TP6 - IA pour la Robotique

Master SETI

Alaf DO NASCIMENTO SANTOS
Simon BERTHOUMIEUX

2023/2024

Contents

1 Introduction

Ce travail est réalisé dans le cadre du module **IA pour la Robotique**, l'objectif principal en est de commencer à étudier l'utilisation des topologies multicouches pour les réseaux de neurones appliqués à la commande d'un robot à mouvement différentiel, de modèle Thymio, simulé avec le logiciel **Webots** version 2021b et **Python** 3.9v. Voici les objectifs spécifiques visés par ce travail pratique :

- Étudier et faire l'implémentation de topologies multicouches;
- Application des réseaux récurrents au robot Thymio sur le simulateur;
- Voir en pratique le filtre spatial dans la navigation du robot.

Cinq fichiers Python ont été créés pour le projet :

- **tp6.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.
  - **multi_layer.py** : les principales fonctions concernant les exercices de multi-couche et leurs solutions.

2 Les topologies multicouches

Dans notre travail précèdent, nous avons étudié les réseaux de neurones composés d'une seul couche. Les résultats obtenus étaient assez intéressant, si nous considérons que la topologie utilisée était la plus simple possible. Lors du présent travail, nous visons à augmenter la complexité des algorithmes, c'est-à-dire, nous allons élargir la topologie du réseau.

Une seule couche de neurones (pas de couches cachées) ne peut produire que des relations monotones entre les entrées et les sorties. Cela veut dire que si on donne des données à notre réseau, celui-ci va toujours produire des résultats qui suivent la même tendance. Par exemple, si on lui donne des informations d'un capteur, celui-ci pourra juste faire croître ou décroître la sortie en fonction de cette entrée. Mais il ne pourra pas produire ces deux tendances pour cette entrée. C'est là qu'intervient la topologie multicouche. Celle-ci se compose de plusieurs couches de neurones, qui permettent des relations entrées-sorties plus complexes et variés (notamment non monotones).

Pour voir comment fonctionne un réseau de neurones multicouche, on peut le comparer à un réseau monocouche. On va utiliser un exemple avec des entrées et des poids. Les entrées varient entre -2.0 et 2.0, et les poids entre -1.0 et 1.0. On va aussi utiliser une fonction spéciale pour décider comment les neurones réagissent, elle limite les valeurs qu'ils peuvent donner entre -1.0 et 1.0.

D'abord on considère le modèle simple montré par la Figure 1. Si on prend $w_1 = -0.5$, pour un vecteur d'entrée $x_1 = [-2.0, -1.8, ..., 0.0, ..., 1.8, 2.0]$, on obtient la Figure 2 comme résultat d'estimation de sortie y_1 en fonction de x_1.

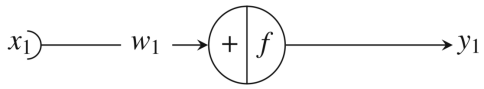


Figure 1: Modèle du neurone.

Le résultat obtenu dans la Figure 2 montre en pratique que le comportement d'un réseau monocouche est forcément monotonique. Nous n'avons qu'une courbe qui ne fais que descendre pour l'intervale d'entrées donné.

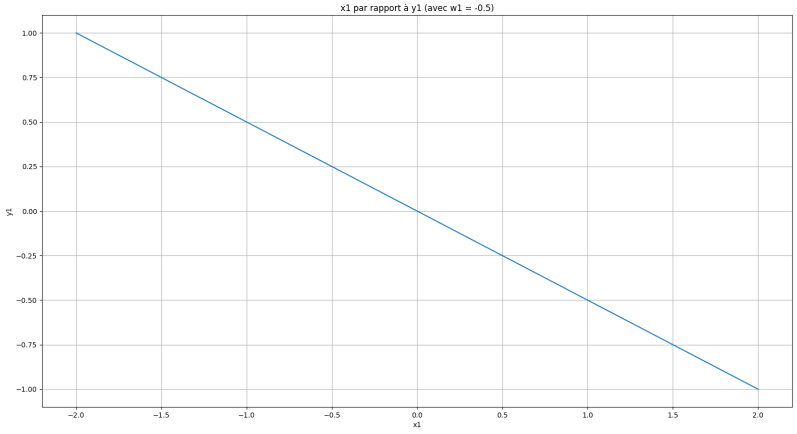


Figure 2: Résultat du modèle simple du neurone.

Pour arriver au résultat présenté, nous avons utilisé la fonction d'activation de type saturation limitant les valeurs en sortie des neurones à l'intervalle $[-1.0, 1.0]$. Sa mise en œuvre est donnée par la suite :

```
def f_activation_sat(x, w):
    s_vec = np.dot(x,w)
    y_vec = []
    for s in s_vec:
        if s < -1:
            y = -1
        elif s > 1:
            y = 1
```

```
        else:
            y = float(s[0])
        y_vec.append(y)

    return y_vec
```

Après, pour le tracé de la courbe, nous avons développé la fonction suivante, qui prend en compte, les valeurs d'entrées, de sorties, leurs labels, en plus d'un titre pour le graphique. Le paramètre "plot" sera plutôt utile dans l'exercice suivant, lors du tracé de plusieurs courbes sur le même graph.

```
def plot_y(x, y, lx, ly, titre, plot=True):
    if plot:
        plt.plot(x, np.reshape(y,21))
    plt.grid()
    plt.xlabel(lx)
    plt.ylabel(ly)
    plt.title(titre)
    if not plot:
        plt.legend()
    plt.show()
```

Avant d'appeler les fonctions présentées, nous devons définir le vecteur d'entrées. Le morceau de code suivant montre la création du vecteur x_1 composé par des valeurs entre -2.0 et 2.0, avec un facteur de incrément égale à 0.2.

```
step = .2
x1 = np.arange(-2.0, 2.0 + step, step)
```

Finalement, nous faisons appel au neurone développé, pour ensuite générer le graphique "y1 par rapport a x1 (avec w1 = -0.5)".

```
if flags["exercice_1"]:
    w1 = [-0.5]
    y1 = f_activation_sat(np.matrix(x1).T, np.matrix(w1))
    titre = "y1 par rapport a x1 (avec w1 = " + str(w1[0]) + ")"
    plot_y(x1, y1, 'x1', 'y1', titre)
```

Pour la suite du travail pratique, on effectue plusieurs tracés similaires à celui de la partie précédente, sauf que maintenant on change les valeurs de w_1. En considérant 15 valeurs de poids w_1 différents, entre -1.0 et 1.0, on obtient la Figure 3. Cette image est une représentation parfaite du fait qu'un réseau de topologie simple, à une seule couche du type feed-forward, peut établir une classification linéaire des informations. Nous pouvons constater que nous avons systèmatiquement un comportement monotonique entre les entrées et les sorties. La zone où nous avons un comportement linéaire pourrait nous permettre de retrouver les poids, même si nous pouvons ici utiliser la légende. En affichant le comportement de poids opposés, la figure est exactement symétrique par rapport aux axes x et y.

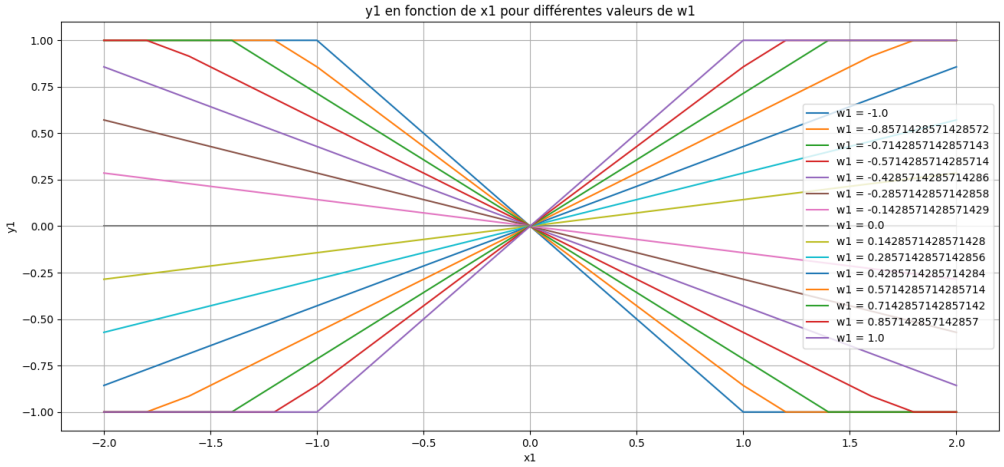


Figure 3: Résultat du modèle simple du neurone avec 15 différentes valeurs de w_1.

Pour faire cette évaluation de plusieurs poids, nous avons développé le morceau de code suivant. D'où une boucle for est utilisée pour changer les valeurs de w_1 et appeler le plot sans faire appel à la fonction `show` de matplotlib, laquelle est appelée lors du appel à `plot_y` de la dernière ligne.

```
if flags["exercice_2"]:
    for i in range(15):
        w1_val = -1.0 + i * (2.0 / 14)
        w1 = [w1_val]
        y1 = f_activation_sat(np.matrix(x1).T, np.matrix(w1))
        plt.plot(x1, np.reshape(y1,21), label=f'w1 = {w1_val}')

    titre = 'y1 en fonction de x1 pour differentes valeurs de w1'
    plot_y(x1, y1, 'x1', 'y1', titre, False)
```

Maintenant, nous nous intéressons à la mise en ouvre du réseau multicouche présenté dans la Figure 4. L'idée ici est de comparer les résultats précédents avec celui d'un réseau plus complexe, contenant des couches cachées.

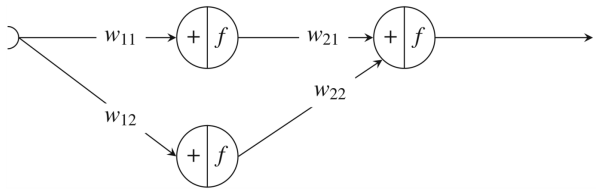


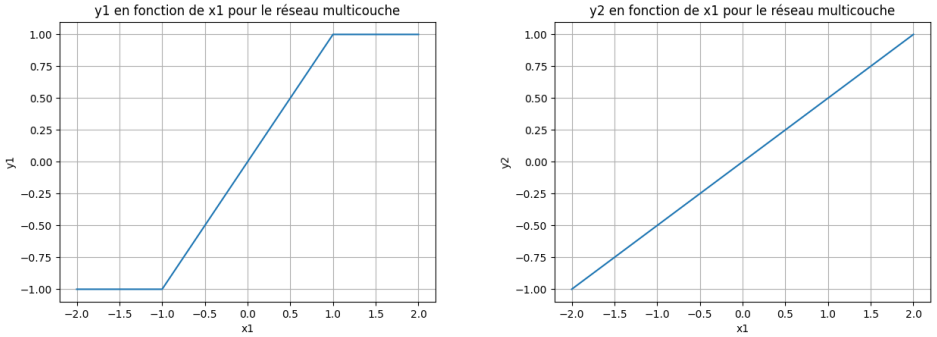
Figure 4: Modèle du réseau multicouche.

Pour la partie code de ce réseau, nous avons fixés les poids suivants :

- $w_{11} = 1$
- $w_{12} = 0.5$

- $w_{21} = 1$
- $w_{22} = -1$

La Figure 5 présente les résultats intermédiaires des couches cachées de notre réseau. En 5a il est possible de vérifier que la sortie y_1 saturera pour $|x_1| > 1.0$, cette sortie, y_1, est celle qui sera liée au poids w_{11} du neurone de couche cachée supérieur de la Figure 4. Pour y_2, sortie du neurone inférieur de la couche cachée de la Figure 4, nous ne verrons pas de saturation sur $[-2; 2]$ (même si nous saturerons pour $\mathbf{R}/[-2; 2]$), ce qui correspond bien au poids w_{12}.



(a) Résultat de la couche cachée y_1. (b) Résultat de la couche cachée y_2.

Figure 5: Résultat du modèle multicouche - couches cachées.

Les deux courbes intermédiaires, pour nos couches cachées sont bien monotonique comme attendu. Leur jonction dans le neurone de la couche de sortie nous donne, après l'application des poids w_{21} et w_{22}, la Figure 6. Ce dernier résultat montre qu'en combinant le comportements de neurones monotinques, nous pouvons avoir des comportements plus complexes que celui d'un simple classifieur linéaire, plus précisément on sera capable d'approcher n'importe quelle fonction, pourvu qu'on ai assez de neurones et que ceux-ci soient assez précis.

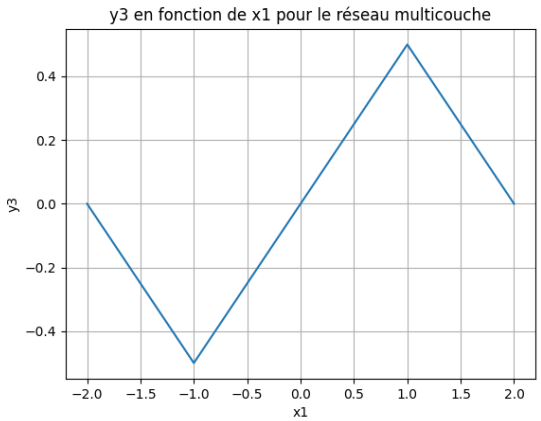


Figure 6: Résultat du modèle multicouche - couche de sortie y_3.

Le code qui génère effectivement le résultat présenté est montré ci-dessous :

```
if flags["exercice_3"]:
    w11 = 1.0
    w12 = 0.5
    w21 = 1.0
    w22 = -1.0

    w1 = [w11]
    w2 = [w12]
    w3 = [w21, w22]

    y1 = f_activation_sat(np.matrix(x1).T, np.matrix(w1))
    y2 = f_activation_sat(np.matrix(x1).T, np.matrix(w2))

    y3 = f_activation_sat(np.matrix([y1,y2]).T, np.matrix(w3).T)

    plot_y(x1, y1, "x1", "y1", "y1 en fonction de x1 pour le reseau multicouche")
    plot_y(x1, y2, "x1", "y2", "y2 en fonction de x1 pour le reseau multicouche")
    plot_y(x1, y3, "x1", "y3", "y3 en fonction de x1 pour le reseau multicouche")
```

2.1 Exercice d'application robotique

Maintenant, comme un exercice d'application robotique des concepts explorés précédemment, nous allons mettre en ouvre le réseau de la Figure 7. Il s'agit d'un schéma de modèle de réseau pouvant servir à l'évitement d'obstacles. L'idée ici est la d'utiliser les mesures issues des deux capteurs situés devant le robot et d'avvoir le comportement suivant :

- si un objet est détecté par l'un des 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.

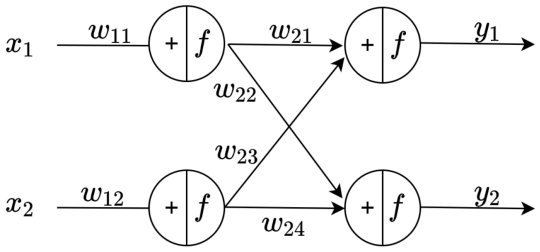


Figure 7: Modèle du réseau pouvant servir à l'évitement d'obstacles.

Pour cet algorithme, nous avons utilisé les poids suivants :

- $w_{11} = 1.0$

- $w_{12} = 1.0$
- $w_{21} = 1.0$
- $w_{22} = -2.0$
- $w_{23} = -2.0$
- $w_{24} = 1.0$

Ce choix est motivé par plusieurs contraintes qui traduisent le cahier des charges :

- $x_1 = x_2 > 0 \iff y_1 = y_2 < 0$
- $x_1 > 0; x_2 = 0 \iff y_1 > 0 y_2 < 0$
- $x_1 = 0; x_2 > 0 \iff y_1 < 0 y_2 > 0$

Ces conditions symétriques vis à vis des entrée et sorties nous imposeront la nature chiral du réseau, même si nous choisissions de partir sur une autre architecture pour les respecter.
On normalisera les entrées et sorties pour faciliter le traitement.
Pour simplifier notre traitement, nous décidons de choisir une rotation qui ne se fait pas sur place, mais avec un léger rayon de braquage dans le cas non saturé.
Le réseau choisit permettra d'avoir une rotation avec recul quand les deux capteurs détectent un obstacle, mais non avec la même intensité.

```
if flags["exercice_4"]:
    s1 = f_activation_sat(np.matrix(x_lf).T, np.matrix(w11))
    s2 = f_activation_sat(np.matrix(x_rf).T, np.matrix(w12))

    w1 = [w21, w23]
    w2 = [w22, w24]

    y1 = f_activation_sat(np.matrix([s1,s2]).T, np.matrix(w1).T)
    y2 = f_activation_sat(np.matrix([s1,s2]).T, np.matrix(w2).T)

    motor_left.setVelocity(y1[0]*speed_max)
    motor_right.setVelocity(y2[0]*speed_max)
```

3 La mémoire : les réseaux récurrents

On commence dans un premier temps par analyser le comportement d'un unique neurone récurent.
On pourra considérer qu'on a les poids d'entrée et de sortie du neurone à 1, et, pour t le nombre de passage dans le neurone, on aura comme entrée du neurone $x = 0.5 \times \delta_t$, où δ_t le symbole de Kronecker (ou Dirac discret).
On étudie le comportement de ce neurone sur les dix premier passages dans le neurone, d'après l'évolution du poids de la liaison récurrente w_{rec}. On obtient :

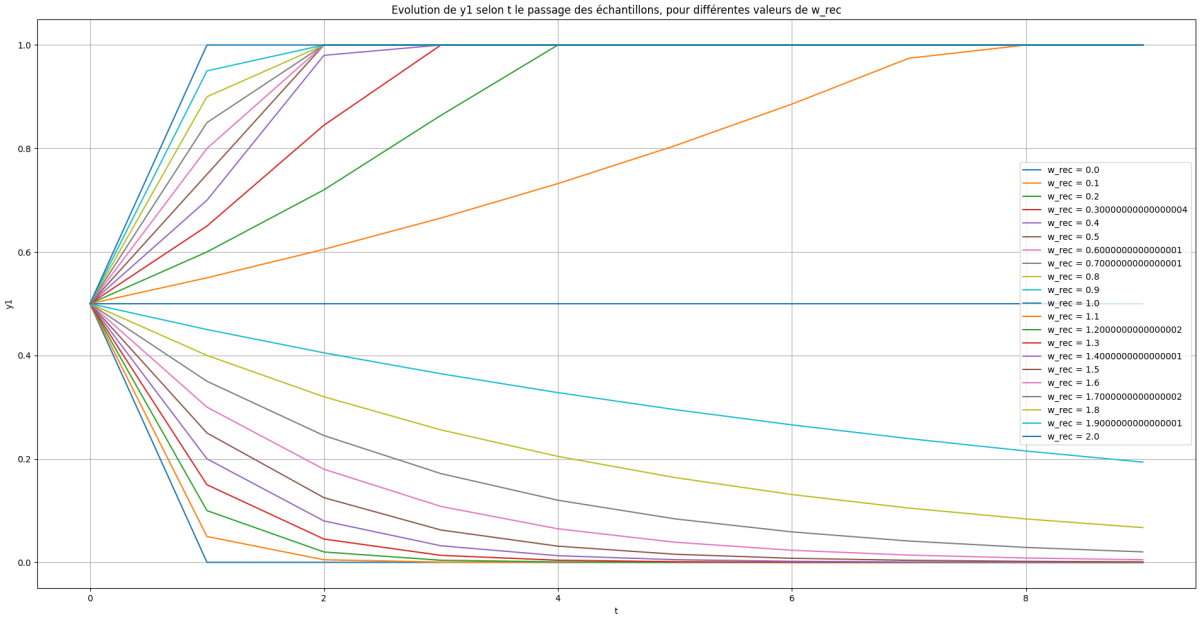


Figure 8: Résultats sur la sortie y pour les dix premiers passage dans un neurone récurent, d'après la valeur de w_{rec}.

On observe que quand $w_{rec} < 1$ on retourne vers la valeur actuelle de entrée (qui est à 0 pour tout les instants hors $t = 0$), de façon plus ou moins rapide selon notre proxité avec 1.
Quand $w_{rec} = 1$, on ne bouge plus de la valeur initiale de x. En vérité on fera la somme de toute les valeurs de x aux instant $< t$. Ce qui correspondra au comportement d'un intégrateur discret sur les entrées.
Quand $w_{rec} > 1$ on convergera vers la saturation, plus ou moins rapidement selon notre proximité avec 1.
De plus, si nous avions $x_t = x$ constant, nous aurions :

$$y_t = x_t + y_{t-1} = \sum_{k=0}^{t} (w_{rec}x)^k = \frac{1 - (w_{rec}x)^{t-1}}{1 - w_{rec}x} \tag{1}$$

Nous observons globalement trois comportements lors d'un passage à travers le neurone récurent :

- $w_{rec} < 1$: Atténuation de l'information qu'on avait précédemment;
- $w_{rec} = 1$: Conservation de l'information qu'on avait précédemment (comportement d'intégrateur);
- $w_{rec} > 1$: Renforcement de l'information qu'on avait précédemment.

3.1 Exercice d'application robotique

Après avoir établi notre modèle de comportement pour un neurone récurent, on essaye d'utiliser celui-ci pour améliorer notre modèle d'évitement obtenu au TP précédent

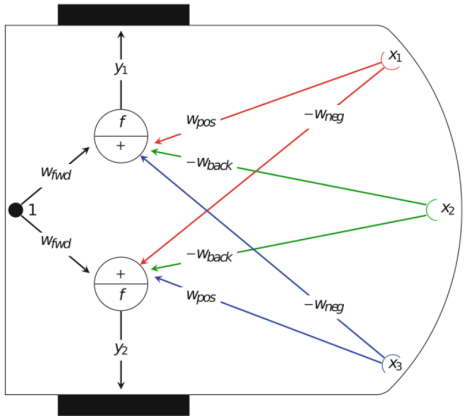


Figure 9: Modèle de réseau récurrent pour l'évitement d'obstacles.

Nous transformons les deux neurones de notre circuit en neurones récurrents, cela nous permettra d'avoir une mémoire sur les dernières sorties et de continuer d'esquiver l'obstacle, même après avoir cessé de le détecter. Nous ne souhaitons ni un effet intégrateur, et il ne serait pas bon d'avoir une sortie qui se renforce, on prendra donc $w_{rec} < 1$, plus précisément, nous choisirons d'avoir $w_{rec} = 0.1$ après avoir constaté que le comportement lié à la mémoire des états précédents est trop important pour $w_rec = 0.5$.
On constate que le réseau avec neurone récurent va plus vite que le réseau sans neurone récurent, cela est compréhensible, car la liaison récurrente amplifie l'effet des biais, comme le montre (1).

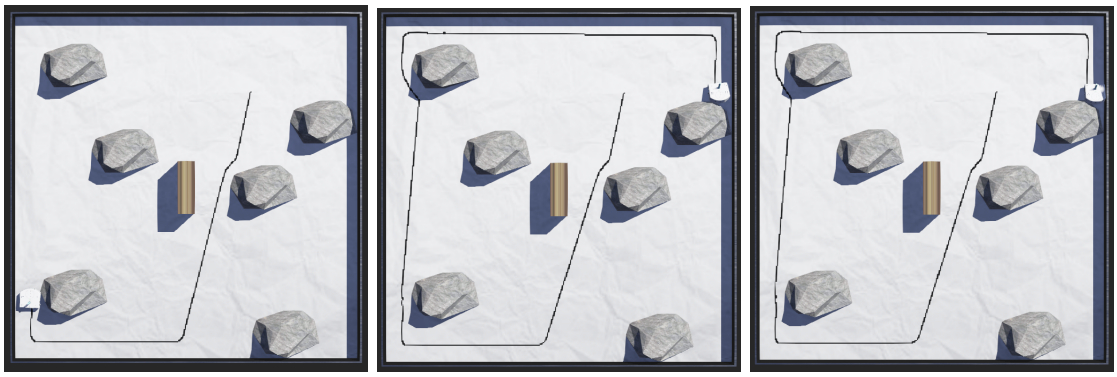


Figure 10: Trajectoires obtenues avec (gauche ($w_{rec} = 0.25$) + centre($w_{rec} = 0.1$)) et sans (droite($w_{rec} = 0.1$)) neurones récurrents.

On constate qu'on ne parvient pas vraiment à exploiter l'intérêt des neurones récurrents dans notre application, même si on peut voir que le fait d'avoir un neurone récurent amplifie bien la trajectoire et accélère bien la vitesse.

4 Le filtre spatial

4.1 Exercice d'application robotique

On implémente le filtre spatiale demandé, en mettant en entrée les capteurs avant du robot.

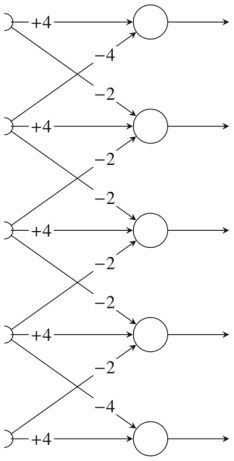


Figure 11: Filtre spatial.

Si nous avions des capteurs alignés, nous aurions [00000] en sortie lorsque l'on se trouve face à un mur, et $[0, -1, 1, -1, 0]$ lorsqu'on détecte un objet juste en face du robot. Le paterne $[-1, 1, -1]$ pourrait être décalé vers la gauche ou la droite si ce n'était pas le capteur central qui détectait l'objet.
Cependant, nous n'avons pas des capteurs alignés, mais des capteurs en arc-de-cercle. La détection d'un objecta correspondra toujours aux cas : $[1, -1, 0, 0, 0]$, $[-1, 1, -1, 0, 0]$, $[0, -1, 1, -1, 0]$, $[0, 0, -1, 1, -1]$, $[0, 0, 0, -1, 1]$. Mais la détection d'un mur correspondra désormais à l'obtention de $[-1, x_i, x_j, x_i, -1]$ en sortie du filtre, avec $x_i \in [0, 1]$ et $x_j \in [-1, 1]$. Un contact avec le mur correspondra a $[-1, 1, -1, 1, -1]$.
Nous pourrions retrouver les sorties du cas où nous avons des capteurs alignés en modifiant les poids des bords de notre filtre, mais nous choisirons de conserver les poids originaux.

On cherche désormais à utiliser ce filtre spatial pour construire une commande pour notre robot vérifiant le cahier des charges suivant :

- Si et seulement si un capteur détecte un objet, alors le robot se tourne et s'oriente vers l'objet;
- Si le capteur central détecte un objet, le robot avance droit;
- Si l'ensemble de capteurs détectent simultanément un objet, alors le robot s'arrête ou recule.

Nous utiliserons l'architecture suivante pour traiter ce cahier des charges :

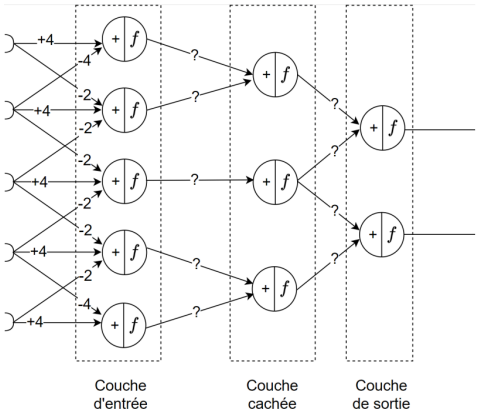


Figure 12: Modèle du réseau pour l'identification du type d'objet.

Pour le filtre utilisé, nous aurons les contraintes suivantes entre la sortie du filtre et la sortie du réseau :

- Quand on aura $[1, -1, 0, 0, 0]$ ou $[0, 0, 0, -1, 1]$ en sortie du filtre, on souhaitera tourner vers la gauche ou la droite respectivement ;
- Quand on aura $[-1, 1, -1, 0, 0]$ ou $[0, 0, -1, 1, -1]$ on souhaitera tourner vers la gauche ou la droite respectivement ;
- Quand on aura $[0, -1, 1, -1, 0]$ on souhaitera continuer tout droit ;
- Quand on aura $[-1, 0.5, 0.5, 0.5, -1]$ on souhaitera s'arrêter ou reculer.

On ignorera la première contrainte, car dans l'architecture choisie avoir $[1, -1, 0, 0, 0]$ ou $[0, 0, 0, -1, 1]$ ne permet de commander qu'une seule roue, et avoir une roue immobile et l'autre roue essayant de déplacer le robot ne nous permettra pas de façon satisfaisante de déplacer celui-ci.
On choisit les poids suivants :

- $w_{15} = -0.9$: Les poids entre les neurones 1 et 5 de la couche d'entrée, et les neurones 1 et 3 de la couche cachée;
- $w_{24} = -1.1$: Les poids entre les neurones 2 et 4 de la couche d'entrée, et les neurones 1 et 3 de la couche cachée;
- $w_3 = -1.0$: Les poids entre le neurone 3 de la couche d'entrée, et le neurone 2 de la couche cachée;
- $w_{ext} = 3.0$: Les poids entre les neurones 1 et 3 de la couche cachée, et les neurones de la couche de sortie;
- $w_{int} = 1.0$: Les poids entre le neurone 2 de la couche cachée, et les neurones de la couche de sortie.

Le choix de poids symétrique est justifié par la nature symétrique du problème et du circuit.
Le choix des poids w_{24}, w_3, w_{ext}, w_{int} vérifie bien la contrainte qui veut qu'on avance (que les deux sorties soient positives) quand on a $[0, -1, 1, -1, 0]$ en sortie de la couche d'entrée.
Le choix de l'ensemble des poids vérifie bien la contrainte qui veut qu'on tourne (qu'on ai une sortie positive et une négative) quand on a $[-1, 1, -1, 0, 0]$ ou $[0, 0, -1, 1, -1]$ en sortie de la couche d'entrée. De plus, on aura toujours la roue opposée au capteur détectant un objet qui tournera dans le sens positif et l'autre roue dans le sens négatif.
La contrainte sur l'arrêt quand on a $[-1, 0.5, 0.5, 0.5, -1]$ est imposée de façon relativement souple, car on souhaitera éviter entrer en contact avec l'objet (situation qui correspondrait à $[-1, 1, -1, 1, -1]$), mais on souhaitera aussi ne pas se retrouver trop loin de l'objet à traquer. Le point d'arrêt choisit pour (x_i, x_j) dans $[-1, x_i, x_j, x_i, -1]$ sera fixé par le rapport entre les poids w_{15} et w_{23} et par le rapport entre w_{ext} et w_{int}. On choisira w_{15} et w_{23} très proches, et w_{ext} assez supérieur à w_{int} pour suivre l'objet de près. Le fait d'avoir w_{ext} assez supérieur à w_{int}, c'est à dire la contributions des poids externes du filtre supérieur à celle du poids central, fera qu'on s'alignera à une vitesse plus élevée avec le robot.

Maintenant que la complexité du circuit à relativement augmenté par rapport aux circuits précédents, le réglage des différents poids du circuit se fait de plus en plus empiriquement.

On rencontrera un problème qui nous semble inévitable avec l'architecture choisit qui est que l'intensité des capteurs est inversement proportionnelle à la distance, alors qu'on souhaiterait idéalement obtenir une vitesse proportionnelle à distance (plus on est loin, plus on va vite), mais l'approximation de la fonction inverse nous semble difficilement réalisable ici, avec un circuit si simple, en l'absence de biais. Nous rencontrerons donc parfois un problème lors de certains longs tournants, où notre robot sera légèrement trop lent à se réajuster et augmentera donc sa distance avec le robot cible, qui se déplace à vitesse constante, ce qui lui fera perdre en vitesse l'amplitude des entrées ayant diminué, et ce qui lui causera de progressivement perdre le robot suivi jusqu'à ce que celui-ci soit hors de portée de capteur.

5 Conclusions

Nous avons vu, dans ce TP, que d'autres structures que le perceptron étaient possibles pour les neurones constituant un réseau de neurones.
Plus particulièrement, nous avons vu qu'il était possible d'introduire des neurones récurent dans notre système. Ces neurones ont parmi leurs entrées la valeur précédente de leur sortie. Ce type de neurones permet d'ajouter au circuit une mémoire des entrées, et a notamment été utilisé pour le traitement du langage. Dans le cas de la robotique, il pourra notamment nous permettre de garder une information sur un obstacle, afin de continuer à éviter celui-ci, même quand celui-ci ne sera plus détecté par les capteurs.

Nous avons aussi explorer des architectures plus complexes à l'aide d'un réseau permettant de faire du filtrage spatial. Ce genre de topologie de réseau permet de détecter certains paternes dans les entrées. Nous avons pu, en travaillant avec ce filtre, conclure qu'il est important de prendre en compte des considérations géométriques sur les entrée du système pour construire notre réseau.
Nous avons aussi vu, en élaborant sur ce réseau pour construire un contrôleur pour le suivie d'objet par notre robot qu'il devenait difficile de placer les poids manuellement au delà d'un certain niveau de complexité.

Les vidéos démonstratives sont disponibles sur les liens :

- exercice 4 : `https://youtu.be/OPnF7pEJzHI`
- exercice 6 partie 1 (avec connexions récurrents): `https://youtu.be/95GW8hfjeos`
- exercice 6 partie 2 (sans connexions récurrents): `https://youtu.be/nmNHgL-wmdg`

```
flags = {
    "debug": True,
    "graphics": False,
    "keyboard": False,
    "exercice_1": False,
    "exercice_2": False,
    "exercice_3": True,
    "exercice_4": False,
    "exercice_5": False,
    "exercice_6": False,
    "exercice_7": False,
    "exercice_8": False,
}
```