TP7 - 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 d'explorer l'apprentissage supervisé pour les réseaux de neurones artificiels 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 les méthodes d'apprentissage;
- Implémenter des perceptrons à travers différentes méthodes;
- Explorer la bibliothèque Python **sklearn**.

Cinq fichiers Python ont été créés pour le projet :

- **tp7.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 multicouche et leurs solutions.

2 Méthodes d'apprentissage

Nous parlons de différentes manières d'apprentissage dans la littérature. Il y a trois principales façons : l'apprentissage supervisé, l'apprentissage par renforcement et l'apprentissage non supervisé. Les exercices pratiques que nous proposons se concentrent sur l'apprentissage supervisé. Nous utilisons des techniques qui aident à ajuster automatiquement les paramètres d'un réseau de neurones, comme les poids et les biais, en se basant sur les données d'entrée et les résultats attendus. Le but de cet apprentissage est de réduire les erreurs du réseau par rapport aux valeurs souhaitées. L'apprentissage supervisé peut avoir différents objectifs, comme la régression, la classification, le marquage (qui est une sorte de classification à plusieurs étapes), la recherche et le classement. Dans ce texte, nous nous concentrerons sur la régression et la classification.

Pour la mise en œuvre, nous utiliserons une bibliothèque Python appelée scikit-learn. Cette bibliothèque comprend des méthodes d'apprentissage, des objets tels que les perceptrons et les perceptrons multicouches, ainsi que des méthodes d'évaluation. Pour commencer, il est nécessaire d'installer cette bibliothèque à la commande suivante :

```
$ pip install sklearn
```

3 Le perceptron

3.1 Exercices de classification

Tout d'abord, nous allons entraîner un perceptron, similaire à celui qui nous avons fait manuellement lors du travail pratique précèdent (tp5). Nous cherchons à comparer les résultats obtenu au préalable avec ceux du apprentissage supervisé. On aura un perceptron dédié à la logique AND, un pour la logique OR, et un troisième pour la logique XOR. Pour ce dernier, nous avions déjà constaté lors du tp précèdent, qu'il ne pouvait pas être réalisé par un simple perceptron, mais nous voulons voir si ce résultat reste le même avec le perceptron obtenu par apprentissage supervisé.

On importe la définition du perceptron de la bibliothèque **scikit-learn** dans notre script à travers :

```
from sklearn.linear_model import Perceptron
```

Ensuite, on définie toutes les valeurs possibles pour les entrées de l'opérateur logique à deux input x_i comme dans le tp précèdent. On ajoute à notre script aussi les labels en sortie attendus selon l'opérateur concerné. Le morceau de code suivant montre cette étape :

```
#liste d'entrees possibles pour l'operateur logique
data = [[0,0], [0,1], [1,0], [1,1]]

#labels correspondants selon liste d'entrees
ORlabels = [0, 1, 1, 1]
ANDlabels = [0, 0, 0, 1]
XORlabels = [0, 1, 1, 0]
```

Après, étant donné que nous allons coder les trois logiques, on définie la fonction `get_perceptron_data()` pour nous montré les informations d'intérêt d'un perceptron par rapport à les labels qui peuvent être lui donné comme sortie attendue. Le biais est donné sur le console de sortie par W_0, les poids W_i par $Weights$, les prédictions par $Prediction$ et finalement le score qui estime la précision moyenne du perceptron après l'entraînement est montré comme $Score$.

```
def get_perceptron_data(operator, labels, MLP=False):
    print("W0: ", operator.intercept_)
    print("Weights: ", operator.coef_)
    print("Predictions: ", operator.predict(data))
    print("Score: ", operator.score(data, labels))
```

Finalement, nous créons une instance du perceptron pour chaque opérateur logique. Ici on utilise un nombre maximum d'itérations admis pour l'apprentissage égale à 40 et un seuil de d'erreur toléré (paramètres de convergence) de 1^{-3}. À la fin, nous appelons la fonction `get_perceptron_data` pour voir les paramètres visés.

```
if flags["exercice_1_2_3"]:
    #Creation du perceptron
    operateurOR = Perceptron(max_iter=40, tol=1e-3)
    operateurAND = Perceptron(max_iter=40, tol=1e-3)
    operateurXOR = Perceptron(max_iter=40, tol=1e-3)
```

```
6
7     operateurOR.fit(data, ORlabels)
8     operateurAND.fit(data, ANDlabels)
9     operateurXOR.fit(data, XORlabels)
10
11    print("\n-------------------------------------------\n\t\t OR
      operator")
12    get_perceptron_data(operateurOR, ORlabels)
13
14    print("\n-------------------------------------------\n\t\t AND
      operator")
15    get_perceptron_data(operateurAND, ANDlabels)
16
17    print("\n-------------------------------------------\n\t\t XOR
      operator")
18    get_perceptron_data(operateurXOR, XORlabels)
```

La sortie sur le terminal est donné la Figure 1. Pour les operateurs OR et AND toutes les prédictions sont bonnes et par conséquence le score est 1,0 (100%). Par contre, pour l'opérateur XOR, comme lors du tp5, nous n'avons que des bonnes prédictions, ce qui s'explique tout simplement par le fait que l'on essaye toujours de traiter un problème non linéaire avec un perceptron par définition monotonique qui ne pourra donner qu'une classification linéaire. Le score, dans l'intervalle $[0.0, 1.0]$, correspond ici à la proportion de classifications correctes obtenue, même si cela pourra pourra ne pas être le cas, particulièrement dans le cas où nous utiliserions un régresseur. Dans le cas XOR, nous avons un score de 0.5 car le perceptron nous donnera toujours 0, et par conséquence la moitié des cas testés seront correcte.

```
-------------------------------------------------
                OR operator
W0:  [-1.]
Weights:  [[2. 2.]]
Prediction:  [0 1 1 1]
Score:  1.0

-------------------------------------------------
                AND operator
W0:  [-2.]
Weights:  [[2. 2.]]
Prediction:  [0 0 0 1]
Score:  1.0

-------------------------------------------------
                XOR operator
W0:  [0.]
Weights:  [[0. 0.]]
Prediction:  [0 0 0 0]
Score:  0.5
```

Figure 1: Exercices de classification pour les perceptrons simples.

Il faut noter que les solutions données ici ne sont pas nécessairement uniques, et qu'il est possible pour le perceptron de converger vers une autre solution en 40 itérations, et

celle-ci pourraient potentiellement être sub-optimale.

3.2 Exercice d'OU-Exclusif comme un classifieur

À cause du résultat précèdent, nous allons utiliser une topologie de perceptron multi-couche à travers le MLPClassifier de la bibliothèque **scikit-learn**. Dans notre script Python, il faudra donc faire l'import suivant :

```
from sklearn.neural_network import MLPClassifier
```

On fait quelques adaptations dans notre fonction `get_perceptron_data` pour pouvoir considérer aussi le cas multi-couche par intermédiaire du paramètre `MLP`. Le morceau de code présenté ci-dessus illustre ce modifications :

```
def get_perceptron_data(operator, labels, MLP=False):
    if MLP:
        print("W0: ", operateurXOR.intercepts_)
        print("Weights: ", operator.coefs_)
    else:
        print("W0: ", operator.intercept_)
        print("Weights: ", operator.coef_)
    print("Predictions: ", operator.predict(data))
    print("Score: ", operator.score(data, labels))
```

Ensuite nous voulons créer et entraîner notre opérateur XOR multi-couche. Pour cela, nous définissons son architecture, en lui donnant une fonction d'activation tanh et une couche cachée à deux neurones, et nous décidons de lui laisser 10000 itérations pour converger. Finalement, on exécute la procédure d'apprentissage à travers la fonction **fit**, comme montré par la suite :

```
operateurXOR =MLPClassifier(hidden_layer_sizes=(2,),activation="tanh",
   max_iter=10000)
operateurXOR.fit(data, XORlabels)
```

Nous avons vérifié expérimentalement que, pour plusieurs essais, les poids et les biais n'étaient jamais les mêmes, et donc les prédictions et les scores changeaient parfois. Ainsi, pour nous assurer de toujours avoir le même score (idéalement un score unitaire), nous avons ajouté une bouclé for dans le code. Le morceau de code final dédié à cette étape est donné par :

```
elif flags["exercice_4_5_6"] :
    operateurXOR =MLPClassifier(hidden_layer_sizes=(2,),activation="tanh
   ",max_iter=10000)
    operateurXOR.fit(data, XORlabels)

    while operateurXOR.score(data, XORlabels) != 1:
        operateurXOR.fit(data, XORlabels)

    print("\n-------------------------------------------------\n\t\t XOR
   operator\n")
    get_perceptron_data(operateurXOR, XORlabels, MLP=True)
```

La Figure 2 montré un résultat possible pour avoir un score égale à 1. Par contre, celui-ci n'est pas unique, même en utilisant le même exact code une deuxième fois, il est

fort probable que les poids et biais obtenus soient différents. Cela est du à la nature de l'algorithme employé, où les poids initiaux et l'ordre d'entraînement sont choisis au hasard.

```
------------------------------------------------
                XOR operator

W0:  [array([-2.54950888,  1.70842912]), array([-2.23559289])]
Weights:  [array([[ 1.7563437 , -4.14860103],
       [ 1.65254691, -3.25737182]]), array([[-3.16997196],
       [-2.45010855]])]
Predictions:  [0 1 1 0]
Score:  1.0
```

Figure 2: Résultat d'un fit pour le perceptron XOR multi-couche.

A partir des résultats de la Figure 2, nous pouvons indiquer les poids pour la topologie représentée par la Figure 3.

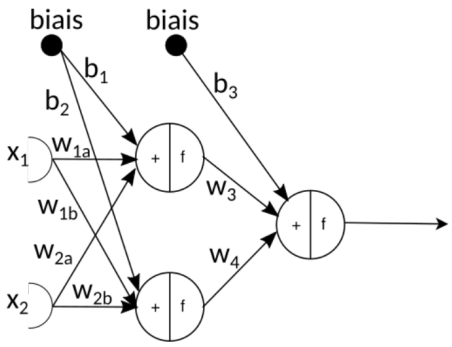

Figure 3: Topologie du perceptron multi-couche XOR.

Dans le cas de notre opérateur XOR, les poids et biais sont donnés par :

- $w_{1a} \approx 1.756$
- $w_{2a} \approx 1.653$
- $w_3 \approx -3.170$
- $b_1 \approx -2.550$
- $b_3 \approx -2.236$
- $w_{1b} \approx -4.149$
- $w_{2b} \approx -3.257$
- $w_4 \approx -2.450$
- $b_2 \approx 1.708$

3.3 Exercice d'OU-Exclusif comme une régression

Maintenant que on est arrivé à une méthode efficace pour réaliser notre operateur XOR par un perceptron multi-couche généré par apprentissage supervisé du type classification, nous voulons essayer aussi l'approche par régression. Ici, le réseau essaie de deviner les bonnes valeurs de sortie au lieu de simplement prédire la classe comme dans le cas du

classifieur. L'objectif est de réduire autant que possible la différence entre les sorties du réseau et les vraies valeurs attendues.

D'abord, dans notre script, on importe la définition de l'entité dédié à cet objet dans la bibliothèque **Sklearn** :

```
from sklearn.neural_network import MLPRegressor
```

Après, on souhait avoir une instance avec un minimum de 2 couches cachées et toujours activation du type tanh. La méthode d'optimisation suggéré a été la "lbfgs" (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) qui est un algorithme d'optimisation sans contrainte utilisé pour minimiser une fonction objectif. Comme avant, nous cherchons entraîner le réseau en utilisant la fonction `fit` et après on montre les résultats avec notre méthode `get_perceptron_data`. Le extrait de code utilisé était le suivant :

```
elif flags["exercice_7"]:
    operateurXOR = MLPRegressor(hidden_layer_sizes=3, activation="tanh",
    solver="lbfgs")
    operateurXOR.fit(data, XORlabels)

    print("\n-----------------------------------------------\n\t\t XOR
    operator\n")
    get_perceptron_data(operateurXOR, XORlabels, MLP=True)
```

La Figure 4 présente les résultats de sortie sur la console à partir du code précédent. Il est à noter qu'à présent, nous n'avons plus un score exact égal à 1.0, ce qui est tout à fait normal. En effet, les prédictions obtenues sont des nombres décimaux assez proches des entiers attendus, mais jamais exacts (ceci est dû à l'erreur numérique engendrée par la méthode de régression). Le score n'est plus ici une proportion de bonne classification, mais une mesure de la distance absolue entre les sorties attendues et les sorties obtenues.

```
-----------------------------------------------
                XOR operator

W0:  [array([ 0.08625448, -2.11517185,  1.85148193]), array([-1.18198958])]
Weights:  [array([[-1.40258134, -0.83544577, -1.0776414 ],
       [-1.36160257, -1.62697368, -1.05712372]]), array([[-1.45430545],
       [-0.09392033],
       [ 1.27749099]])]
Predictions:  [1.32073568e-04 9.99862826e-01 9.99946460e-01 3.88273139e-05]
Score:  0.9999999593656811
```

Figure 4: Résultat d'un fit pour le perceptron XOR multi-couche.

3.4 Exercice d'application robotique

Dans cette partie du travail pratique, nous avons comme objectif le développement d'une solution basée sur un réseau de neurones du type MLP pour la navigation autonome du robot Thymio évoluant dans un labyrinthe. On commence le processus de validation de l'apprentissage supervisé en utilisant l'ensemble des capteurs de proximité.

À l'aide d'un réseau travaillé lors des tps précedents, illustré par la Figure 5, d'évitement d'obstacles, nous avons fais évoluer le robot dans le labyrinthe pour collecter des données provenant des sept capteurs de proximité ainsi que les consignes de vitesse des moteurs.

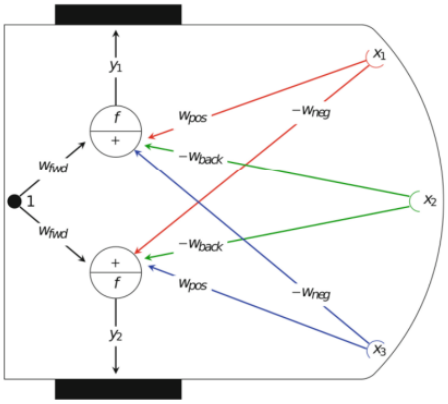

Figure 5: Modèle du réseau d'évitement d'obstacles.

Les poids utilisés sont donnés par :

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

Le morceau de code suivant montre la partie principale de cette phase du projet, nous avons stocké l'ensemble des données dans des listes (`speed_list` et `proximeters_list`), puis nous les avons enregistrés dans un fichier HDF (Hierarchical Data File) à l'aide de l'utilitaire h5py. Pour choisir le moment de sauvegarde du dataset, c'est à dire le moment jusque auquel on enregistre, il faut que l'utilisateur appuie sur la touche "DOWN". Cela devra être fait après avoir complété un tour dans le labyrinthe, et il faudra repérer l'opération une fois pour chaque sens. Nous avons sauvé deux datasets, l'un pour le sens horaire et l'autre pour le sens anti-horaire. Ils se trouvent dans le dossier "datasets" de notre contrôleur.

```
if flags["exercice_8"]:
    y_l = f_activation_sat(np.matrix(X_f), np.matrix(W_l).T)
    y_r = f_activation_sat(np.matrix(X_f), np.matrix(W_r).T)

    sl = y_l[0]*speed_max
    sr = y_r[0]*speed_max

    motor_left.setVelocity(sl)
    motor_right.setVelocity(sr)

    speed_list.append([sl, sr])
    proximeters_list.append([x_lf, x_l2f, x_cf, x_rf,
    x_r2f, x_lb, x_rb])

    if (key==Keyboard.DOWN) and not saved:
```

```
        print("Saving dataset...")
        with h5py.File("datasets/dataset_webots.hdf5", "w") as f:
            speed_array = np.array(speed_list)
            proximeters_array = np.array(proximeters_list)

            speed_dataset = f.create_dataset('thymio_speed',
            data=speed_array)
            proximeters_dataset = f.create_dataset('thymio_prox',
            data=proximeters_array)

            speed_list = []
            proximeters_list = []
            saved = True
            f.close()
```

Une fois ces mesures réalisées, on les récupèrent dans le fichier `perceptron.py`, puis, après les avoir normalisées, on utilise la fonction `train_test_split` pour les séparer aléatoirement en échantillons d'entraînement et de test.

```
    # Ouverture du fichier anti-horaire
    with h5py.File('datasets/dataset_webots_anti-horaire.hdf5', 'r') as
f:
        # Recuperation des datasets
        speed_dataset = f['thymio_speed']
        proximeters_dataset = f['thymio_prox']

        # Conversion des datasets en listes
        speed_array_ah = speed_dataset[:]
        proximeters_array_ah = proximeters_dataset[:]

    with h5py.File('datasets/dataset_webots_horaire.hdf5', 'r') as f:
        # Recuperation des datasets
        speed_dataset = f['thymio_speed']
        proximeters_dataset = f['thymio_prox']

        # Conversion des datasets en listes
        speed_array_h = speed_dataset[:]
        proximeters_array_h = proximeters_dataset[:]

    # Concatenation des listes
    thymio_commands = np.concatenate((speed_array_ah, speed_array_h),
axis=0)
    thymio_proximeters = np.concatenate((proximeters_array_ah,
proximeters_array_h), axis=0)

    # Normalisation
    speed_max = 9.53
    thymio_commands = thymio_commands / speed_max

    # Split du dataset
    prox_x_train, prox_x_test, prox_y_train, prox_y_test =
train_test_split(thymio_proximeters, thymio_commands)
```

Après la séparation du dataset, on défini un réseau de deux couches cachées de deux

neurones. Celui-ci correspond au réseau minimum qui nous permettra d'obtenir de façon consistante un score supérieur à 0.97.

```
    # Definition du reseau et entrainement
    controller = MLPRegressor(hidden_layer_sizes=(2,2), activation="tanh
   ", solver="lbfgs")
    controller.fit(prox_x_train, prox_y_train)

    # Affichage des resultats
    print("W0: ", controller.intercepts_)
    print("Weights: ", controller.coefs_)
    print("Score: ", controller.score(prox_x_test, prox_y_test))
    print("Predictions: ", controller.predict([[0,0,0,0,0,0,0]]))


    # Pickle le modele
    filename = 'ai_controller_model_hyper_prox.model'
    pickle.dump(controller, open(filename, 'wb'))
```

On évalue le score obtenu pour un réseau obtenu avec cette procédure, et on trouve un score de 0.9977830745542975, correspondant aux poids suivants :
Poids :

- Poids entrant de la première couche cachée : [[1.34217836e+00, -9.27272620e-01], [-1.67545958e-01, -9.44109140e-03], [-8.69027771e-01, -8.93971590e-01], [-7.46572850e-01, 1.53539897e+00], [5.40966150e-04, -1.56212023e-01], [-3.52834173e-01, -7.24735304e-01], [3.23976939e-01, 4.21430955e-01]]
- Poids entrant de la deuxième couche cachée : [[-0.10541683, -0.99874366], [1.06645967, 0.23904906]]
- Poids entrant de la couche de sortie : [[0.26044084, 1.11680691], [-1.3911212 , -0.23417072]]

Biais :

- Biais de la première couche cachée : [0.12888508, 0.31037895
- Biais de la deuxième couche cachée : [0.09092965, -0.51690698]
- Biais de la couche de sortie : [-0.1199145 , 0.15497101]

Le très haut score obtenu peut s'expliquer par le fait qu'on essaye ici d'approcher le comportement d'un réseau de neurones très simple (deux neurones, une couche) par un réseau plus complexe. Le fait d'avoir un bottleneck d'un neurone entre les entrée et les sorties diminuerait le score, car nous n'avions pas cette contrainte dans notre réseau de départ (avoir un bottleneck donne des scores vers 94%).

On test le réseau ainsi créé en simulation, et on trouve que le perceptron nous donne parfois des valeurs de vitesse normalisée légèrement supérieur à 1, car nous avions beaucoup de vitesse à 1 dans notre dataset. Nous ajoutons donc une saturation sur notre modèle. Nous avons donc le code suivant pour avancer à l'aide de notre MLP :

```
            s=controller.predict([[x_lf, x_l2f, x_cf, x_rf, x_r2f, x_lb,
    x_rb]])
            s=np.clip(s, -1, 1)
            motor_left.setVelocity(s[0][0]*speed_max)
            motor_right.setVelocity(s[0][1]*speed_max)
```

Nous observons le même comportement qu'avec le réseau que nous avions utilisé pour les enregistrements. Une vidéo de démonstration du résultat obtenu est disponible sur le lien : `https://youtu.be/UL8ZTWwn_j8`.

4 Conclusions

En conclusion, notre projet a utilisé diverses bibliothèques d'apprentissage automatique telles que Perceptron, MLPClassifier et MLPRegressor de sklearn, ainsi que des outils essentiels de traitement des données tels que `train_test_split`, h5py pour la manipulation des données, numpy pour les opérations numériques et pickle pour la sérialisation. En utilisant ces outils, nous avons collecté des données simulées à partir d'un robot autonome équipé d'un système de perception codé à la main, inspiré du modèle du véhicule de Braitenberg. 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_1_2_3": False,
    "exercice_4_5_6": False,
    "exercice_7": False,
    "exercice_8": False,
    "exercice_9": False,
    "exercice_10": True,
}
```

À travers des expérimentations approfondies et l'entraînement de modèles, nous avons atteint une étape importante : la navigation autonome du robot à travers un labyrinthe. Le résultat de notre projet démontre l'efficacité des techniques d'apprentissage automatique avec des systèmes de perception attentivement conçus pour des tâches de navigation autonome. Ce succès souligne le potentiel d'intégration des principes classiques de la robotique avec des approches modernes d'apprentissage automatique pour permettre un comportement intelligent dans des systèmes autonomes.

Nous avions vu précédemment que pour les réseaux trop larges, il devenait difficile de fixer les poids manuellement, et nous avons donc appris à les fixer par apprentissage par renforcement. Nous avons vu qu'il était possible d'utiliser un réseau entrainé par renforcement pour obtenir un comportement désiré.

Pour l'avenir, des affinements et des améliorations supplémentaires de notre approche pourraient conduire à des systèmes autonomes encore plus robustes et polyvalents capables de naviguer dans des environnements complexes avec précision et fiabilité.