INTEG: Identification paramètres dynamiques Approche Deep Learning

Nathan **FRAPPEREAU**
Jonathan **DELACOUX**

Pour comprendre au mieux ce rapport, nous vous invitons à ouvrir notre code python en parallèle et à suivre les indications pour l'exécuter

Sommaire :

I / Justification de l’approche

Dans le cadre du projet d’année en année de l’option robotique nous avons choisi de travailler sur la question de l’identification des paramètres dynamiques (masses, inerties et positions des centres de masse) du système. Il est important en robotique de connaître ces paramètres afin de prédire avec précision le comportement dynamique des robots. Sachant que le robot est construit de pièces avec une géométrie particulière et que l’actionnement des joints n’est pas ponctuel ni de masse négligeable, les masses, les inerties et les positions des centres de masse sont parfois très différents dans la réalité comparé au robot sur le papier.

Le système étudié est un robot RR, nous avons fait le choix de l’étudier dans le plan horizontal mais notre approche devrait être adaptable pour le robot dans le plan vertical.

En étudiant le travail de l’année précédente réalisé par Noé **MASSON** et François **LALUBIN** nous avons décidé de prendre un pas de recul par rapport à leur méthode analytique pour comparer leurs résultats à ceux que nous obtiendrions par une méthode expérimentale. Nous avons donc fait le choix d’étudier le système comme une "boîte noire” dont on ne connaît pas les paramètres ou les équations et d'entraîner un réseau de neurones pour qu’il arrive à identifier les paramètres inconnus.

II / Définition du modèle

Pour commencer, il est important de réfléchir à quels sont les paramètres que nous voulons déterminer (nos sorties) et quels sont les paramètres connus (nos entrées).

Dans le cas de notre problème, nous voulons déterminer les masses des bras, leur inertie par rapport à l'axe Z (qui est l'axe de rotation des liaisons pivot) et la position du centre de masse de chaque membre. Nous avons donc **6 sorties à estimer**. En ce qui concerne nos entrées, le robot est actionné en couples donc nous avons 2 couples d'entrée (un par joint), nous avons aussi accès à la position articulaire, la vitesse articulaire et il nous est possible de calculer l'accélération articulaire des 2 joints. Au total nous avons donc **8 paramètres d'entrée**.

Nous envisageons une approche d'apprentissage supervisé. C'est-à-dire qu'on entraîne notre réseau à identifier exactement les paramètres que nous voulons estimer. Une approche d'apprentissage non-supervisé serait aussi possible en laissant au réseau la liberté d'identifier les paramètres/combinaisons linéaires sans que nous les lui donnions explicitement, mais ne sachant pas comment mettre en œuvre une telle méthode, nous avons opté pour l'apprentissage supervisé.

Maintenant que nous savons quelles sont nos entrées et nos sorties, il faut maintenant générer des jeux de données.

III / Récupération des données de simulation

Sachant qu'il serait impossible de générer des jeux de données avec plusieurs combinaisons de paramètres dynamiques connues en utilisant un robot réel, nous avons travaillé principalement avec des données de simulation GAZEBO.

Une première étape dans notre projet était donc d'arriver à générer un grand nombre de données, toutes avec des paramètres de simulations différentes (mais réalistes) afin que ces données puissent être utilisées pour entraîner notre réseau.

Nous avons besoin d'une méthode robuste, rapide et statistiquement complète de génération de données de simulation, c'est pourquoi "les générer à la main" n'était pas une solution possible.

Nous avons donc écrit un script python ***data_generation.py*** qui:

- Génère une combinaison aléatoire de nos paramètres de sortie
- Lance la simulation gazebo pour ces paramètres
- Lit un fichier de contrôle du robot (couples en fonction du temps)
- Enregistre les données du topic /joint_states (positions articulaires, vitesses, couples) dans un fichier .bag ayant dans son noms les paramètres de simulation
- Attend quelques secondes le temps qu'on récupère suffisamment de données
- Arrête l'enregistrement et la simulation Gazebo
- Attend que Gazebo soit complètement éteint
- Recommence

Cette étape de notre projet à été très longue a cause de notre dépendance aux autres groupes, notamment pour la rédaction du launch file et du xacro qui ne permettaient pas initialement de changer les paramètres dynamiques du robot.

Ce script fonctionne, il suffit de choisir le nombre de simulations que l'on veut réaliser (en moyenne, il faut compter 30s pour chaque simulation) dans le script lui-même puis l'exécuter dans une console python avec la commande « *run* ***data_generation.py*** ».

Avec le recul que nous avons maintenant à la fin de ce projet, il serait intéressant de se repencher sur ce script de génération de données. Nous pourrions le rendre plus "général" en le transformant en fonction qui prend en argument les paramètres à générer et leurs limites, les topics à enregistrer, le nombre de points que l'on veut réaliser, etc… Et en ce qui concerne les performances, nous avons vu qu'il était possible de faire tourner Gazebo plus vite cf. "Can I run Gazebo faster than real time" mais n'avons pas eu le temps d'explorer cette piste.

Toutes les données générées par le script data_generation.py sont enregistrées dans le dossier /dataset du répertoire de travail.

IV / Extraction et pré-traitement des données

Une fois qu'on a généré suffisamment de données, on peut s'intéresser à comment ces données brutes seront interprétées par notre réseau de neurones. Nos données sont jusqu'ici contenues dans des fichier .bag certes lisibles en python mais ne possédant pas les opérations algébriques que des données dans un tableau numpy auraient.

Nous avons donc écrit un 2ème script python, ***pre_processing.py***, qui permet d'extraire les données des fichiers .bag générées par le script ***data_generation.py*** et de les mettre dans un format adapté: un tableau numpy.

Le code étant bien commenté, nous ne détaillerons pas ici le fonctionnement de l'algorithme.

Un point à éclaircir peut être est le choix de supprimer les points autour des butées mécaniques. Nous avons fait le choix au début du projet de travailler avec un robot de simulation doté de butées mécaniques, comme notre robot réel. Or, lors de la simulation, nous avons remarqué que le robot venait frapper brutalement la butée mécanique, provoquant une inconsistance et une discontinuité physique dues aux limites de l'outil de simulation. Nous avons donc supprimé tous les points où le robot s'approchait des butées mécaniques à un angle inférieur à ~10°, c'est-à-dire quand **|theta|>3.0** .

Pour tester le fonctionnement de ce script, il faut tout d'abord générer des données à traiter en exécutant le script ***data_generation.py***, puis appeler la librairie ***pre_processing.py*** et utiliser la méthode «pre_process()» dans une variable.

V / Choix de la structure du réseau et entraînement

Le script en rapport avec cette partie est le script ***train.py***.

Cette partie du projet est potentiellement la plus complexe même si notre script python ne semble pas l'indiquer. Par souci de temps et de connaissances sur le sujet, nous avons choisi l'architecture la plus basique possible, une architecture de type Multi-Layer Perceptron Regressor est utilisée avec les paramètres de base. Aucun hyperparamètre n'a été affiné pour améliorer les résultats de notre réseau.

Il serait intéressant de revenir sur ce script dans un futur projet afin d'affiner les paramètres du réseau et potentiellement d'en changer l'architecture pour un modèle plus adapté.

Il serait aussi intéressant d'étudier la convergence du modèle choisi en affichant des courbes d'erreurs et autres, afin de valider ou non la performance de notre approche.

VI / Résultats et analyse

Pour observer les résultats de notre approche il faut executer le script ***train.py*** avec la commande « *run* ***train.py*** ».

Pour être complètement honnête, les résultats ne sont pas bons (pour l'instant).

On trouve des coefficients de détermination r^2 peu satisfaisants : entre 0.16 et 0.40 pour la plupart avec l'exception de I_2 qui est déterminée avec un coeff r^2=0.85 en moyenne.

Plusieurs facteurs sont sans doute responsables de ces résultats:

- Comme dit dans la partie précédente, le modèle n'a pas du tout été adapté à notre problème, une optimisation des hyperparamètres et de l'architecture de notre système améliorerait sans aucun doute les performances de notre estimateur.

- On essaie de calculer des valeurs qui ne sont pas directement observables. En effet, après avoir discuté avec l'autre groupe travaillant aussi sur le sujet de l'identification, ils ont trouvé que **les paramètres réellement observables ne sont pas les paramètres I_1, I_2, m_1, m_2, com_1, com_2 mais certaines combinaisons linéaires de ces paramètres.**

  Il serait donc très intéressant de changer l'output désirée de notre réseau à ces combinaisons linéaires de paramètres :
  - $Iz_1 + m_1com_1^2 + m_2I_1^2$
  - $Iz_2 + m_2com_2^2$
  - $I_1com_2m_2$
  - $g(m_1com_1 + m_2I_1)$
  - gm_2com_2

Ici g=0 car le robot est placé horizontalement, donc il n'y a pas d'influence de la gravité sur le modèle dynamique.

Pour conclure, la méthode deep learning pour l'identification de paramètres est intéressante mais n'a pas encore donné des preuves solides dans la résolution de notre problème. En effet, il est nécessaire de savoir exactement la forme des résultats désirés afin qu'ils soient utilisables, ce qui est arrivé trop tard dans notre projet. Nous avons commencé à développer cette méthode en mettant en place des scripts pour extraire les données et les pré-traiter mais l'approche gagnerait à être poursuivie l'année prochaine. Il reste ainsi à choisir un modèle convenable avec tous ses hyperparamètres ainsi que de déterminer la forme de la sortie du réseau de neurones en fonction des combinaisons linéaires réellement observables.