TP5 - IA pour la Robotique

Master SETI

Alaf DO NASCIMENTO SANTOS
Simon BERTHOMIEUX

2023/2024

Contents

1 Introduction

Dans le cadre du module **IA pour la Robotique**, l'objectif principal de ce travail est d'étudier le modèle neuronal à l'aide d'un robot à mouvement différentiel, modèle Thymio, simulé avec le logiciel **Webots** version 2021b et **Python** 3.9v. Voici les objectifs visés par les travaux pratiques :

- Implantation d'une porte logique à travers un modèle de perceptron simple
- Application du modèle au robot Thymio sur le simulateur
- Dévelopment d'un modèle de perceptron analogique

Quatre fichiers Python ont été créés pour le projet :

- **tp5.py** : ce fichier sert de contrôleur principal, orchestrant le comportement du système.
- le contrôleur principal s'interface avec :
  - **flags_file.py**: Il permet d'activer ou de désactiver des fonctions telles que le débogage et le contrôle du clavier.
  - **motors_controller.py** : les fonctions liées au contrôle des moteurs du robot sont implémentées ici.
  - **perceptron.py** : les fonctions du modèle logique de perceptron.

2 Perceptron : implantation d'une porte logique

Le modèle du perceptron est répresenté par la Figure 1. Il est composé par deux entrées x_1 et x_2, pouvant être de valeur unitaire ou nulle, et d'une sortie aussi binaire y en fonction des valeurs d'entrée et des poids w_i choisis.

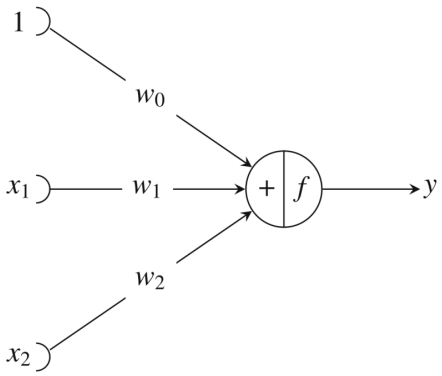


Figure 1: Perceptron à deux entrées.

Ici nous utilisons la fonction d'activation échelon, dont l'équation 1 est utilisé pour une représentation formelle. Finalement, l'équation 2 décrit ce modèle de perceptron mathématiquement.

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{1}$$

$$x = w_0 + \sum_{i=1}^{2} w_i \cdot x_i \tag{2}$$

D'abord, pour explorer le modèle proposé, nous allons essayer de assigner les poids w_i de sorte à assurer une sortie telle que celles indiquées par le Tableau 1. En d'autres termes, on souhaite avoir des valeurs w_0, w_1, et w_2 afin d'obtenir un perceptron qui remplit la fonction des portes logiques ET, OU et XOR.

x_1	x_2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 1: Opérations logiques à deux entrées

On commence par définir les variables x1 et x2 avec des valeurs 0 et/ou 1, comme donné par le code Python suivant :

```
x1 = [0, 0, 1, 1]
x2 = [0, 1, 0, 1]
```

Ensuite, dans une fonction suivante, à partir de l'expression $s = w_0 + x_1 \cdot w_1 + x_2 \cdot w_2$, laquelle est tout simplement un produit scalaire entre deux vecteur (avec la première position de X éguale à 1), on calcule la valeur de s :

```
def get_s(x, w):
    s = np.dot(x,w)
    return s
```

Après, dans une deuxième fonction prenant en entrée s, on calcule y à travers la définition 3. Le code Python peut être simplifié en une seule opération logique, comme le montre la fonction suivante :

```
def get_y(s):
    return s > 0
```

$$y = \begin{cases} 1 & \text{if } s > 0 \\ 0 & \text{if } s \leq 0 \end{cases} \tag{3}$$

Finalement, nous voulons effectuer dans le script tous les cas indiqués dans le Tableau 1. Dans un premier moment, le code suivant a été écris tout simplement pour les trouvé d'une façon empirique avec des boucles :

```python
def find_weights(gate):
    # Iterate through each x value one at a time to find weights
    for w0 in [i * 0.1 for i in range(-10, 11)]: # from -1 up to 1, step
    = 0.1
        for w1 in [i * 0.1 for i in range(-10, 11)]:
            for w2 in [i * 0.1 for i in range(-10, 11)]:
                all_correct = True
                for i in range(len(x1)):
                    s = get_s([1,x1[i],x2[i]],[w0,w1,w2])
                    y = get_y(s)
                    if gate == "OR":
                        # Check if output matches OR gate truth table
                        if not (y == (x1[i] or x2[i])):
                            all_correct = False
                            break
                    if gate == "AND":
                        # Check if output matches AND gate truth table
                        if not (y == (x1[i] and x2[i])):
                            all_correct = False
                            break
                    elif gate == "XOR":
                        # Check if output matches XOR gate truth table
                        if not (y == (x1[i] ^ x2[i])):
                            all_correct = False
                            break
                if all_correct:
                    print("Found weights for", gate, "gate:", "w0 =", w0
    , ", w1 =", w1, ", w2 =", w2)
                    test_gate(x1, x2, [w0, w1, w2])
                    break
            if all_correct:
                break
        if all_correct:
            break
```

Le code précédent fait des appels à la fonction suivante. Cela est nécessaire pour mieux visualiser les résultats obtenus pour chaque ensemble de poids obtenus.

```python
def test_gate(X1, X2, W):
    for i, xi in enumerate(X1):
        s = get_s([1, X1[i], X2[i]], W)
        y = get_y(s)
        print("x = ", X1[i], X2[i], "y = ", y)
```

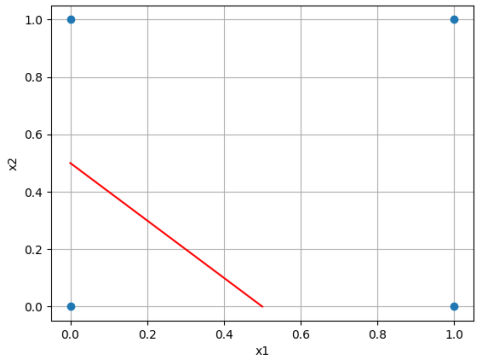
La Figure 2 montre les valeurs de w_i obtenues, bien que les résultats respectifs pour chaque combinaison d'entrées. Il est possible d'observer que cette sortie est égale au Table 1 pour les cas AND et OR.

```
----------------------------------------------------------------------
Found weights for OR gate: w0 = -0.9 , w1 = 1.0 , w2 = 1.0
x =  0 0 y =  False
x =  0 1 y =  True
x =  1 0 y =  True
x =  1 1 y =  True
----------------------------------------------------------------------
Found weights for AND gate: w0 = -1.0 , w1 = 0.1 , w2 = 1.0
x =  0 0 y =  False
x =  0 1 y =  False
x =  1 0 y =  False
x =  1 1 y =  True
```

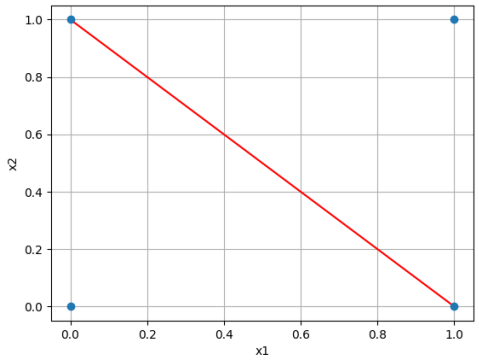
Figure 2: Sortie de la fonction find_weights() pour les portes OR et AND.

Ce qui a été observé expérimentalement, c'est que l'implémentation réalisée ne fonctionne pas pour le cas XOR et nous avons l'intention de démontrer pourquoi cela arrive à l'aide de graphiques 2D pour les courbes linéaires du perceptron dans les 2 premiers cas (OR et AND).

Comme expliqué par [1], le perceptron sépare simplement les données d'entrée en deux catégories. La première c'est l'ensemble des points qui provoquent ont une sortie unitaire, alors que la deuxième classe et composée par les points qui n'en provoquent pas (et par conséquence, la sortie est nulle). Sachant que le perceptron trace effectivement une ligne dans l'espace d'entrée bidimensionnel (caractérisée par s, ou, le threshold), et que les entrées d'un côté de la ligne sont classées dans une catégorie, tandis que celles de l'autre côté sont classées dans une autre catégorie, nous pouvons aussi trouver des valeurs w_i analytiquement. Par exemple, la Figure 3a montre un trace pour la logique OR, où $w_0 = 0$, $w_1 = 1$ et $w_2 = 1$, un seuil égal à 0.5 (ligne rouge). La Figure 3b présente la courbe pour la porte logique AND, dont le seuil unitaire définit prend en compte que le point où $x_1 = x_2 = 1$.



(a) Plan d'entrées pour la logique OR



(b) Plan d'entrées pour la logique AND

Figure 3: Plan d'entrées pour les perceptrons

Pour généraliser, considérant $w_0 = 0$, nous pouvons dire que pour la porte AND nous n'avons que besoin de $w_1 + w_2 \geq seuil$ et pour la porte OR $w_1 > seuil$ ou $w_2 > seuil$. Ce type de généralisation ne peut pas être appliquée à la logique XOR, parce que cette

dernière est une logique non-linéaire, et par conséquence ne peut pas être implémentée par une réseau de neurone d'une seule couche (perceptron). En d'autres termes, pour prendre en compte que les points (0,1) ou (0,1) à la fois, il fallait plus qu'un seul curbe lineaire sur le plan d'entrées.

3 Exercice d'application

Comme un exercice d'application du modèle développé lors de l'étape précédente, sur le simulateur **Webots**, nous implémentons les perceptrons dans notre robot Thymio. Dans cette partie du TP, nous utilisons deux capteurs de proximité situés à l'arrière pour détecter des obstacle de façon à pouvoir faire le robot avancé selon tel détection.

Pour cela, d'abord nous initialisons deux capteurs nécessaire, l'un pour la gauche et l'autre pour la droite, comme donné par la suite :

```
sensor_left_back = robot.getDevice('prox.horizontal.5')
sensor_right_back = robot.getDevice('prox.horizontal.6')
```

Après, dans la boucle principale, nous lisons ces capteurs. Ici, il est important de normalisé les valeurs lus pour avoir une entrée comprise entre 0 et 1, après on fait un cast pour un valeur entier (on aura forcement 0 ou 1 comme entrée, entrées de type tout-ou-rien).

```
x_b1 = int(sensor_left_back.getValue()/distance_max > 0.01)
x_b2 = int(sensor_right_back.getValue()/distance_max > 0.01)
X_b =[1, x_b1, x_b2]
```

Pour conclure, le vecteur d'entrées est utilisé pour la prise de décision, telle que décrites avant. On s'en sert de la logique AND et les moteurs sont actionnés si un obstacle est détecter, sinon ils sont arrêtés, comme montre le code suivant :

```
if flags["exercice_2"]:
    s = get_s(X_b, w_and)
    y = get_y(s)

    if flags["debug"]:
        print("X_b, s, y = ", X_b, s, y)

    if y == 1:
        forward(motor_left, motor_right, 4)
    else:
        stop(motor_left, motor_right)
```

4 Perceptron analogique

Dans cette partie, on souhaite explorer les concepts d'un percetron qui peut travailler avec des valeurs analogiques, c'est-à-dire, qui peut avoir des entrées et/ou sortie différentes de valeurs binaires, pouvant être des flottants.

Pour commencer, la Figure 4 montre un modèle simple de perceptron à une seule entrée. Nous voulons l'implémenter dans notre robot, où l'entrée sera la valeur lue par un

capteur de proximité situé devant le robot et la sortie sera utilisé pour réguler la vitesse de moteurs afin de reculer le robot si un objet est détecté en face à lui.

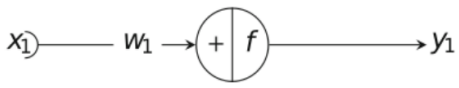


Figure 4: Perceptron analogique à une entrée.

La fonction d'activation que nous utilisons ici est un peu différénte de celle de la première partie du TP, elle est donne par la fonction Python `f_analogique()`. On pourrait aussi avoir utilisé la fonction tangente hyperbolique, mais nous avons opté par la simplicité de la logique suivante :

```
def f_analogique(x):
    y = x

    if x < -1:
        y = -1
    elif x > 1:
        y = 1

    return y
```

Dans notre code du controlleur, dans un premier moment, nous initialison le capteur qui se trouve devant le robot en position centralisé (prox.horizontal.2) et après, dans la boucle principale, nous le lisons comme donné par :

```
x_cf = sensor_center_front.getValue()/distance_max
```

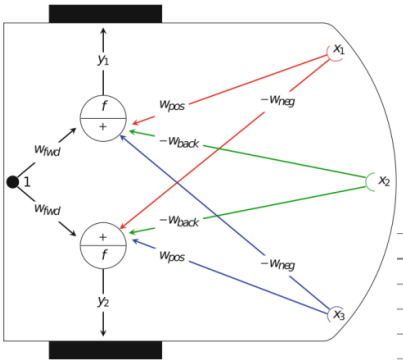
La valeur lue est appliqué dans la fonction d'activation présentée précédemment. Ici pour avoir une valeur de sortie proportionnelle, nous utilisons la vitesse maximale comme facteur lors de l'activation des moteurs par la fonction `backward` utilisée pour reculer le robot, comme indiqué par la suite :

```
elif flags["exercice_3_0_1"]:
    s = get_s([x_cf], [1])
    y = f_analogique(s)
    backward(motor_left, motor_right, y*speed_max)
```

5 Application des réseaux de neurones artificiels : Véhicule de Braitenberg

La Figure 5 représente le modèle de réseau de neurones artificiel (ANN) que nous allons implémenter. Chaque moteur de roue aura un neurone lui associé avec 4 entrées chacun. La première entrée est utilisée pour s'assurer que le robot avancera toujours en l'absence d'entraves, on la donne une valeur est unitaire. Les autres 3 entrées proviennent des capteurs x_1, x_2, et x_3, situés devant le robot, servent à identifier la présence des obstacles.

Les données acquises par les capteurs sont normalisées pour avoir une plage analogique comprise entre 0 et 1, et les poids w_{fwd}, w_{back}, w_{pos}, w_{neg} sont défini selon l'application cible. Dans la suite, nous allons discuter deux application différentes de ce réseau : l'évitement d'obstacles dans la section 5.1, et le suivi d'objet dans la section 5.2.



Symbole	Description
w_{fwd}	Poids associé au mouvement avancer
w_{back}	Poids associé au mouvement reculer
w_{pos}	Poids associé à une rotation positive de la roue
w_{neg}	Poids associé à une rotation negative de la roue

Figure 5: Modèle de Réseaux de Neurones Artificiel proposé.

5.1 Conception d'une stratégie d'évitement d'obstacles

Les relations 4, 5 et 6 ont été établies expérimentalement.

$$w_{fwd} < w_{back} \tag{4}$$

$$w_{neg} = w_{pos} \tag{5}$$

$$w_{back} < w_{pos} \implies w_{back} < w_{neg} \tag{6}$$

L'idée utilisé pour obtenir les résultat précédents est la suivante :

- Relation 4 : Forte réactivité au recul en présence d'obstacles au front du robot, réactivité légèrement réduite à l'avancement pour modérer la vitesse. Donc, le poids l'idée à l'avancer doit être plus petit.
- Relation 5 : Il s'agit de poids symétriques selon la figure 5. Sachant qu'ils servent à faire tourner le robot en cas d'obstacle sur ses diagonales, ils doivent être identiques pour ne pas rendre un côté plus prédominant que l'autre.
- Relation 6 : lorsque un capteur diagonal (soit celui de la gauche, ou de la droite), et le capteur du centre détectent un obstacle simultanément, le robot doit privilégier la rotation plutôt que le recul, car le recul peut souvent le faire tomber dans une boucle d'avance et de recul au même endroit, ce qui oblige, surtout dans le cas du problème du labyrinthe, à s'ouvrir à une nouvelle région lacunaire (une rotation à droite ou à gauche résoudrait ce cas).

On peut résumer les relations trouvées par la suite :

$$\boxed{w_{fwd} < w_{back} < w_{neg} = w_{pos}}$$

Les poids utilisé pour notre stratégie d'évitement d'obstacles sont donné par :

- $w_{fwd} = 0.7$
- $w_{back} = 0.9$
- $w_{pos} = 1.0$
- $w_{neg} = 1.0$

5.2 Conception d'une stratégie de suivi d'objet

À partir du travail développé lors de la section 5.1, nous allons définir des nouveaux poids pour au lieu d'un système d'évitement d'obstacles, avoir un robot capable de suivre un objet. Les caractéristiques souhaitées sont las suivantes, avec les relations de poids y liées :

- Le robot avance en absence d'objets :

$$w_{fwd} > 0 \tag{7}$$

- Si le capteur centrale détecte un objet très proche, il s'arrête :

$$w_{fwd} = w_{back} \tag{8}$$

- Si un objet est détecté par le capteur situé sur un côté (soit gauche, soit droite), le robot turne à direction pour assurer son suivi.

$$w_{pos} < 0 \text{ et } w_{neg} < 0 \tag{9}$$

- Et pour maintenir la symétrie susmentionnée, entre w_{pos} et w_{neg}, nous utilisons toujours l'équation 5.

Finalement, pour donner priorité à avancer toujours, si on le capteur de gauche et le capteur du centre détectent un obstacle simultanément, on suit celui du centre. Pour cela, une dernière relation est donné par :

$$|w_{fwd}| > |w_{pos}| \therefore |w_{fwd}| > |w_{neg}| \tag{10}$$

On peut résumer les relations trouvées par la suite :

$$\boxed{w_{neg} = w_{pos} < 0 < w_{fwd} = w_{back}}$$

Pour prouver la véracité de ces rapports, l'environnement de simulation a été utilisé avec deux robots, l'un contrôlé par le clavier et l'autre par le contrôleur développé dans ce TP. Les poids utilisé pour notre stratégie de suivi d'objet sont donné par :

- $w_{fwd} = 1.0$
- $w_{back} = 1.0$
- $w_{pos} = -0.5$
- $w_{neg} = -0.5$

6 Conclusions

Ce projet met en évidence la capacité des perceptrons à résoudre des problèmes complexes de robotique avec une approche simple mais efficace. En utilisant des réseaux de neurones à une seule couche, notre robot a pu acquérir des capacités de suivi d'objet et d'évitement d'obstacles qui améliorent considérablement son autonomie et sa capacité à interagir avec son environnement. Pour choisir entre les comportements souhaités du robot, ainsi que pour voir les résultats pratiques de chaque exercice, utilisez la structure ci-dessous qui est contenue dans le fichier **flags_file.py**. Il suffit de mettre l'élément désiré à True.

```
flags = {
    "debug": True,
    "graphics": False,
    "keyboard": False,
    "exercice_2": False,
    "exercice_3_0_1": False,
    "exercice_3_0_2": False,
    "exercice_3_1": False,
    "exercice_3_2": True
}
```

Malgré sa simplicité, cette architecture de réseau de neurones a démontré une adaptabilité surprenante, permettant au robot de généraliser efficacement à différents environnements et à diverses configurations d'objets et d'obstacles. De plus, la facilité de mise en œuvre et de formation des perceptrons à une seule couche en fait une solution attrayante pour des applications robotiques pratiques.

Ce projet souligne également l'importance de la simplicité et de l'efficacité dans la conception de systèmes robotiques autonomes. En privilégiant des approches minimalistes mais robustes comme celle-ci, nous sommes en mesure de créer des robots intelligents capables de fonctionner de manière fiable dans le monde réel, tout en minimisant la complexité et les coûts de développement.

En poursuivant cette voie de recherche, nous ouvrons la porte à de nouvelles avancées dans le domaine de la robotique, où des solutions élégantes et efficaces basées sur des perceptrons à une seule couche pourraient être déployées dans une multitude d'applications pratiques, de la logistique à l'assistance domestique.

References

[1] Dr. Mark Humphrys. *Single-layer Neural Networks (Perceptrons)*. 1987. URL: https://humphryscomputing.com/Notes/Neural/single.neural.html (visited on 02/20/2024).