TP III : IA Robotique
Topologies des réseaux de neurones artificiels

Enseignant : S. Rodriguez
Date de fin de séance : le 9 mai 2022 (nombre total de séances consacrés à cette partie: 3)
Rédigé par Jeong Hyun AHN (APP5 – ESR)

Table des matières

Introduction

Les réseaux de neurones composés d'une simple couche représentent la plus simple topologie. Afin d'augmenter la complexité des algorithmes, il est donc nécessaire d'augmenter la topologie du réseau. Ainsi, les techniques d'apprentissage profond se déploient en présence de milliers ou des millions de neurones.

1. Les topologies multicouches

La nécessité d'une topologie multicouche émerge par le fait qu'une seule couche de neurones peut seulement établir une relation monotonique entre les entrées et les sorties. Autrement dit, la fonction représentée par une seule couche est une courbe qui « monte » constamment ou « descend » constamment dans l'intervalle des entrées.

Par exemple, nous pouvons associer les entrées d'un capteur à un réseau monocouche afin d'augmenter ou décroitre la vitesse d'un moteur quand le signal du capteur augmente. Néanmoins, il n'est pas possible d'assurer deux tendances différentes par ce type de réseau.

1.1. Modèle d'une topologie multicouche

Pour comprendre le mode de fonctionnement d'un réseau de neurones multicouche, nous le confronterons à un réseau monocouche. Pour ce faire, nous allons considérer un vecteur d'entrées avec une gamme définie par l'intervalle [−2.0, 2.0] et un vecteur de poids défini entre [−1.0, 1.0]. La fonction d'activation dans cet exercice sera une saturation limitant les valeurs en sortie des neurones entre [−1.0, 1.0].

1.2. Implantation du modèle d'une seule entrée

Nous allons implanter le modèle illustré par la Figure 1 avec w1 = -0.5.

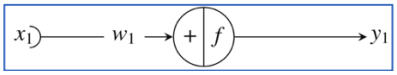
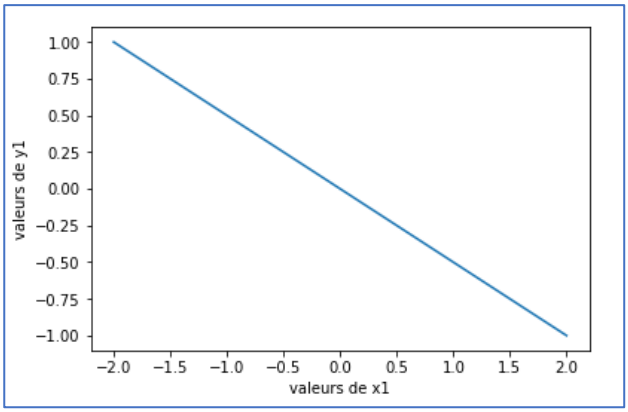


FIG. 1 : Modèle du réseau monocouche

Voici le code ainsi que le graphe généré par ce code :

```python
import numpy as np
import matplotlib.pyplot as plt

def action(s):
    if(s<=-1.0):
        return -1.0
    elif(s>=1.0):
        return 1.0
    else:
        return s


def perceptron(x1,x2,w0,w1,w2):
    s=w0 + w1*x1 + w2*x2
    y = action(s)
    format_x1 = "{:.1f}".format(x1)
    format_y = "{:.1f}".format(y)
    print("x1 = ",format_x1,"y1 = ",format_y)
    return y

x1 = np.arange(-2.0,2.0,0.2)
w1 = -0.5
y = [0]*x1.size # create same size as x1
for i in range(0,x1.size):
    y[i] = perceptron(x1[i],0,0,w1,0)


plt.plot(x1,y)
plt.xlabel('valeurs de x1')
plt.ylabel('valeurs de y1')
plt.show()
```



Nous avons un vecteur d'entrée qui incrémente de 0.2 comme suit : x1 = [−2.0,−1.8,...,0.0,...,1.8,2.0].

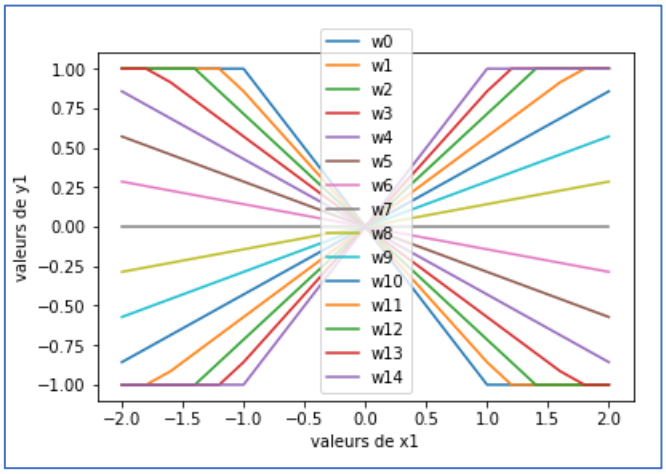
Nous pouvons observer une relation monotonique de ce réseau monocouche qui a une sortie y1 qui « descend » constamment de +1 à -1.

1.3. Étude de l'effet de poids

Nous allons maintenant faire varier le poids w1 de ce neurone monocouche pour déterminer la relation entre les entrées et les sorties selon la valeur de poids.

Voici le code et le graphe généré à partir de ce code :

```python
import numpy as np
import matplotlib.pyplot as plt

def action(s):
    if(s<=-1.0):
        return -1.0
    elif(s>=1.0):
        return 1.0
    else:
        return s

def perceptron(x1,x2,w0,w1,w2):
    s=w0 + w1*x1 + w2*x2
    y = action(s)
#     format_x1 = "{:.1f}".format(x1)
#     format_y = "{:.1f}".format(y)
    # print("x1 = ",format_x1,"y1 = ",format_y)
    return y

x1 = np.arange(-2.0,2.1,0.2)
w1 = np.linspace(-1.0,1.0,15)
y = [0]*x1.size # create same size as x1

for j in range(0,w1.size):
    w = w1[j]
    for i in range(0,x1.size):
        y[i] = perceptron(x1[i],0,0,w,0)
    plt.plot(x1,y,label="w"+str(j))

plt.xlabel('valeurs de x1')
plt.ylabel('valeurs de y1')
plt.legend()
plt.show()
```



Nous pouvons voir que w0 (en bleu) représente poids de -1 sur le graphe et w14 (en violet) poids de +1. Nous avons 15 valeurs de poids différents qui varient de -1 à +1 sur ce graphe.

Nous pouvons remarquer que la sortie varie de -1 à +1 dans tous les cas. Ce qui est un comportement attendu vu que notre fonction d'activation est une fonction de saturation dans l'intervalle [-1, +1].

Le poids de 0 (poids 7 sur le graphe) ne fait pas varier la sortie peu importe l'entrée. Ce qui est assez évident vu que nous avons une seule entrée avec un poids de 0. *(y1 = x1 * w1 = x1 * 0)*

Quand le poids est inférieur à 0, la sortie est décroissante de manière linéaire avec la pente qui devient moins importante lors que le poids s'approche de 0. Et inversement pour les poids supérieurs à 0.

1.4. Modèle du réseau multicouche

Nous allons maintenant implanter le modèle du réseau à deux couches comme illustré par la Figure 2 en imposant les poids du réseau comme suit : w_{11} =1, w_{12} =0.5, w_{21} =1, w_{22} =−1.

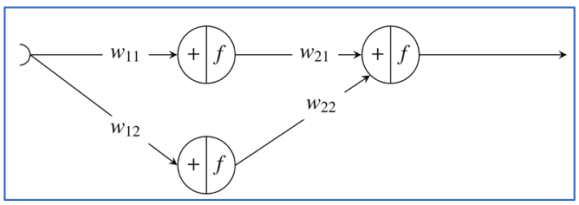
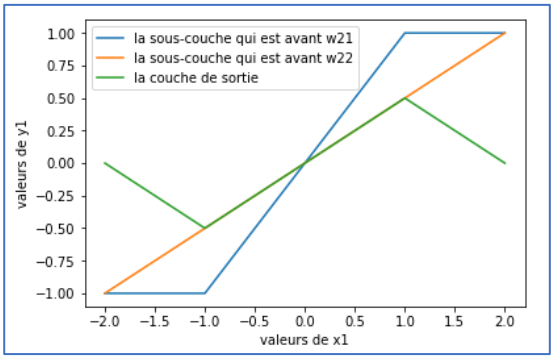


FIG. 2 : Modèle du réseau multicouche

```python
import numpy as np
import matplotlib.pyplot as plt

def action(s):
    if(s<=-1.0):
        return -1.0
    elif(s>=1.0):
        return 1.0
    else:
        return s

def perceptron(x1,x2,w0,w1,w2):
    s=w0 + w1*x1 + w2*x2
    y = action(s)
    return y

x1 = np.arange(-2.0,2.1,0.2)

y1 = [0]*x1.size # create same size as x1
y2 = [0]*x1.size
y3 = [0]*x1.size

for i in range(0,x1.size):
    y1[i] = perceptron(x1[i],0,0,1,0)
    y2[i] = perceptron(x1[i],0,0,0.5,0)
plt.plot(x1,y1,label="la sous-couche qui est avant w21")
plt.plot(x1,y2,label="la sous-couche qui est avant w22")

for i in range(0,x1.size):
    y3[i] = perceptron(y1[i],y2[i],0,1,-1)
plt.plot(x1,y3,label="la couche de sortie")

plt.xlabel('valeurs de x1')
plt.ylabel('valeurs de y1')
plt.legend()
plt.show()
```



Nous avons tracé les sorties des neurones de la couche cachée et celles de la couche de sortie par rapport aux valeurs en entrée x1.

Ce graphe nous permet de faire une sorte de confrontation avec le modèle monocouche représenté dans la Figure 1 de la partie précédente.

Nous pouvons voir ici une relation dite non-monotone.

Malgré les deux entrées qui ont une forme « croissante » venant des deux couches cachées, nous avons une sortie de dernier neurone qui « décroit » au début, et qui « croît » pendant un moment, puis qui « décroit » à nouveau vers la fin.

Autrement dit, nous avons vu que les réseaux multicouches permettent d'adresser des tâches plus complexes par rapport aux réseaux monocouches.

1.5. Application du modèle multicouche sur le robot

Nous allons maintenant concevoir un réseau capable d'effectuer le comportement décrit ci-dessous en utilisant les mesures issues des deux capteurs situés devant le robot :

> Si un objet est détecté par l'un de deux capteurs, le robot effectue un virage afin d'éviter l'obstacle.
> Si l'objet est détecté par les deux capteurs, le robot devra reculer.

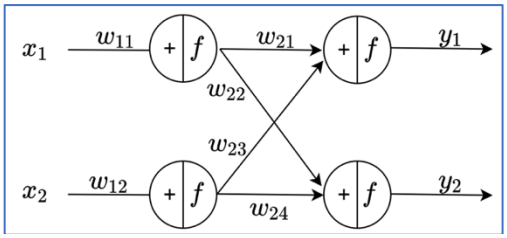


FIG. 3 : Modèle du réseau multicouche utilisé pour l'évitement d'obstacles du robot

Nous avons utilisé les poids suivants pour la validation de ce modèle et pour l'application sur le robot :

w11 = 1 ; w12 = 1 ; w21 = 0.3 ; w23 = -0.5 ; w22 = -0.3 ; w24 = 0.5

Voici le code appliqué sur le robot :

```python
def action(s):
    if(s<=-1.0):
        return -1.0
    elif(s>=1.0):
        return 1.0
    else:
        return s

def perceptron(x1,x2,w0,w1,w2):
    s=w0 + w1*x1 + w2*x2
    y = action(s)
    return y

## Thymio callback
def obs(node_id):
        global done

        if not(done):
            cap0 = th[node_id]["prox.horizontal"][0]
            cap4 = th[node_id]["prox.horizontal"][4]

            x1 = cap0*(1/4500) # conversion de [0,4500] à [0,500]
            x2 = cap4*(1/4500)

            y1=perceptron(x1,0,0,1,0) # 1e étape
            y2=perceptron(0,x2,0,0,1)
            y3=perceptron(y1,y2,0,0.3,-0.5) # moteur gauche
            y4=perceptron(y1,y2,0,-0.3,0.5) # moteur droite
            # ajouter votre code ici
            vit_g = int(y3*500) # conversion [-1,1] à [-500,500]
            vit_d = int(y4*500)

            th[node_id]["motor.left.target"] = vit_g
            th[node_id]["motor.right.target"] = vit_d
```

Dans ce code,
y1 et y3 représentent les deux sorties qui se trouvent sur la partie haute du modèle de Figure 3.
y2 et y4 représentent celles qui se trouvent sur la partie basse.

Nous utilisons deux capteurs ici, celui de gauche (capteur 0) et celui de droite (capteur 4). Le robot tourne à gauche sur lui-même si un obstacle se trouve à droite. Le robot tourne à droite sur lui-même si un obstacle se trouve à gauche. Il recule quand ces deux capteurs voient des obstacles en même temps.

Vous trouverez un vidéo démonstratif du robot Thymio II programmé avec le code en haut qui montre ce comportement dans le fichier qui va contenir ce compte-rendu.

2. La mémoire : les réseaux récurrents

Les modèles et les topologies étudiées se caractérisent par la propagation d'information au travers du réseau. Les sorties du réseau sont donc indépendantes du temps. Afin d'introduire une dépendance temporelle, il est nécessaire d'introduire la définition des « connexions récurrentes ». Ainsi, les sorties d'un neurone peuvent constituer une entrée du réseau ou de lui-même. Les connexions récurrentes peuvent être utilisées pour l'implantation d'une mémoire au sein du réseau.

Un cas d'usage dans le cadre de la navigation robotique est celui de l'évitement d'obstacles. En effet, les réseaux abordés en début de ce cours d'IA robotique, effectuent leur activation en présence des signaux en entrée confirmant la présence d'un obstacle. Ces actions sont arrêtées lors que l'obstacle n'est plus détecté. Or, cette condition n'assure pas l'évitement complet de l'obstacle. Il est nécessaire de conserver dans le temps les actions évasives.

2.1. Modèle de réseau récurrent

Nous allons mettre en œuvre ce type de réseaux et nous allons les analyser.

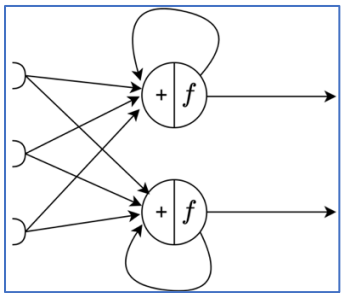


FIG. 4 : Modèle de réseau récurrent

Nous allons d'abord implanter le réseau illustré dans la Figure 4. La fonction d'activation est toujours une fonction de saturation dans l'intervalle [0, 1]. Le vecteur d'entrées est défini entre [0, 1] et les poids du réseau sont, pour la plupart, comprissent également entre [0, 1].

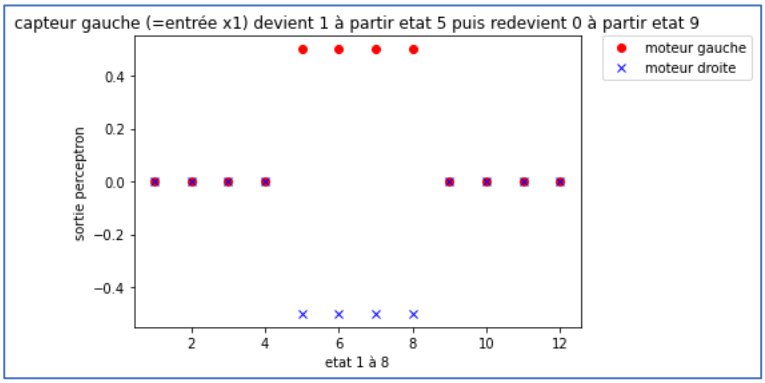
Voici le code utilisé pour la validation de ce modèle récurrent :

```
11  def action(s): # saturation dans [0,1]
12      if(s<=-1.0):
13          return -1.0
14      elif(s>=1.0):
15          return 1.0
16      else:
17          return s
18
19  def perceptron(x1,x2,x3,xrec,wrec,w1,w2,w3):
20      s=w1*x1 + w2*x2 + w3*x3 + wrec*xrec
21      y = action(s)
22      return y
23
24  x = [1,2,3,4,5,6,7,8] # etat 1 à 8 pour représentation graphique
25  # x1 = [0,0,0,0,1,1,1,1] # cap G
26  # x2 = [0,0,1,1,0,0,1,1] # cap centre
27  # x3 = [0,1,0,1,0,1,0,1] # cap D
28
29  x1 = [0,0,0,0,1,1,1,1] # cap G
30  x2 = [0,0,0,0,0,0,0,0] # cap centre
31  x3 = [0,0,0,0,0,0,0,0] # cap D
32
33  wrec = 1.3 # pour recurrence = memoire
34
35  wpos = 0.5 # pour acceler quand le robot tourne
36  wneg = -0.5 # pour tourner sur le robot même
37  wback = -0.8 # pour reculer
38
39  y1 = [0.0]*8
40  y2 = [0.0]*8
41
42  y1[0] = perceptron(x1[0],x2[0],x3[0],y1[0],wrec,wpos,wback,wneg)
43  y2[0] = perceptron(x1[0],x2[0],x3[0],y1[0],wrec,wneg,wback,wpos)
44
45  for i in range(1,8): # 1 to 7
46      y1[i] = perceptron(x1[i],x2[i],x3[i],y1[i-1],wrec,wpos,wback,wneg) #
47      y2[i] = perceptron(x1[i],x2[i],x3[i],y1[i-1],wrec,wneg,wback,wpos) #
```

Étudions l'influence des poids de connexions récurrentes sur les sorties du réseau. Nous avons trois cas :

cas 1 - poids de récurrence de 0 ; cas 2 - poids de récurrence de 0.5 ; cas 3 - poids de récurrence de 1.5

Cas 1 : poids de récurrence de 0

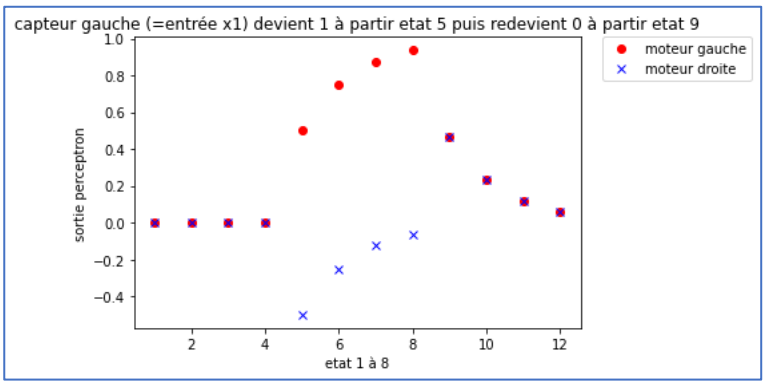


Il y a un typo sur les graphes qui seront présentés ici : en abscisse nous avons 12 états et pas 8. Nous pouvons considérer que c'est une sorte de temps. Autrement dit, nous pouvons considérer que cette simulation dure 12 s, et entre 5 et 8 secondes, le capteur de gauche observe un obstacle qui est très proche de lui. Il y a une conversion avant l'entrée dans ce réseau de neurones. Donc le fait qu'il y a un obstacle très proche de lui est traduit de manière suivante dans ce modèle :

capteur gauche (ou entrée x1) qui devient 1 à partir de l'état 5 puis redevient 0 à partir de l'état 9

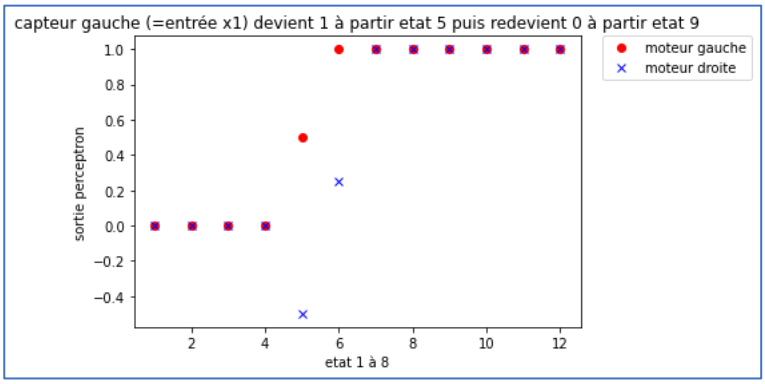
Nous pouvons remarquer qu'il n'y a pas de retard entre le moment de l'observation d'obstacle et le déclenchement d'évitement d'obstacle dans le cas où il n'y a pas de récurrence. Nous avons une réaction immédiate.

Cas 2 : poids de récurrence de 0.5



Cependant, dans le cas où il y a de récurrence, il y a un petit retard ou un décalage avant d'atteindre la vitesse des moteurs souhaitée pour l'évitement d'obstacles. Puis, même après la disparition d'obstacle (i.e. à partir de l'état 9), il y a un retard avant d'arrêter complètement les moteurs.

Cas 3 : poids de récurrence de 1.5



Finalement, dans le cas où nous avons un poids de récurrence supérieur à 1 (ici, valeur de 1.5), ce modèle récurrent garde en mémoire le fait qu'il y a un obstacle et ceci avec un retard comme dans le cas précédent. Il n'arrive pas à oublier le fait qu'il y avait un obstacle.

2.2. Application du modèle de réseau récurrent sur le robot

Nous allons maintenant implanter ce modèle sur le robot.

Nous avons rajouté au modèle illustré dans la Figure 5 (modèle de réseau pour l'évitement d'obstacles) en rajoutant deux connexions récurrentes pour les neurones de sortie.

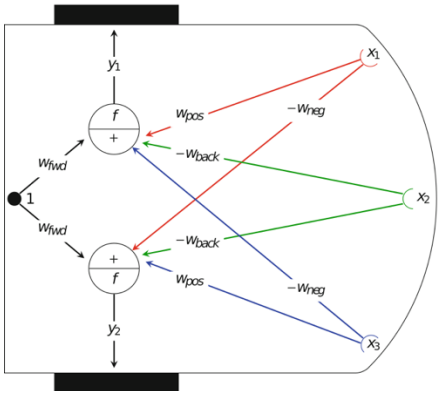


FIG. 5 : Modèle de réseau récurrent pour l'évitement d'obstacles

Nous avons un vidéo et un tracé sur papier comme démonstration. Le vidéo se trouvera dans le même fichier que celui qui va contenir ce compte-rendu.

Nous avons déclaré 2 variables globales nommés : « y1_memo » et « y2_memo » qui gardent en mémoire des sorties précédentes.

Voici le code utilisé pour déploiement du réseau récurrent sur le robot :

```python
## Thymio callback
def obs(node_id):
        global done, wrec, wpos, wneg, wback, wfwd, y1mem, y2mem

        if not(done):
            cap0 = th[node_id]["prox.horizontal"][0] # cap gauche
            cap2 = th[node_id]["prox.horizontal"][2] # cap centre
            cap4 = th[node_id]["prox.horizontal"][4] # cap droite

            x1 = cap0*(1/4500) # conversion de [0,4500] à [0,1]
            x2 = cap2*(1/4500)
            x3 = cap4*(1/4500)

            y1=perceptron(x1,x2,x3,y1mem,wrec,wpos,wback,wneg,wfwd)
            y2=perceptron(x1,x2,x3,y2mem,wrec,wneg,wback,wpos,wfwd)

            y1mem = y1
            y2mem = y2

            # ajouter votre code ici
            vit_g = int(y1*500) # conversion [-1,1] à [-500,500]
            vit_d = int(y2*500)

            th[node_id]["motor.left.target"] = vit_g
            th[node_id]["motor.right.target"] = vit_d

            if th[node_id]["button.center"]:
```

Et, voici le tracé sur papier :

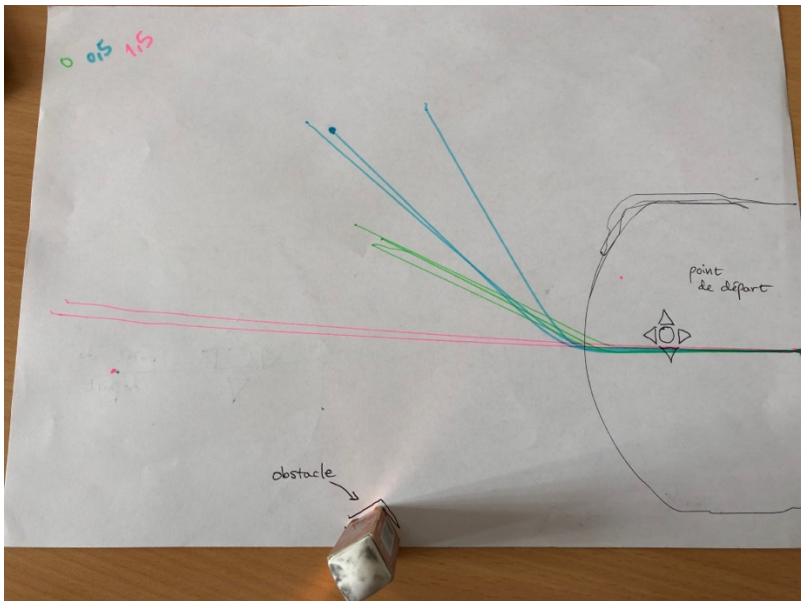


Figure : tracé sur papier – modèle récurrent
en vert : sans récurrence ; en bleu : poids de récurrence de valeur 0.5 ;
en rose : poids de récurrence de valeur 1.5

Grâce à notre tracé sur papier, nous pouvons voir que dans le cas où le poids de récurrence dépasse 1, comme il n'y a pas d'obstacle au démarrage, il garde en mémoire l'absence d'obstacle et avance tout droit en dérivant très peu malgré la détection d'obstacle.

Puis, dans le cas de poids de récurrence dans l'intervalle]0,1], nous pouvons observer qu'il y a un léger retard avant l'évitement d'obstacles et le fait qu'il y a une déviation plus importante vu que le robot garde en mémoire la présence d'obstacles même après la disparition de ce dernier.

3. Le filtre spatial

Certaines configurations du réseau de neurones permettent l'analyse d'une ou plusieurs paternes à partir des informations en entrée. Ces configurations assument une relation spatiale de voisinage entre les entrées et l'adjacence des neurones dans le réseau. Ce type de filtre a eu classiquement vocation à la détection de contours pour des applications dans le traitement d'image. En toute simplicité, un filtre spatial permettrait à un robot de faire la différence entre un obstacle ponctuel et un mur dans une application de navigation.

Afin de valider ces concepts à l'expérimental, la démarche suivante est proposée.

Objectif :
Concevoir un réseau de neurones capable d'assurer le suivi d'un objet. Pour cela, il sera nécessaire de différentier un objet d'un obstacle, ex. un mur. Le réseau fera usage de 5 capteurs de proximité situés devant. La stratégie de navigation peut se synthétiser par :

- si seulement un capteur détecte un objet, alors le robot turne et s'oriente ver l'objet,
- si le capteur central détecte un objet, il avance droit et
- si l'ensemble de capteurs détectent simultanément un objet, alors le robot s'arrête ou recule.

3.1. Modèle de filtre spatial

Nous allons implanter le réseau de la Figure 6 en respectant les poids proposés pour le réseau.

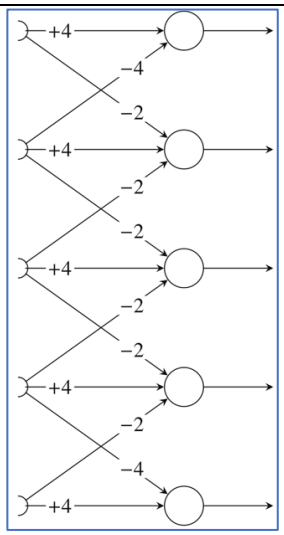


FIG. 6 : Filtre spatial

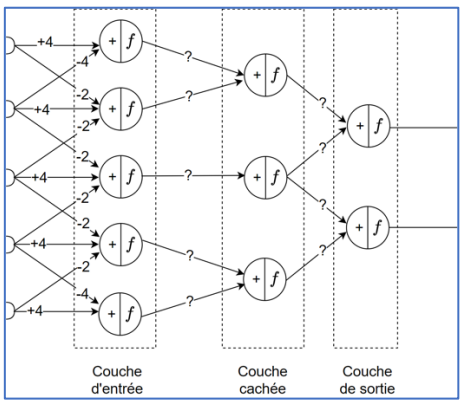
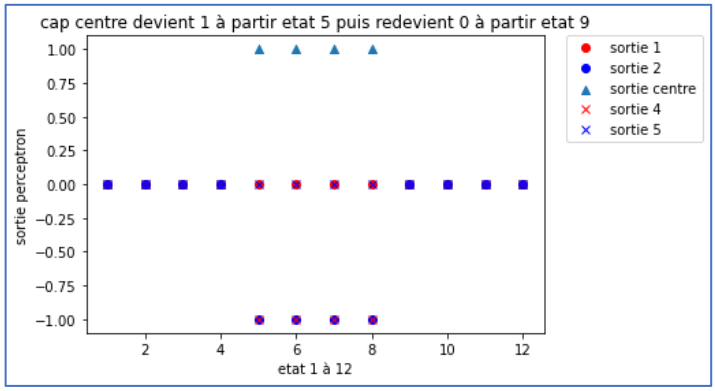


FIG. 7 : Modèle du réseau pour l'identification du type d'objet

Nous pouvons rapporter les sorties du réseau sur trois cas d'usage comme suit :

Cas 1 : objet est devant le robot

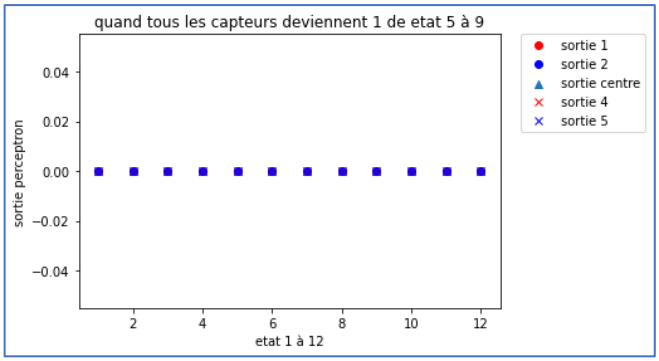


Sur le graphe présenté ici, nous pouvons voir en absicsses 12 états qui fonctionnent un peu comme le temps dans cette simulation pour valider ce modèle de filtre spatial. Puis, nous avons en ordonnés, les 5 sorties de filtre spatial. Sur le robot, la sortie 1 représentera le capteur de gauche, la sortie centre, le capteur de centre et ainsi de suite.

La situation est la suivante : le capteur centre voit un obstacle de l'état 5 à l'état 9.

Nous pouvons voir que comme son nom l'indique, le filtre spatial permet de faire une sorte de détection de contours. Ici, quand la sortie de centre est à 1, les deux sorties adjacentes au centre retournent -1.

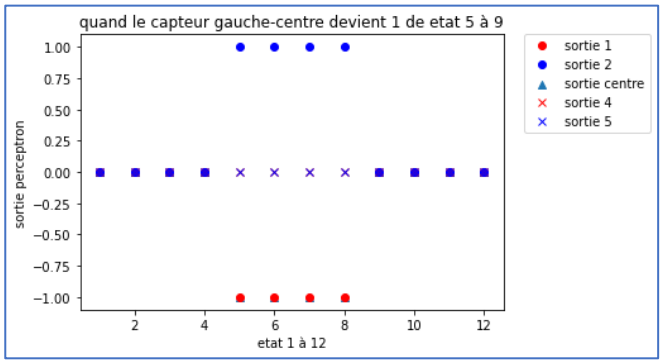
Cas 2 : robot est devant un mur



Dans cette deuxième situation, le robot est devant un mur. Autrement dit, tous les entrées sont à 1 de l'état 5 à 9 selon le concept de simulation présenté dans le cas 1.

Nous pouvons voir que le filtre n'arrive pas à trouver un contour et que toutes les sorties retournent 0 vu qu'il n'y a pas de contour.

Cas 3 : objet est devant le capteur gauche-centre (ou capteur 1)



Ce n'est pas très visible vu que les points qui représentent la valeur de sortie centre sont cachés par ceux de sortie 1 (capteur gauche) mais quand le capteur gauche-centre (autrement dit 'capteur 2') observe un obstacle, les 2 capteurs qui l'entournet (i.e. capteur gauche et capteur centre) retournent une valeur inversée de sortie de ce capteur.

Donc, nous avons la détection de contours qui fonctionne peu importe si l'objet est devant le centre ou un peu vers la gauche ou droite.

3.2. Application du modèle de filtre spatial sur le robot

Nous allons maintenant déployer ce réseau doté d'un « filtre spatial » sur le robot en complétant le réseau comme celui illustré par la Figure 7. On remarquera que le filtre spatial est la couche d'entrée du réseau. Une couche cachée et une couche en sortie complémentent la structure. Nous allons à nouveau utiliser la fonction de saturation dans l'intervalle [−1, 1] comme fonction d'activation.

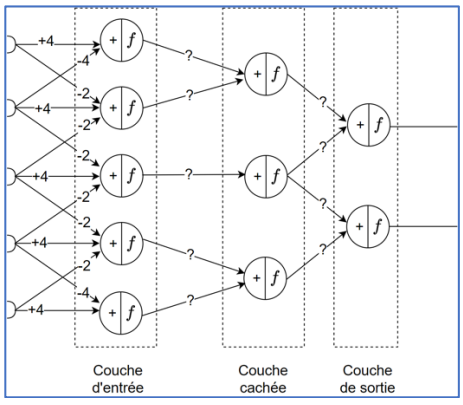


FIG. 7 : Modèle du réseau pour l'identification du type d'objet

Nous avons assigné la sortie située en haut du modèle de réseau au moteur gauche et l'autre au moteur droit. Ceci bien évidemment après la conversion nécessaire comme nous pouvons le voir à travers le code présenté ici :

```python
## Thymio callback
def obs(node_id):
        global done

        if not(done):
            cap0 = th[node_id]["prox.horizontal"][0] # cap gauche 1
            cap1 = th[node_id]["prox.horizontal"][1] # cap gauche 2
            cap2 = th[node_id]["prox.horizontal"][2] # cap centre 3
            cap3 = th[node_id]["prox.horizontal"][3] # cap droite 4
            cap4 = th[node_id]["prox.horizontal"][4] # cap droite 5

            x1 = cap0*(1/4500) # conversion de [0,4500] à [0,1]
            x2 = cap1*(1/4500)
            x3 = cap2*(1/4500)
            x4 = cap3*(1/4500) # conversion de [0,4500] à [0,1]
            x5 = cap4*(1/4500)

            y1=perceptron(x1,x2,0,   4,-4, 0)
            y2=perceptron(x1,x2,x3, -2, 4,-2)
            y3=perceptron(x2,x3,x4, -2, 4,-2)
            y4=perceptron(x3,x4,x5, -2, 4,-2)
            y5=perceptron(0,x4,x5,   0,-4, 4)

            y6=perceptron(y1,y2,0,  2,-2,0)
            y7=perceptron(y3 ,0,0,  2, 0,0)
            y8=perceptron(y4,y5,0, -2, 2,0)

            y9 =perceptron(y6,y7,0,  2,-1, 0)
            y10=perceptron(y7,y8,0, -1, 2, 0)

            # ajouter votre code ici
            vit_g = int(y9*500) # conversion [-1,1] à [-500,500]
            vit_d = int(y10*500)

            th[node_id]["motor.left.target"] = vit_g
            th[node_id]["motor.right.target"] = vit_d
```

Le choix des poids a été inspiré des poids déjà définis pour le modèle de filtre spatial. Les poids de la couche cachée et de la couche de sortie ont été choisi de la manière suivante :

Les poids des extrémités sont positifs et double des poids se trouvant vers l'intérieur et le centre du réseau pour que le robot tourne au lieu d'avancer tout droit.

X1	2	
X2	-2	2
		-1
X3	2	
		-1
X4	-2	2
X5	2	

Tableau : le choix des poids

Avec cette conception, le robot tourne vers l'objet (pour le suivi d'objet) si l'objet est détecté par deux capteurs de gauche. Puis, il avance tout droit si l'objet est détecté par le capteur de centre et avec l'un des capteurs de gauche ou de droit.

Conclusion

Nous avons vu les différentes topologies des réseaux de neurones à travers cette longue séance de TP3 de cours de IA robotique.

Nous avons pu voir la différence entre les réseaux multicouches et les réseaux monocouches. La notion de récurrence qui permet d'assurer l'évitement d'obstacles. Puis, le filtre spatial qui permet de faire une sorte de détection de contours pour le suivi d'un objet.

J'ai surtout consacré beaucoup de temps pour finir la partie « filtre spatial » vu que le choix des poids n'était pas très évident au début.

J'ai appris vers la fin de cette séance que ce choix des poids est normalement géré par la machine dans la vie réelle. Le but de ce TP étant l'attaque de base de ANN et d'IA pour comprendre ses principes de fonctionnement.

TP IV: Apprentissage supervisé pour les réseaux de neurones artificiels

Enseignant : S. Rodriguez
Date de fin de séance : le 16 mai 2022
Rédigé par Jeong Hyun AHN (APP5 – ESR)

Table des matières

Introduction : méthodes d'apprentissage

Dans la littérature, les méthodes d'apprentissage peuvent se structurer en trois catégories : *i) l'apprentissage supervisé, ii) l'apprentissage par renforcement et iii) l'apprentissage non supervisé.* Ce TP est orienté vers **l'apprentissage supervisé**. Les techniques utilisées dans ce TP permettent l'identification automatique d'hyperparamètres d'un réseau de neurones (ex. poids et biais) à partir des données en entrée et des sorties attendues. Le processus d'apprentissage s'oriente donc vers la réduction de l'erreur en sortie du réseau par rapport aux valeurs attendues.

Les problèmes abordés par l'apprentissage supervisé peuvent s'orienter sur divers objectifs. Ces objectives peuvent être **la régression, la classification, la labellisation** (multi-label classification) et **la recherche et le classement**. Dans ce cadre, nous restreindrons le travail à des **problèmes de régression et de classification**.

Dans le but de rendre accessible et efficace le travail d'implantation, nous utiliserons une bibliothèque python (sklearn) intégrant les méthodes d'apprentissage, des objets type perceptron et perceptron multi- couches ainsi que de méthodes d'évaluation.

L'installation de la bibliothèque se fait
grâce à la commande suivante dans la console d'IDE : pip install sklearn

---

1. Modèle Linéaire - Perceptron

Dans un premier temps, nous nous intéresserons à l'entraînement d'un perceptron pour l'implantation d'opérateur logiques OR, AND et XOR. Ces opérateurs ont été paramétrés manuellement au début du cours.

Nous allons comparer ainsi les paramètres obtenus par apprentissage automatique et ceux définis manuellement.

1.1. Procédure pour entamer l'apprentissage automatique en Python

Pour entamer l'apprentissage automatique, nous avons suivi la procédure détaillée ci-dessous :

• Dans un script python, nous avons importé la définition du perceptron de la bibliothèque Sklearn
• Nous avons défini toutes les valeurs possibles pour les entrées de l'opérateur logique
• Nous avons défini les labels en sortie attendus selon l'opérateur concerné (ex. pour l'opérateur OR) : *ORlabels = [0, 1, 1, 1]*

• Nous avons créé une instance du perceptron et nous avons indiqué le nombre d'itérations max. admis pour l'apprentissage ainsi que le seuil de d'erreur toléré (paramètres de convergence).
• Nous avons exécuté la procédure d'apprentissage en utilisant la fonction **fit** de la variable perceptron et nous avons indiqué en paramètres les entrées et les labels préalablement définis.
• La fonction **predict** renvoie la valeur en sortie du réseau pour une entrée spécifiée.
• La fonction **score** renvoie l'estimation sur la précision moyenne du perceptron après entraînement.

1.2. Code Python du modèle linéaire

```
from sklearn.linear_model import Perceptron
#Importation de la définition du perceptron multi-couche
from sklearn.neural_network import MLPClassifier

#liste d'entrées possibles pour l'opérateur logique
data = [[0,0], [0,1], [1,0], [1,1]]

#labels correspondants selon liste d'entrées
ORlabels = [0,1,1,1]
ETlabels = [0,0,0,1]
XORlabels= [0,1,1,0]

#Création du perceptron
operateurOR = Perceptron(max_iter=40, tol=1e-3)
operateurET = Perceptron(max_iter=40, tol=1e-3)
operateurXOR = Perceptron(max_iter=40, tol=1e-3)

# Fit the classifier to the input training data
operateurOR.fit(data, ORlabels)
operateurET.fit(data, ETlabels)
operateurXOR.fit(data, XORlabels)

print('predicted  OR logic : ',operateurOR.predict(data))
print('predicted  ET logic : ',operateurET.predict(data))
print('predicted XOR logic : ',operateurXOR.predict(data))

print('score of  OR logic perceptron : ',operateurOR.score(data, ORlabels))
print('score of  ET logic perceptron : ',operateurET.score(data, ETlabels))
print('score of XOR logic perceptron : ',operateurXOR.score(data, XORlabels))
```

Nous avons conçu et entraîné les perceptrons pour les opérateurs logiques OU, ET et OU-Exclusif. Puis, nous avons vérifié les prédictions du perceptron à l'aide de la fonction **predict**.

1.3. Résultats du modèle linéaire pour la logique XOR

Nous pouvons voir que comme dans les TP précédents, la logique OU-exclusif ne peut pas être satisfait par un classifeur linéaire. (cf. nous avons importé Perceptron de librairie sklearn.linear_model)

Nous avons utilisé la fonction **score** pour estimer la précision moyenne du perceptron après l'entraînement. Voici les résultats obtenus :

```
In [1]: runfile('C:/Users/jahn/Desktop/thymio-direct/tp4_ex1.py', wdir='C:/Users/jahn/Desktop/thymio-
direct')
predicted  OR logic :  [0 1 1 1]
predicted  ET logic :  [0 0 0 1]
predicted XOR logic :  [0 0 0 0]
score of  OR logic perceptron :  1.0
score of  ET logic perceptron :  1.0
score of XOR logic perceptron :  0.5
```

Nous pouvons voir que le score reflète la qualité des résultats du perceptron. Plus le score est proche de 1, plus la précision (ou la qualité) du perceptron est élevé.

La gamme du score rapporté par la bibliothèque varie de 0.0 à 1.0.

2. Classifieur multi-couche

Exercice d'OU-Exclusif comme un classifieur

Comme il a été constaté dans l'exercice précédent, l'opérateur OU-Exclusif ne peut pas être satisfait par un classifieur linéaire. Il est donc nécessaire d'utiliser une topologie de perceptron multi-couche. Pour le faire, nous allons utiliser le MLPClassifier dans la bibliothèque Sklearn : **from** sklearn.neural_network **import** MLPClassifier

2.1. Code et modèle du Classifieur multi-couche

Nous allons choisir une fonction d'activation de type **tangente hyperbolique** ('tanh'), **une couche** cachée et deux neurones et une **limite d'itérations** de 20000 comme sur la Figure suivante :

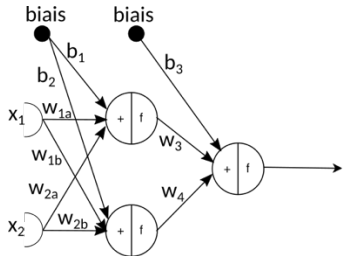


Fig. 1 : Topologie du perceptron multi-couche - Opérateur logique OU-Exclusif

Et, voici le code associé :

```
#Importation de la définition du perceptron multi-couche
from sklearn.neural_network import MLPClassifier

#liste d'entrées possibles pour l'opérateur logique
data = [[0,0], [0,1], [1,0], [1,1]]

#labels correspondants selon liste d'entrées
XORlabels= [0,1,1,0]

#Création du perceptron
operateurXOR = MLPClassifier(max_iter=20000, tol=1e-5, activation='tanh', hidden_layer_sizes=(2))

# Fit the classifier to the input training data
operateurXOR.fit(data, XORlabels)

print('predicted XOR logic : ',operateurXOR.predict(data))

print('score of XOR logic perceptron : ',operateurXOR.score(data, XORlabels))

print('les poids du perceptron XOR : ',operateurXOR.coefs_)
print('les biais du perceptron XOR : ',operateurXOR.intercepts_)
```

2.2. Résultats du Classifieur multi-couche

Voici les résultats obtenus :

```
In [5]: runfile('C:/Users/jahn/Desktop/thymio-direct/tp4_ex4_multicouche.py', wdir='C:/Users/jahn/
Desktop/thymio-direct')
predicted XOR logic :  [0 1 1 0]
score of XOR logic perceptron :  1.0
les poids du perceptron XOR :  [array([[-2.81678514, -3.1511965 ],
       [ 2.91314016,  3.05240924]]), array([[-4.63890629],
       [ 4.49855271]])]
les biais du perceptron XOR :  [array([ 1.52665405, -1.59840279]), array([4.38650433])]
```

Nous pouvons voir que le modèle converge lors de l'étape d'apprentisage. Nous avons un score de 1.

Les valeurs attendues sont : [0, 1, 1, 0] pour la logique XOR.
Et, les valeurs prédites sont : [0, 1, 1, 0] avec notre Classifieur multi-couche.

Nous avons donc bien appris le perceptron à prédire les valeurs attendues.

2.3. Variation des résultats selon la tolérance pour Classifieur multi-couche

Mais, avec une tolérance de 1e-5, nous obtenons de temps en temps un score de 0.5. Autrement dit, il y a des moments quand on obtient [1, 0, 1, 0] comme résultats pour la logique OU-Exclusif alors que nous devrions obtenir [0, 1, 1, 0].

```
In [4]: runfile('C:/Users/jahn/Desktop/thymio-direct/tp4_ex4_multicouche.py', wdir='C:/Users/jahn/
Desktop/thymio-direct')
predicted XOR logic :  [1 0 1 0]
score of XOR logic perceptron :  0.5
les poids du perceptron XOR :  [array([[-3.22544646,  2.94760433],
       [ 2.7689913 ,  1.21126401]]), array([[-3.13405043],
       [-3.57941733]])]
les biais du perceptron XOR :  [array([ 2.55571112, -3.86432147]), array([-0.4283552])]
```

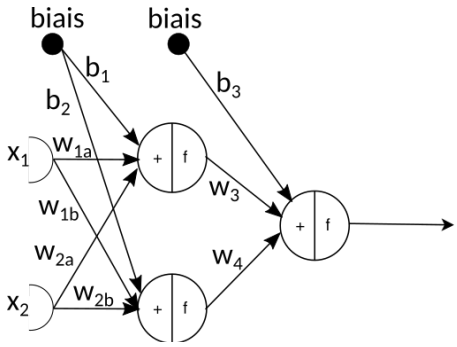
Donc, nous n'obtenons pas les mêmes résultats à chaque cycle d'apprentissage avec notre Classifieur multi-couche.

Nous allons le voir dans la partie suivante mais malgré le fait que nous pouvons obtenir un score exact de 1.0 dans ce modèle « Classifieur » contrairement à un « modèle de régression ». Comme les labels étant des valeurs binaires contrairement aux valeurs réelles dans le cas de « régression », si la tolérance est mal choisie, nous pouvons obtenir un résultat qui est totalement faux.

C'est pour cette raison qu'il faut bien choisir un modèle qui est adapté à son problème lors de choix d'une méthode d'apprentissage.

2.4. Utilisation des attributs pour la récupération des poids

A partir des attributs, nous pouvons récupérer les poids de notre modèle présenté dans la Figure 1. Les poids du réseau correspondent à l'attribut coefs_ de l'objet (w_{1a}, w_{1b}, w_{2a}, w_{2b}, w_3, w_4). Les biais du réseau sont définis dans l'attribut intercepts_ (b_1, b_2, b_3).



Rappel de Fig. 1 : Topologie du perceptron multi-couche - Opérateur logique OU-Exclusif

Pour obtenir les poids du perceptron, il suffit d'afficher les attributs ainsi :

```
print(operateurXOR.coefs_)
```

Cette commande affiche les poids dans l'ordre suivant : (w1a, w1b, w2a, w2b, w3, w4)

```
print(operateurXOR.intercepts_)
```

Cette commande renvoie les biais dans l'ordre suivant : (b1, b2, b3)

3.1. Perceptron - Régression

Exercice d'OU-Exclusif comme une régression

Une approche alternative dans l'implantation de l'opérateur logique OU-Exclusif est celle de le représenter sous la forme d'une régression. En effet, le réseau devra alors prédire les valeurs en sortie **(*valeur dans l'intervalle [0,1] ici*)** souhaitées et non la classe **(*classe 0 ou 1, approche binaire*)** comme dans le cas du classifieur.

L'entité disponible à cet objet dans la bibliothèque Sklearn est celle du MLPRegressor :

```
#Importation de la définition du perceptron multi-couche pour la régression
from sklearn.neural_network import MLPRegressor
```

3.1.2. Code – Régression – logique XOR

Nous avons défini une instance en indiquant un minimum de 2 couches cachées, une fonction d'activation 'tanh' et l'utilisation de la méthode d'optimisation 'lbfgs'.

Nous avons entrainé le réseau (fonction fit) en indiquant les entrées et la valeur de sortie correspondante. Puis, nous avons estimé le score et vérifié les prédictions.

Voici le code :

```
from sklearn.neural_network import MLPRegressor

#liste d'entrées possibles pour l'opérateur logique
data = [[0,0], [0,1], [1,0], [1,1]]

#labels correspondants selon liste d'entrées
ORlabels = [0,1,1,1]
ETlabels = [0,0,0,1]
XORlabels= [0,1,1,0]

#Création du perceptron
operateurXOR= MLPRegressor(max_iter=20000, tol=1e-5, activation='tanh', hidden_layer_sizes=(2),solver='lbfgs')

# Fit the classifier to the input training data
operateurXOR.fit(data, XORlabels)

print('predicted XOR logic : ',operateurXOR.predict(data))

print('score of XOR logic perceptron : ',operateurXOR.score(data, XORlabels))

print('les poids du perceptron XOR : ',operateurXOR.coefs_)
print('les biais du perceptron XOR : ',operateurXOR.intercepts_)
```

3.1.1. Résultats – Régression – logique XOR

Voici les résultats obtenus :

```
predicted XOR logic :  [2.24187316e-04 9.99747490e-01 9.99789052e-01
3.08493298e-04]
score of XOR logic perceptron :  0.9999997463116449
les poids du perceptron XOR :  [array([[1.18209833, 1.21956068],
       [1.16752398, 1.20427465]]), array([[-1.35718499],
       [ 1.34132201]])]
les biais du perceptron XOR :  [array([-1.97548317, -0.38727815]),
array([-0.81076925])]
```

Nous pouvons deviner la topologie et les poids associés du réseau ainsi créé avec ces résultats obtenus. Pour rappel, Nous avons défini une instance en indiquant un minimum de 2 couches cachées, une fonction d'activation 'tanh' et l'utilisation de la méthode d'optimisation 'lbfgs' dans l'étape précédente.

Contrairement, au cas où nous avons considéré ce problème de OU-Exclusif comme un problème de « Classifieur », nous obtenons **toujours** (constamment) un score très proche de 1 avec ce modèle de « **régression** ».

Certes, les poids différent très légèrement mais c'est car il y a un aspect aléatoire dans l'initialisation des poids lors de l'apprentissage.

3.2. Perceptron de régression – Application sur le robot

Nous allons implanter un perceptron **multicouche** de **régression** permettant le suivi de ligne en utilisant en entrée les mesures **normalisées** des capteurs prévus à cet effet dans le robot Thymio (grâce à la variable "prox.ground.delta"). Le réseau prédira les vitesses de moteurs appropriées (normalisées dans une gamme [-1,1]).

3.2.1. Apprentissage dans le code principal – code lancé une seule fois

Le code suivant est un code qui est lancé qu'une seule fois lors de cette application robotique

```
121   # [0,1] ligne noir à gauche
122   data1= [[0,0],[0,0.2],[0,0.4],[0,0.6],[0,0.8],[0,1]]
123   data2= [[0.2,0],[0.4,0],[0.6,0],[0.8,0],[1,0]]
124   data3= [[0.2,0.2],[0.4,0.4],[0.6,0.6],[0.8,0.8],[1,1]]
125
126   # [-1,1] tourne à gauche pour suivre la ligne
127   label1=[[0,0],[0,0.2],[0,0.4],[0,0.6],[0,0.6],[0,0.6]]
128   label2=[[0.2,0],[0.4,0],[0.6,0],[0.6,0],[0.6,0]]
129   label3=[[0.2,0.2],[0.4,0.4],[0.6,0.6],[0.6,0.6],[0.6,0.6]]
130
131   data  =[0]*16
132   labels=[0]*16
133
134   for i in range(0,6):
135       data[i]=data1[i]
136       labels[i]=label1[i]
137
138   for j in range(0,5):
139       data[j+6]=data2[j]
140       labels[j+6]=label2[j]
141
142   for k in range(0,5):
143       data[k+11]=data3[k]
144       labels[k+11]=label3[k]
145
146   perceptron = MLPRegressor(max_iter=200000, tol=1e-5, activation='tanh', hidden_layer_sizes=(4), solver='lbfgs')
147   perceptron.fit(data, labels)
```

Le « suivi de ligne » est beaucoup plus fonctionnel et précis lorsque les données en entrée et les résultats attendus (ou les labels) sont plus nombreux et complets.

Nous avons ici les données et les résultats attendus qui sont déclarés comme des vecteurs. Nous avons créé des vecteurs data1, data2 et data3 pour former un vecteur data qui rassemble tous ces données. Par exemple, la dernière case de data1 [0,1] correspond à la situation où la ligne noire est vue par le capteur de sol à gauche du robot.

Dans ce cas, notre résultat attendu est le comportement suivant : [-1,1] (i.e. nous voulons que le robot tourne à gauche pour suivre la ligne).

3.2.2. Utilisation du perceptron dans la partie Callback

```
## Thymio callback
def obs(node_id):
        global done, perceptron
        try:
            if not(done):
                sol1 = th[node_id]["prox.ground.delta"][0]*(1/1023) # conversion de [0,1023] à [0,1]
                sol2 = th[node_id]["prox.ground.delta"][1]*(1/1023)

                # ajouter votre code ici
                pred = perceptron.predict([[sol1,sol2]])
                vit_g = int(pred[0][0]*350) # conversion [-1,1] à [-500,500]
                vit_d = int(pred[0][1]*350)

                th[node_id]["motor.left.target"] = vit_g
                th[node_id]["motor.right.target"] = vit_d

                if th[node_id]["button.center"]:
```

Cette partie du code est celui qui est répété à chaque callback. Nous récupérons les données de 2 capteurs de sol, puis nous effecutons une conversion. (conversion de [0,1023] à [0,1])

Avec la fonction predict() de bibliothèque Sklearn, le perceptron va essayer de prévoir des valeurs attendues à la sortie. Ces valeurs attendues seront utilisés pour contrôler les 2 moteurs du robot après une deuxième conversion. (conversion de [-1,1] à [-500,500])

Nous pouvons voir qu'en faisant varier la valeur des labels qui varie de 0 à 1 et la plage de conversion de vitesse, que le robot effectue un suivi de ligne plus ou moins vite. Il faut noter que quand la rotation du robot ou la vitesse du robot est trop grande, que le suivi de ligne ne fonctionne pas très correctement.

Nous avons un vidéo démonstratif inclus dans le même fichier que ce compte-rendu pour montrer le bon fonctionnement de ce suivi de ligne.

3.2.3. La fonction d'activation, les poids et les biais de ce modèle

Voici les résultats obtenus :

```
In [3]: runfile('C:/Users/jahn/Desktop/thymio-direct/test.py', wdir='C:/Users/jahn/
Desktop/thymio-direct')
score of perceptron SL :  0.9967353944497312
les poids du perceptron SL:  [array([[-0.23071706, -0.12787874, -1.05626316,
-2.02493986],
       [ 0.56156072, -2.252754  , -0.70305405, -0.48874005]]), array([[ 0.39493481,
-0.87805302],
       [-0.00510621, -0.82672352],
       [ 0.93395194,  0.50082287],
       [-1.0937941 , -0.14057491]])]
les biais du perceptron SL :  [array([0.38066846, 0.57722782, 0.32704642, 0.57949351]),
array([0.14249159, 0.66499942])]
la fonction activation SL :  identity
```

La fonction d'activation est une fonction nommée : ***'identity'*** dans la bibliothèque Sklearn.

D'après la documentation, c'est une fonction « no-op activation » **(i.e. retourne f(x) = x)**

Nous avons un vidéo démonstratif, un graphe de topologie, les données sur les poids, les biais et la fonction d'activation.

Les poids ainsi que les biais de ce perceptron sont indiqués en haut.

3.3. Suivi de ligne avec évitement d'obstacles avec un seul réseau multicouche

3.3.1. Code de suivi de ligne avec évitement d'obstacles

Pour aller plus loin, nous allons implanter dans un seul réseau multicouche, une stratégie de suivi de ligne avec évitement d'obstacles. Le robot sera ainsi capable d'éviter des obstacles placés sur la ligne. Pour ce faire, le réseau fera usages de proximètres et des capteurs au sol.

```
# question bonus

# si obstacle devant
# avance en tournan a gauche
data4= [[0.7,0.7,0.6,0.6],[0.7,0.7,0.6,0.6],[0.7,0.7,0.6,0.6]]
label4=[[0.2,0.8],[0.2,0.8],[0.2,0.8]]

# si obstacle devant
# avance en tournan a gauche
data5= [[0.6,0,0.6,0.6],[0.8,0,0.6,0.6],[1,0,0.6,0.6]]
label5=[[0.2,0.8],[0.2,0.8],[0.2,0.8]]

data6= [[0.6,0,0.6,0],[0.8,0,0.6,0],[1,0,0.6,0]]
label6=[[0.2,0.8],[0.2,0.8],[0.2,0.8]]

data7= [[0.6,0,0,0.6],[0.8,0,0,0.6],[1,0,0,0.6]]
label7=[[0.2,0.8],[0.2,0.8],[0.2,0.8]]

#revenir au SL après 0,0,0,0 comme entrée
#tourne a droite
data8= [[0,0,0.1,0.1],[0,0,0.1,0],[0,0,0,0.1]]
label8=[[0.8,-0.2],[0.8,-0.2],[0.8,-0.2]]

data  =[0]*31
labels=[0]*31

for i in range(0,6):
    data[i]=data1[i]
    labels[i]=label1[i]

for j in range(0,5):
    data[j+6]=data2[j]
    labels[j+6]=label2[j]

for k in range(0,5):
    data[k+11]=data3[k]
    labels[k+11]=label3[k]

for l in range(0,3):
    data[l+16]=data4[l]
    labels[l+16]=label4[l]
```

```
## Thymio callback
def obs(node_id):
        global done, perceptron
#        try:
        if not(done):
                sol1 = th[node_id]["prox.ground.delta"][0]*(1/1023) # conversion de [0,1023] à [0,1]
                sol2 = th[node_id]["prox.ground.delta"][1]*(1/1023)

                cap_centre = th[node_id]["prox.horizontal"][2]
                cap_droite = th[node_id]["prox.horizontal"][4]

                centre = cap_centre*(1/4500)
                droite = cap_droite*(1/4500)

                # ajouter votre code ici
                pred = perceptron.predict([[sol1,sol2,centre,droite]])
                vit_g = int(pred[0][0]*500) # conversion [-1,1] à [-500,500]
                vit_d = int(pred[0][1]*500)

                th[node_id]["motor.left.target"] = vit_g
                th[node_id]["motor.right.target"] = vit_d

        if th[node_id]["button.center"]:
                print("button.center pressed")
                #Arret du robot
                th[node_id]["motor.left.target"] = 0
                th[node_id]["motor.right.target"] = 0
                set_leds(th, id, 0, 0, 32)
                done = True
```

Nous avons ajouté dans les variables « data » et « labels » du code principal, deux cases supplémentaires pour chaque case de vecteur. Ces deux cases supplémentaires correspondent aux capteurs centre et droite.

En fait, notre évitement d'obstacles ne fera qu'en un seul sens : vers gauche

Donc, nous avons choisi les capteurs centre et droite pour vérifier quand l'obstacle (qui se trouvera toujours à droite du robot vu que le robot ne tournera qu'à gauche pour éviter des obstacles) disparaît.

3.3.2. Résultats de suivi de ligne avec évitement d'obstacles

Nous avons réussi à faire l'évitement d'obstacles au cours d'un suivi de ligne. Mais, le robot n'avance pas correctement vers droite pour retrouver la ligne à suivre après l'évitement. Alors que ce cas est bien précisé dans les variables « data » et « labels ».

C'est probablement dû au fait qu'il y a des « data » et des « labels » qui décrivent une situation similaire mais qui demandent des résultats différents. (le résultat ne peut pas converger dans ce cas là !)

Par exemple, notre « data » est défini ainsi : [sol1, sol2, cap_centre, cap_droite]

si nous avons le cas suivant : [0,0,0,0]

nous devons demandé au robot d'avancer vers droite, vu que l'obstacle a disparu et le robot devrait retrouver la ligne.

Mais, dans le cas où : [0, 0.2, 0.1, 0.3] le « capteur de sol à droite » voit légèrement la ligne noire et il y a toujours l'obstacle, nous allons demander au robot d'avancer vers gauche, pour éviter complètement l'obstacle.

Donc, il nous demandons des résultats différents pour une situation qui ressemble beaucoup. D'où l'importance de bien choisir son modèle au départ pour éviter des événements contradictoires comme celui-ci pour l'apprentissage correct du robot.

Nous avons un vidéo démonstratif inclus dans le même fichier que ce compte-rendu pour montrer ce comportement.

Conclusion

Nous avons étudié dans ce TP, deux principales méthodes pour **l'apprentissage supervisé :** Régression et Classifier.

Nous avons vu que leur principale différence est : régression peut prédire une sortie binaire et le classifieur une sortie de valeur réelle.

Nous avons vu que pour choisir une de ces méthodes, il faut bien comprendre le problème à résoudre. Par exemple, dans le cas de notre robot Thymio, nous avons souvent utilisé la méthode de Régression vu que la sortie (= la vitesse des moteurs) devait être une valeur réelle.

TP 5 : Apprentissage par renforcement : La règle de Hebb

Enseignant : S. Rodriguez
Date de fin de séance : le 17 mai 2022
Rédigé par Jeong Hyun AHN (APP5 – ESR)

Table of Contents

Introduction

Dans ce dernier TP de cours IA Robotique, nous allons faire apprendre le robot à éviter des obstacles en lui indiquant le comportement attendu par la saisie des boutons (forward, backward, left, right).

La première partie de ce TP est consacrée aux aspects théoriques du système de récompense pour l'apprentissage machine. La deuxième partie sera consacrée aux aspects pratiques.

1. La règle de Hebb

C'est une technique d'apprentissage pour les réseaux de neurones permettant l'apprentissage par renforcement. Elle permet de modifier les poids des connexions entre les neurones. Cette technique s'appuie sur le postulat suivant : Si un réseau effectue une action correcte alors on conforte cette action. Cela est décliné de la manière suivante :

• Si la sortie de deux neurones connectés est similaire, les poids de leur connexion doit augmenter. • Si la sortie de deux neurones connectés est différente, les poids de leur connexion doit décroître.

Le variation introduite dans les poids des connexions entre le neurone a et le neurone b est formalisée par l'équation :

$$\Delta w_{ab} = \alpha \cdot y_a x_b \quad (1)$$

où w_{ab} représente le poids reliant les neurones a et b, Δw_{ab} est la variation de w_{ab}, y_a est la sortie du neurone a, x_b représente l'entrée du neurone b et α est une constante modélisant la vitesse d'apprentissage.

L'application de la règle de Hebb est possible en assumant deux hypothèses :

- Le robot explore son environnement et rencontre des situations pour lesquelles il obtient des entrées qui le permettent à son réseau de neurones d'estimer un ensemble de sorties.
- Le robot reçoit des informations lui indiquant si son comportement est bon ou non.

L'évaluation de la qualité du comportement du robot sera effectué par un observateur humain. Il indiquera au robot manuellement un retour concernant son comportement. Il est important de différentier le fait que l'évaluation ne porte pas sur l'état du robot (ex. proche/loin d'un obstacle) mais sur le comportement du robot (ex. il s'approche ou évite un obstacle). Ainsi, les connexions du réseau peuvent générer un comportement sur la base de l'état mesuré par les capteurs.

1.1. Modèle – Indication du comportement attendu au robot

Apprendre à éviter un obstacle

Nous souhaitons faire apprendre à notre robot Thymio à éviter un obstacle. Pour cela on pourrait permettre au robot de se déplacer librement dans un environnement et on récompensera le robot quand il évite correctement un obstacle par l'appuie d'une touche. Alternativement, on pourra faire l'appui d'une autre touche s'il collisionne avec l'obstacle. Cette méthode peut demander un temps considérable pour converger.

De manière alternative, nous présenterons au robot multiples situations et on indiquera le comportement attendu comme par exemple :

• Détecter un obstacle à gauche et tourner à droite est un bon comportement ;

• Détecter un obstacle à droite et tourner à gauche est un bon comportement ;

• Détecter un obstacle devant et avancer est un mauvais comportement.

Cette stratégie renforcera les poids du réseau associés à un bon comportement grâce à un retour d'apprentissage binaire. En contraste, un apprentissage supervisé quantifie l'erreur entre la sortie désirée et la sortie actuelle pour appliquer une correction des poids et obtenir ainsi la sortie exacte souhaité.

1.2. Algorithmes

1.2.1. Algoritme du réseau

Afin de valider la règle de Hebb à l'expérimentale et de l'appliquer à l'apprentissage de l'évitement d'obstacle, il est nécessaire de déployer le réseau illustré dans la Figure 1.

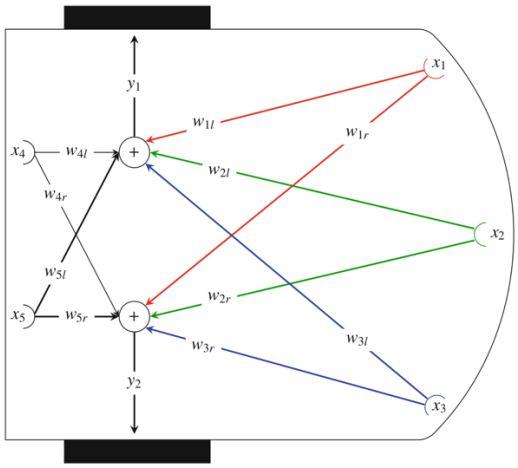


Fig. 1 : Modèle du réseau pour la validation de la règle de Hebb

Pour ce faire, la structure logiciel proposée par l'algorithme 1 préconise un échantillonnage du réseau de 100ms où les sorties y_1 et y_2 sont mises à jours en appliquant l'équation . Ces sorties sont appliquées aux moteurs gauche et droit.

Algorithm 1 Algorithme du réseau de la Fig. 1

```
1 : float période=0.1 // Période d'échantillonage (s)
2 : if période du timer est expirée then
3 :     x ← Valeurs mesurées par le capteurs de proximité
4 :     y ←  W · x
5 :     Vitesse roue gauche ← y[1]
6 :     Vitesse roue droite ← y[2]
7 : end if
```

Les capteurs qui intervient en tant qu'entrées du réseau de la Fig. 1 sont cinq, identifiées ci-dessous :

$$\begin{array}{rcl} x_1 & \leftarrow & \text{Capteur frontal gauche} \\ x_2 & \leftarrow & \text{Capteur central} \\ x_3 & \leftarrow & \text{Capteur frontal droit} \\ x_4 & \leftarrow & \text{Capteur arrière gauche} \\ x_5 & \leftarrow & \text{Capteur arrière droit} \end{array} \quad (2)$$

Nous assumons que les valeurs issues de capteurs sont normalisées et définies dans l'intervalle continu [0, 100] (0 représente l'absence d'obstacle et 1 la présence très proche d'un obstacle). Les sorties du réseau sont défis entre [−100, 100], tout valeur en dehors de l'intervalle sera saturé à la valeur 100 si y > 100 et −100 si y < −100.

L'algorithme 1 sous la notation vectorielle considère alors le vecteur d'entrées :

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \quad (3)$$

alors, la sortie du réseau est définie comme suit :

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{1l} & w_{2l} & w_{3l} & w_{4l} & w_{5l} \\ w_{1r} & w_{2r} & w_{3r} & w_{4r} & w_{5r} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \mathbf{Wx} \quad (4)$$

1.2.2. Algorithme de récompense

La récompense

Afin d'introduire la notion d'apprentissage dans l'algorithme, il est nécessaire de modifier les poids du modèle, W. Nous utiliserons alors 4 boutons pour indiquer au robot le comportement souhaité face à une situation. Par exemple, si le capteur frontal gauche détecte un obstacle, alors le robot devra tourner à droite. L'implantation de ce mécanisme devra associer les actions à chaque bouton. L'algorithme 2 synthétise cela comme suit :

Algorithm 2 Algorithme de récompense

```
1 : if le bouton avancer est appuyé then
2 :     y1 ← 100
3 :     y2 ← 100
4 : end if
5 : if le bouton reculer est appuyé then
6 :     y1 ← −100
7 :     y2 ← −100
8 : end if
9 : if le bouton tourner à gauche est appuyé then
10 :    y1 ← −100
11 :    y2 ← 100
12 : end if
13 : if le bouton tourner à droite est appuyé then
14 :    y1 ← 100
15 :    y2 ← −100
16 : end if
```

La règle de Hebb

La dernière phase de l'implantation de l'apprentissage concerne la mise à jour des poids suivant la règle de Hebb. L'algorithme 3 décrit cette dernière phase :

Algorithm 3 Algorithme de récompense

```
1 : x ← valeurs issues des capteurs
2 : for j dans {1,2,3,4,5} do
3 :     wjl ← wjl + αy1xj
4 :     wjr ← wjr + αy2xj
5 : end for
```

2. Implémentation des algorithmes sous Python

2.1. Code Python – Évitement d'obstacles – modèle récompense

Nous allons faire apprendre à notre robot à éviter des obstacles.

```python
def action(s):
    return np.clip(s,-100,100) # saturation d'un vecteur 2D

def perceptron(X,poids):
    s= np.matmul(poids,X)
    y = action(s) # s = produit vectoriel
    return y
```

```python
poids = np.array([[0,0,0,0,0],[0,0,0,0,0]])
alpha = 0.01
memo = np.array([[1,0,0,0,0],[0,0,0,0,0]])
```

Nous avons déclarés : poids, alpha, memo comme des variables globales. « memo » garde en mémoire les vecteurs de poids et permet de vérifier si les vecteurs de poids a changé ou pas. (afin de les imprimer qu'une seule fois après un changement).

La variable « alpha » est celle qui a été présentéé dans l'explication d' « Algorithme 3 ».

```
global done, perceptron, poids, alpha, memo

if not(done):
    cap_gauche = th[node_id]["prox.horizontal"][0]
    cap_centre = th[node_id]["prox.horizontal"][2]
    cap_droite = th[node_id]["prox.horizontal"][4]
    arriere_gauche = th[node_id]["prox.horizontal"][5]
    arriere_droite = th[node_id]["prox.horizontal"][6]

    cap1 =       cap_gauche*(100/4500)
    cap2 =       cap_centre*(100/4500)
    cap3 =       cap_droite*(100/4500)
    cap4 = arriere_gauche*(100/4500)
    cap5 = arriere_droite*(100/4500) # conversion [0,4500] à [0,100]

    X = np.array([[cap1],[cap2],[cap3],[cap4],[cap5]])
    Y = np.array([[0],[0]])

    if th[node_id]["button.forward"]:
        print("button.up pressed")
        Y[0] = 100
        Y[1] = 100
        for j in range(0,5):
            poids[0][j] = poids[0][j] + alpha*Y[0]*X[j] #
            poids[1][j] = poids[1][j] + alpha*Y[1]*X[j]

    if th[node_id]["button.backward"]:
        print("button.down pressed")
```

Contrairement aux TP précédents, nous utilisons des valeurs des capteurs normalisées entre [0, 100] ici. Puis, nous utilisons des vecteurs comme des entrées et des sorties du perceptron. La fonction de saturation est normalisée entre [-100, 100] aussi.

```
    if th[node_id]["button.right"]:
        print("button.right pressed")
        Y[0] = 100
        Y[1] = -100
        for j in range(0,5):
            poids[0][j] = poids[0][j] + alpha*Y[0]*X[j]
            poids[1][j] = poids[1][j] + alpha*Y[1]*X[j]

    resultat = perceptron(X,poids)

    if(not(np.array_equal(memo,poids))):
        print('poids = ', poids)
        memo = np.copy(poids)

    vit_g = int(resultat[0])
    vit_d = int(resultat[1])

    th[node_id]["motor.left.target"] = vit_g
    th[node_id]["motor.right.target"] = vit_d

    if th[node_id]["button.center"]:
        print("button.center pressed")
        #Arret du robot
        th[node_id]["motor.left.target"] = 0
        th[node_id]["motor.right.target"] = 0
        set_leds(th, id, 0, 0, 32)
        done = True
```

Nous avons des conditions « if » pour la saisie des boutons. Nous modifions les poids en fonction du bouton appuyé. Les poids ainsi modifiés sont utilisés par le perceptron pour fournir aux moteurs les valeurs de vitesse attendues par l'humain.

2.2. Résultats – Poids et Comportement

On apprend au robot à eviter des obstacles en le récompensant par l'appuie d'une touche.

Nous utilisons quatre boutons ici : right, left, forward et backward pour montrer au robot quelle action à réaliser dans chaque situation (obstacle devant, derrière, à gauche, à droite…)

	poids de X1	poids de X2	poids de X3	poids de X4	poids de X5						
alpha = 0.5											
début	0	0	0	0	0						
	0	0	0	0	0						
bouton right	3708	0	0	0	0						
	-3708	0	0	0	0						
bouton left	3708	0	-4127	0	0						
	-3708	0	4127	0	0						
bouton back	-1008	-4822	-9058	0	0						
	-8424	-4822	-804	0	0						
intermediaire	…	…	…	…	…						
							2e prise				
final	4052	-4822	-13926	3927	4125		9445	-9720	-24823	4264	4453
	-13482	-4822	4069	3927	4125		-29016	-9720	4882	4264	4453

Au début, tous les poids sont initialisés à 0. Au début, nous avons mis en place un obstacle devant les deux capteurs de gauche du robot. En appuyant sur le bouton droite du robot, nous avons appris au robot de tourner vers droite dans le cas où il y a un obstacle à gauche.

Cette action est traduite par le changement du poids sur les connexions qui sont reliées au capteur qui se trouve à gauche : poids de 3708 pour le moteur gauche, -3708 pour le moteur droite.

En mettant un obstacle devant le capteur droite, nous mettons à -4127 (moteur guahce) 4127(moteur droite) aux poids sur les connexions reliées au capteur de droite.

Ainsi, à chaque changement de l'emplacement d'objets et de bouton appuyé, il y a un changement de poids.

Nous obtenons des poids suivants une fois que l'apprentissage du robot est terminé :

final	4052	-4822	-13926	3927	4125
	-13482	-4822	4069	3927	4125

La première colonne montre que : le robot tourne à droite quand le capteur gauche détecte un objet. La deuxième colonne montre que : le robot recule quand le capteur centre détecte un objet. Les deux dernières colonnes montrent que : le robot avance quand un objet se trouve derrière le robot.

2.3. Résultats avec alpha = 0.01

Voici le même tableau mais avec la valeur de variable alpha = 0.01 cette fois-ci.

alpha = 0.01											
début	0	0	0	0	0						
	0	0	0	0	0						
bouton right	0	0	0	0	0						
	0	0	0	0	0						
bouton left	193	0	-93	0	0						
	-193	0	93	0	0						
bouton back	192	-4822	-9058	0	0						
	-8424	-4822	-804	0	0						
intermediaire	...	...	...	...	...						
							2e prise				
final	217	-67	-690	229	97		189	-300	-681	195	195
	-801	-206	99	297	97		-775	-300	99	195	195

Nous pouvons remarquer que la grandeur des poids est relativement petite par rapport au cas où alpha = 0.5. Nous pouvons aussi voir que les poids ne varient pas autant lors d'un nouveau apprentissage.

Donc, en diminiuant la variable alpha, nous devons apprendre plusieurs fois pour que le robot apprenne correctement le comportement voulu par l'humain. Mais, l'avantage d'une valeur d'alpha petite est que la nouvelle apprentissage perturbe moins l'apprentissage précédente.

2.4. Apprendre le robot à avancer en absence d'obstacles

Dans cette dernière partie du TP, nous allons modifier le programme afin qu'il puisse apprendre à avancer en absence d'obstacle.

Cette modification ne demande pas beaucoup de changements.

Nous allons tout simplement ajouter une case supplémentaire dans les vecteurs de poids et d'entrée X.

```
poids = np.array([[0,0,0,0,0,1],[0,0,0,0,0,1]])
alpha = 0.5
memo = np.array([[1,0,0,0,0,0],[0,0,0,0,0,0]])
```

Nous avons ajouté un biais avec poids de 1, ce qui correspond à la dernière case du vecteur poids sur le code au-dessus.

```
X = np.array([[cap1],[cap2],[cap3],[cap4],[cap5],[biais]])
Y = np.array([[0],[0]])
```

Nous avons ajouté une entrée nommée [biais] dans le vecteur d'entrées. Sa valeur étant 50 à chaque fois.

Nous obtenons ainsi les poids très similaires que dans le cas précédent mais avec un ajout d'un biais qui fait que même en absence des obstacles la sortie du perceptron est constamment : Y = X * poids = [1 x 50 ; 1 x 50] = **[50 ; 50]** comme sortie qui est utilisé directement pour contrôler les deux moteurs pour faire avancer le robot.

Nous avons un vidéo démonstratif inclus dans le même fichier que ce compte-rendu qui montre ce comportement.

Conclusion

Nous avons vu à travers ce dernier TP d'IA Robotique le fonctionnement du système de récompense pour l'apprentissage robotique.

Le système de récompense classique est quand on permet au robot de se déplacer librement dans un environnement et on récompense le robot quand il évite correctement un obstacle par l'appuie d'une touche.

Mais, comme la méthode classique peut demander un temps considérable pour converger, nous avons utilisé dans ce TP le système de récompense alternative où nous avons indiqué au robot le comportement attendu par l'appuie d'une touche.

Nous avons aussi appris comment utiliser les vecteurs comme entrées et sorties du perceptron grâce aux fonctions du bibiliothèque *numpy* comme :

matmul() : pour le produit matriciel, clip() : pour la saturation…