous les points de la population sont pas « morts » (ligne 2). Si cette condition s'avère vraie, alors, on bouge tous les points de la population et on affiche le résultat. Si la condition n'est pas remplie (ligne 5), c'est que tous les points de la population sont « morts » et que l'on se trouve donc à la fin d'une génération. On a donc une série d'étapes à effectuer pour reformer une nouvelle population. On commence par calculer la fitness de tous les points de la population (ligne 6). Ensuite, on définit une nouvelle population en faisant appel à la méthode de la classe

`Population` vue plus haut : `natural_selection` (ligne 7). Cela nous renvoie une nouvelle population qui devrait être meilleure que la précédente. On mute ensuite cette nouvelle population pour éviter les doublons et créer une plus grande mixité (ligne 8). Après cela, on supprime l'ancienne population (ligne 9) et l'on définit la nouvelle population en tant que population actuelle (ligne 10). On peut alors exécuter la fonction `gen`, qui va actualiser le nombre de générations en haut à gauche de la fenêtre (ligne 11). On se retrouve donc avec une population à nouveau prête à entrer dans la boucle. On termine en actualisant la fenêtre (ligne 13) et en attendant le nombre de secondes défini par la variable `frame_rate` (ligne 16). L'instruction `try/except` (lignes 12 et 14) permet simplement de réduire le nombre d'erreurs dans la console du module `tkinter` lorsque l'on clique sur la croix pour fermer la fenêtre.

1.8 Conclusion

Les résultats obtenus par le programme sont très concluants. On arrive la plupart du temps à toucher l'objectif avant une dizaine de générations. Plus on laisse le programme tourner, plus les points s'améliorent, mais le nombre de pas minimal sera de l'ordre de grandeur d'environ 140 (après 50 générations). Cependant, il est très important de comprendre que ces résultats ne sont pas les résultats optimaux du programme et dépendent complètement des valeurs définies dans le code. Par exemple, en triplant le nombre de points dans la population, on atteint l'objectif avant cinq générations la plupart du temps. Mais le but de ce programme n'était justement pas de trouver le moyen le plus efficace de parcourir ce chemin en particulier mais bien de développer une intelligence artificielle capable de résoudre ce type de problème. Et une fois ce programme écrit, il y a des démonstrations bien plus impressionnantes à réaliser en changeant simplement le nombre et la disposition des obstacles. C'est là que le programme devient réellement intéressant. Un exemple beaucoup plus complexe comportant 3 obstacles à contourner chacun du bon coté se trouve à la page suivante.

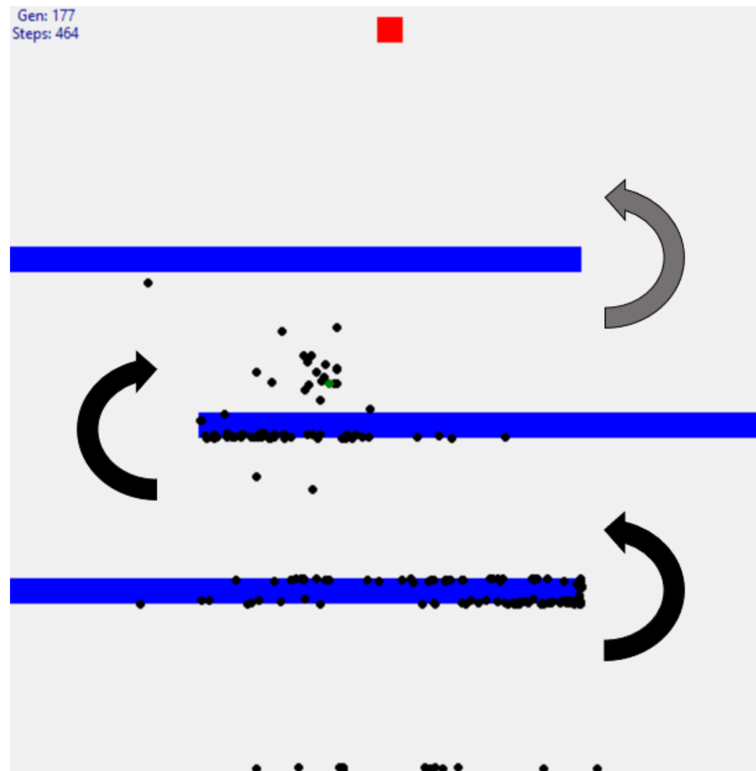


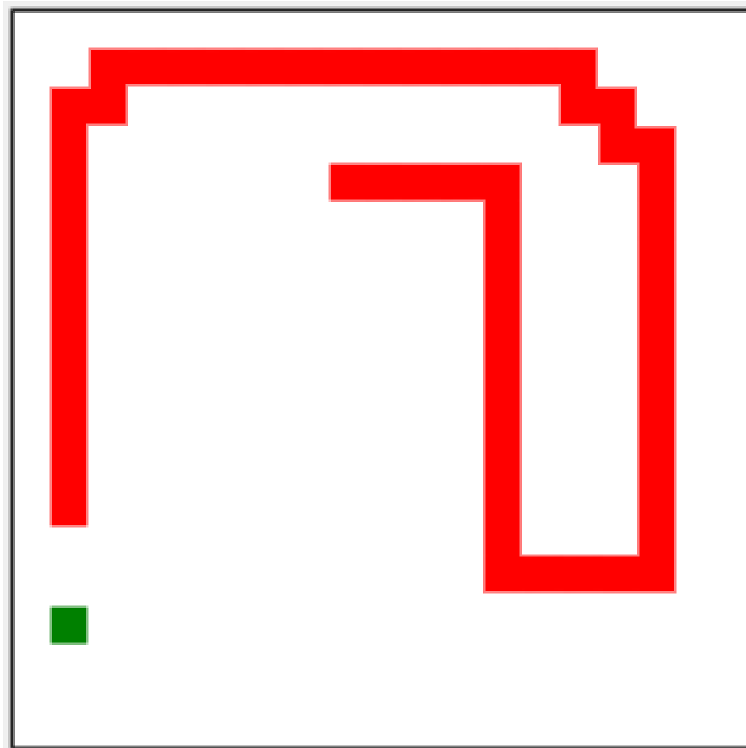
Figure 1.3 – Contournement de plusieurs obstacles

On voit dans cet exemple bien plus compliqué que les points sont parvenus à contourner les 3 objectifs en moins de 200 générations.

C'est exactement cela que le programme devait donner : réussir à faire apprendre à un point à contourner un ou de multiples obstacles pour atteindre un objectif.

Chapitre 2

Snake Ai



2.1 Objectif

L'objectif que je souhaitais atteindre avec ce second programme codé en Python est le suivant : faire apprendre à un ordinateur à jouer au fameux jeu Snake. Snake est un jeu basique, facile à comprendre, mais difficile à maîtriser. Dans ce jeu on dirige un serpent de petite taille (au début) dans une zone délimitée. Des fruits apparaissent un par un, dans cette zone, de manière aléatoire. Le but du jeu est de manger le plus de fruits possible. La difficulté provient du fait que lorsque l'on mange un fruit, la taille de notre serpent augmente. Si la tête de notre serpent vient à entrer en contact avec sa queue, la partie se termine. La partie se termine également si l'on essaye de faire sortir le serpent de la zone de jeu.
Le programme devra donc apprendre à faire face à ces difficultés. Pour atteindre cet objectif, j'utiliserai de nouveau un algorithme de type génétique mais, cette fois, chaque serpent disposera d'un réseau de neurones artificiels.

2.2 Programmer le jeu

Pour pouvoir faire apprendre à un ordinateur à jouer à Snake, il a fallu dans un premier temps reprogrammer le jeu. J'ai donc commencé par coder Snake en python. Ce programme permet à un utilisateur de jouer au jeu en utilisant les flèches directionnelles du clavier. Je ne vais pas détailler le code du jeu (disponible en annexe) car il a été beaucoup modifié par la suite, notamment pour pouvoir faire jouer une machine.

2.3 Fonctionnement d'un réseau neuronal artificiel

2.3.1 Concept et explications relatifs au réseau utilisé

Un réseau neuronal tel que celui que j'ai utilisé se présente par couches : les couches d'entrées, les couches dites « cachées » et les couches de sorties. La figure de la page suivante montre un réseau neuronal basique qui permet de se représenter le réseau de neurones utilisé dans le programme de manière simplifiée.

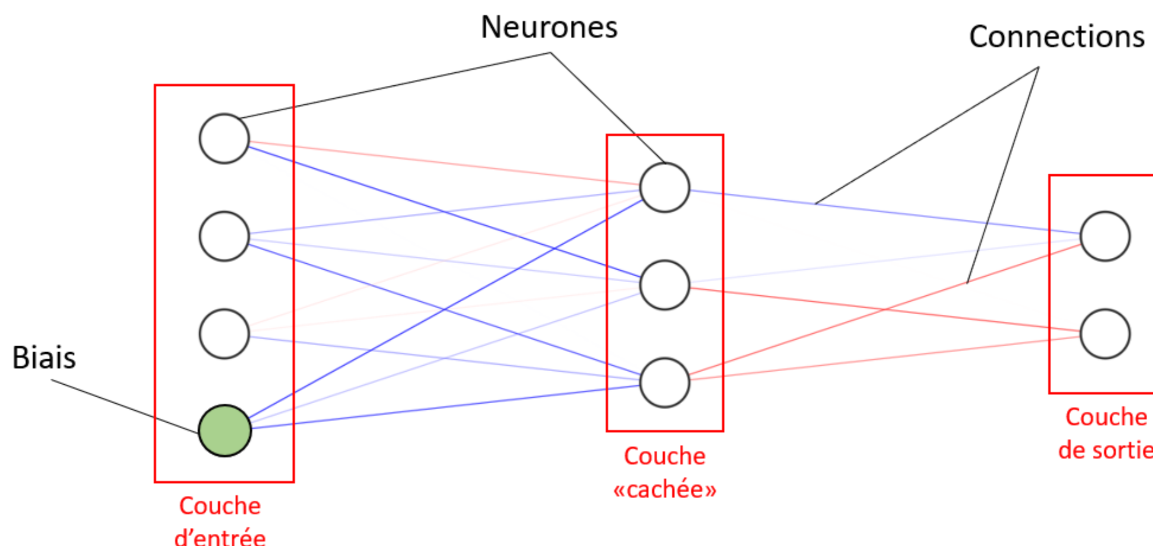


FIGURE 2.1 – Exemple de réseau neuronal

Un réseau de neurones se compose de deux parties principales : les neurones, représentés par des disques, et les connections, représentées par des traits. Les neurones sont organisés par groupes ; on appelle ces groupes des couches (représenté par des colonnes de neurones). Chaque connexion se voit attribuer un poids. Ce poids est en réalité un simple nombre décimal compris entre -1 et 1. Plus un poids est proche de -1 ou de 1, plus il est considéré comme important. On le représente alors sur la figure ci-dessus à l'aide d'un trait plus épais et moins transparent. Les connections ayant un poids négatif sont représentées en bleu ; celles ayant un poids positif en rouge. On décrit un réseau de neurones par sa forme et son type : sa forme étant le nombre de couches et son type la manière dont les neurones sont connectés.
Dans ce cas, la forme du réseau est $4 \times 3 \times 2$ car la première couche contient 4 neurones, la deuxième 3 et la dernière 2. On choisit la forme d'un réseau de manière plus ou moins subjective : elle dépend bien évidemment du nombres d'entrées que l'on a (on ne vas pas passer d'une couche de 100 neurones à une couche de 3 neurones), mais il n'y a pas de règle précise pour définir la forme d'un réseau. Il est expliqué plus bas dans mon travail comment j'ai procédé pour trouver une forme de réseau convenant à ma problématique.
Quand au type du réseau ci-dessus, c'est un réseau dit « complètement connecté ». Cela signifie que tous les neurones d'une couche sont reliés à tous les neurones de la couche d'après. C'est le type de réseau le plus utilisé et le plus facile à mettre en place.
La première colonne à gauche représente la couche d'entrée du réseau. C'est

là que l'on injecte les données. Cette colonne représente en quelque sorte les organes sensoriels du réseau. La dernière colonne à droite représente la couche de sortie du réseau. Chaque sortie représente une décision à prendre potentiellement. Par exemple, dans la figure ci-dessus, le premier neurone de la couche de sortie pourrait représenter la décision de tourner à droite et le second neurone la décision de tourner à gauche. C'est cette couche là qui détermine réellement quoi faire. Ce sont donc uniquement les couches d'entrées et de sorties qui importent réellement. Les couches cachées, elles, permettent de faire le passage de l'une à l'autre.
Un dernier point important concernant ce réseau de neurones est la présence d'un biais (représenté en vert). C'est simplement une entrée de valeur fixe qui permet de recalibrer les données.
Un réseau de neurones est donc plus simplement une ou plusieurs matrices de poids. C'est en changeant ces différents poids que l'on obtient les résultats que l'on veut. C'est donc sur ce point que devra travailler l'algorithme génétique : trouver les meilleurs poids possibles de manière à ce que le réseau neuronal permette au serpent de prendre les bonnes décisions au bon moment.

2.3.2 Mathématiques

Une fois le concept compris, on peut observer le fonctionnement d'un réseau du point de vue mathématique. Comme le type du réseau utilisé est « complètement connecté », les calculs employés sont plutôt simples. Pour ce faire, on emploie le calcul matriciel.

[1x4] x [4x3] = [1x3]
E1 E2 E3 B
X
W1 W2 W3 W4
W5 W6 W7 W8
W9 W10 W11 W12
=
N1 N2 N3
tanh
N'1 N'2 N'3
Couche d'entrée
Poids des connections entre deux couches
Couche «cachée»

Figure 2.2 – Matrices associées à un réseau neuronal

Pour la première couche du réseau, on forme une matrice (comme montré par la figure de la page précédente) de manière à ce qu'il n'y ait que les valeurs d'entrée (E1 → E4). Pour calculer les valeurs de la seconde couche, il faut créer une deuxième matrice contenant tous les poids des connexions entre la première et la deuxième couche (W1 → W12). La forme de cette matrice se trouve être le nombre de neurones dans la couche d'entrée fois le nombre de neurones dans la seconde couche. Une fois que l'on a les deux matrices, il suffit de faire un produit matriciel entre elles. Il faut ensuite faire passer les valeurs dans une fonction non linéaire (tangente hyperbolique dans ce cas). On obtient alors les valeurs de chaque neurone de la seconde couche dans une nouvelle matrice.
Une fois cela fait pour toutes les couches du réseau, on obtient, lors de la dernière opération, la couche de sortie. C'est grâce à elle que l'on peut savoir quelle décision il est mieux de prendre. Pour ce faire, il suffit de regarder les valeurs de chaque neurone de la couche de sortie : plus le chiffre est élevé pour une sortie, plus la décision qui y est associée a de probabilité d'être la bonne par rapport au réseau courant.

2.4 Apprentissage

Lorsqu'on lance le programme, une population de serpents est générée. Ils sont tous dotés d'un réseau neuronal dont les poids sont générés aléatoirement. Les décisions « prisent » par ces réseaux n'auront donc probablement aucun sens au début. Mais grâce à l'algorithme génétique (voir 1.3 Apprentissage), les serpents auront des réseaux de neurones de plus en plus performants ; ils prendront donc des décisions de plus en plus sensées.

2.5 Entrées du réseau de neurones

Passons maintenant à un point très important, car il conditionnera complètement les décisions du réseau de neurones. En effet, « on vit par ce que l'on voit » il est donc important d'apporter une bonne « vision » à l'ordinateur. Le joueur humain, lui, a accès à chaque pixels du jeu grâce à son sens de la vue. Il a donc une perception parfaite car il reçoit toutes les données du jeu. Cependant, il n'est pas possible de donner autant d'informations à mon ordinateur. Il ne supporterait simplement pas la charge de calcul nécessaire. En effet, cela représenterait plusieurs dizaines de millions d'opérations par

serpents à chaque mouvement. Il a donc fallu limiter les entrées du réseau. J'ai choisi d'utiliser un système d'axes prenant la tête du serpent comme origine. Le serpent va pouvoir percevoir trois types d'éléments différents : les murs, le fruit et sa queue. A chaque type d'éléments sont associés 4 entrées : une pour le Nord, une pour le Sud, une pour l'Est et une pour l'Ouest.

1. **Les murs** : Les 4 premières entrées sont données par la distance de la tête du serpent à chaque mur. Grâce à cela, le serpent pourra savoir ou il se situe dans la fenêtre.

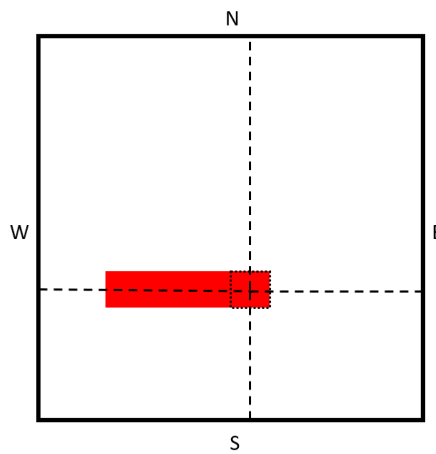


FIGURE 2.3 – Détection des murs

2. **Le fruit** : S'il se situe sur l'un des quatre axes, l'entrée correspondant à cet axe aura la valeur de la distance de la tête du serpent au fruit. Si le fruit ne se trouve pas sur un des axes, alors l'entrée correspondante sera nulle.

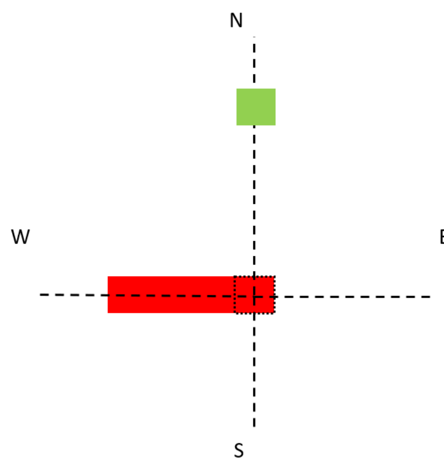


FIGURE 2.4 – Détection du fruit

3. **Sa queue** : Si la queue du serpent se trouve sur l'un des quatre axes, l'entrée correspondant à l'axe aura la valeur de la distance de la tête à sa queue. Si sa queue ne figure pas sur un des axes, alors l'entrée correspondante sera nulle.

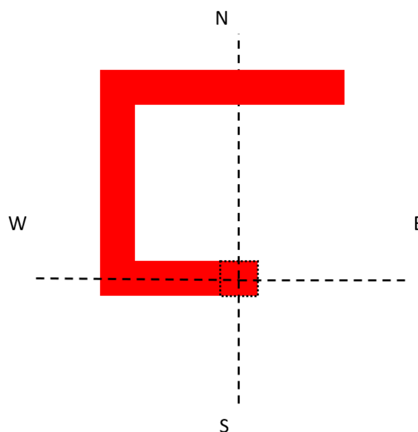


FIGURE 2.5 – Détection de sa queue

On se retrouvera donc avec une matrice d'entrées 13×1 qui contiendra les informations suivantes :

$$\begin{bmatrix} \text{Distance au mur Nord} \\ \text{Distance au mur Sud} \\ \text{Distance au mur Ouest} \\ \text{Distance au mur Est} \\ \text{Distance au fruit Nord} \\ \text{Distance au fruit Sud} \\ \text{Distance au fruit Ouest} \\ \text{Distance au fruit Est} \\ \text{Distance à sa queue Nord} \\ \text{Distance à sa queue Sud} \\ \text{Distance à sa queue Ouest} \\ \text{Distance à sa queue Est} \\ \text{Biais} \end{bmatrix}$$

2.6 Affichage

Lors du lancement du programme, une fenêtre s'ouvre. On y retrouve la zone de jeu ainsi que des informations concernant le serpent que l'on voit en dessous. Chaque génération se joue en arrière plan. Une fois tous les serpents

morts, on revoit la partie du meilleur des serpents de la génération précédente. Les informations sous la zone de jeu se mettent alors à jour pour afficher les données correspondant au serpent montré à l'écran.

2.7 Concrètement dans le code

La suite du travail sera consacrée à une description partielle du programme. En effet, dû à l'ampleur de la tâche et des similitudes avec le premier programme, il serait trop long de tout détailler. Cependant, le programme (disponible en annexe) à été commenté avec soin.
Concrètement, cela se matérialise dans le code sous forme de quatre classes principales : `Brain` et `Fruit`, utilisées par `Snake`, ainsi que `Population`.

2.7.1 Initialisation

Quelques étapes d'initialisation sont nécessaires au bon fonctionnement du programme.

Importation des modules

Dans ce programme, j'ai eu besoin d'utiliser les modules suivants :

```
1 # import all the libraries
2 import random as rdm  # to generate random numbers
3 from tkinter import *  # for the display window
4 import time  # to wait some time when needed
5 import numpy as np  # to compute operations on arrays
6 import math  # for unlinear functions and infiny number
7 import pickle  # for saving and loading python objects
8 import sys  # to use interpreter arguments
```

Le module `random` (ligne 2) sert à générer des nombres aléatoires. Il sera utilisé plusieurs fois pour divers occasions. Comme dans le premier chapitre, j'ai utilisé le module `tkinter` (ligne 3) pour l'affichage du programme. Le module `time` (ligne 4) sera utilisé dans le but de laisser un délai pour que l'utilisateur puisse voir ce qui se passe à l'écran. L'un des modules les plus important est `numpy` (ligne 5). C'est grâce à lui que sera gérée la partie mathématique du réseau neuronal dans le programme. Le module `math` (ligne 6) est sollicité de multiples fois tout au long du programme. `Pickle` (ligne 7) est un module utilisé pour sauvegarder des objets python. Il s'agira des serpents

et de leur cerveau dans le cas de mon programme. Le module `sys` (ligne 7) sert à récupérer des arguments en ligne de commande que l'utilisateur aurait insérés lors du lancement du programme.

Variables globales

Voici ci-dessous quelques variables globales qui reviendront tout au long des explications.

```
1 # global variables
2 directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # the different possibilities of
  ↪ directions [N,S,W,E]
3 dimensions = (0, 20)  # the dimensions of the window (in decade of pixels)
4 maxsteps = 300  # how many steps they have at the beginning
5 save_path = time.strftime("saves/snake_%d.%m.%y_%Hh%Mmin%Ssec.pkl")  # the path where
  ↪ it saves the snake
6 gen = 1 # the number of generations
7 display = Display("Snake AI", 10 * dimensions[1], gen, maxsteps)  # Creating the
  ↪ window
8 pop = Population(maxsteps)  # creates the first population
```

La liste nommée `directions` (ligne 2) contient la liste de 4 vecteurs unitaires. Ces 4 vecteurs représentent les différentes directions que le serpent peut suivre. On les utilisera plus tard en spécifiant l'indice de la liste correspondant au vecteur voulu. Le tuple `dimensions` (ligne 3) définit simplement la zone jouable de la fenêtre à l'aide de nombres entiers de dizaines de pixels. La variable `maxsteps` (ligne 4) définit le nombre de mouvements que le serpent à le droit de faire. Une fois cette limite dépassée, la partie se termine. Cette précaution est nécessaire car il ne faut pas que la partie dure indéfiniment à cause d'un serpent tournant en rond. La variable `save_path` (ligne 5) contient le chemin d'accès du fichier où les serpents seront stockés. La variable `gen` (ligne 6) garde en mémoire le nombre de génération(s) qui se sont écoulée(s) depuis le lancement du programme. Les deux instances `display` et `pop` (lignes 7 et 8) stockent respectivement l'objet d'affichage et l'objet de population qui seront détaillés plus bas.

2.7.2 Classes mineures

Ces classes mineures ne sont pas des classes très intéressantes à détailler mais il est tout de même important de connaître certains points pour pouvoir comprendre la suite de mes explications.

Affichage

La classe d'affichage nommée `Display` est la classe qui s'occupe de générer une fenêtre à l'aide du module `tkinter` (voir 2.7.1 Importation des modules). C'est notamment dans cette classes que seront stockées les variables d'instance permettant de modifier les textes écrits dans la fenêtre.

```
1 # informations to display
2 self.borders = self.canv.create_rectangle(10 * dimensions[0] + 4, 10 * dimensions[0] +
  ↪ 4, 10 * dimensions[1] + 5, 10 * dimensions[1] + 5, width=1, fill="white")
3 self.gen = self.canv.create_text(5, size + 6, fill="darkblue", text="Gen:
  ↪ {}".format(gen), anchor="nw")
4 self.maxsteps = self.canv.create_text(70, size + 6, fill="darkblue", text="Max steps:
  ↪ {}".format(maxsteps), anchor="nw")
5 self.score = self.canv.create_text(5, size + 22, fill="darkblue", text="Score: [...]",
  ↪ anchor="nw")
6 self.killedby = self.canv.create_text(70, size + 22, fill="darkblue", text="KB:
  ↪ [...]", anchor="nw")
```

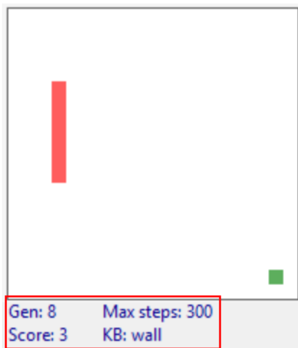


Figure 2.6 – Informations affichées sous la zone jouable

La première variable d'instance nommée `borders` (ligne 2) permet de dessiner un rectangle autour de la zone jouable de la fenêtre pour rendre visible la limite à ne pas dépasser. La deuxième est nommée `gen` (ligne 3) et elle permet d'afficher le nombre de générations écoulées en temps réel. La variable d'instance suivante nommée `maxsteps` (ligne 4) permet d'afficher le nombre total de mouvements ou « pas » que les serpents ont à disposition. La variable d'instance `score` (ligne 5) affiche le score du meilleur serpent et `killedby` (ligne 6) affiche la raison de sa mort.

Cette classe dispose également d'une méthode (voir ci-dessous) qui permet d'actualiser la fenêtre ainsi que tous les éléments qui s'y trouvent.

```
1 def update(self):
2         """
3         Updates the canvas.
4         :return: None
5         """
6         self.canv.update()
```

Carrés

Cette classe intitulée `Square` permet de générer des carrés dans la fenêtre.

```
self.show = display.canv.create_rectangle(10 * x + 5, 10 * y + 5, 10 * x + 10 + 5, 10
    * y + 10 + 5, width=0, fill=color)
```

La ligne de code ci-dessus est tirée du constructeur de la classe. On y voit la variable d'instance `show` qui dessine simplement un carré dans la fenêtre. Ce carré est rempli avec la couleur spécifiée au moment de l'appel du constructeur.

```
display.canv.delete(self.show)
```

Cette ligne de code est exécutée lorsque l'on tente de supprimer l'objet. Elle supprime le carré de la fenêtre, ce qui a pour effet de ne plus l'afficher lorsque la fenêtre est actualisée.

2.7.3 Serpent

On passe maintenant à l'une des, si ce n'est la, classe la plus importante du programme : `Snake`. C'est cette classe qui s'occupera de créer, déplacer, tuer, faire manger ou encore sauvegarder un serpent pour ne donner que quelques exemples.

Initialisation

Le constructeur de cette classe est complexe, mais on ne peut plus important, car il pose les bases pour le bon fonctionnement de la classe.

```
def __init__(self, hposx=int(dimensions[1]/2), hposy=int(dimensions[1]/2), length=4,
    direc=3, display=False, maxsteps=300, ow=False):
    """
    Initialisation method at the creation of a snake.
    :param int hposx: starting position of the snake on the X axis
    :param int hposy: starting position of the snake on the Y axis
    :param int length: the starting length of the snake
    :param int (between 0 and 3) direc: the starting direction the snake is facing
    :param bool display: if we want it to be displayed in the window or not
    :param int maxsteps: how many steps it has (maximum)
```

```
10      :param bool ow: if it is an overwatch snake or not
11      """
12      if direc not in [0, 1, 2, 3]:
13          raise Exception("The starting direction of the snake has to be 0, 1, 2 or 3
            ↪  (N, S, W, E).")
14      self.direc = directions[direc]  # its direction vector
15      self.coords = []  # the list of all its points' coordinates
16      self.show = []  # the list of its points in the canvas
17      for i in range(length):
18          self.coords.append((hposx - i * self.direc[0], hposy - i * self.direc[1]))  #
            ↪  adds every coordinate to its list
19      self.score = 0  # the number of fruits that it ate
20      self.display = False  # if it is displayed or not
21      self.display_new = display  # if we want to hide or show it
22      self.fruit = self.newfruit()  # its fruit
23      self.fruitslist = [self.fruit]  # all its fruits
24      self.fruit_index = 1  # which fruit will be the next in the list
25      self.ow = ow  # if it is an overwath snake
26      self.brain = Brain(13)  # its brain
27      self.dead = False  # if it is dead
28      self.step = 0  # how many steps it has done yet
29      self.maxsteps = maxsteps  # how many steps it has at disposal
30      self.fitness = 0  # how good it is
31      self.kb = None  # what killed it
32      self.displayupdate()
```

Le constructeur demande plusieurs paramètres lors de la création d'un serpent (ligne 1). Il est d'abord demandé la position initiale du serpent dans la fenêtre sur l'axe x (`hposx`) et sur l'axe y (`hposy`). Il est ensuite demandé la longueur initiale du serpent (`length`) et la direction dans laquelle il avancera par défaut (`direc`). Le paramètre suivant (`display`) définit si l'on veut afficher le serpent, et celui d'après (`maxsteps`) le nombre de « pas » que le serpent est autorisé à faire. Le dernier paramètre (`ow`) sert à définir si l'on rejoue la partie du serpent en question ou si c'est la première fois que ce serpent joue.

On commence par vérifier que la direction entrée soit bel et bien comprise entre 0 et 3 (lignes 12 et 13), car il n'y a que 4 possibilités de directions (Nord, Sud, Ouest, Est). On affecte ensuite un vecteur unitaire à la variable `direc` (ligne 14) en fonction du chiffre entré en paramètre, ce qui lui donne une direction. On initialise ensuite deux listes vides attribuées au variables `coords` et `show` (lignes 15 et 16). La liste attribuée à `coords` étant la liste de chaque points dont le serpent dispose pour son corps et sa tête ; et celle

attribuée à `show` étant la liste des carrés (voir 2.7.2 Carrés) dans la fenêtre. On remplit donc la première liste avec autant de points que précisés par le paramètre `length` (lignes 17 et 18). La variable suivante nommée `score` (ligne 19) stocke simplement le nombre de fruits que le serpent en question a mangés. Les variables `display` et `display_new` (lignes 20 et 21) sont utiles lorsque l'on souhaite afficher ou cacher le serpent. Si la variable `display` (ligne 20) est « vraie », on voit le serpent et si elle est « fausse », on ne le voit pas dans la fenêtre. Les processus d'affichage et de disparition ne seront pas plus détaillés. La variable `fruit` (ligne 22) continent l'objet de type `Fruit` courant que le serpent doit manger pour pouvoir grandir. Cet objet est retourné par la méthode `newfruit` expliquée plus bas. La liste `fruitslist` (ligne 23) garde en mémoire tous les fruits dont le serpent a eu besoin. La variable `fruit_index` (ligne 24) indique l'indice du prochain fruit à mettre dans la liste `fruitlist`. Les variables `ow` (ligne 25) et `maxsteps` (ligne 29) stockent les paramètres correspondants entrés lors de la création de l'objet. Le « cerveau » du serpent est créé puis stocké dans la variable d'instance `brain` (ligne 26). C'est cette partie là qui contiendra le réseau de neurones et qui s'occupera des décisions prises par le serpent (voir 2.7.5 Cerveau). La variable `dead` (ligne 27) sera « vraie » quand le serpent sera mort. La variable `step` (ligne 28) gardera en mémoire le nombre de « pas » qu'aura fait le serpent. La valeur de `fitness` (ligne 29) est calculée par une fonction expliquée plus bas (voir 2.7.3 Calcul de la fitness). Elle représente, à l'aide d'un nombre réel, à quel point le serpent est fort. Plus un serpent a une grande fitness, meilleur il est. La dernière variable est nommée `kb` (ligne 30), pour « killed by », et indique ce qui a tué le serpent. On finit en affichant (ligne 31) le nouveau serpent si besoin.

Déplacement

Cette partie du code est appelée lorsqu'on souhaite déplacer le serpent dans une direction connue.

```
1 def move(self):
2     """
3     Moves the snake.
4     :return: None
5     """
6     if not self.dead:
7         self.step += 1
```

```
8           self.changedirec(self.brain.output(self.calculinputs()))  # asks the brain to
        ↪   return a value based on its input
9           self.check()
```

Avant de déplacer le serpent, on commence par vérifier qu'il est toujours vivant (ligne 6). On incrémente ensuite la variable `step` de 1 (ligne 7) car il aura fait un « pas » de plus. Ensuite, on change la direction dans laquelle le serpent ira (ligne 8). Pour ce faire, on appel la méthode `changedirec` (voir 2.7.3 Changement de direction) en entrant l'indice le la nouvelle direction (compris entre 0 et 3). Cet indice est donné par une méthode de la classe `Brain` nommée `output` (voir 2.7.5 Sortie). Cette méthode demande également un paramètre qui est une matrice contenant les entrées du réseau neuronal. Cette matrice est calculée et retournée par la méthode `calculinputs` (voir 2.7.3 Calcul des entrées). Le serpent change alors de direction, ou garde la même si la nouvelle n'est pas différente de la précédente. On appelle ensuite la méthode `check` (ligne 9) qui vérifie plusieurs choses et qui exécute les actions nécessaires s'il y en a (voir 2.7.3 Vérification).

Changement de direction

Cette méthode permet de changer la direction vers laquelle le serpent se dirige.

```
1  def changedirec(self, new_direc):
2      """
3      Changes the direction of the snake.
4      :param int (between 0 and 3) new_direc: the new directions the snake will be
   ↪   facing
5      :return: None
6      """
7      if not directions[new_direc] == self.direc and not directions[new_direc] ==
       ↪  (-self.direc[0], -self.direc[1]):
8          self.direc = directions[new_direc]
```

Cette méthode demande un paramètre : la nouvelle direction que l'on veut pour le serpent. On regarde alors si la nouvelle direction demandée n'est pas la même que la direction actuelle pour ne pas changer de direction inutilement. On regarde également si la nouvelle direction n'est pas l'inverse de la direction actuelle pour éviter que le serpent ne recule sur lui-même.

Vérification

Cette méthode est très importante car c'est elle qui gère tout ce qui se produit lors du mouvement du serpent.

```
def check(self):
    """
    Checks if it is going out of the window, touching its tail or eating the fruit.
    Shortens its tail otherwise.
    :return: None
    """
    newhead = (self.coords[0][0] + self.direc[0], self.coords[0][1] + self.direc[1])
    if newhead[0] < dimensions[0] or newhead[0] > dimensions[1] - 1 or newhead[1] <
    ↪   dimensions[0] or newhead[1] > dimensions[1] - 1:  # goes out of the window
        self.kb = "wall"
        self.dead = True
    elif self.coords.count(newhead) == 1:  # touchs its tail
        self.kb = "tail"
        self.dead = True
    elif self.coords[0][0] == self.fruit.pos[0] and self.coords[0][1] ==
    ↪   self.fruit.pos[1]:  # eats the fruit
        self.newhead()
        self.score += 1
        self.fruit.__del__()
        self.fruit = self.newfruit()
        self.fruitslist.append(self.fruit)
    else:
        self.newhead()
        self.shorten()
    if self.step == self.maxsteps:  # reaches the maximum of steps
        self.kb = "steps"
        self.dead = True
```

Avant de déplacer le serpent, on va générer une nouvelle tête temporairement : `newhead` (ligne 7). On vérifie alors premièrement que cette nouvelle tête n'est pas en dehors de la zone jouable (ligne 8), auquel cas le serpent est tué par un des murs (lignes 9 et 10). On vérifie ensuite que le serpent n'est pas en train de se couper la route en regardant si les coordonnées de cette nouvelle tête n'existent pas déjà dans la liste des coordonnées de tous les points du corps du serpent (ligne 11). Le cas échéant, le serpent est tué par sa queue (lignes 12 et 13). A partir de là, on sait que le serpent n'est pas mort à cause d'une mauvaise décision. On peut maintenant regarder si la nouvelle tête se trouve sur le fruit (ligne 14). Si oui,

1. on ajoute alors la nouvelle tête au corps du serpent (ligne 15),

2. on augmente son score de 1 (ligne 16),
3. on supprime le fruit mangé (ligne 17),
4. on définit un nouveau fruit (ligne 18) (voir 2.7.3 Nouveau fruit),
5. et on ajoute ce nouveau fruit à la liste de tous les fruits (ligne 19).

Si aucune de ces vérifications ne s'avère vraie, on ajoute la nouvelle tête au corps du serpent (ligne 21) et on le raccourcit (ligne 22) également pour donner l'illusion d'un mouvement. On finit avec une ultime vérification : est-ce qu'il reste des pas à disposition (ligne 23) ? Si non, le serpent est tué car son nombre de « pas » dépasse la limite maximale fixée par la variable `maxsteps` (lignes 24 et 25).

Nouveau fruit

Cette méthode est appelée lors de la création d'un nouveau fruit.

```
1  def newfruit(self):
2      """
3      Generates a new fruit and returns it
4      :return: Fruit fruit: the new fruit
5      """
6      fruit = Fruit()  # creates a new fruit
7      while (fruit.pos[0], fruit.pos[1]) in self.coords:  # executes while the fruit is
   ↪   in the snake body
8          fruit.pos = fruit.coordsfruit()
9      return fruit
```

On commence par créer un nouveau fruit et on le stocke dans la variable temporaire `fruit` (ligne 6). Ensuite, on change les coordonnées du fruit (ligne 8) (voir 2.7.4 Génération des coordonnées du fruit) tant que le fruit se trouve sur le serpent (ligne 7). Ainsi, le fruit ne peut pas apparaître sur le serpent. On retourne ensuite ce fruit (ligne 9).

Calcul des entrées

Cette méthode de la classe serpent est primordiale pour le bon fonctionnement du réseau neuronal car c'est elle qui calcule les données qui seront ensuite utilisées comme entrées du réseau. Les différentes entrées ont été définies plus haut (voir 2.5 Entrées du réseau de neurones).

```
def calculinputs(self):
    """
    Calculates all the snake's inputs.
    :return: np.ndarray inputs: two-dimensional array wich contains all the inputs + a
    ↪ bias
    """
    eyes_w = [0, 0, 0, 0]  # its vision till a wall (N, S, W, E)
    eyes_t = [0, 0, 0, 0]  # its vision till its tail
    eyes_f = [0, 0, 0, 0]  # its vision till the fruit
    eyes_w[0] = 1 / (self.coords[0][1] - dimensions[0] + 1)  # North wall
    eyes_w[1] = 1 / (dimensions[1] - self.coords[0][1])   # South wall
    eyes_w[2] = 1 / (self.coords[0][0] - dimensions[0] + 1)  # West wall
    eyes_w[3] = 1 / (dimensions[1] - self.coords[0][0])  # East wall
    for i in range(dimensions[1]):  # North tale
        if (self.coords[0][0], self.coords[0][1] - i - 1) in self.coords:
            eyes_t[0] = 1 / (i + 1)
            break
    for i in range(dimensions[1]):  # South tale
        if (self.coords[0][0], self.coords[0][1] + i + 1) in self.coords:
            eyes_t[1] = 1 / (i + 1)
            break
    for i in range(dimensions[1]):  # West tale
        if (self.coords[0][0] - i - 1, self.coords[0][1]) in self.coords:
            eyes_t[2] = 1 / (i + 1)
        break
    for i in range(dimensions[1]):  # East tale
        if (self.coords[0][0] + i + 1, self.coords[0][1]) in self.coords:
            eyes_t[3] = 1 / (i + 1)
            break
    if self.fruit.pos[0] == self.coords[0][0]:  # North or South fruit
        dist = self.coords[0][1] - self.fruit.pos[1]
        if dist > 0:
            eyes_f[0] = dist
        else:
            eyes_f[1] = - dist
    elif self.fruit.pos[1] == self.coords[0][1]:  # West or East tale
        dist = self.coords[0][0] - self.fruit.pos[0]
        if dist > 0:
            eyes_f[2] = dist
        else:
            eyes_f[3] = - dist
    inputs = np.array([[
            eyes_w[0], eyes_w[1], eyes_w[2], eyes_w[3],
            eyes_t[0], eyes_t[1], eyes_t[2], eyes_t[3],
            eyes_f[0], eyes_f[1], eyes_f[2], eyes_f[3], 1  # bias
    ]])
```

```
46        return inputs
```

On commence cette méthode en créant 3 listes (lignes 6,7 et 8). Chacune des listes représente un type d'entrées :

1. `eyes_w` : les murs (ligne 6),
2. `eyes_t` : sa queue (ligne 7) et
3. `eyes_f` : le fruit (ligne 8).

Ces listes sont toutes les trois organisées sous forme de 4 directions : le premier chiffre de la liste représentant le Nord, le deuxième le Sud, le troisième l'Ouest et le quatrième l'Est. Une fois cela établi, on peut passer à la manière de calculer les données à mettre dans ces listes.

1. **Les murs** : Le fonctionnement du calcul pour les murs est le suivant : plus on se rapproche d'un mur, correspondant à une certaine direction, plus la valeur associée à cette direction se rapproche de 1. Par conséquent, si le serpent longe le mur Nord par exemple, la première valeur de la liste `eyes_w` vaudra 1. Les quatre valeurs sont calculées aux lignes 9 à 12. Pour les calculer, on utilise les coordonnées de la tête du serpent (`self.coords[0]`) en x et en y ainsi que les dimensions de la zone jouable (`dimensions`) en x et en y.
2. **Sa queue** : Le fonctionnement du calcul pour sa queue est un peu différent car un serpent n'a pas toujours sa queue dans son visuel. Dans ce cas, la valeur calculée sera 0. Dans le cas ou sa queue se trouve dans une des directions, plus elle est proche, plus la valeur correspondante de la direction concernée sera proche de 1. Pour ce faire, on regarde tour à tour chaque carré unité de la zone jouable le long d'une des directions (lignes 13, 17, 21 et 25). Lorsque sa queue figure sur l'un des carrés analysés (lignes 14, 18, 22 et 26), on utilise le nombre de carrés parcourus comme distance : `i` (lignes 15, 19, 23 et 27). On passe ensuite à la direction suivante (lignes 16, 20, 24 et 28 : on ne passera pas à la direction suivante à cette ligne car il s'agit de la dernière direction ; on arrêtera alors juste la recherche).
3. **Le fruit** : En ce qui concerne le fonctionnement du fruit, on utilise simplement la distance de la tête du serpent au fruit dans une certaine direction, s'il s'y trouve. De la même manière qu'avec sa queue, s'il n'y a pas de fruit le long de la direction, la valeur sera nulle. Pour calculer cette distance, il suffit de regarder si les valeurs x et y de la positon du

fruit sont respectivement égales aux valeurs x et y de la position de la tête du serpent (lignes 29 et 35). Cela permet de déterminer si le fruit ce situe sur l'axe x ou y. Pour spécifier s'il s'agit du Nord plutôt que du Sud ou de l'Est plutôt que de l'Ouest, il faut simplement regarder si la distance obtenue est positive ou négative (lignes 31 et 37). On peut finalement entrer la valeur dans la liste à l'indice correspondant à la bonne direction (lignes 32, 34, 38 et 40).

Une fois que toutes ces valeurs ont été calculées et rangées dans leur liste locale, elle seront stockées (lignes 42 à 44) dans une matrice nommée `inputs` (ligne 41). On va ensuite ajouter le biais (voir 2.3 Fonctionnement d'un réseau neuronal artificiel) à la fin de la matrice (ligne 44). On conclut cette méthode en retournant cette matrice (ligne 46).

Calcul de la fitness

Le calcul de la fitness est un autre élément essentiel du programme. Même si cette méthode n'est qu'une très petite partie de code, elle n'est absolument pas négligeable et résulte d'un raisonnement complexe.

```
1 def calculfitness(self):
2     """
3     Calculates the fitness of the snake.
4     :return: None
5     """
6     self.fitness = (self.score**2)*(1 + (self.score/math.sqrt(self.step) + 1)**3)
```

Cette méthode contient simplement une fonction mathématiques à deux variables : `score` et `step` (ligne 6). Cette fonction est extrêmement importante car c'est elle qui définira si un serpent est fort ou non. Plus un serpent « mange » de fruits, plus il est fort (`score`). Cependant, tout n'est pas si simple, car même si un serpent « mange » beaucoup de fruits, s'il met énormément de temps à le faire, cela ne fait pas forcément de lui un très bon serpent. C'est là qu'intervient le second paramètre : le nombre de mouvements qu'un serpent fait (`step`). A partir de là, il a fallu déterminer un lien entre ces deux paramètres qui soit à la fois juste mais également discriminant. Ainsi, on pourra éliminer les serpents sans « avenir » et ne garder que ceux qui ont une chance.
C'est donc après une certaine réflexion et quelques tests pratiques que j'en

suis arrivé à définir la fonction de deux variables suivante :

$$f(x,y) = x^2 \cdot \left(1 + \left(\frac{x}{\sqrt{y}} + 1\right)^3\right)$$

Cette fonction donne une grande importance au score : la fitness (`fitness` ou $f(x,y)$) augmente de plus en plus si le score (`score` ou x) augmente. Le ratio du score sur le nombre de mouvements (`step` ou y) est également présent dans la formule.
Voici ci-dessous la représentation graphique de la fonction qui permettra sans doute de mieux la comprendre :

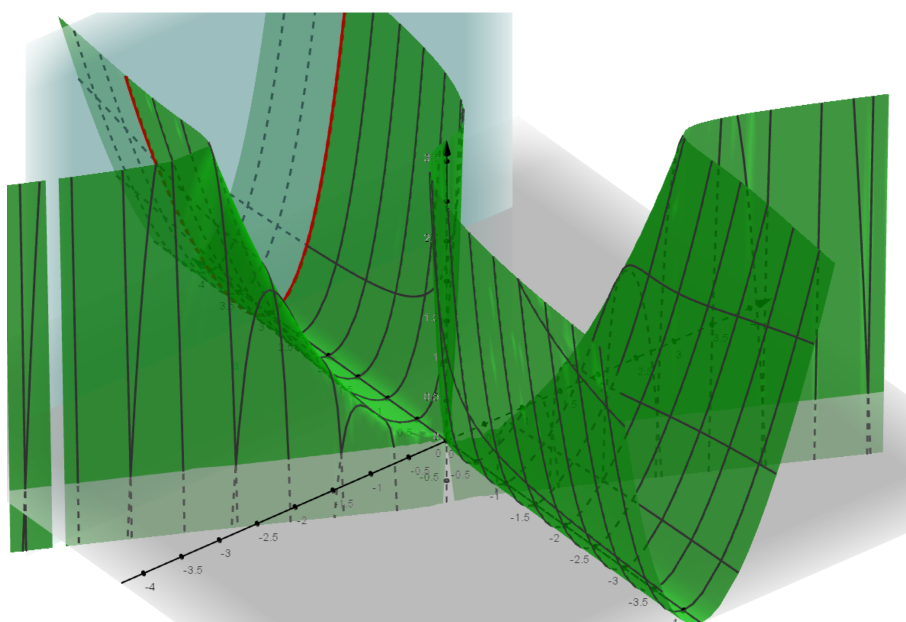


Figure 2.7 – Représentation graphique de la fitness

On peut voir qu'il y a une courbe mise en évidence sur le graphe. Cette courbe résulte de l'intersection entre la fonction et un plan. Ce plan est fixé à $y = 3$. De tel sorte, on voit à la fois la partie de la courbe qui montre l'évolution du score pour un nombre de pas donné, ainsi que la fonction dans sa globalité.

Sauvegarde d'un serpent

Cette méthode permet de sauvegarder un serpent. Grâce à un programme à part, il sera ensuite possible de revisionner une partie jouée par serpent ou de le faire jouer dans un nouvel environnement (voir 2.7.7 Rejouer).

```python
def save(self):
    """
    Saves the snake with pickle.
    :return: None
    """
    pickle.dump(self, open(save_path, 'wb'))
```

Dans cette méthode, on utilise le module `pickle` (voir 2.7.1 Importation des modules) pour sauvegarder le serpent dans un fichier (voir 2.7.1 Variables globales) (ligne 6). On pourra ainsi utiliser ce fichier pour importer ce même serpent dans un autre programme.

Suppression

Cette méthode est exécutée juste avant la surpression d'un serpent.

```python
def __del__(self):
    """
    Deletes its fruit and its display.
    :return: None
    """
    try:
        for i in range(len(self.show)):  # deletes its display
            del self.show
    except:
        pass
    del self.fruit
```

Il est important de supprimer chaque élément d'affichage du serpent avant de le supprimer lui même car sinon, le serpent serait toujours visible dans la fenêtre alors qu'il n'existerait plus. Il faut donc l'effacer de l'interface. Pour ce faire, une instruction s'exécute pour chaque élément d'affichage du serpent (contenus dans la liste `show`, voir 2.7.3 Initialisation) (ligne 7), il s'agit de supprimer l'élément en question (ligne 8).
Un autre élément à supprimer de l'affichage est le fruit actuel que le serpent tentait de « manger » (ligne 11).
Les instructions `try` et `except` permettent d'ignorer certaines erreurs du module `tkinter` (voir 2.7.1 Importation des modules) (lignes 6 et 9).

2.7.4 Fruit

Cette classe permet de générer un fruit que le serpent pourra manger.

Initialisation

Le constructeur de la classe permet d'initier les 2 variables d'instances importantes pour cette classe.

```python
def __init__(self):
    """
    Initialisation method at the creation of a fruit.
    """
    self.pos = self.coordsfruit()  # its position
    self.show = []  # its display
```

La première variable `pos` (ligne 5) concerne la position du fruit dans la fenêtre. Elle est retournée par la méthode `coordsfruit` (voir 2.7.4 Génération des coordonnées du fruit). La seconde variable `show` contiendra l'instance de l'affichage du fruit dans la fenêtre.

Génération des coordonnées du fruit

C'est cette méthode qui va générer les coordonnées aléatoires du fruit dans la fenêtre.

```python
def coordsfruit(self):
    """
    Randomize a position in the window.
    :return: list: a random postion in the window
    """
    return [rdm.randint(dimensions[0], dimensions[1] - 1), rdm.randint(dimensions[0],
    ↪  dimensions[1] - 1)]
```

Dans cette méthode, on retourne directement une liste de longueur 2 (ligne 6). Le premier élément de la liste étant les coordonnées du fruit sur l'axe x et le deuxième élément, celles sur l'axe y. Pour générer ces coordonnées, on utilise le module `random` (voir 2.7.1 Importation des modules). On utilise donc ce module pour générer un nombre aléatoire sur chacun des axes, compris dans les intervalles donnés par la zone jouable.

Suppression

Cette méthode est appelée lors de la suppression du fruit.

```python
def __del__(self):
    """
    Deletes its display.
    :return: None
    """
    try:
        del self.show[0]
    except:
        pass
```

On supprime simplement l'instance d'affichage du fruit stockée dans la liste `show` (ligne 7). Cela est nécessaire car si on ne le fait pas, le fruit n'existera plus mais sera toujours affiché dans la fenêtre.

2.7.5 Cerveau

La classe `Brain` est la classe qui contient le réseau de neurones. C'est cette classe qui reçoit les entrées, exécutes les opérations nécessaires et indique la décision à prendre.

Initialisation

Cette méthode s'exécute lors de la création d'un cerveau. C'est dans cette méthode qu'est fixée la forme du réseau neuronal.

```python
def __init__(self, nb_of_inputs):
    """
    Initialisation method at the creation of a fruit.
    :param int nb_of_inputs: the number of different inputs in the array given by the
    ↪ calculinputs method
    """
    self.w1 = np.random.uniform(-1, 1, (nb_of_inputs, 10))  # the first neurons layer
    self.w2 = np.random.uniform(-1, 1, (10, 8))  # the second neurons layer
    self.w3 = np.random.uniform(-1, 1, (8, 4))  # the third neurons layer
    self.mutation_chance = 0.01  # the mutation probability
    self.id = rdm.uniform(0, 1)  # its unique id
```

On remarque la création de 3 variable d'instances : `w1` (ligne 6), `w2` (ligne 7) et `w3` (ligne 8). Ce sont ces trois variables qui définissent la forme du réseau (voir 2.8 Choix de la forme du réseau). Ce sont donc en réalité (voir 2.3.1 Concept et explications relatif au réseau utilisé) 3 matrices de vecteurs de

poids. À la création d'un cerveau, ces trois matrices seront remplies de manière aléatoire. Le module `numpy` (voir 2.7.1 Importation des modules) est utilisé pour générer et remplir les matrices. La variable `mutation_chance` (ligne 9) définit la probabilité qu'une mutation se produise (voir 2.7.5 Mutation d'une matrice). La variable `id` (ligne 10) stocke simplement un nombre réel généré aléatoirement par le module `random`. Ce nombre sera utilisé en tant qu'identifient unique pour un réseau de neurone.

Tangente hyperbolique et sigmoïde

Ces deux méthodes sont simplement des fonctions mathématiques utilisées dans le réseau de neurones (voir 2.3.2 Mathématiques). J'ai programmé la sigmoïde même si, au final, je n'utilise que la tangente hyperbolique.

```
1 def tanh(self, x):
2     """
3     Hyperbolic tangent function.
4     :param x: float
5     :return: float
6     """
7     return math.tanh(x)
```

La méthode `tanh` (ligne 1) demande un nombre décimal en paramètre. On utilise ensuite le module `math` (voir 2.7.1 Importation des modules) pour retourner la tangente hyperbolique du paramètre (ligne 7). Voici à quoi ressemble le graphique de la tangente hyperbolique :

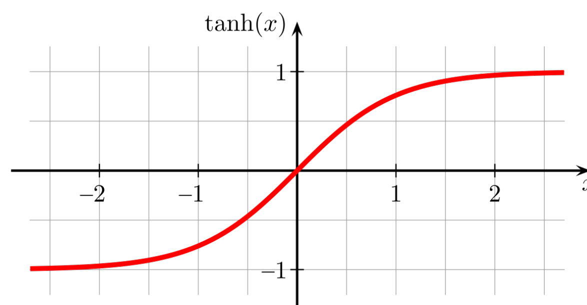


FIGURE 2.8 – Graphique de la tangente hyperbolique

```python
def sigmoid(self, x):
    """
    Sigmoid function.
    :param x: float
    :return: float
    """
    return 1 / (1 + math.exp(-x))
```

La méthode `sigmoid` (ligne 1) demande un nombre décimal en paramètre. On utilise ensuite le module `math` (voir 2.7.1 Importation des modules) pour retourner la sigmoïde du paramètre (ligne 7). Voici à quoi ressemble le graphique de la sigmoïde :

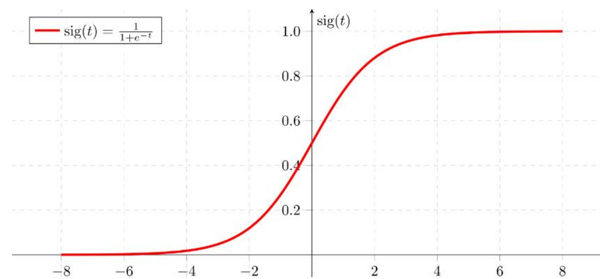


FIGURE 2.9 – Graphique de la sigmoïde

Sortie

C'est précisément dans cette méthode que les calculs mathématiques du réseau de neurones opèrent pour finalement indiquer une décision à prendre.

```python
def output(self, data):
    """
    Calculates the neural network output using matrix calculation and the different
    ↪ weights of the arrays.
    :param np.ndarray data: the input array
    :return: int: the index of the maximum value in the output array
    """
    h1 = np.dot(data, self.w1)  # matrix dot product
    for c in range(h1.shape[1]):
        h1.itemset(c, self.tanh(h1.item(c)))  # unlinear function
    h2 = np.dot(h1, self.w2)  # matrix dot product
    for c in range(h2.shape[1]):
        h2.itemset(c, self.tanh(h2.item(c)))  # unlinear function
    output = np.dot(h2, self.w3)  # matrix dot product
    for c in range(output.shape[1]):
```

```
15            output.itemset(c, self.tanh(output.item(c)))  # unlinear function
16        return self.max(output)
```

La méthode `output` (ligne 1) demande un paramètre nommé `data`. Ce paramètre requis se trouve être la couche d'entrée du réseau de neurones (voir 2.7.3 Calcul des entrées). On utilise alors à nouveau le module `numpy` (voir 2.7.1 Importation des modules) pour faire un produit matriciel entre la couche d'entrée et la première matrice de poids. On stocke alors le résultat dans la variable nommée `h1` (ligne 7). On passe ensuite chacun des nombres obtenus dans la fonction tangente hyperbolique (voir 2.7.5 Tangente hyperbolique et sigmoïde). On obtient finalement notre première couche de neurones dite « cachée ». On réitère ensuite l'opération 2 nouvelles fois pour finir par obtenir la couche de sortie du réseau définie par la variable `output` (ligne 13). Il suffit alors de retourner l'indice de la plus grande valeur de cette matrice de sortie (ligne 16). C'est la méthode `max` qui s'en charge.

Mutation d'une matrice

Cette méthode permet de muter certains éléments d'une des matrices de poids.

```
1   def arraymutate(self, array):
2       """
3       Potentially (depending on the rate: self.mutation_chance) mutates a given array
    ↪   and modifies the brain id if a mutation took place.
4       :param np.ndarray array: a full array
5       :return: np.ndarray array: the potentially modified array
6       """
7       mut = False
8       for r in range(array.shape[0]):
9           for c in range(array.shape[1]):
10              x = rdm.uniform(0, 1)
11              if x < self.mutation_chance:  # if the random number is bigger than the
                ↪  rate => mutation
12                  array[r][c] = rdm.uniform(-1, 1)  # mutates the weight
13                  mut = True
14      if mut:
15          self.id = rdm.uniform(0, 1)  # new unique id
16      return array
```

La méthode `arraymutate` demande une matrice en paramètre : `array` (ligne 1). Pour commencer, on crée une variable nommée `mut` (ligne 7). C'est une

variable qui stockera uniquement des valeurs booléennes. Elle vaudra « faux » tant qu'une mutation ne se sera pas produite. Ensuite, on parcourt la matrice donnée en paramètre en suivant une série d'étapes, ceci pour chaque poids (ligne 8 et 9) :

1. On génère un nombre `x` aléatoirement (ligne 10),
2. On regarde si ce nombre est plus petit que la valeur de la variable `mutation_chance` (voir 2.7.5 Initialisation). Si oui :
   (a) On « mute » le poids en question : on le redéfinit aléatoirement (ligne 12),
   (b) la variable `mut` devient « vraie »(ligne 13).

Si une mutation a eu lieu (ligne 14), le réseau de neurones n'est plus le même. On doit donc définir un nouvel identifiant unique (ligne 15). Pour finir, on retourne la matrice mutée (ligne 16).

Mutations

Cette méthode s'occupe juste de muter chacune des matrices tour à tour.

```
1 def mutate(self):
2         """
3         Applies the arraymutate method to all the weights of the arrays.
4         :return: None
5         """
6         self.w1 = self.arraymutate(self.w1)
7         self.w2 = self.arraymutate(self.w2)
8         self.w3 = self.arraymutate(self.w3)
```

(voir 2.7.5 Mutation d'une matrice et 2.7.5 Initialisation)

2.7.6 Population

Cette classe permet de générer un grand nombre de serpents à la fois et d'interagir avec eux.

Initialisation

Cette méthode est appelée quand on crée une nouvelle population.

```python
def __init__(self, maxsteps, nb_of_snakes=500):
    """
    Initialisation method at the creation of a population.
    :param int maxsteps: how many steps the snakes in the population will have
    ↪ (maximum)
    :param int (even) nb_of_snakes: how many snakes are there in the population
    """
    self.pop = []  # the list of all the snakes in the population
    self.maxsteps = maxsteps  # maximum of steps for each snake
    for i in range(nb_of_snakes):
        self.pop.append(Snake(maxsteps=maxsteps))
    self.fitness_sum = 0  # the sum of all the snakes' fitness
```

Lors de la création d'une population, il faut indiquer deux paramètres (ligne 1) :

1. `maxsteps` : le nombre de pas maximum qu'un serpent à le droit de faire,
2. `nb_of_snakes` : le nombre de serpents que l'on veut dans notre population.

On commence par créer une liste `pop` (ligne 7). Cette liste contiendra tous les serpents de la population. On stocke le paramètre `maxsteps` dans une variable du même nom (ligne 8). Il faut ensuite remplir la liste vide que l'on vient de créer avec le nombre de serpents que l'on souhaite avoir (ligne 9 et 10). Pour finir, on définit une dernière variable nommée `fitness_sum`. Elle sera utilisée par la suite pour contenir la somme de toutes les fitness de tous les serpents.

Déplacements

Cette méthode permet de déplacer tous les serpents de la population.

```python
def move(self):
    """
    Moves all the snakes in the population.
    :return: None
    """
    for i in range(len(self.pop)):
        self.pop[i].move()  # moves the snake
```

Pour chaque serpent de la population (ligne 6), on exécute sa méthode `move` (voir 2.7.3 Déplacement) (ligne 7).

Sont-ils tous morts ?

Cette méthode permet de savoir si tous les serpents d'une certaine population sont morts où s'il reste des serpents vivants.

```
1  def all_dead(self):
2      """
3      Returns True if all the snakes are dead.
4      :return: bool res
5      """
6      res = 0  # how many snakes are dead
7      for i in range(len(self.pop)):
8          if self.pop[i].dead:
9              res += 1
10     return res == len(self.pop)
```

On définit une variable locale `res`. On regarde ensuite pour chaque serpent de la population (ligne 7) s'il est mort (ligne 8). Si oui, on incrémente la variable `res` de 1 (ligne 9). On retourne ensuite une valeur booléenne résultant de la comparaison entre le nombre de serpents morts (`res`) et le nombre total de serpents dans la population en question (ligne 10).

Affichage

La programmation de cette partie n'est pas très intéressante à détailler,je me contenterai donc de décrire le processus d'affichage.
Lorsque tous les serpents jouent leur partie, rien n'est affiché dans la fenêtre. Cela permet entre autres d'exécuter les parties le plus vite possible. Ce n'est qu'une fois que tous les serpents sont morts que l'on affiche quelque chose : on rejoue la partie du meilleur serpent de la population. Ainsi, on peut voir à quel point une génération est forte ou non.
Il existe également la possibilité de voir tous les serpents d'une population de manière simultanée pendant qu'ils jouent. Cependant je ne détaillerai pas plus que cela cette option car, une fois activée, il devient difficile de voir ce qui se passe dans la fenêtre vu le nombre de serpents affichés simultanément.

Calcul des fitness

Cette méthode permet de calculer la fitness de tous les serpents de la population.

```python
def calculfitness(self):
    """
    Calculates the fitness of each snake.
    :return: None
    """
    for i in range(len(self.pop)):
        self.pop[i].calculfitness()
```

On exécute la méthode `calculfitness` (voir 2.7.3 Calcul de la fitness) (ligne 7), pour chaque serpent de la population (ligne 6).

Croisement

Le croisement ou « crossover » en anglais fait partie du processus d'application d'un algorithme génétique. C'est un procédé qui consiste à prendre deux individus, appelés « parents », et à intervertir de manière aléatoire certains de leurs gènes pour ensuite donner ce que l'on appelle des « enfants ». Il existe plusieurs techniques de croisement ; celle que j'ai choisie se nomme « croisement uniforme » : on définit la probabilité que deux gènes soient intervertis. Voici ci-dessous un petit schéma explicatif :

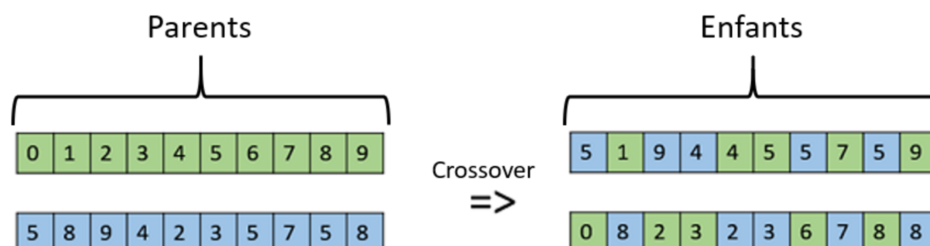


FIGURE 2.10 – Visualisation du croisement

On retrouve les deux parents à droite et les deux enfants à gauche. Chaque gène est représenté par un carré contenant un chiffre et une couleur. Il y a ensuite une chance sur deux d'être intervertis pour chacun des couples de gènes parents. On obtient finalement deux enfants.
Dans le cas de mon programme, les parents sont les serpents d'une population et les gènes sont chacun des poids de leur réseau neuronal. La partie du code correspondant au croisement se trouve à la page suivante.

```python
def crossover(self):
    """
    Mixing individuals in the population.
    :return: None
    """
    for i in range(int(len(self.pop) / 2)):
        for r in range(self.pop[2 * i].brain.w1.shape[0]):
            for c in range(self.pop[2 * i].brain.w1.shape[1]):
                if rdm.randint(0, 1):
                    self.pop[2 * i].brain.w1[r][c], self.pop[2 * i + 1].brain.w1[r][c]
                    ↪ = self.pop[2 * i + 1].brain.w1[r][c], self.pop[2 *
                    ↪ i].brain.w1[r][c]
        for r in range(self.pop[2 * i].brain.w2.shape[0]):
            for c in range(self.pop[2 * i].brain.w2.shape[1]):
                if rdm.randint(0, 1):
                    self.pop[2 * i].brain.w2[r][c], self.pop[2 * i + 1].brain.w2[r][c]
                    ↪ = self.pop[2 * i + 1].brain.w2[r][c], self.pop[2 *
                    ↪ i].brain.w2[r][c]
        for r in range(self.pop[2 * i].brain.w3.shape[0]):
            for c in range(self.pop[2 * i].brain.w3.shape[1]):
                if rdm.randint(0, 1):
                    self.pop[2 * i].brain.w3[r][c], self.pop[2 * i + 1].brain.w3[r][c]
                    ↪ = self.pop[2 * i + 1].brain.w3[r][c], self.pop[2 *
                    ↪ i].brain.w3[r][c]
```

Comme le croisement de serpents fonctionne par couples, on commence par diviser la population en deux : l'ensemble des serpents d'indice pair $(2 \cdot i)$ et celui correspondant aux indices impairs $(2 \cdot i + 1)$ avec i variant entre 0 et le nombre total de serpents de la population divisé par 2 (sans compter le dernier). Pour chacun des couples formés (ligne 6), on va parcourir leurs matrices de poids respectives (lignes 7 et 8, 11 et 12 ainsi que 15 et 16). On génère ensuite un nombre entier aléatoire entre 0 et 1 pour chaque élément de la matrice (ligne 9, 13 et 17). Si le résultat obtenu est 0, on ne fait rien et on passe au poids suivant. Cependant, si le résultat est 1, on inverse alors les poids respectifs des deux parents (lignes 10, 14 et 18) :

```
poids_1, poids_2 = poids_2, poids_1
```

Sélection naturelle

Cette méthode intervient lorsque tous les serpents d'une population sont morts. Elle exécute les actions nécessaires et renvoie une nouvelle population.

```
def naturalselection(self, maxsteps):
    """
    Generates a new population based on itself.
    :param int maxsteps: How many steps the old poppulation had
    :return: Population new_pop: The new population
    """
    self.calculfitness()
    self.sumfitness()
    self.pop.sort(key=self.fitness)  # sorts the snake by fitness
    self.pop[-1].save()  # saves the best snake
    overwatch(self.pop[-1])  # displays the best snake
    if pop.pop[-1].kb == "steps":
        maxsteps += 20
    new_pop = Population(maxsteps)  # creates a new population
    display.canv.itemconfig(display.maxsteps, text="Max steps: {}".format(maxsteps))
    for i in range(len(new_pop.pop)):
        individual = self.selectindividual()  # slections a snake
        new_pop.pop[i].brain.w1 = individual.brain.w1.copy()  # copies its neural
        ↪ network
        new_pop.pop[i].brain.w2 = individual.brain.w2.copy()
        new_pop.pop[i].brain.w3 = individual.brain.w3.copy()
    new_pop.crossover()  # mixes the snakes with each other
    for i in range(len(new_pop.pop)):
        new_pop.pop[i].brain.mutate()  # mutates all the snakes
    new_pop.pop[-1].brain.w1 = self.pop[-1].brain.w1.copy()  # inculdes the best snake
    ↪ of this population in the new one (without mutation)
    new_pop.pop[-1].brain.w2 = self.pop[-1].brain.w2.copy()
    new_pop.pop[-1].brain.w3 = self.pop[-1].brain.w3.copy()
    return new_pop
```

La méthode `naturalselection` demande le paramètre `maxsteps` (ligne 1), ce qui est logique, car lors de la création d'une nouvelle population, on doit donner le nombre de pas maximum (voir 2.7.6 Initialisation). On commence la méthode en calculant la fitness de chacun des serpents (ligne 7) (voir 2.7.6 Calcul des fitness). On calcule ensuite la somme de toutes les fitness (ligne 8). Après, on trie la population de serpents pour qu'elle soit organisée par ordre croissant de fitness (ligne 9). Le dernier serpent de la liste `pop` sera donc le meilleur de la population. On sauvegarde alors ce serpent (ligne 10) (voir 2.7.3 Sauvegarde d'un serpent). On peut maintenant rejouer la partie de ce même serpent (ligne 11) (voir 2.7.7 Rejouer).
Si le meilleur des serpents a été tué parce qu'il lui manquait des « pas » pour continuer (ligne 12), on donnera alors 20 « pas » de plus à la génération suivante (ligne 13). Cela permet d'éviter de donner systématiquement plus

de « pas » aux serpents d'une certaine population et ainsi de réduire le temps de calcul liée à la durée de vie de cette génération.
On continue en créant une population entièrement nouvelle (ligne 14). On actualise l'affichage du texte du nombre maximal de « pas » au cas où il aurait été augmenté (ligne 15). On sélectionne ensuite des nouveaux serpents (ligne 17) autant de fois que l'on veut de serpents dans la nouvelle population (ligne 16). Le processus de sélection (`selectindividual`) ne sera pas plus détaillé, car j'ai utilisé exactement le même que lors de mon premier programme (voir 1.4.2 La sélection des candidats). On copie ensuite les trois matrices de poids du cerveau du serpent sélectionné dans le cerveau d'un serpent de la nouvelle population (lignes 18, 19 et 20). Une fois cela fait, on croise les serpents de cette nouvelle population entre eux (voir 2.7.6 Croisement) (ligne 21). Pour éviter de tourner en rond et pour faire évoluer les serpents, il est nécessaire d'inclure une phase de mutation. On va donc muter (ligne 23) chaque serpent de la nouvelle population (ligne 22). Pour finir, on copie également le cerveau du meilleur serpent de l'ancienne population dans un serpent de la nouvelle population pour éviter une régression (lignes 24, 25 et 26). On termine en retournant cette nouvelle population (ligne 27).

Suppression

Cette méthode est exécutée juste avant la suppression d'une population.

```
1 def __del__(self):
2     """
3     Deletes all the snakes in the population.
4     :return: None
5     """
6     for i in range(len(self.pop)):
7         del self.pop[0]
```

On y supprime (ligne 7) simplement tous les serpents contenus dans la population (ligne 6).

2.7.7 Rejouer

La fonction `overwatch` permet, une fois que tous les serpents d'une population sont morts, de rejouer directement la partie d'un des serpents.

```
1 def overwatch(snake):
2     """
```

```
3        Replays the game of a given snake.
4        :param snake: Snake
5        :return: None
6        """
7        display.canv.itemconfig(display.score, text="Score: {}".format(snake.score))  #
         ↪  displays it score
8        display.canv.itemconfig(display.killedby, text="KB: {}".format(snake.kb))  #
         ↪  displays what killed it
9        ow = Snake(maxsteps=snake.maxsteps, display=True, ow=True)  # creates a new snake
10       ow.brain.w1 = snake.brain.w1.copy()  # copies its neural network
11       ow.brain.w2 = snake.brain.w2.copy()
12       ow.brain.w3 = snake.brain.w3.copy()
13       ow.fruitslist = snake.fruitslist[:]  # copies its fruits list
14       ow.fruit = ow.fruitslist[0]  # its first fruit
15       ow.fruit.show = [Square(ow.fruit.pos[0], ow.fruit.pos[1], "green")]
16       while not ow.dead:
17           ow.move()
18           display.update()
19           time.sleep(0.03)
20       del ow
21       display.canv.itemconfig(display.score, text="Score: [...]")
22       display.canv.itemconfig(display.killedby, text="KB: [...]")
```

Cette fonction prend un paramètre `snake` (ligne 1) de type `Snake`. C'est ce serpent qui sera rejoué et affiché. On commence par afficher des informations relatives à la partie du serpent (lignes 7 et 8). Comme il a déjà joué et qu'on se contente juste de lui faire rejouer exactement la même partie, on sait « à l'avance » ce qui l'a tué et le score qu'il a fait.

On crée ensuite un serpent temporaire nommé `ow` (ligne 9). On donne en paramètre autant de « pas » que le serpent que l'on veut rejouer disposait (`maxsteps`), on dit qu'on veut l'afficher (`display`) et que c'est un « overwatch » (`ow`). On copie ensuite les matrices du réseau de neurones du serpent que l'on veut rejouer dans ce serpent temporaire (`w1`, `w2`, `w2`, lignes 10, 11 et 12), ainsi que la liste de tous ses fruits (`fruitslist`, ligne 13). On indique ensuite au serpent temporaire le premier fruit qu'il doit utiliser (`fruit`, ligne 14). Après cela, on affiche le fruit dans la fenêtre (`fruit.show`, ligne 15).

On entre maintenant dans une boucle qui continuera de s'exécuter tant que le serpent n'est pas mort (ligne 16). Le contenu de la boucle est formé de la série d'instructions suivante :

1. on fait bouger le serpent (voir 2.7.3 Déplacement) (ligne 17),
2. on actualise l'affichage (voir 2.7.2 Affichage) (ligne 18),

3. et on attend un très bref délai (ligne 19) permettant à l'utilisateur du programme de voir ce qui se passe.

Une fois que la partie se termine, on supprime le serpent temporaire (ligne 20) et on actualise les textes affichés dans la fenêtre (lignes 21 et 22).

2.7.8 Boucle principale

Passons maintenant à la boucle exécutée dans le programme. C'est là que sont données les instructions et que sont gérés les multiples objets.

```
while True:
    try:
        if not pop.all_dead():
            pop.move()
        else:
            new_pop = pop.naturalselection(pop.maxsteps)
            del pop
            gen += 1
            display.canv.itemconfig(display.gen, text="Gen: {}".format(gen))
            pop = new_pop
            display.update()
    except:
        del pop
        break
```

On commence par entrer dans une boucle sans fin (ligne 1). En réalité, elle n'est pas sans fin car les instructions `try` (ligne 2) et `except` (ligne 12) sont là pour intercepter une éventuelle fermeture de la fenêtre par l'utilisateur et arrêter le programme.
Tant que la fenêtre reste ouverte, les instructions suivantes s'exécutent :

1. On regarde s'il reste des serpents vivants (ligne 3). Si oui, on les fait bouger (voir 2.7.3 Déplacement) (ligne 4).
2. Si non, on commence par créer une nouvelle population `new_pop` (ligne 6) retournée par la méthode `naturalselection` appliquée à l'ancienne population (voir 2.7.6 Sélection naturelle).
3. On supprime ensuite l'ancienne population (ligne 7).
4. Une fois cela fait, on incrémente la variable `gen` de 1 (voir 2.7.1 Variables globales) (ligne 8) et on actualise son affichage (ligne 9).
5. On définit alors la nouvelle population `new_pop` comme étant la population courante `pop` (ligne 10).

6. On termine en actualisant l'affichage de la fenêtre (ligne 11).

Si l'utilisateur ferme la fenêtre, la population courante se voit alors supprimée (ligne 13) et la boucle est interrompue (ligne 14). C'est la fin du programme.

2.8 Choix de la forme du réseau

Une fois tout le programme rédigé, il a encore fallu optimiser la forme du réseau de neurones (voir 2.3 Fonctionnement d'un réseau neuronal artificiel). Comme il n'y a pas de manière de définir précisément la forme d'un réseau en fonction d'un problème donné, j'ai du tester différentes formes à fin d'obtenir une intelligence artificielle optimale. J'ai donc laissé tourner mon programme quelques temps tout en relevant le score maximal de chaque génération. J'ai réalisé chacun de mes tests sur un total d'un peu plus de 1000 générations. La durée moyenne d'un test est d'environ 8 heures. L'ampleur de cette durée peut paraître démesurée car les premières générations ne durent que quelques secondes seulement. Cela s'explique par le fait que plus les serpents deviennent forts, plus les calculs liés aux générations prennent de temps. Voici maintenant les résultats que j'ai obtenu en testant plusieurs formes de réseaux différentes :

1. J'ai commencé mes tests en choisissant la forme de réseau neuronal la plus simple possible : la couche d'entrée reliée directement à la couche de sortie. Cela donne la forme 13×4.

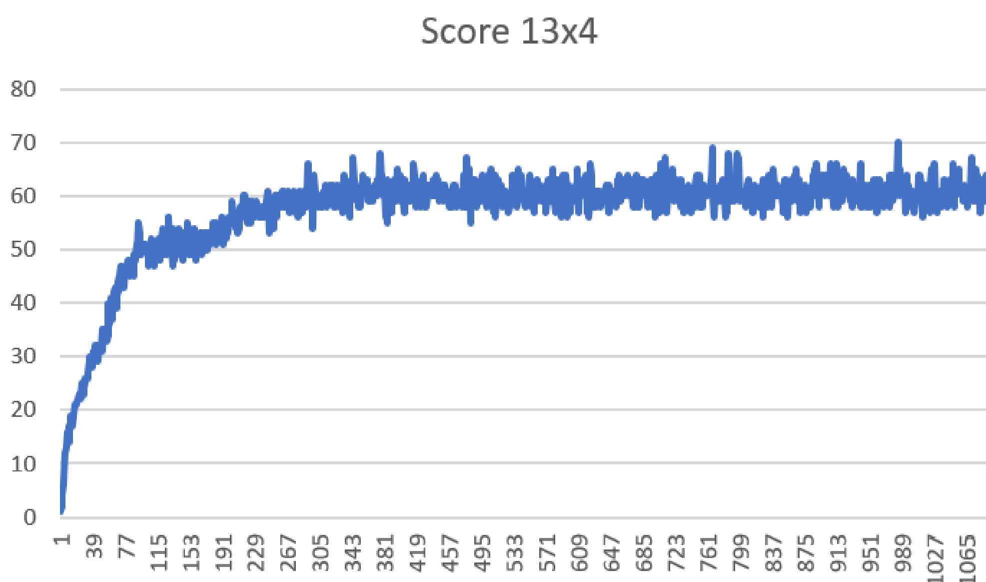


Figure 2.11 – Score en fonction du nombre de générations avec un réseau 13x4

Ce que l'on remarque ici c'est que le début de l'apprentissage se passe de manière très rapide. Les serpents apprennent très vite les règles du jeu et tiennent compte des raisons de leur mort. Cependant, le score plafonne autour de 60 points.

2. J'ai continué mes tests en ajoutant une couche de neurones « cachée ». J'ai décidé d'y mettre 8 neurones. Cela donne alors un réseau de forme $13 \times 8 \times 4$.

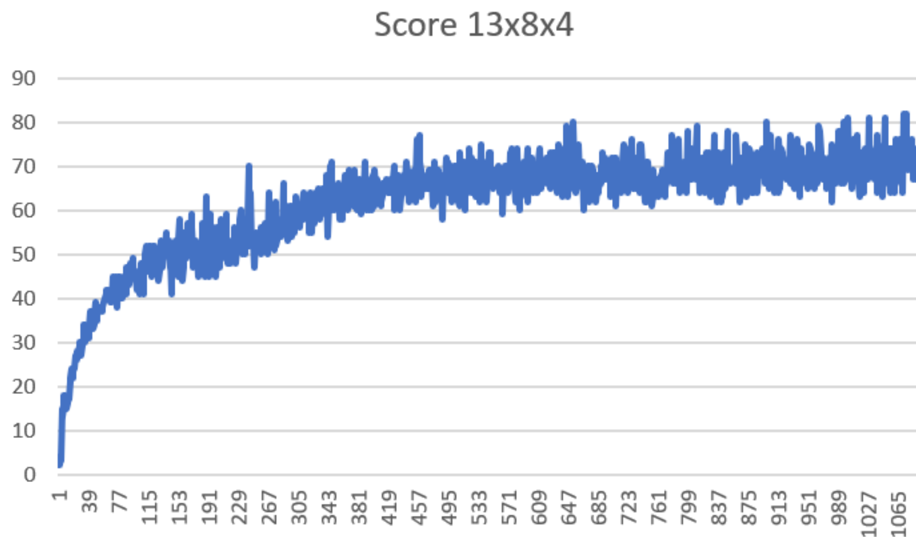


Figure 2.12 – Score en fonction du nombre de générations avec un réseau 13x8x4

On remarque de nouveau un apprentissage initial très rapide suivi d'une stagnation du score. Cependant, cette fois ce dernier se stabilise autour de 70 points.

3. J'ai donc décidé de réitérer l'opération en rajoutant à nouveau une couche « cachée » au réseau. Je donne donc au réseau la forme $13 \times 10 \times 6 \times 4$.

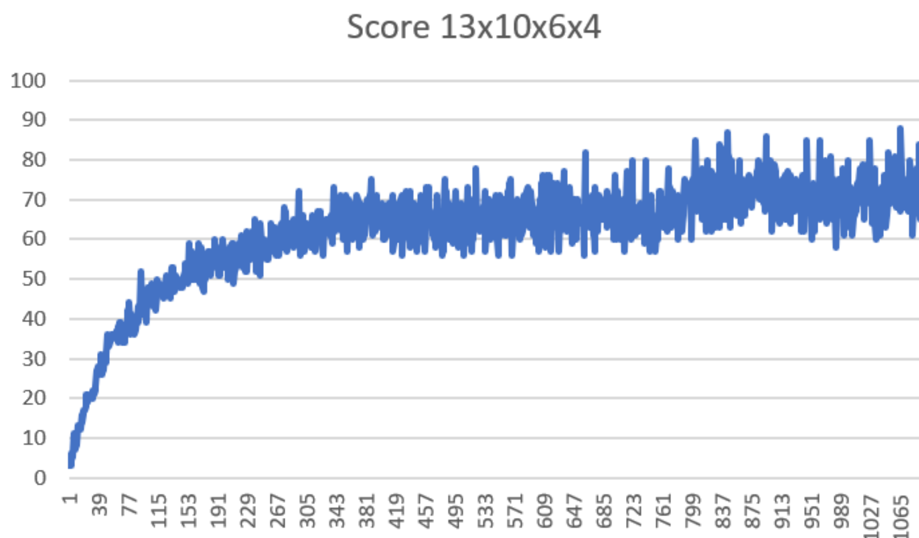


Figure 2.13 – Score en fonction du nombre de générations avec un réseau 13x10x6x4

Sans compter quelques pics autour de 90 points, le graphe semble très similaire à celui du test précédent : forte augmentation au début puis stabilisation à 70 points.

D'autres tests ont bien évidemment été réalisés mais j'ai choisi de ne détailler que les résultats les plus concluants et intéressants.

En généralisant ces observations, on conclut que les serpents apprennent de manière très intense au début mais semblent avoir beaucoup de peine à s'améliorer par la suite. Cela est sans doute dû à la limite placée sur la quantité d'entrées du réseau de neurones (voir 2.5 Entrées du réseau de neurones). En effet, si l'on donnait toutes les informations de la fenêtre en entrées au réseau, on atteindrait sans doute le score maximal mais cela n'est pas possible vu la capacité de calcul limitée de mon ordinateur. Cependant, malgré les entrées extrêmement limitées que l'on donne aux serpents, ils parviennent tout de même à réaliser un très bon score ; 103 points pour le meilleur des essais que j'ai pu enregistrer dans ce travail. Pour conclure cette section, je dirais donc que la meilleure forme de réseau est la dernière car même si le nombre de neurones augmente drastiquement le temps de calcul, cela permet d'obtenir de très bon résultats avec le peu de valeurs données aux serpents.

2.9 Programme de récupération des serpents

J'ai écrit ce nouveau programme dans le but de pouvoir interagir avec les serpents sauvegardés. Je vais expliquer son fonctionnent mais je ne dé-

taillerai pas le code en lui-même. Lorsqu'on lance le programme, l'utilisateur doit saisir le nom du fichier du serpent qu'il veut récupérer. Pour ce faire, l'utilisateur a deux options :

1. entrer « last », ce qui aura pour effet de récupérer le dernier serpent enregistré,
2. ou entrer le nom du fichier du serpent qu'il veut récupérer.

Ensuite, il faut entrer une nouvelle information avant de voir le serpent jouer. On a de nouveau 2 possibilités :

1. entrer « load » pour simplement rejouer la dernière partie du serpent récupéré,
2. ou « play » pour faire jouer le serpent avec de nouveaux fruits générés aléatoirement jusqu'à ce qu'il meurt.

Une fois ces deux paramètres spécifiés, on voit la fenêtre s'ouvrir avec le serpent qui joue. Des informations relatives à la partie sont affichées en dessous de la zone jouable.

Conclusion

Ce travail m'a permis d'avancer énormément dans les domaines combinés de la programmation et de d'intelligence artificielle. Beaucoup de temps a été investi dans ce dossier, mais cela valait vraiment la peine. Les deux programmes mis en œuvre donnent tous deux des résultats très intéressants. La plus grande difficulté que j'ai rencontré fût la résolution de certains bugs qui paraissait parfois impossible. Je n'avais jamais entrepris un projet aussi important que celui-ci et cela m'a appris à travailler de manière régulière et assidue. De plus, l'apprentissage du langage de programmation LaTeX, au sujet duquel je restais sceptique au début, m'a semblé de plus en plus utile et intuitif au fur et à mesure de la rédaction de ce dossier. Je suis convaincu que j'utiliserai à nouveau ce langage dans mes projets futurs. Je pense sans doute poursuivre plus tard mon travail sur l'intelligence artificielle dans une optique plus théorique et philosophique en abordant ce thème d'un point de vue éthique (rôles et implications dans notre société). En effet, si j'arrive a programmer un réseau de neurones artificiel à 17 ans dans ma chambre, je me demande où vont mener les recherches futures des nombreux scientifiques autour de ce sujet encore très récent.

Annexes

Voici ci-dessous les liens « github » des quatre programmes que j'ai écrits tout au long de ce travail. Ils y sont présentés dans leur intégralité. J'ai pris soin de les commenter afin qu'on puisse les parcourir sans trop de difficultés.

1. Le premier programme que j'ai écrit est intitulé `Dots_ai` (voir 1 Dots Ai). C'est le programme dans lequel les points d'une population apprennent par eux mêmes à contourner un ou plusieurs obstacles pour arriver à un objectif.
   Dots Ai : `https://github.com/Lucas-Cours/TM/blob/master/Dots_ai.py` (277 lignes)
2. Le second programme que j'ai rédigé est intitulé `Snake_game` (voir 2.2 Programmer le jeu). J'ai eu besoin d'écrire ce programme pour avoir le contrôle total du jeu et pouvoir implanter l'intelligence artificielle de manière maîtrisée. Cette version du programme est jouable avec les touches directionnelles du clavier.
   Snake Game : `https://github.com/Lucas-Cours/TM/blob/master/Snake_game.py` (240 lignes)
3. Le troisième programme que j'ai écrit est intitulé `Snake_ai` (voir 2 Snake Ai). C'est le programme dans lequel une population de serpents apprend les conditions à respecter pour rester en vie, tout en « mangeant » le plus de fruits que possible.
   Snake Ai : `https://github.com/Lucas-Cours/TM/blob/master/Snake_ai.py` (649 lignes)
4. J'ai nommé `Snake_replay` le quatrième et dernier programme que j'ai rédigé (voir 2.9 Programme de récupération des serpents). Ce programme apporte la possibilité d'interagir avec les serpents sauvegardés dans le programme `Snake_ai`.
   Snake Replay : `https://github.com/Lucas-Cours/TM/blob/master/Snake_replay.py` (510 lignes)

Bibliographie[1]

Réseau neuronal :

- Introduction aux réseaux de neurones : https://towardsdatascience.com/understanding-neural-networks-19020b758230
- En savoir un peu plus sur les réseaux de neurones : https://towardsdatascience.com/nns-aynk-c34efe37f15a
- Explications simples à propos de l'algorithme génétique : https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html
- Explications détaillées à propos du croisement : https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm
- Explications du fonctionnement d'un réseau de neurones : https://www.youtube.com/watch?v=aircAruvnKk
- Rôle du biais dans un réseau neuronal artificiel : https://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks
- Chaîne Youtube de Code Bullet : https://www.youtube.com/channel/UC0e3QhIYukixgh5VVpKHH9Q
- Génération graphique d'un réseau de neurones : http://alexlenail.me/NN-SVG/index.html

LaTeX :

- Bases de LaTeX : https://fr.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes
- Documentation complète du langage de programmation : https://en.wikibooks.org/wiki/LaTeX
- Modèle de format de documents : https://github.com/aroquemaurel/LaTeX-Template
- En-têtes et pieds de page : http://ww2.ac-poitiers.fr/math/spip.php?article267

Images :

- Couverture : https://engineering.fb.com/ml-applications/facebook-to-open-source-ai-hardware-design/
- Fonction sigmoïde : https://e-nature.org/machine-learning/derivee-de-la-fonction-sigmoide/
- Fonction tangente hyperbolique : https://fr.wikipedia.org/wiki/Tangente_hyperbolique

1. Toutes les sources citées ici ont été consultées pour la dernière fois le 16 octobre 2019.