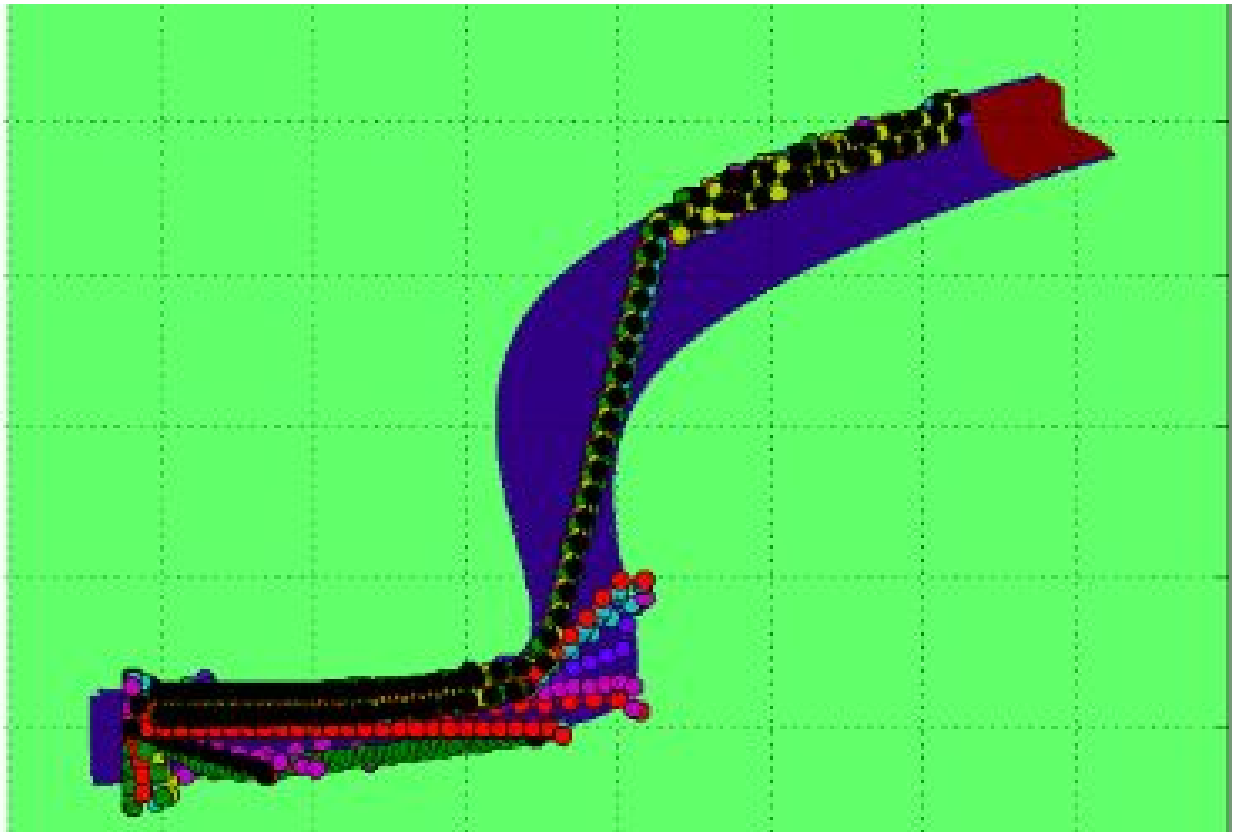


Apprentissage artificiel appliqué au contrôle d'un véhicule autonome

2019

OZDEMIR Serdar

I- Introduction

Les voitures autonomes sont en passe de devenir une réalité , Tesla Motors prévoit l'activation d'une conduite autonome de niveau 5 (aucune intervention humaine nécessaire) pour le grand public d'ici 2020 et beaucoup d'autres constructeurs ont des travaux en cours sur le sujet. Devant l'actualité il devient donc essentiel de comprendre comment de tels véhicules permettent le transport des biens et des personnes sans aucune assistance et notamment les algorithmes d'apprentissage qui permettent de réagir à toutes les situations.

Pour ce faire j'ai créée un modèle simpliste se focalisant uniquement sur l'évitement d'obstacles sur un circuit pour couvrir l'aspect le plus basique de la conduite autonome : éviter un accident.

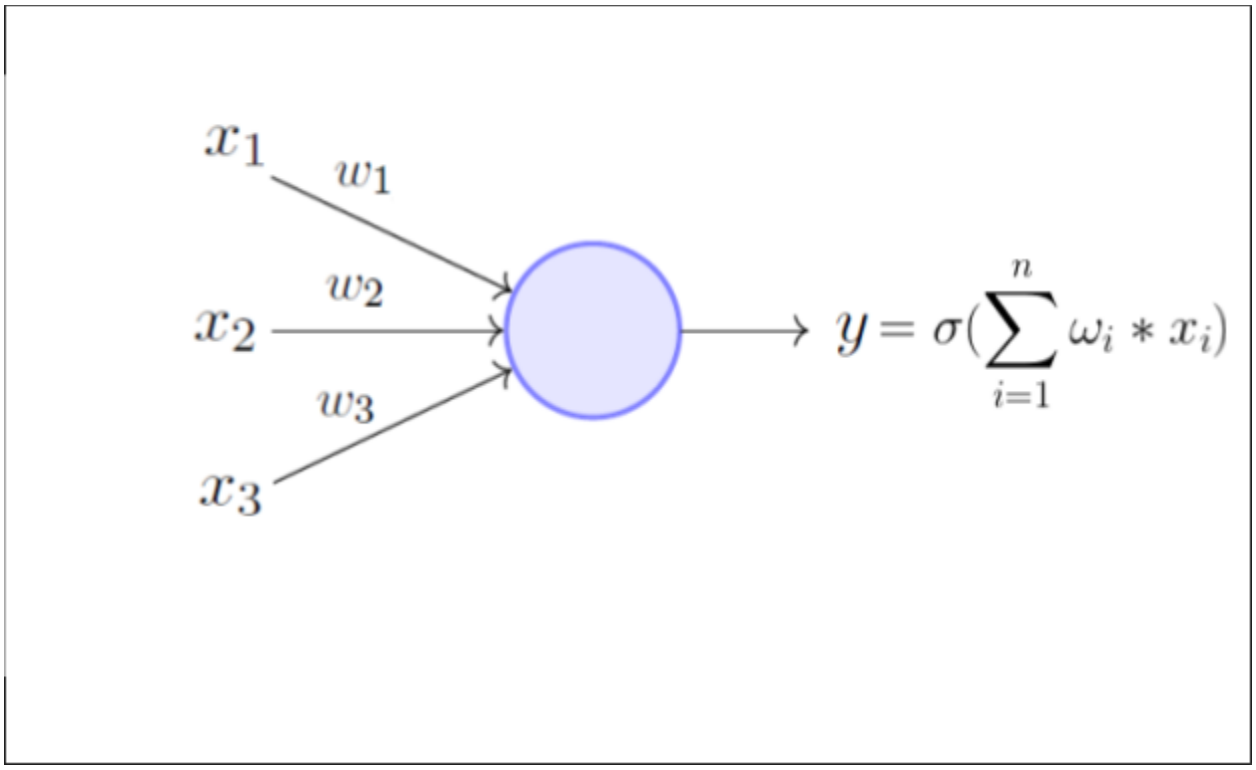
Dans mon modèle, basé sur la NeuroEvolution telle que décrite par Ronald & Schoenauer [1] , les voitures sont des individus assimilés à des points avec un champ de vision, une vitesse et un angle. Ces voitures se déplacent sur un circuit qui sera dans les faits un circuit dessiné puis interprété mais dont les voitures n'auront pas connaissance. Le déplacement se fait au travers d'un réseau de neurones gérant l'angle et la vitesse en temps réel et dont les poids sont amenés à évoluer par un algorithme génétique jusqu'à former un ou des individus capables de conduire seuls.

II- L'exécution

1) Le réseau de neurones

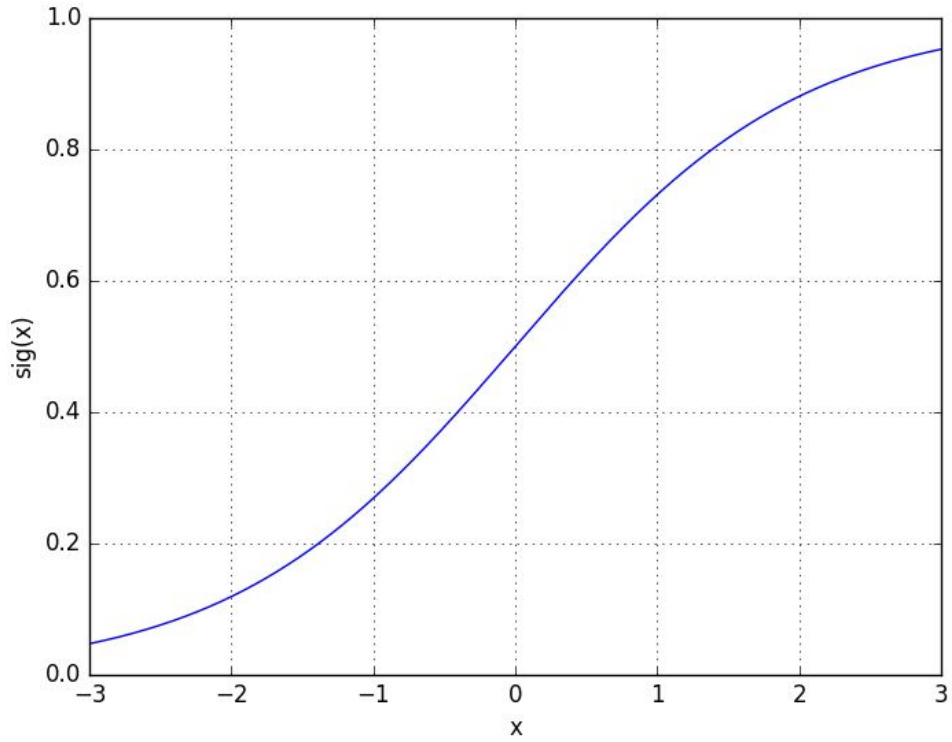
Avant de pouvoir s'intéresser aux déplacements du véhicule, il faut créer son "cerveau" c'est à dire le réseau de neurones qui en fonction d'une entrée (son champ de vision) choisit un angle et une vitesse adéquate pour ne pas avoir d'accident.

Tout d'abord il faut comprendre le fonctionnement d'un neurone: un neurone est un objet mathématique prenant des entrées multiples (x) avec des poids multiples (w) et donnant un nombre en sortie (borné selon les valeurs d'une fonction d'activation σ)



Ici la fonction d'activation retenue est la fonction sigmoide : il s'agit d'une bijection ramenant notre combinaison linéaire à un nombre unique entre 0 et 1 :

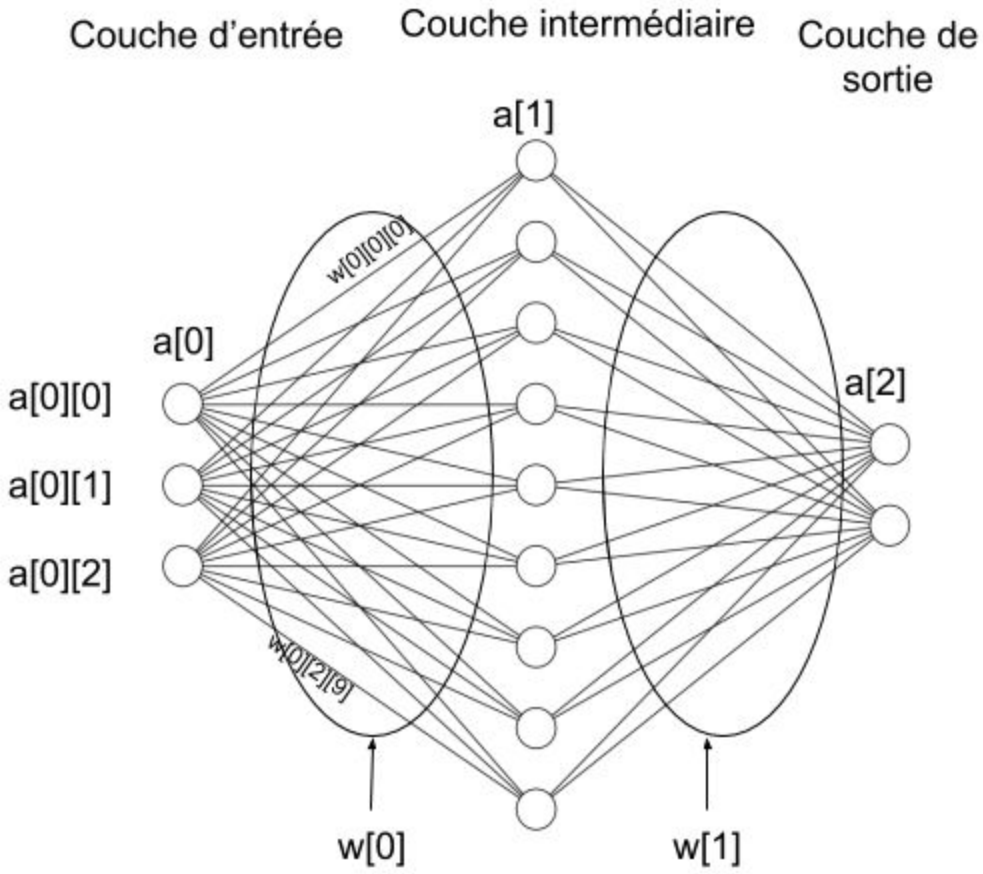
$$\begin{aligned} \sigma : \mathbb{R} &\to [0;1] \\ x &\mapsto \frac{1}{1+e^{-x}} \end{aligned}$$



D’autres fonctions d’activation existent et peuvent être utilisés (notamment la fonction tangente hyperbolique) du moment que l’ensemble d’arrivée est [0;1] ou [-1,1] mais la sigmoide est plus généralement utilisée car très simple à calculer.

Un réseau de neurone est simplement un empilement de neurones en couches , notamment en couches d’entrée et d’arrivée dont le choix de la topologie est un problème à lui seul. C’est pourquoi nous avons choisi un réseau en [3;9,2] (après avoir essayé plusieurs topologies) car c’est celui qui offrait les meilleures performances sans trop complexifier les calculs .

En voici le graphe :



On code ce réseau de neurones en deux parties :

- la liste des activations ***a*** telle que ***a[i][j]*** contienne la valeur de sortie du j ème neurone de la couche i
- la liste des poids ***w*** telle que ***w[i][j][k]*** contienne la valeur du poids de la connexion entre le neurone **j** de la couche **i** et le neurone **k** de la couche **i+1**

Le résultat en fonction de l'entrée d'un réseau de neurones se fait alors par **propagation** : on calcule les activations des neurones de proche en proche jusqu'à la couche de sortie selon la formule suivante:

$$a[i+1][j] = \sigma(\sum_{k=1}^{n} w[i][k][j] * a[i][k])$$

Cette disposition des couches nous permet alors de traduire la propagation en termes matriciels et d'accélérer les calculs :

$$a[i+1] = \sigma(\begin{bmatrix} a[i][0] \\ \dots \\ a[i][n] \end{bmatrix} \begin{bmatrix} w[i][0][0] & \dots & w[i][0][m] \\ \dots & \dots & \dots \\ w[i][n][0] & \dots & w[i][n][m] \end{bmatrix})$$

$$\Leftrightarrow$$

$$a[i+1] = \sigma(a[i] \cdot w[i])$$

```python
class Neurones:

    def __init__( self , tailles, fct_activation):

        ainit=[[random.random() for k in range(tailles[i])]for i in range(len(tailles))]
        self.a=np.array(ainit)                                          #a[i][j] valeur du neurone j de la couche i ,
                                                                        #ce n'est pas une matrice , on initialise aléatoirement
        self.tailles=tailles
        winit=np.empty(len(tailles),dtype=object)   #On initialise le tableau des poids , pour L couches , elle contient L matrices de poids entre la couche L+1 et L

        for l in range(len(tailles)-1):

            winit[l]=((np.random.rand(tailles[l],tailles[l+1]))-0.5)*normalisation_poids # np.random.rand(i,j) renvoie une matrice de nombres aléatoires entre 0 et 1 de taille i*j

        self.w=np.array(winit) #w[L][i,j] est le poids de la connexion entre le neurone i de la couche L et le neurone j de la couche L+1

        self.sig=fct_activation     #on appelle sig comme sigma la fonction d'activation du réseau
        self.memo={}    #memoisation pour ameliorer les performances

    def propagation(self,entree):
        if tuple(entree) not in self.memo: #si la valeur du calcul n'est pas connue on la calcule
            """Attention: l'entree sera une liste/un tableau ne pouvant pas etre utilise comme cle de dictionnaire
            c'est pourquoi il faut le transformer d'abord en tuple"""
            self.a[0]=copy.deepcopy(entree)
            for l in range(1,len(self.tailles)):
                #print(self.a[l-1],self.w[l])
                self.a[l]=self.sig(np.dot(self.a[l-1],self.w[l-1])) #éventuellement introduire une matrice de biais ici
            self.memo[tuple(entree)]=self.a[-1] #on ajoute la valeur calculee au dictionnaire pour une eventuelle utilisation future
            return self.a[-1]
        else:
            return self.memo[tuple(entree)]
```

D'où la classe Neurones ci dessus *(voir annexe neurones.py , classe Neurones)*

2) Les véhicules

Le véhicule est programmé comme un objet avec différents attributs:

- vehicule.vitesse : ***(flottant)*** norme de la vitesse du véhicule
- vehicule.vmax : ***(flottant)*** vitesse maximale du véhicule
- vehicule.position : ***(tuple)*** (x,y) de position du véhicule sur le circuit
- vehicule.distance : ***(flottant)*** distance parcourue par le véhicule
- vehicule.vivant: ***(booléen)*** si le véhicule est vivant ou a eu un accident
- vehicule.reseau: ***(type Réseau de neurones)*** il s'agit du réseau de neurones qui permet la prise de "décision" , on a attribué à ces véhicules un réseau [3;9;2] où l'on remplacera la première couche par une entrée (calculée selon une fonction ***detect_entree***) et la dernière couche donnera le tuple résultant **(angle,vitesse)**

```
class Vehicules:
    def __init__(self,position):
        self.position=position
        #vitesse maximale du vehicule , on multipliera cette valeur par la sortie du réseau
        self.vmax=45
        self.vitesse=0
        self.angle=0.0
        self.reseau=Neurones([angles_vision,9,2],sigmoide)#modifier le premier coefficient de la liste
                                                          #en raccord avec le nombre de sorties de detect_entree

        self.distance=0.0
        self.vivant=True
```

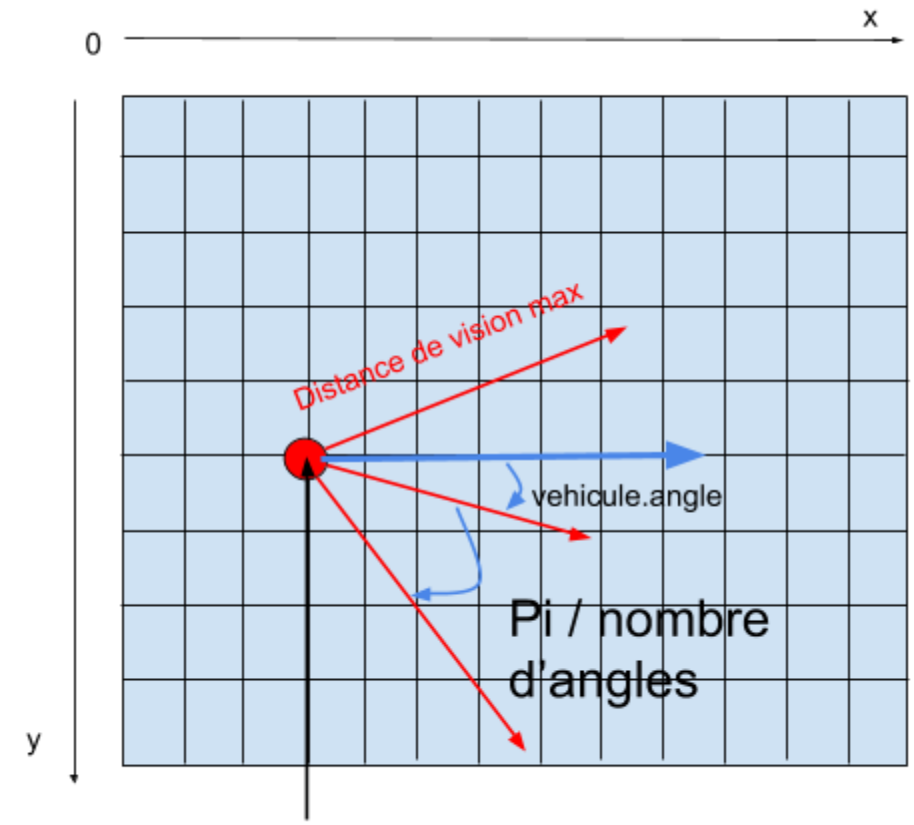
(voir annexe Neurones.py , ligne 160)

a) La détection des entrées: la "vision" du véhicule

Pour que notre réseau de neurones puisse calculer un angle et une vitesse adéquats , il faut lui fournir une entrée cohérente avec son déplacement , j'ai donc décidé de créer un système de détection d'obstacles : la voiture projette son "regard" jusqu'à une distance **dmax** et cela dans trois directions (voir schéma) :

- tout droit : c'est à dire à l'angle de la voiture **voiture.angle**
- à gauche : à l'angle **voiture.angle + pi/3**
- à droite : à l'angle **voiture.angle - pi/3**

Le véhicule regarde ensuite pour tout $k \in [1; dmax]$ si au point **(voiture.position, k , voiture.angle + angle de visée)** (en repérage cylindrique type (origine, rayon , angle)) il y a un obstacle. Cette vérification est rapide : notre circuit est assimilé à une matrice et la



présence d'un **"1"** dans cette matrice à une position donne la présence d'un obstacle à cette position.

Nous avons besoin de nombres entre 0 et 1 dans chaque direction : en cas d'obstacle dans une direction , on enregistre pour le **k** ayant donné une collision le rapport **k/dmax**.

Le tableau de vision résultant est donc de la forme

[k_gauche/dmax ; k_devant/dmax ; k_droite/dmax]

De cette manière nous dotons notre véhicule d'une "vision" simpliste , voici comment cet algorithme est codé dans la fonction ***detect_entree***:

```
def detect_entree(self,dmax,nbangles):
    distances=[i*0.0 for i in range (nbangles)] #tableau qui contiendra la distance à un obstacle POUR CHAQUE ANGLE
    angles=[i*np.pi/nbangles for i in range(-int((nbangles)/2),int((nbangles)/2))] #tableau des angles d'observation

    for i in range(len(angles)): #on fixe un angle d'observation , on va regarder dans cette direction
        x,y=self.position       #on copie la position actuelle du véhicule
        for k in range(dmax):   #on itere jusqu'a dmax
            if x>=0 and x<xmax and y>=0 and y<ymax and not circuit[int(x)][int(y)]==1: #conditions a laquelle on continue de regarder dans la directio
                                                                                       # en gros tant qu'il n'y a pas d'obstacle ou qu'on ne tombe pas sur un
                x+=np.cos(self.angle+angles[i])     # x devient x+ projection selon x dans la direction choisie
                y+=np.sin(self.angle+angles[i])     # y devient y+ projection selon y dans la direction choisie
                distances[i]=k/dmax                 # la distance pour l'angle d'observation choisi devient k/dmax pour avoir un quotient entre 0 et 1

    return np.array(distances) #on convertit en tableau numpy pour que la fonction de propagation puisse calculer rapidement dessus
```

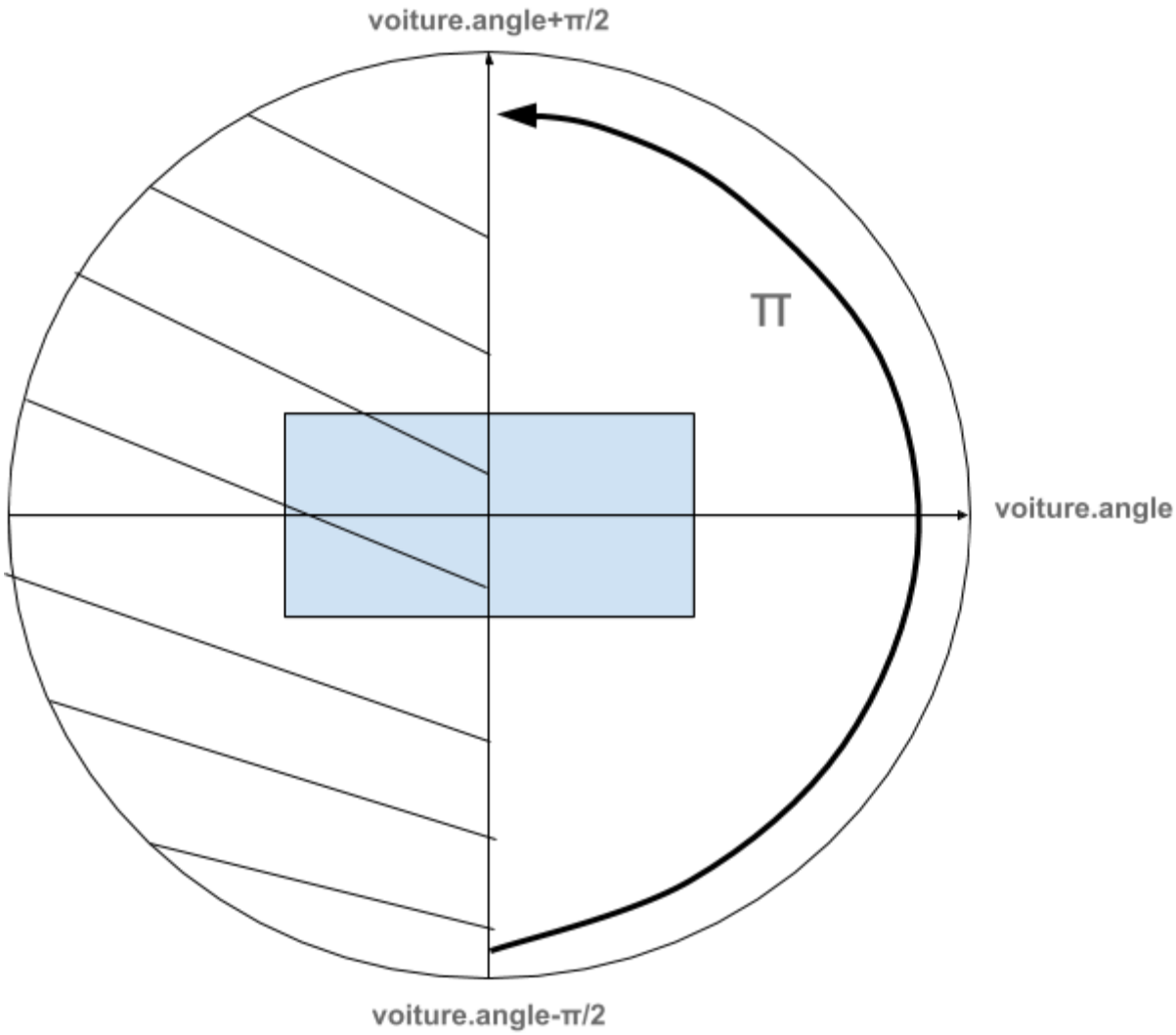
(voir annexe à partir de la ligne 170)

b) Le déplacement du vehicule : mise en pratique

Une fois que nous avons une entrée valide , nous pouvons laisser le réseau de neurones faire son travail et décider d'un déplacement pour le véhicule :

Les sorties du réseau de neurones étant bornées entre **0** et **1** , on veut avoir une vitesse et un angle utilisables, c'est pourquoi on multiplie l'un de ces nombres par **voiture.vmax** pour avoir une vitesse réelle .

L'autre sortie sera assimilée à un angle : j'ai pris le parti de ramener le nombre produit entre -1 et +1 au lieu de 0 et 1 puis le multiplier par pi: une voiture ne devrait pas faire d'angle à gauche/droite de plus de **π/2** , de plus cela empechera nos individus d'être bloqués à tourner en rond.



Pour effectuer un déplacement il faut:

1) Que le véhicule observe son entourage et obtient une entrée grace à ***detect_entree***
2) Calcule le résultat associé à cette entrée par le réseau de neurones grâce à ***voiture.reseau.propagation***
3) Transformer les résultats du réseau en angle et vitesse utilisables
4) Calculer dx,dy déplacement en position cartésienne selon x et y en fonction de l'angle et la vitesse de sortie en projetant selon l'axe **x** et l'axe **y**
5) Vérifier si la nouvelle position **(x+dx , y+dy)** induit une collision ou une sortie du circuit
    a) si c'est le cas : le véhicule n'est plus vivant ***voiture.vivant=False***
    b) si la voiture arrive sur la zone d'arrivée , sa distance augmente de **10000** (valeur arbitraire pour forcer l'importance ce point d'arrivée)
    c) sinon : la position est actualisée **(x,y) = (x+dx, y+dy)** ainsi que la distance parcourue (négative pour pénaliser les individus qui tourneraient en rond

Cet algorithme est programmé de la façon suivante, quelques conditions supplémentaires ont été introduites après les essais pour éviter les blocages du programme et un système de "récompense" a été mis en place pour récompenser les véhicules arrivant jusqu'à une zone d'arrivée (signalée par un "**2"** dans la matrice du circuit):

```
def deplacement(self):
    if self.vivant:
        #le véhicule "regarde" autour de lui : on veut lui donner des entrées
        entree=self.detect_entree(distance_vision,angles_vision)
        #print(entree)

        x,y=self.position

        #on calcule le résultat par le réseau de neurones
        resultat_reseau = self.reseau.propagation(entree)
        #print("res=",resultat_reseau)

        self.vitesse,self.angle=(resultat_reseau[0])*self.vmax, (resultat_reseau[1]-0.5)*2*(np.pi/2)

        dx,dy=int(self.vitesse*dt*np.cos(self.angle)) , int(self.vitesse*dt*np.sin(self.angle))
        #print("d=",dx,dy)

        if x<0 or x>=xmax or y<0 or y>=ymax:
            self.distance -=200 #punir les individus qui rentrent dans les murs
        if x<0 or x>=xmax or y<0 or y>=ymax or circuit[x][y] or (dx,dy)==(0,0) or self.distance< -3000:
            #dx,dy==0,0 est unecondition pour éviter les blocages,
            #on préferera que les individus soient constamment en mouvement
            if circuit[x][y]==2:
                self.distance+=10000
            else:
                self.distance-=4000
            self.vivant=False
        else:
            self.distance+=np.sqrt(dx**2+dy**2)
            self.position= x+dx,y+dy
            #xinit,yinit= position_initiale
            #self.distance = np.sqrt((xinit-(x+dx))**2+(yinit-(y+dy))**2)
```

(voir annexe , à partir de la ligne 183)

3) Générations et évolution

En partant de cette définition d'un véhicule , on effectue des **générations** successives : on génère un certain nombre (50 dans notre cas) d'individus ayant chacun un réseau de neurones aléatoire puis on les fait tous déplacer dans le circuit jusqu'à ce qu'il n'y en ait plus aucun en vie.

```
def generation(la_horde,nb_individus,tracer):
    nbvoit_vivantes=len(la_horde)
    distances_par_vehicule=[]

    tps=time.time()
    for k in range(len(la_horde)):
        vuatur=la_horde[k]
        positions=[vuatur.position]
        while nbvoit_vivantes>0 and not vuatur.mort():
            if vuatur.mort() :
                nbvoit_vivantes-=1
            elif (time.time()-tps)>3.0: #introduire une duree de vie limite, certains individus sont sinon capables de ne jamais mourir
                print('je suis mort patron')
                vuatur.vivant=False
            else:
                vuatur.deplacement()
                #print(vuatur.reseau.w)
            positions.append(vuatur.position)
            #print(vuatur.distance)

        distances_par_vehicule.append(vuatur.distance)
        x_val = [x[0] for x in positions]
        y_val = [x[1] for x in positions]
        if tracer or (time.time()-tps)>3.0:
            ax1.plot(x_val,y_val,marker='o')
            ax1.set_title("Deplacements")
            ax1.set_xlabel("x")
            ax1.set_ylabel("y")
    #(sorted(distances_par_vehicule)[-nb_meilleurs:])
    return la_horde,distances_par_vehicule,[x_val,y_val]
    #plt.show()
```

Une fois cela accompli , on trie les individus selon la distance parcourue : les **meilleurs individus** étant ceux qui ont parcouru le plus de distance.

Ces individus sont ensuite choisis selon un algorithme génétique défini par les constantes suivantes:

- le **taux de mutation** : compris entre 0 et 1 , il s'agit de la probabilité qu'un individu subisse une **mutation** c'est à dire une modification aléatoire d'un ou plusieurs de ses gènes , les gènes de nos voitures sont ici **les poids des connexions neuronales.**

- le **taux de rétention** : compris entre 0 et 1 , c'est la proportion des meilleurs individus sur le nombre d'individus total qui seront conservés comme **parents** pour la génération suivante

- le **taux de sauvetage** : compris entre 0 et 1, il s'agit de la probabilité qu'un individu qui ne soit pas forcément parmi les meilleurs soit retenu comme parent pour la génération suivante, cela permet de conserver une certaine diversité génétique

Les taux sélectionnés dans mon programme après divers essais sont :

- taux de sauvetage = 0.05

- taux de rétention = 0.6

- taux de mutation = 0.2

Ces valeurs ont été sélectionnés en partie selon le papier de *Ronald & Schoenauer*[1] et en partie d'après les valeurs généralement admises pour les algorithmes génétiques classiques.

L'algorithme génétique prend place entre deux générations:

1) On **trie** les individus en fonction de leur **distance parcourue**
2) On crée une **liste de parents** à laquelle on ajoute la proportion de meilleurs individus désirée selon le **taux de rétention**
3) On ajoute au hasard quelques individus selon le **taux de sauvetage** pour garder une diversité génétique qui nous permettra à termes de sortir de solutions locales non globales
4) On fait **muter** quelques parents au hasard selon le **taux de mutation** : la mutation s'effectue en ajoutant à des poids connexionnels choisis au hasard d'un parent un bruit aléatoire gaussien grâce à ***np.random.normal()*** et on **normalise** c'est à dire que l'on ramène ce poids dans des bornes définies pour éviter le calcul avec des valeurs numériques trop grandes . Pour ces bornes une valeur cohérente de **normalisation** semble être **8 ou 10.** On fait décroitre ce bruit géométriquement en fonction du nombre de générations par un facteur **0.999** pour pouvoir affiner au fil des générations : si les poids ont convergé vers une valeur précise , un bruit trop élevé pourrait en dévier de manière trop importante

```
def mutation(individu,nbmutat):
    for i in range(nbmutat): #on va modifier aléatoirement plusieurs poids du reseau neuronal de l'individu
        couche_random=random.randint(0,len(individu.reseau.w)-2)
        #print('len',len(individu.reseau.w)-1)
        #print('couche',couche_random)
        #print('poids',individu.reseau.w[couche_random])
        #print('type',type(individu.reseau.w[couche_random]))
        i_random=random.randint(0,len(individu.reseau.w[couche_random])-1)
        j_random=random.randint(0,len(individu.reseau.w[couche_random][i_random])-1)
        #rajout d'un nombre aleatoire decroissant a chaque gen pour affiner au fil des générations
        individu.reseau.w[couche_random][i_random,j_random]+= np.random.normal(scale=normalisation_poids/2)*.999**no_generation
        if individu.reseau.w[couche_random][i_random,j_random]>normalisation_poids:
            individu.reseau.w[couche_random][i_random,j_random]=normalisation_poids
        elif individu.reseau.w[couche_random][i_random,j_random]<-normalisation_poids:
            individu.reseau.w[couche_random][i_random,j_random]=-normalisation_poids
```

(voir annexe à partir de la ligne 79)

5) On génère autant d'individus que nécessaire par **reproduction**: d'après les travaux de *Ronald & Schoenauer*[1] , pour un algorithme neurogénétique il est intéressant d'avoir deux opérateurs de reproduction choisis au hasard :
    a) l'un génère un individu "enfant" puis attribue à quelques poids connexionnels au hasard une **combinaison barycentrique** des poids

connectionnels de chacun de ses parents : c'est la **reproduction barycentrique**

b) l'autre remplace aléatoirement quelques poids connectionnels de l'enfant par le même poids de l'individu père ou mère

```
 96  def reproduction(pere,mere,nb_modifs):
 97      enfant=Vehicules(position_initiale)
 98
 99      #on a deux opérateurs de reproduction choisis au hasard
100      choix=random.random()
101      #La reproduction barycentrique
102      if choix > .5 :
103          enfant.reseau.w = 0.5*(np.add(pere.reseau.w , mere.reseau.w ))
104      #La reproduction par copie : l'enfant reçoit aléatoirement des poids du pere ou de la mere
105      else:
106          for i in range( len(enfant.reseau.w)):
107              for j in range( len(enfant.reseau.w[i])):
108                  for k in range(len(enfant.reseau.w[i][j])):
109                      p=random.choice([pere.reseau.w[i][j][k],mere.reseau.w[i][j][k]])
110                      enfant.reseau.w[i][j][k]=p
111
112      return enfant
113
```

(voir annexe ligne 96)

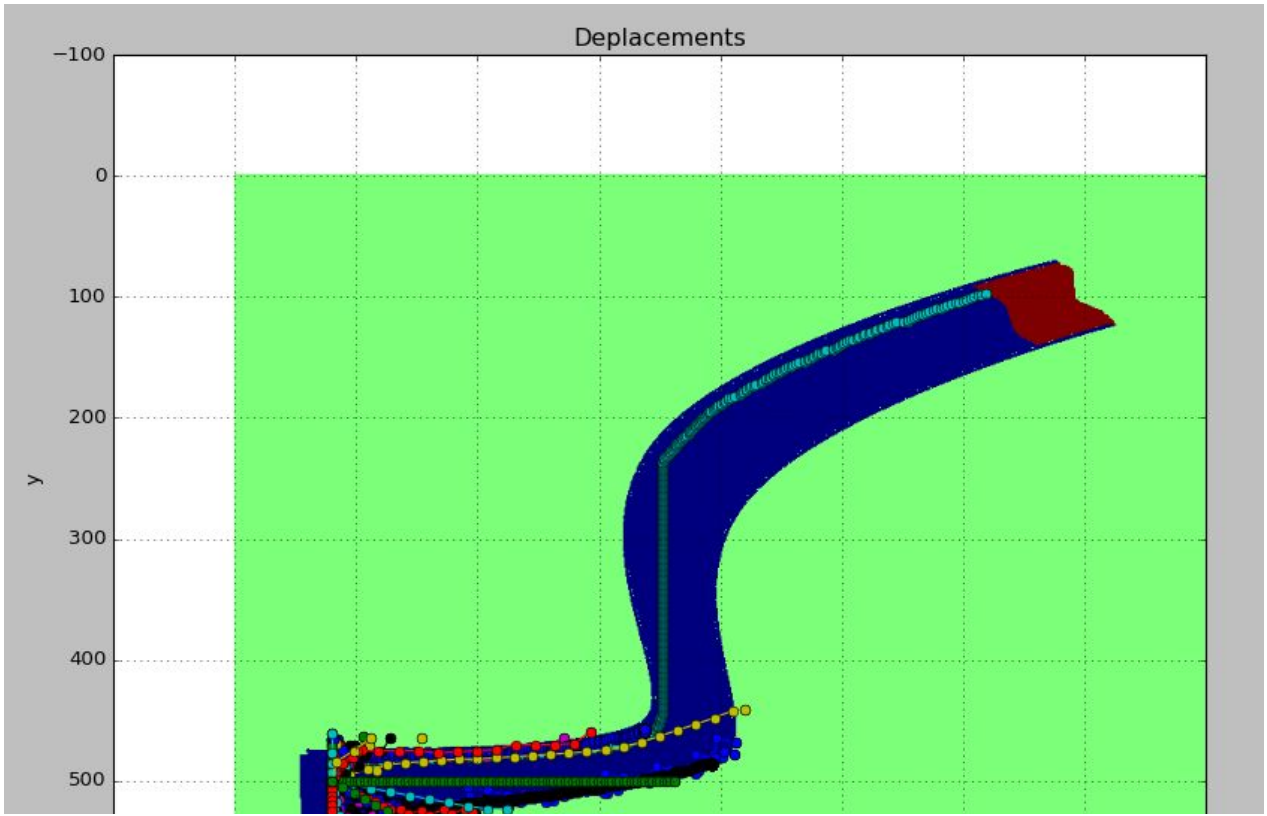
Le programme complet d'évolution d'une génération est alors le suivant :

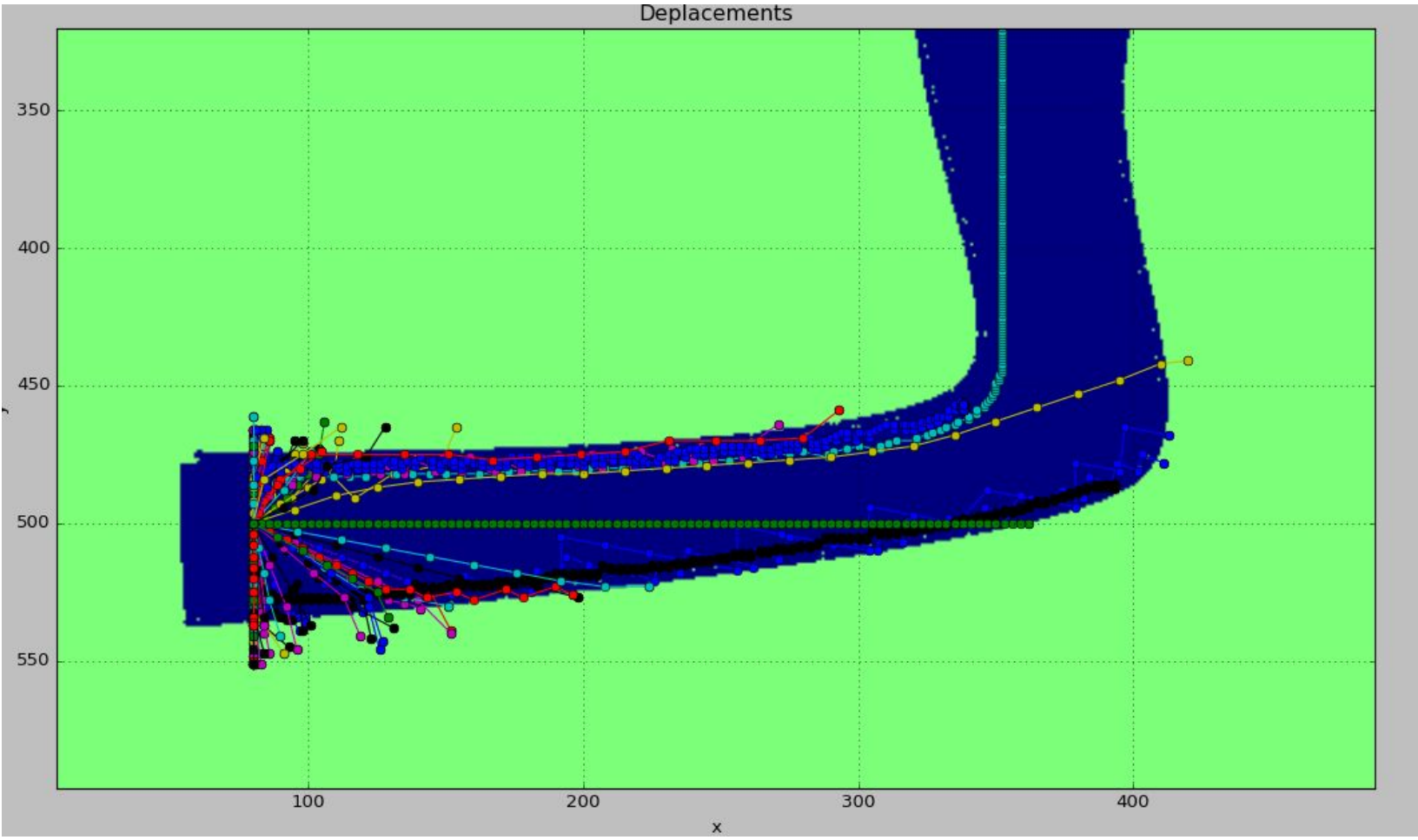
```
def evolution(individus,distances):

    distance_moyenne=sum(distances)/len(distances)
    variance=sum([d**2-distance_moyenne**2 for d in distances])/len(distances)
    print('moy=',distance_moyenne,'\n variance=',variance)

    indices_tries=np.argsort(distances)
    #argsort renvoie une liste contenant les indices des elements tries par ordre croissant sans modifier la liste
    individus_tries=[individus[k] for k in indices_tries] #les individus sont ici mis dans l'ordre croissant
    taux_mutation=0.2
    taux_meilleurs=0.6 #proportion des meilleurs individus conservés pour la génération suivante
    taux_sauvetage=0.05 #chances qu'un "mauvais" individu soit conservé

    #on prend les meilleurs individus
    nombre_meilleurs=int(taux_meilleurs*len(individus_tries))
    parents=individus_tries[nombre_meilleurs:]

    #on ajoute quelques individus moins bons au hasard pour la diversité
    for k in range(len(individus_tries[:nombre_meilleurs])):
        if random.random() < taux_sauvetage:
            parents.append(individus_tries[k])

    enfants=[]

    #on fait muter quelques parents au hasard
    for k in range(len(parents)):
        if random.random() < taux_mutation:
            mutation(parents[k],2)

    while len(enfants)<len(individus):
        enfants.append(reproduction(random.choice(parents),random.choice(parents),5))

    return enfants,distance_moyenne,variance
```

Puis on relance une nouvelle génération sur cette nouvelle population.

```
nbind=50
nbgen=100
individus,distances,plot= generation([Vehicules(position_initiale)for i in range(nbind)],nbind,5)
print(time.time()-debut_tps)
drap=False
liste_dist=[]
liste_var=[]
for k in range(nbgen):
    no_generation=k
    if k==nbgen-1:
        drap=True
    debut_tps=time.time()
    new_gen,dmoy,variance=evolution(individus,distances)
    individus,distances,plot=generation(new_gen,nbind,drap)
    liste_dist.append(dmoy)
    liste_var.append(variance)
    #liste_temps.append(time.time()-debut_tps)
    #if time.time()-debut_tps>3:
    #   break
    print('generation_numero:',no_generation,'temps=',time.time()-debut_tps)
#plt.plot(plot[0],plot[1])
ax1.grid(True)
#fig2.subplot(211)
ax2.plot(range(no_generation+1),liste_dist)
#fig2.subplot(212)
#ax2.plot(range(no_generation+1),liste_var)
plt.show()
```

à la fin du nombre de générations indiquées, toutes les positions des différents individus de la dernière génération sont visibles: voici le résultat d'une évolution de 70 individus sur 300 générations:





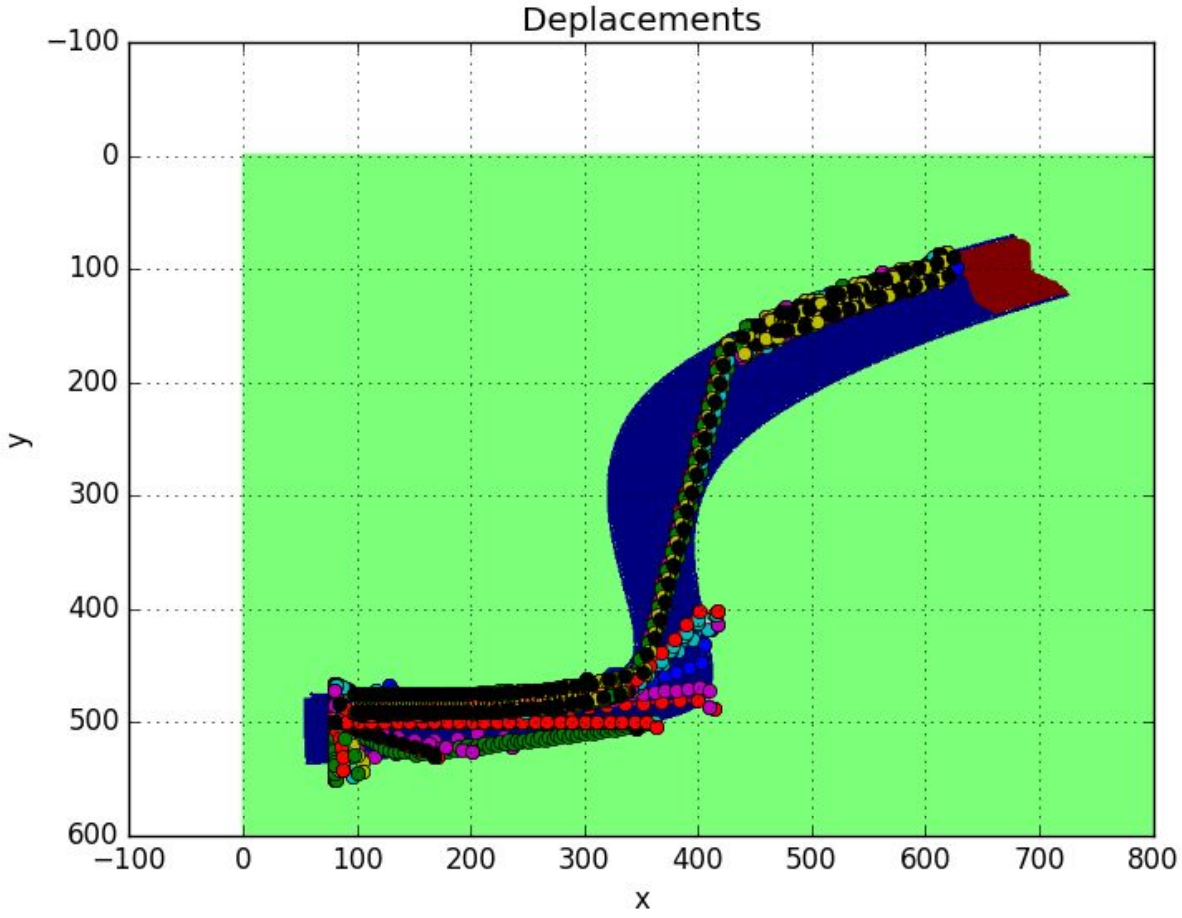
En vert : zone d'obstacle

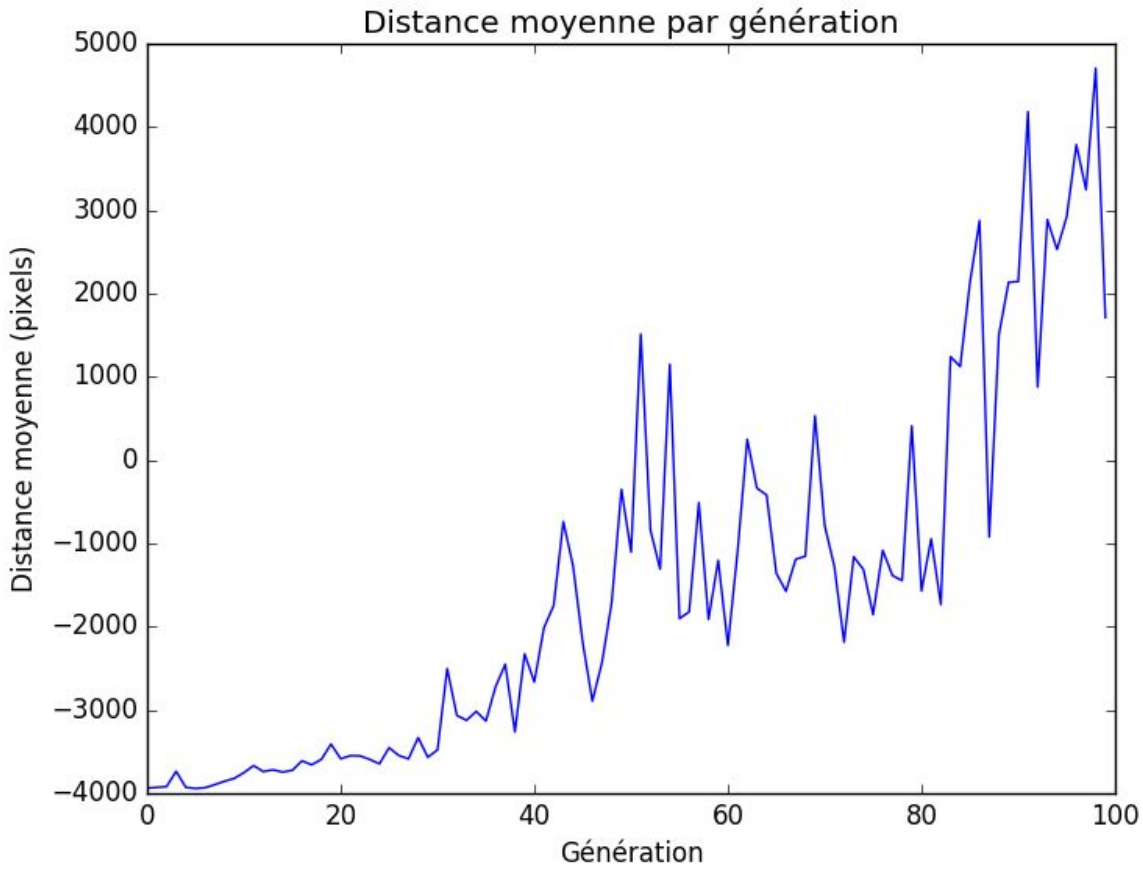
En bleu nuit : zone de déplacement autorisé

En rouge : zone d'arrivée

III- Les résultats :

On obtient bien les résultats attendus c'est à dire des individus capables d'évitement de murs et d'arriver jusqu'à la ligne d'arrivée : voici le résultat au bout de 100 générations de 70 individus. Le premier graphique montre les déplacements de chaque individu de la dernière génération , chaque ligne en couleur représentant un individu.


Deplacements
−100
0
100
200
300
400
500
600
y
−100
0
100
200
300
400
500
600
700
800
x



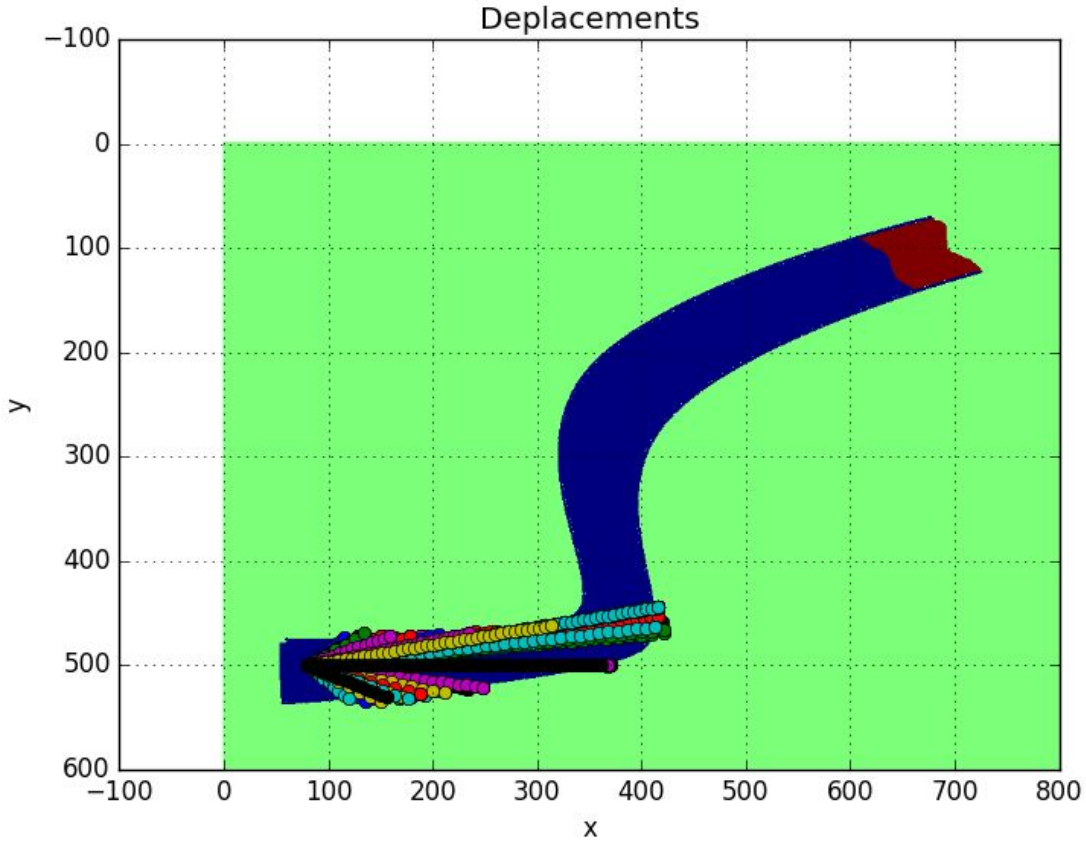
On observe bien une évolution générale des individus, toutefois très hachée du fait de l'intervention des mutations aléatoires. Ces résultats ont été obtenus avec une **distance de vision** de **20** cases et chose très importante : **un facteur de normalisation à 10.**

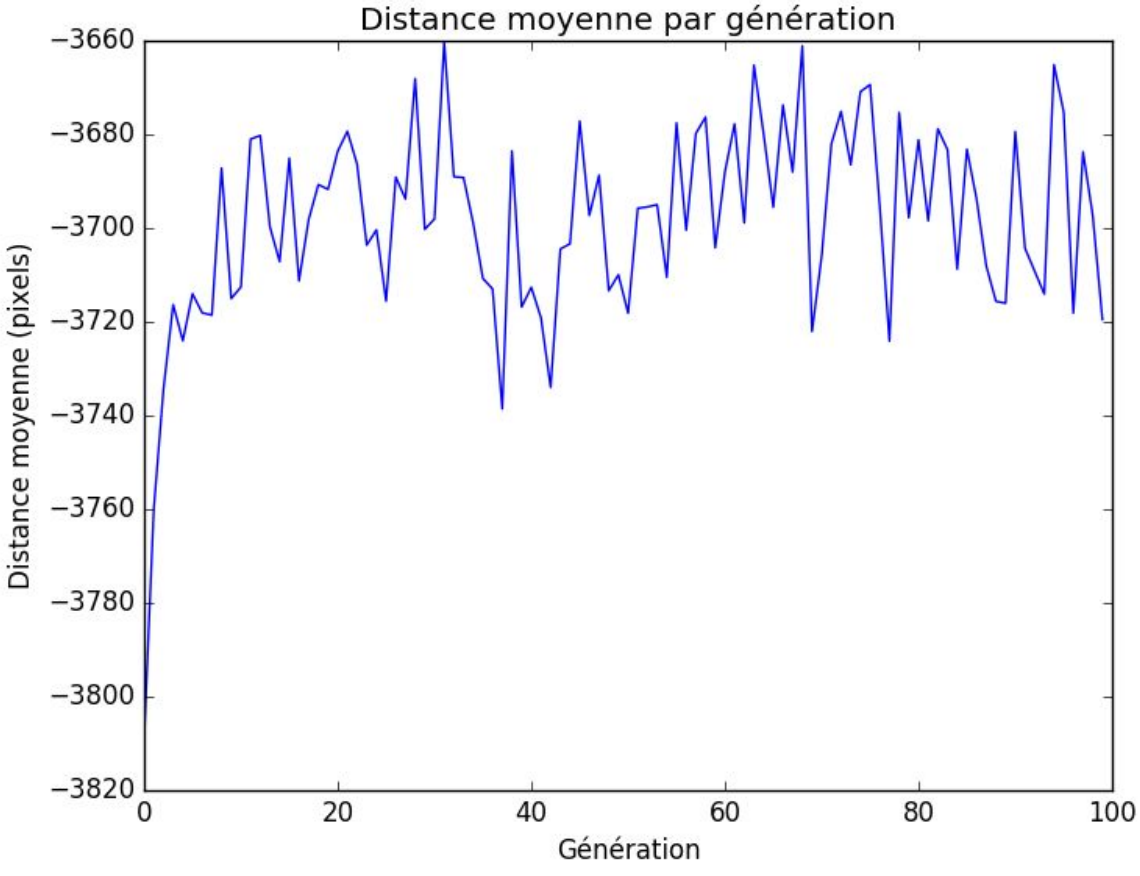
L'importance du facteur de normalisation

Si ce facteur peut paraître anodin et survolé dans la plupart des aspects de la neuro évolution , dans le cas d'un contrôle de véhicule le papier de *Ronald&Schoenauer*[1] insiste sur l'importance de ce facteur, qui non seulement peut permettre de trouver une solution plus rapide , mais de plus dans mon cas pouvait totalement empêcher le franchissement de certains virages.

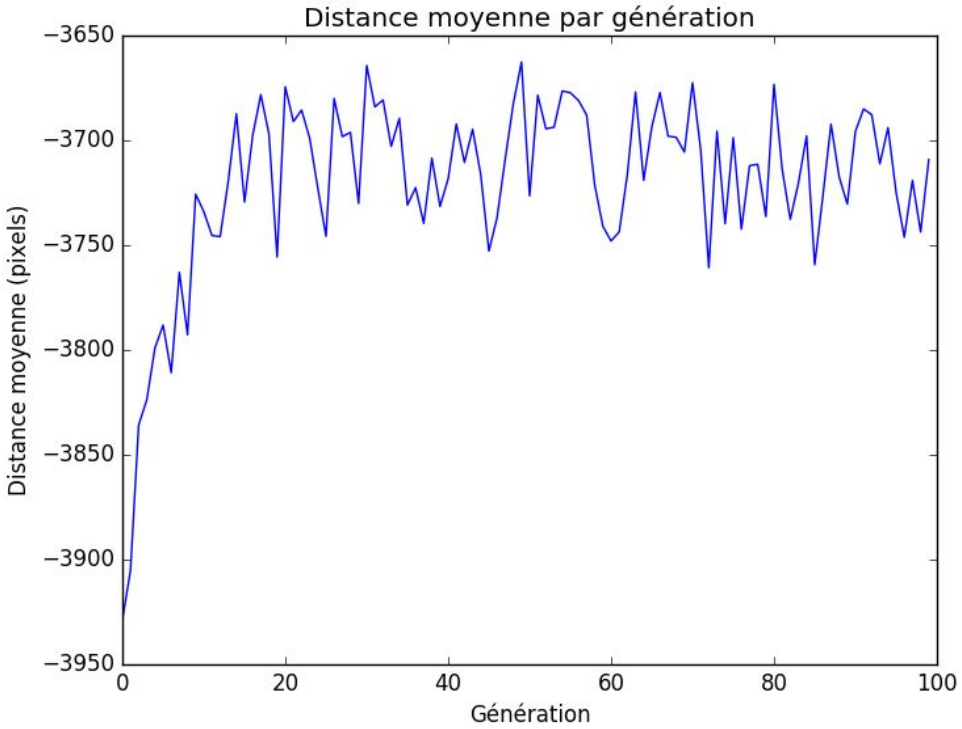
Voici une rapide comparaison des résultats pour des facteurs de normalisation de 1, 5 , 7

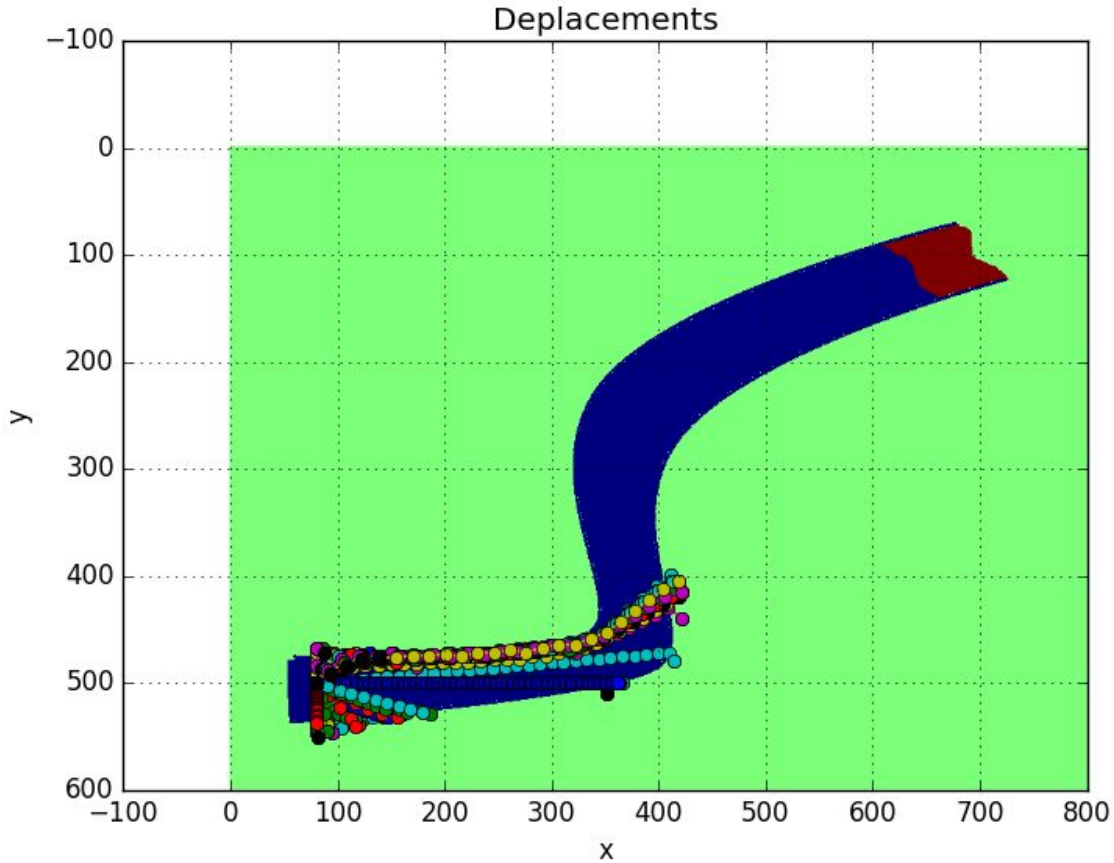
Facteur de normalisation = 1:


Deplacements
−100
0
100
200
300
400
500
600
y
−100
0
100
200
300
400
500
600
700
800
x

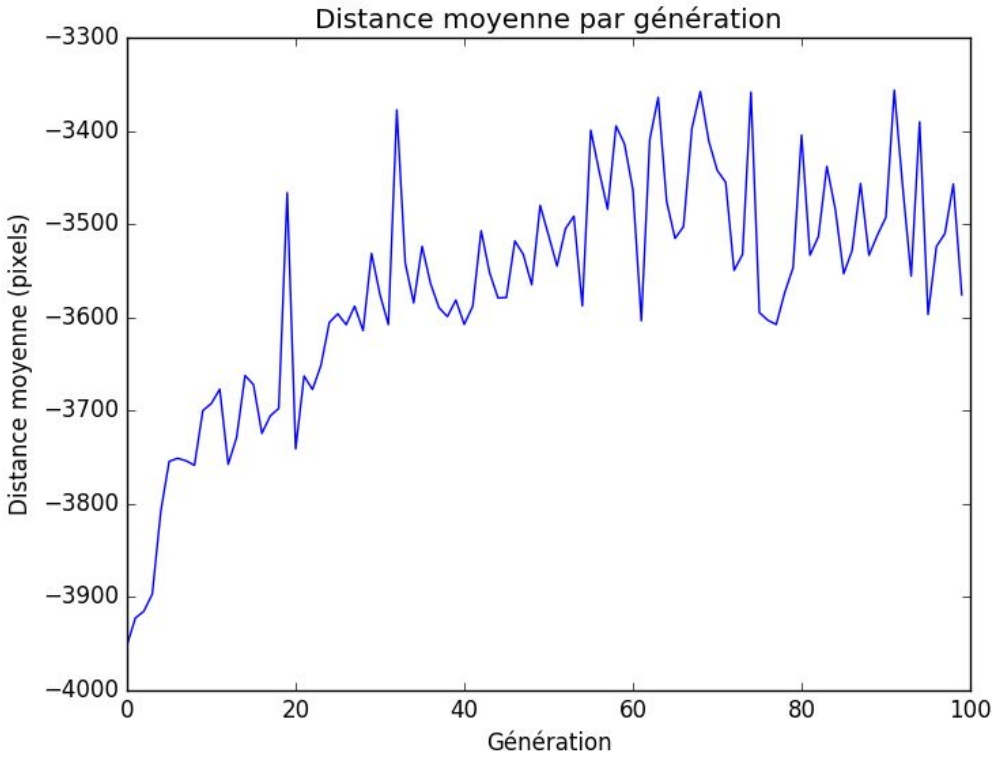

Distance moyenne par génération
−3660
−3680
−3700
−3720
−3740
−3760
−3780
−3800
−3820
Distance moyenne (pixels)
0
20
40
60
80
100
Génération

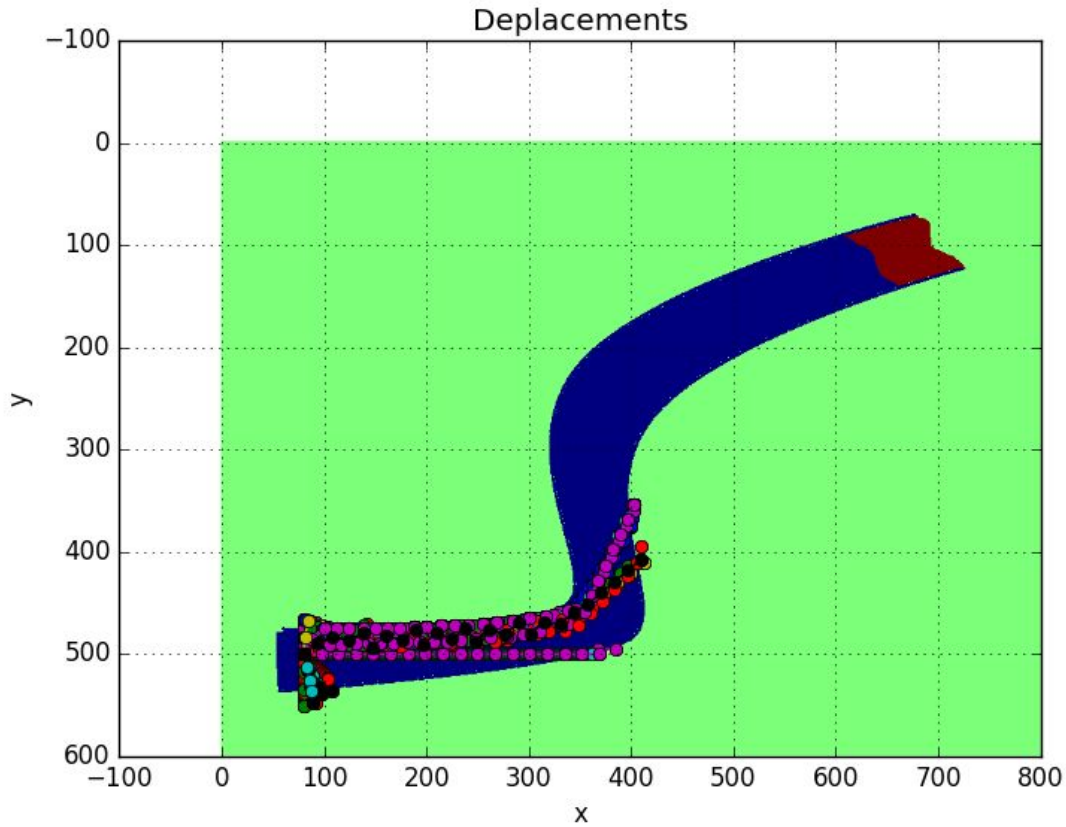
facteur de normalisation = 5 :





facteur de normalisation = 7



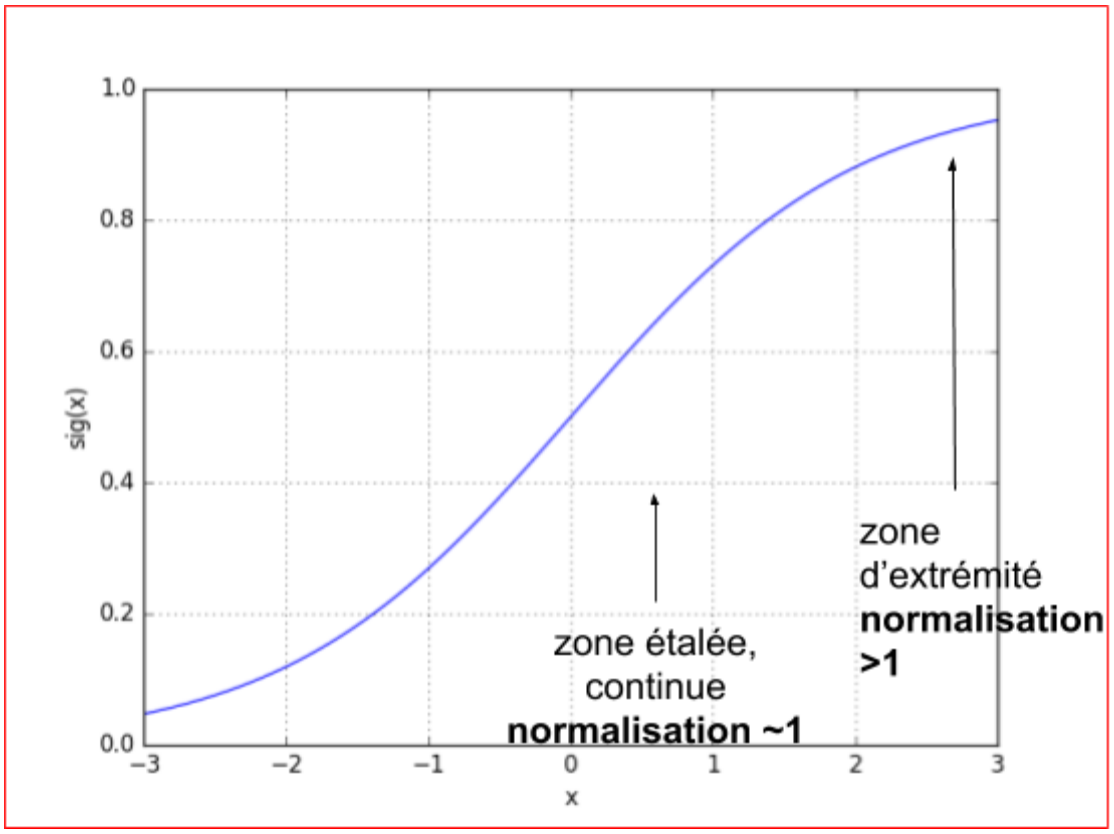


J'ai pu constater que plus le facteur de normalisation était grand , et donc plus les poids étaient grands, plus les angles que le véhicule est capable de prendre sont forts.

Une interprétation est que puisque :

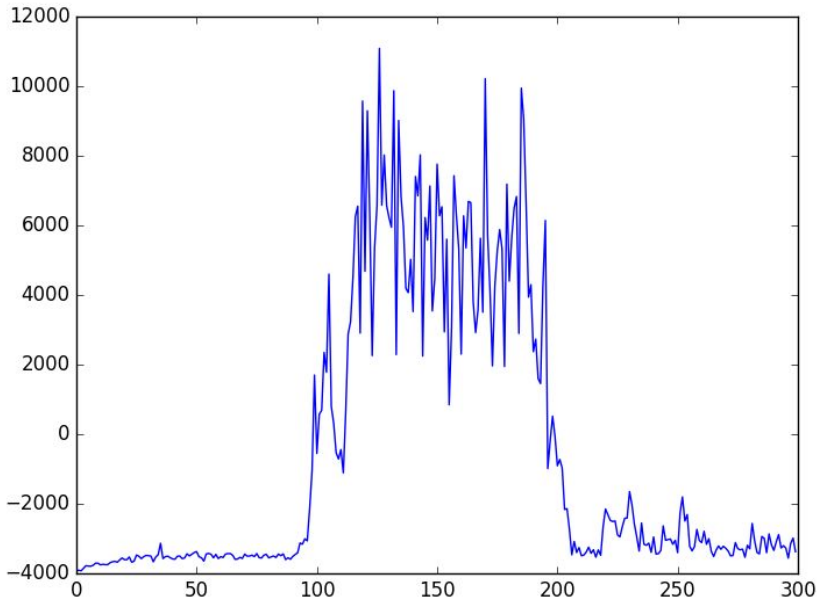
$$a[i+1][j] = \sigma(\sum_{k=1}^{n} w[i][k][j] * a[i][k])$$

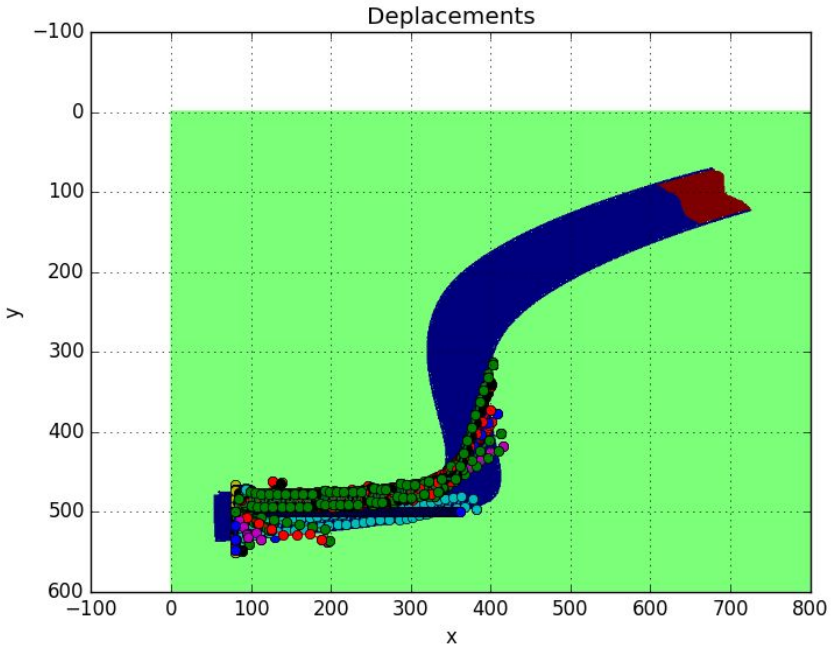
On observe alors que plus les poids seront grands dans la somme, plus la sortie de la sigmoïde se trouvera aux extrémités et plus le réseau de neurones pourra donner des réponses non linéaires de l'entrée



Comportements étranges : mutations trop fortes

Au fil de mes essais il m'est arrivé de toucher aux facteurs de mutation , notamment au bruit aléatoire qui est ajouté aux poids , il est important que ce bruit soit décroissant du nombre de générations sinon une mutation trop forte peut faire sortir toute une génération de la solution précédemment trouvée : en voici un exemple





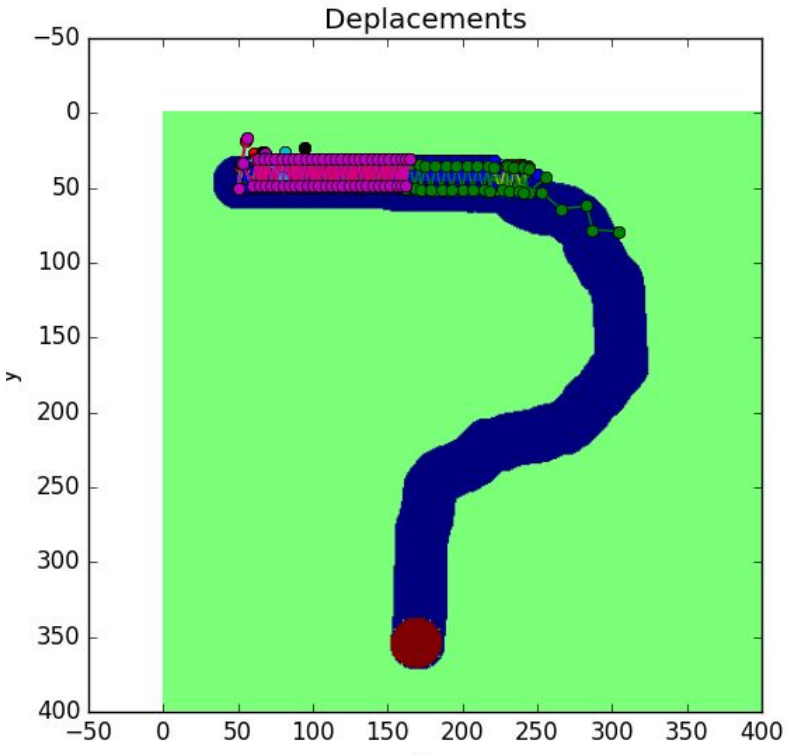
On peut très clairement voir que la capacité à tourner efficacement pour éviter les murs a été perdue aux alentours de la génération 200 du fait de mutations trop fortes , les enfants étant basés sur les parents de la génération précédente, cette capacité disparaît progressivement de toute la population. La dernière génération est ainsi incapable d'arriver au bout du circuit.

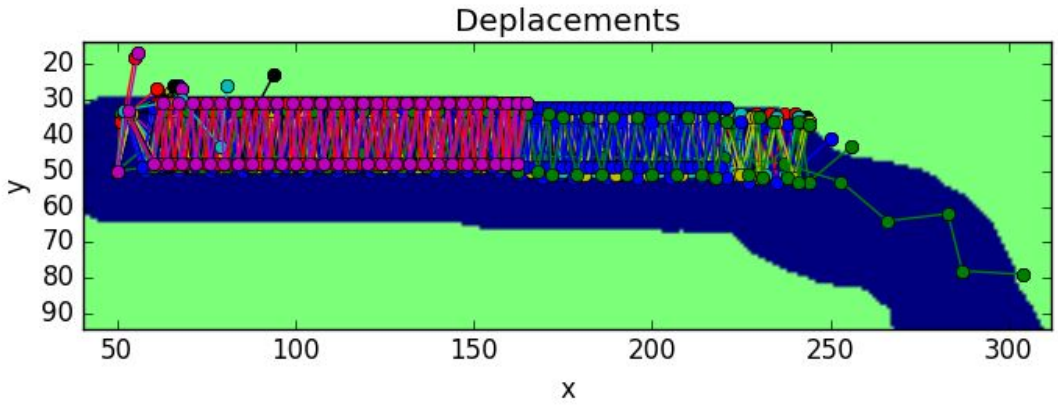
Conservation des meilleurs éléments : incapacité à l'adaptation et apprentissage biaisé , un problème

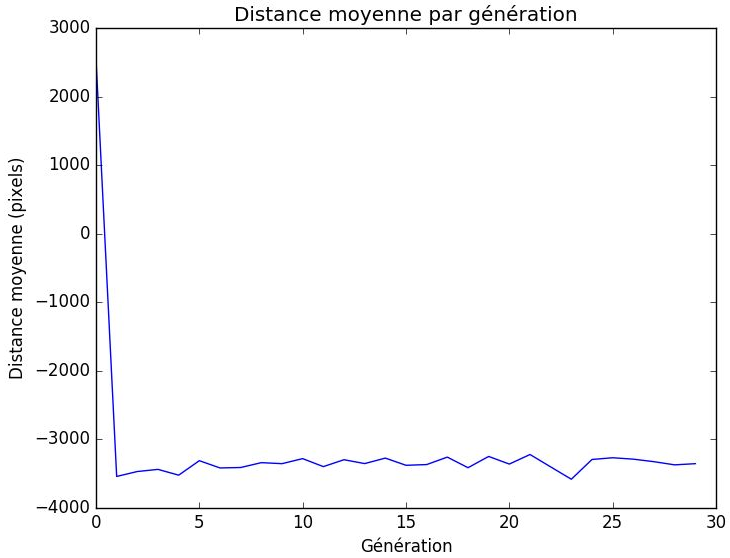
Une fois qu'une génération était à mon sens assez douée , je l'ai conservée et enregistrée dans des fichiers binaires grâce à la librairie **Pickle** et à mes fonctions **sauvegarde_generation** et **lecture_generation** que voici

```
316
317 def sauvegarde_generation(chemin,gen):
318     with open(chemin,'wb') as fichier :
319         pickle.dump(gen,fichier)
320         fichier.close()
321
322 def lecture_generation(chemin):
323     with open(chemin,'rb') as fichier:
324         generation_enreg=pickle.load(fichier)
325         return generation_enreg
326
```

Puis j'ai relancé le processus d'évolution à partir de ces individus **déjà entraînés** sur un autre circuit indépendant du premier :







Un problème apparaît alors: il semblerait que les individus soient incapables de s’adapter à un nouveau milieu de manière simple.