CentraleSupélec

RAPPORT PROJET S8
CONTRÔLE GARANTI PAR RÉSEAUX DE NEURONES POUR DES ROBOTS MOBILES

Tarek OMRAN
Ali RAMLAOUI

Pôle Systèmes Cyber-Physiques
Encadré par Adnane SAOUD

11 juin 2022

Table des matières

1 Mise en contexte **3**
1.1 Objectifs du projet . . . 3
1.2 Formulation du problème . . . 3
1.3 Outils et démarches . . . 4
1.4 Notions utilisées . . . 4

2 Entraînement d'un contrôleur DDPG **5**
2.1 Mountain car . . . 5
2.2 Pendule . . . 6
2.3 Environnement d'entraînement . . . 7
2.4 Deep Deterministic Policy Gradient (DDPG) . . . 7
2.5 Performances du contrôleur . . . 8
2.6 Régularisation continue . . . 10
2.7 Dubins Car . . . 12

3 Entraînement d'un contrôleur MPC explicit **13**
3.1 Double intégrateur . . . 13
3.2 Entraînement d'un contrôleur MPC (Model Predictive Control) . . . 14

4 Atteignabilité du réseau de neurones **15**
4.1 Principe de la démarche . . . 15
4.2 Algorithme d'atteignabilité . . . 17
4.3 Premiers résultats et performances . . . 18
4.4 Algorithmes alternatifs . . . 20

5 Atteignabilité du système dynamique **20**
5.1 Arithmétique d'intervalles . . . 20
5.2 Arithmétique d'intervalles appliquée au système . . . 21

6 Contrôle garanti du système **23**
6.1 Contrôle garanti . . . 23
6.2 Résultats . . . 23
6.3 Algorithme d'atteignabilité . . . 24
6.4 Résultats finaux . . . 26
6.5 Stabilité par analyse de Lyapunov . . . 27

A Appendix **29**

1 Mise en contexte

1.1 Objectifs du projet

Les systèmes physiques deviennent de plus en plus autonomes et intègrent des modules et logiques de contrôle de plus en plus complexes et difficiles à expliquer. Cela complique la vérification de leur spécification. Le problème est encore plus visible et important dans le cadre de systèmes critiques contrôlés par des outils de type boîte noire tels que des réseaux de neurones. L'intérêt de tels outils est la possibilité de résoudre des problèmes complexes du type apprentissage par vision, ou apprentissage dans un environnement complexe et évoluant de manière imprévisible.
Nous voyons clairement dans le cas des véhicules autonomes que les algorithmes deviennent de plus en plus intelligents et prennent des décisions que les concepteurs ne peuvent que très rarement expliquer ce qui pousse à chercher plus d'interprétabilité de ces systèmes [1]. Cette démarche s'oppose aux systèmes critiques dans le domaine ferroviaire par exemple [2], où les spécificités et normes imposent une certaine méthode de développement des spécifications du système ainsi que des garanties de fonctionnement prenant en compte tous les imprévus possibles tels que des dysfonctionnements informatiques ou des comportements extérieurs rares.

1.2 Formulation du problème

Dans ce projet, nous allons considérer un système dynamique défini par sa représentation d'état en temps discret

$$\begin{aligned} F : \mathbb{R}^n \times \mathbb{R}^m &\to \mathbb{R}^n \\ (x(k), u(k)) &\mapsto F(x(k), u(k)) = x(k+1) \end{aligned} \tag{1}$$

x(k) est ainsi le vecteur d'état du système à l'itération k, et u(k) et l'entrée du système, avec n la dimension de l'état et m la dimension de l'entrée.

Nous souhaitons obtenir un contrôleur qui est capable de piloter le système de manière autonome, c'est-à-dire qu'à partir de l'observation $x(k)$, pouvoir prédire une action $u(k)$ lui permettant d'atteindre un objectif. Le schéma bloc du système final est montré dans la figure 1.

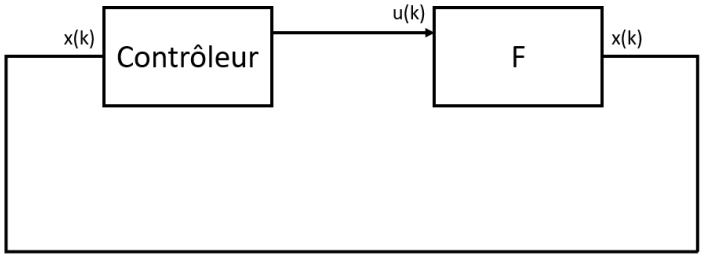


FIGURE 1 – Schéma Bloc du système avec son contrôleur

On souhaite obtenir un contrôleur sous la forme d'un réseau de neurones pour profiter des techniques de Deep Reinforcement Learning, et ensuite prouver que les sorties de la fonction F répondent à certaines spécifications. Cependant, les réseaux neurones sont entraînés uniquement utilisant des données en entrée souvent labellisées et leur interprétabilité est limitée. Pour résoudre ce problème, nous agissons par analyse d'atteignabilité qui consiste à montrer que le réseau de neurones peut répondre à la spécification, sans nécessairement s'intéresser aux raisons pour lesquelles il est capable d'y répondre. On considère un ensemble de départ dans lequel les observations se situent, et on cherche à estimer l'ensemble atteignable au bout d'un nombre fini d'itérations du système contrôlé.

1.3 Outils et démarches

Nous allons nous intéresser dans ce projet à des problèmes simples initialement et l'objectif sera de le résoudre le plus rapidement possible, afin de passer sur des exemples plus complexes et tester la résilience et la portée des modèles et stratégies adoptées lors du projet. L'idée est de construire les modules indépendamment puis de les regrouper afin de pouvoir vérifier les spécificités du système considéré.
Les deux premiers problèmes simples qui nous intéressent sont implémentés dans la librairie Gym [3], ce qui permet de visualiser rapidement les performances du contrôleurs et de garantir que le problème considéré est solvable sans grosses difficultés. L'objectif est alors d'y entraîner le contrôleur en utilisant des méthodes de Deep Learning pour garantir de bonnes performances et un entraînement facile. Ensuite, il s'agit de présenter et expliquer les méthodes utilisées pour prouver les spécifications du système : la démarche consiste à effectuer des analyses d'atteignabilité en itérant sur des temps discrets et vérifier que ces ensembles d'atteignabilité sont bien inclus dans un ensemble correspondant au respect d'une contrainte du système.

1.4 Notions utilisées

Les méthodes que nous allons employer consistent à ne considérer que des hyper-rectangles des espaces $\mathbb{R}^n$, pour n'importe quel n, que nous appellerons tout simplement intervalles.

Définition 1.1 (Intervalle de dimension n)**.** *Considérons l'espace $\mathbb{R}^n$, avec $n \in \mathbb{N}^*$. Alors un intervalle de dimension n est un ensemble pouvant s'écrire $\left[\underline{a_1}, \overline{a_1}\right] \times \ldots \times \left[\underline{a_n}, \overline{a_n}\right]$, où $(\underline{a_i} \leq \overline{a_i}) \in \mathbb{R}^2$, $\forall i \in [\![1, n]\!]$. On pourra également simplifier l'expression en notant l'intervalle $[a]$.*

Définition 1.2 (Sur-approximation par un intervalle)**.** *Pour tout ensemble $\mathcal{H} \subset R^n$, on appelle $\mathcal{I}_\mathcal{H}$, le plus petit intervalle de $\mathbb{R}^n$ tel que $\mathcal{H} \subset \mathcal{I}_\mathcal{H}$.*

L'objectif du projet est alors de définir un ensemble de faisabilité $\mathcal{S}$ qui désigne l'ensemble des points de l'espace d'arrivée à atteindre afin de rendre $\mathcal{S}$ faisable. Un ensemble $\mathcal{H}$ respecte la spécification $\int$ si et seulement si $\mathcal{H} \subset \mathcal{S}$.

Proposition 1.1. *Pour que $\mathcal{H} \subset \mathbb{R}^n$ vérifie la spécification $\mathcal{S} \subset \mathbb{R}^n$, il suffit que $\mathcal{I}_\mathcal{H} \subset \mathcal{S}$.*

Tout au long de ce projet, nous allons approcher tous les ensembles par leur sur-approximation par un intervalle et montrer que ces intervalles vérifient la spécification voulue afin de garantir le contrôle de notre système.

Définition 1.3 (Longueur d'un intervalle)**.** *On appellera longueur d'un intervalle $\mathcal{I} = \left[\underline{a_1}, \overline{a_1}\right] \times \ldots \times \left[\underline{a_n}, \overline{a_n}\right] \subset \mathbb{R}^n$, la quantité*

$$\rho(\mathcal{I}) = \max_{i \in [\![1,n]\!]} (\underline{a_i} - \overline{a_i})$$

Définition 1.4 (Ensemble des intervalles)**.** *On appelle $\mathbb{IR}^n$ l'ensemble de tous les intervalles définis sur $\mathbb{R}^n$.*

Définition 1.5 (Extension aux intervalles d'une fonction)**.** *Soit $f : \mathbb{R}^n \to \mathbb{R}^m$ une fonction. Une extension aux intervalles de la fonction f est une fonction $[f] : \mathbb{IR}^n \to \mathbb{IR}^m$ telle que $\forall \mathcal{I} \in \mathbb{IR}^n$, $\mathcal{I}_{f(\mathcal{I})} \subset [f](\mathcal{I})$.*

Nous constatons à partir de cette définition que l'extension aux intervalles d'une fonction n'est pas unique. L'extension qui nous intéresse est la meilleure extension aux intervalles, c'est à dire celle pour laquelle $[f](\mathcal{I}) = \mathcal{I}_{f(\mathcal{I})}$ est la sur-approximation de l'image de l'intervalle $\mathcal{I}$ par la fonction f.

Définition 1.6 (Fonction monotone à l'inclusion)**.** *Soit [f] une extension aux intervalles d'une fonction f. On dit que [f] est monotone à inclusion si pour* $[x_1], [x_2] \in \mathbb{IR}^n$*, avec* $[x_1] \subset [x_2]$*, alors* $[f]([x_1]) \subset [f]([x_2])$*. Plus particulièrement, cette propriété reste valable lorsque l'intervalle est un singleton.*

Proposition 1.2 (Image d'un intervalle par une fonction monotone)**.** *Soit* $f : \mathbb{R} \to \mathbb{R}$ *une fonction continue croissante (resp. décroissante). Alors* $[f]([\underline{x}, \overline{x}] = [f(\underline{x}), f(\overline{x})])$ *(resp.* $[f]([\underline{x}, \overline{x}] = [f(\overline{x}), f(\underline{x})])$*.*

Démonstration. Utiliser le théorème des valeurs intermédiaires et la monotonicité de f. □

Proposition 1.3 (Image d'un intervalle par une fonction continue)**.** *Soit* $f : \mathbb{R}^n \to \mathbb{R}^m$ *une fonction continue. On désigne pour* $i \in [\![1, m]\!]$*, la composante i de la fonction f par* $f_i : \mathbb{R}^n \to \mathbb{R}$*. Notons* $[f_i]$ *l'extension aux intervalles optimale de* f_i*. Alors pour tout intervalle* $[x] \in \mathbb{IR}^n$*,*

$$[f_i]([x]) = \left[\min_{u \in [x]}(f_i(u)), \max_{u \in [x]}(f_i(u))\right]$$

Démonstration. L'image d'un ensemble connexe par une fonction connexe est encore connexe. Or un intervalle étant un ensemble connexe de $\mathbb{R}^n$, $[f_i]([x])$ est encore un connexe. Comme les connexes de $\mathbb{R}$ sont exactement les intervalles, on obtient le résultat. □

Cette proposition permet de trouver l'extension aux intervalles optimale d'une fonction lorsqu'il les min et les max de chaque composante sont faciles à calculer sans avoir recourt à un problème d'optimisation complexe. Le cas où f est une application linéaire ou affine permet par exemple de calculer facilement cette image, ou lorsque $f : \mathbb{R}^n \to \mathbb{R}^n$ et que chaque composante de f est monotone par rapport à la même composante de x en utilisant en plus la proposition 1.2.

2 Entraînement d'un contrôleur DDPG

2.1 Mountain car

Le premier problème que nous allons considérer est le système *mountain car continuous* implémenté sur Gym [3]. L'objectif est de faire sortir un chariot disposant d'un moteur d'une position d'équilibre stable. L'intérêt du problème réside dans le fait que la gravité est plus forte que la force induite par le moteur du chariot, donc la stratégie qui consiste à avancer vers une direction de la colline n'est pas acceptable. Le problème est uni-dimensionnel car la voiture ne peut que se déplacer vers la droite ou vers la gauche avec une certaine puissance. Le seul moyen de résoudre le problème est alors d'osciller autour de la position d'équilibre stable afin d'accumuler de la vitesse et pouvoir sortir de la colline.

Présentons alors la version formalisée du problème :
Le vecteur d'état est représenté par

$$X(k) = \begin{bmatrix} x(k) \\ \dot{x}(k) = v(k) \end{bmatrix}$$

où x est la position du chariot sur l'axe horizontal, $x \in [-1.2, 0.6]$ et v est sa vitesse sur le même axe, $v \in [-0.07, 0.07]$.

L'entrée du système est représentée par le vecteur u unidimensionnel, qui représente l'action que le chariot effectue, $u \in [-1, 1]$.

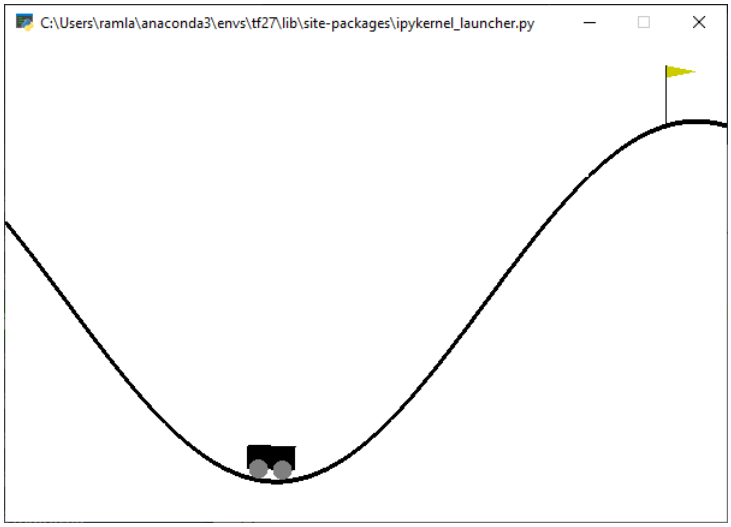


FIGURE 2 – Environnement du problème Mountain Car

La dynamique du système est représentée par

$$\begin{cases} x(k+1) & = x(k) + v(k+1) \\ v(k+1) & = u(k)P - 0.0025\cos(3x(k)) \end{cases} \tag{2}$$

où $P = 0.0015$ est lié à la puissance du moteur, et 0.0025 à la gravitation.
L'objectif dans le problème est d'atteindre la zone $x \geq 0.6$. Dans le cadre de l'entraînement par Reinforcement Learning, une fonction de *reward* est définie afin d'aider l'agent à l'apprentissage et mieux récompenser les actions qui aident l'état du système à respecter l'objectif. Lorsque l'objectif est atteint, la reward prend immédiatement une valeur grande. Sinon, l'objectif de cette fonction est de guider l'action vers la bonne direction, c'est-à-dire prendre des valeurs qui seraient améliorées par les bonnes actions et pénalisées par les mauvaises actions. Dans ce cas, il s'agit de pénaliser les fortes actions afin de garantir que le chariot n'avance pas vers une direction avec une pleine puissance mais lui permettre d'explorer autour de puissances faibles en ne faisant pas la différence entre les deux directions.

$$q(k) = \begin{cases} 100 & \text{si } x(k) \geq 0.6 \\ q(k-1) - 0.1u(k)^2 & \text{sinon} \end{cases} \tag{3}$$

2.2 Pendule

Le second problème auquel nous allons nous intéresser afin de tester les performances de notre démarche est le système du pendule à un degré de liberté disposant d'un moteur afin de permettre sa rotation. Ce problème est également implémenté dans le module Gym [3]. L'objectif est de stabiliser le pendule verticalement et vers le haut.

Le pendule est identifié par l'angle θ qu'il fait avec la verticale ascendante dans le sens anti-trigonométrique et sa vitesse $\dot{\theta}$. Le vecteur d'état est donc

$$X(k) = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}, \; X(k) \in [0, 2\pi] \times [-8, 8]$$

Le vecteur d'entrée du contrôleur est unidimensionnel et correspond à la puissance fournie par le moteur, $u(k) \in [-2, 2]$. La dynamique du système est la suivante

$$\begin{cases} \theta(k+1) & = \theta(k) + \delta_t \dot{\theta}(k) \\ \dot{\theta}(k+1) & = \dot{\theta}(k) + \frac{3g\delta_t}{2l}\sin(\theta(k)) + \frac{3}{ml^2}u(k)\delta_t \end{cases} \tag{4}$$

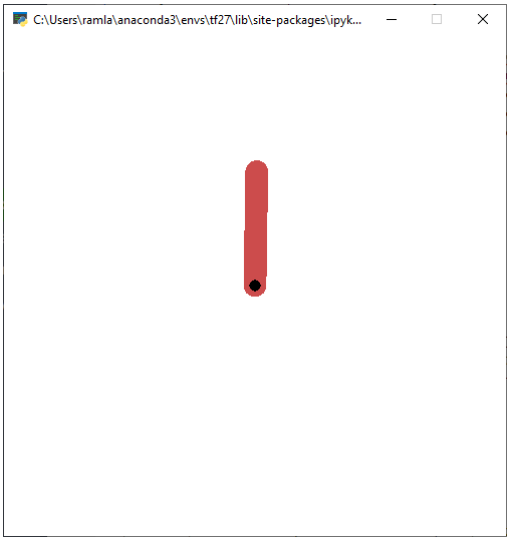


FIGURE 3 – Environnement du problème du pendule

où $\delta_t = 1s$ est le pas de temps du système, g est la constante de gravitation, l la longueur du pendule et m sa masse.
La fonction de récompense est de plus en plus élevée lorsque la mesure principale de l'angle θ vaut 0.

$$q(k+1) = m(\theta(k))^2 + 0.01\dot{\theta}(k)^2 + 0.001(u(k))^2, \text{ où } \theta(k) \in [-\pi, \pi] \tag{5}$$

Le fait de rajouter la vitesse angulaire ainsi que la puissance fournie u permet de pénaliser les situations où le pendule est bien dirigé vers le haut mais qu'il a beaucoup d'inertie ou beaucoup de potentiel d'inertie faisant qu'il risque de tomber dans les itérations suivantes.

2.3 Environnement d'entraînement

L'objectif est de pouvoir entraîner le contrôleur sur différents problèmes sans avoir à réadapter l'écriture du module permettant la génération et l'entraînement du contrôleur à chaque problème. C'est la raison pour laquelle chaque problème est associé à un environnement comportant toute la dynamique du système, la fonction *reward* et étant capable de réinitialiser le problème, garder en mémoire le vecteur d'état actuel du système, renvoyer le vecteur d'observation et effectuer une action à partir d'un vecteur d'entrée pour passer à une étape suivante. Ainsi, l'environnement permet d'initialiser une nouvelle partie avec un vecteur d'état et d'observation initiaux et de fournir un nombre fini d'actions avant d'arriver à une position après un nombre d'actions dépendant du système considéré.
Les deux problèmes présentés plus haut sont déjà implémentés dans un environnement permettant également de visualiser les étapes d'une partie figure 2. Chaque partie dure au maximum 200 étapes pour le système du chariot ou s'arrête immédiatement lorsque l'objectif est atteint. Afin d'adapter les environnements à notre problème, nous modifions légèrement les environnements du module afin d'inclure la possibilité d'initialiser le vecteur d'état sur une position aléatoire dans un intervalle donné. Étant donné que les spécifications que nous voudrons vérifier se font à partir d'intervalles, l'idée est alors de pouvoir modifier l'intervalle sur lequel on entraîne le système et non pas juste démarrer sur une position similaire à chaque épisode.

2.4 Deep Deterministic Policy Gradient (DDPG)

Le contrôleur qui nous intéresse utilise la méthode Deep Deterministic Policy Gradient [4], basée sur une combinaison de plusieurs réseaux de neurones permettant pour certains de ren-

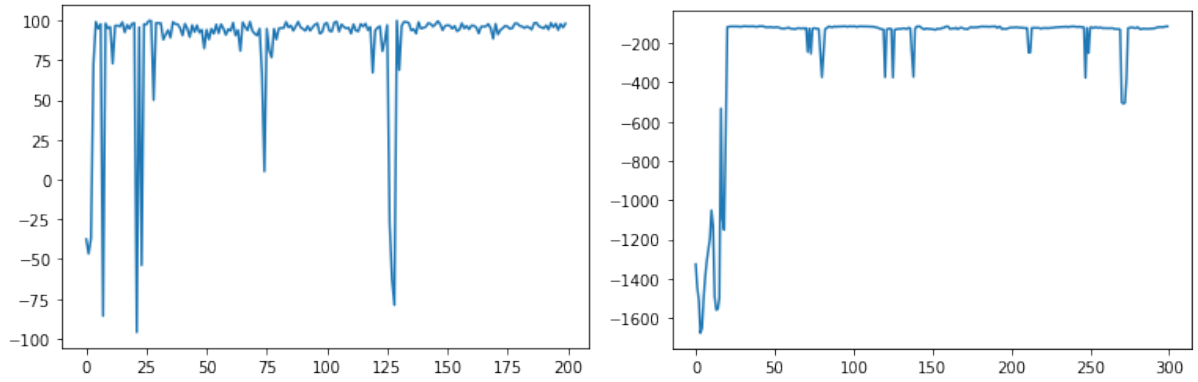
voyer l'action en prenant le vecteur d'observation en entrée, et pour d'autres de critiquer l'action prise par ce premier réseau de neurones afin de lui indiquer, lors de l'entraînement, à quel point sa décision est bonne. Le réseau de neurones critique prend donc en entrée un vecteur d'observation et une action correspondant à cette observation afin de renvoyer un scalaire qui correspond à sa valeur critique. L'objectif de ce critique est donc d'approcher la fonction *reward* le mieux possible. Le réseau de l'acteur prend uniquement en entrée l'observation du vecteur d'état, et renvoie l'action à effectuer. Le contrôleur est donc l'acteur, et le critique est uniquement utile pendant l'entraînement.
L'objectif est d'utiliser, en plus de l'acteur et du critique, des réseaux *target critic* et *target actor* dont l'objectif est de garder un tour d'avance sur les autres réseaux afin de permettre leur entraînement. L'intérêt d'utiliser un acteur DDPG au lieu d'un acteur Q-learning est que, dans notre cas, nous entraînons notre agent sur un espace d'action continu et non discret. Le principe de ces deux méthodes est d'utiliser l'équation de Bellman qui consiste à introduire une Q-value, qui représente le "potentiel" de l'action, c'est-à-dire la *reward* qu'il est possible d'espérer avoir si l'on effectue une certaine action. Elle est multipliée par un facteur $0 < \gamma < 1$ appelé *discount factor*, qui permet de faire diminuer la *reward* lorsque la série d'observations que l'acteur peut prendre mène à des Q-valeurs faibles.
Pour entraîner ces réseaux de neurones, on définit leurs fonctions de perte. Pour l'acteur critique, la fonction de perte est l'erreur quadratique entre sa sortie et la récompense obtenue avec l'équation de Bellman à partir du *target critic*. La fonction de perte de l'acteur est tout simplement l'opposée de la valeur du critique, c'est-à-dire que l'acteur a pour objectif de maximiser la récompense. Les mises à jour de ces deux réseaux de neurones se fait par backpropagation, alors que la mise à jour des réseaux *target*, se fait par moyennage des deux réseaux.
Pour permettre à l'agent d'explorer de nouveaux états lors de l'entraînement, l'idée est de rajouter un bruit à l'action renvoyée par l'acteur. Le bruit utilisé dans l'article est un bruit de type Ornstein-Uhlenbeck qui est un processus stochastique qui permet d'évoluer autour de sa moyenne en inversion la tendance des éléments générés de la manière suivante :

$$x_{t+1} = x_t + dt\theta(\mu - x_t) + \sigma\sqrt{dt}N(0,1) \tag{6}$$

2.5 Performances du contrôleur

Nous allons implémenter l'agent DDPG sur Python en utilisant Keras et TensorFlow. Étant donné que la boucle d'itération de l'agent est réecrite manuellement et qu'à chaque étape de l'entraînement on fait un appel à l'environnement d'entraînement, l'objectif est de paralléliser l'implémentation afin d'accélérer le temps d'exécution et d'entraînement. En effet, sans modification, l'entraînement du réseau de neurones n'est plus parallélisé car la boucle est exécutée séquentiellement et chaque calcul de gradient est espacé par une étape d'itération du système dynamique. Nous avons alors utilisé les décorateurs permettant de passer toutes ces fonctions et toute la boucle d'itération en mode graphe de TensorFlow afin de profiter de la parallélisation automatique du module et de l'accélération matérielle tout au long de l'entraînement. Cela a permis d'obtenir un speed-up d'environ x50 en parallélisant la boucle d'entraînement permettant de passer l'entraînement d'un agent sur chacun des deux problèmes avec 200 épisodes de 200 minutes à quelques minutes seulement.
L'agent arrive très rapidement à trouver une solution au problème et apprendre à le résoudre la plupart du temps. Nous avons réalisé plusieurs figures présentant les performances d'entraînement sur chaque jeu, le taux de succès et les stratégies ou chemins adoptés lors de l'entraînement ainsi que lors des parties sans mémoire de l'agent.
De plus, nous avons remarqué que commencer la partie sur un intervalle aléatoire permet dans les deux cas d'avoir des performances presque similaires au cas où toute les parties sont commencées dans une position déterminée préalablement. Le premier cas étant le plus intéressant pour le sujet de ce projet, on prendra toujours l'intervalle le plus gros possible pour l'entraînement

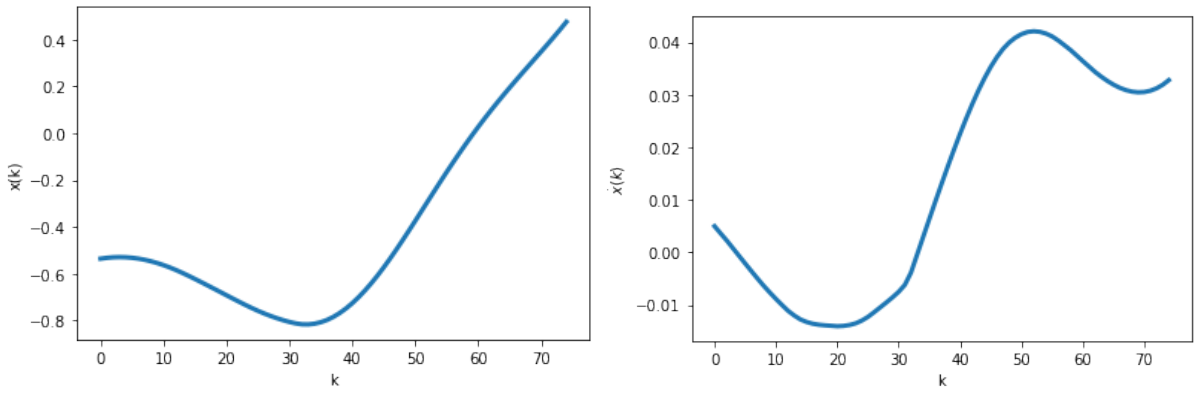
permettant de garantir des résultats acceptables dans tous les cas.



(a) Mountain car (b) Pendule

FIGURE 4 – Récompenses obtenues pendant l'entraînement

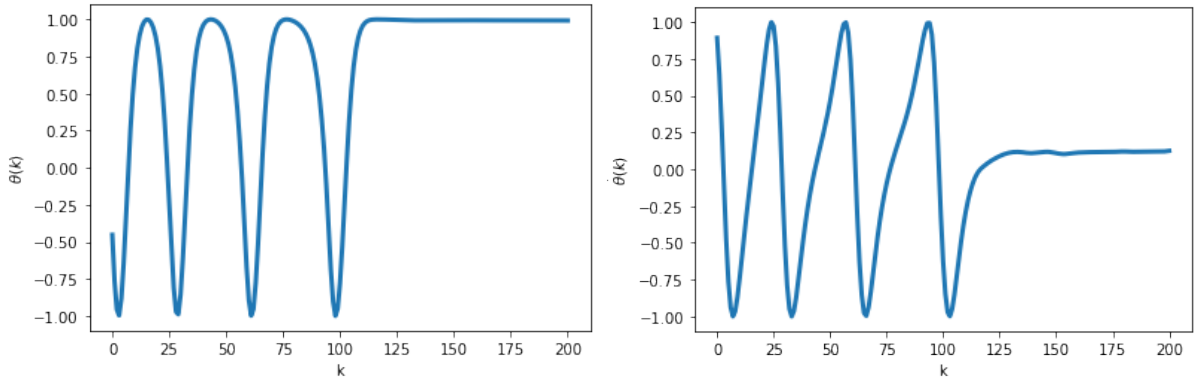
La figure 19 montre les différentes récompenses obtenues à l'issue de chaque épisode de l'entraînement. On constate que dans les deux problèmes, l'agent arrive à converger rapidement au bout d'une trentaine d'épisodes environ. Cependant, il reste des cas où l'agent ne parvient toujours pas à résoudre le problème qui se manifestent par des pics vers le bas.



(a) Position $x(k)$ du chariot (b) Vitesse $\dot{x}(k)$

FIGURE 5 – Mountain car - Variables d'état durant un épisode

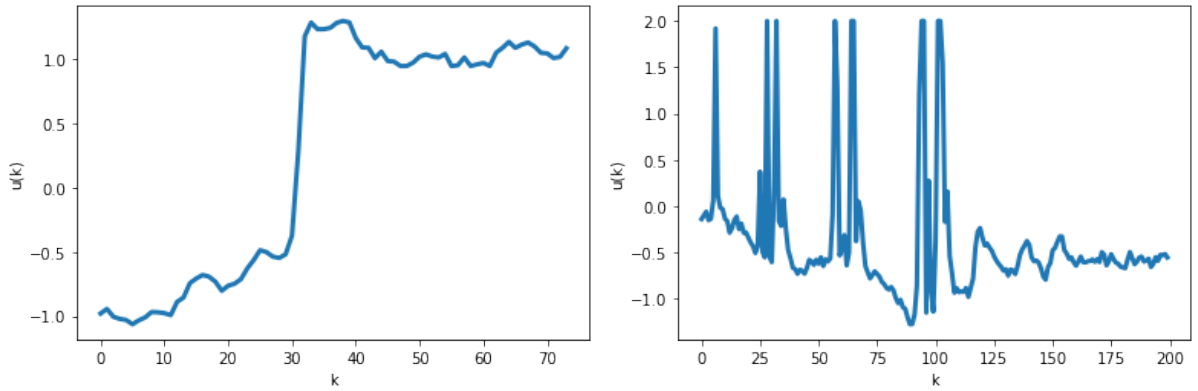
On représente également dans les figure 5 et figure 6 l'évolution des variables d'état du système au cours d'un épisode qui a été résolu avec succès par l'agent. Dans le cas de la voiture, on constate bien que la position atteint l'objectif à la toute fin, et que la voiture est passée à gauche avant de parvenir vers la droite. De même, la vitesse a pu être augmentée grâce à ce passage de la voiture à gauche afin de profiter de l'inertie qu'elle possède pour augmenter sa vitesse. Dans le cas du pendule, on constate la périodicité de θ au début, qui explique que l'agent a eu du mal à stabiliser le pendule pendant trois tours, avant d'y parvenir.
La figure 7 montre beaucoup de discontinuité dans les actions renvoyées par l'agent du pendule, ce qui risquerait de poser problème pour vérifier qu'il est correct sur des intervalles.



(a) Angle $\theta(k)$)

(b) Vitesse $\dot{\theta}(k)$

FIGURE 6 – Pendule - Variables d'état



(a) Mountain car

(b) Pendule

FIGURE 7 – Actions du contrôleur lors d'un épisode

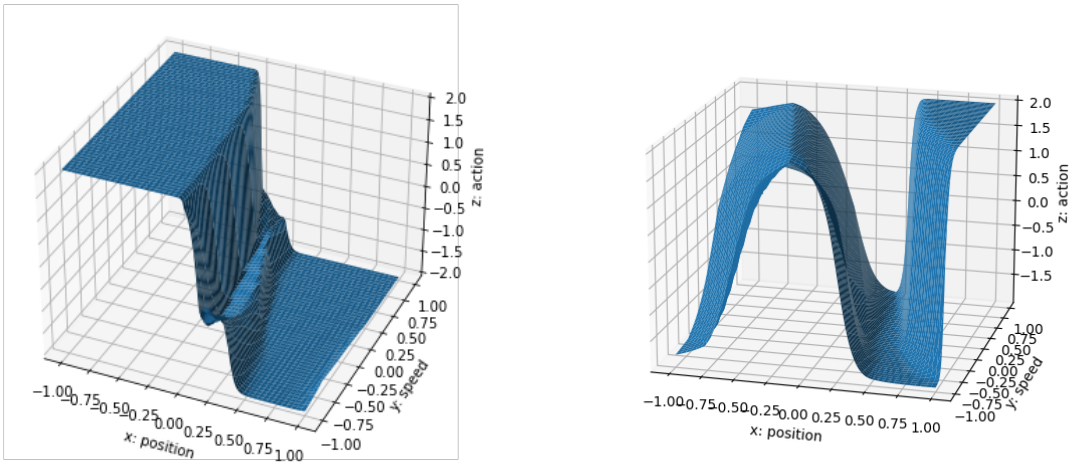
2.6 Régularisation continue

En observant les discontinuités dans le cas du pendule par exemple des actions renvoyées d'une itération à l'autre, on se doute que le contrôleur apprend des représentations possiblement discontinues des actions dans l'espace des états. Nous traçons dans la figure 8 qu'il existe des zones ou la pente est très élevée. Cela pose problème dans le cas d'une analyse d'atteignabilité sur les intervalles car cette discontinuité pourrait causer une grosse perte en précision. Afin de palier à ce problème, on rajoute une condition de régularisation à l'agent DDPG basée sur l'approche de Qianli Shen et al [5]. Elle consiste à rajouter une pénalisation dans la fonction de perte de l'acteur qui pénalise l'entraînement lorsque les actions renvoyées par l'acteur pour des états proches (au sens de la norme euclidienne classique) sont éloignées.

$$\mathcal{R}_s = \lambda_s \mathop{\mathbb{E}}_{s\in\mathcal{B}} \max_{s'\in\mathbb{B}(s,\epsilon)} ||\mu(s) - \mu(s')||_2^2 \tag{7}$$

où λ_s est une constante de régularisation, $\mathcal{B}$ est le batch d'entraînement utilisé, $\mathbb{B}(s,\epsilon)$ est la boule centrée en s et de rayon ϵ, μ est l'acteur qui renvoie les actions en fonction de l'état du système s. Plus on souhaite imposer la continuité, plus il faut augmenter λ_s et ϵ_s (continuité sur un plus grand espace).
Pour implémenter cette pénalisation, étant donné qu'il n'est pas facile de calculer la maximum sur la boule de la distance entre les actions du contrôleur, l'idée est de tirer D_s points aux hasards de la boule pour calculer ce maximum. Plus sa valeur est grande, et plus la valeur du maximum

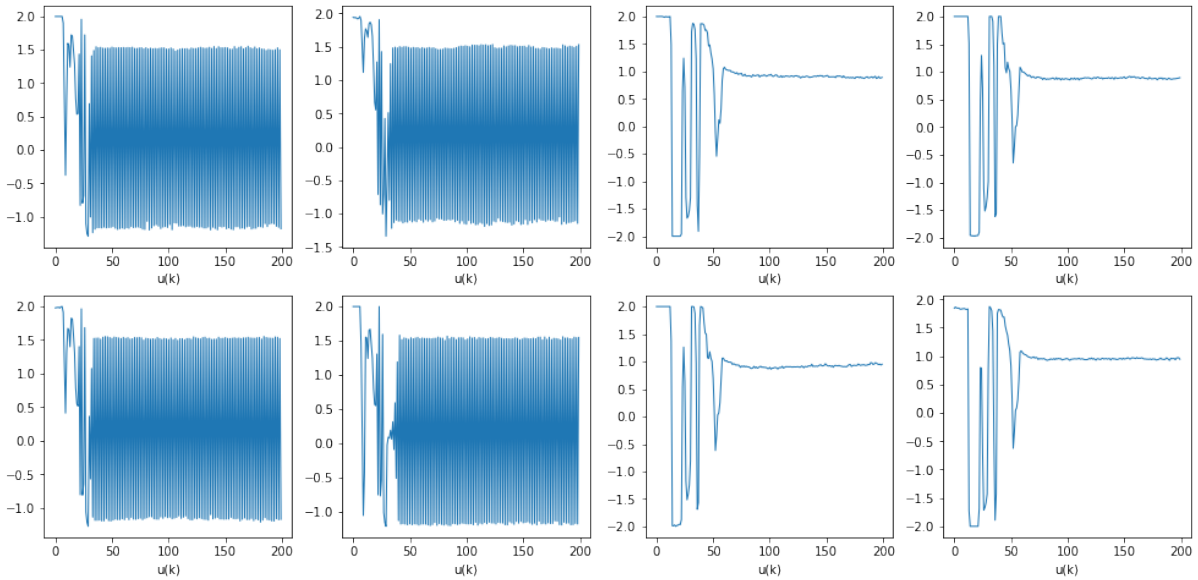
sera précise. Le gradient de cette régularisation peut ensuite être calculé en considérant l'indice s_{max} où cette valeur est atteinte.



(a) Sans régularisation (b) Avec régularisation

FIGURE 8 – Pendule - Actions du contrôleur en fonction de l'état

Nous voyons en pratique que les actions du contrôleur sont beaucoup plus régulières dans la figure 9. Le contrôleur a réussi à apprendre une nouvelle manière de stabiliser le pendule : plutôt que de le stabiliser en faisant osciller ses actions dans les deux extrémités, la condition de régularisation l'oblige à stabiliser le pendule sans avoir à intervenir pour y fournir plus de puissance.



(a) Sans régularisation (b) Avec régularisation

FIGURE 9 – Pendule - Actions sur quelques épisodes

2.7 Dubins Car

Enfin, nous avons décidé d'entraîner un contrôleur DDPG sur un problème non répandu dans la communauté Reinforcement Learning dans lequel nous avons dû nous-même associer une fonction de récompense à une spécification décidée. Il s'agit du système intitulé *Dubins car*. Il représente une voiture se déplaçant en deux dimensions, et qui peut subir des rotations pendant qu'elle se déplace ou en restant immobile. On agit ainsi, dans le cas discret, à chaque pas de temps, sur la vitesse angulaire de la voiture, ainsi que sur la norme de la vitesse du véhicule suivant sa direction actuelle. On représente alors le système par la dynamique suivante :

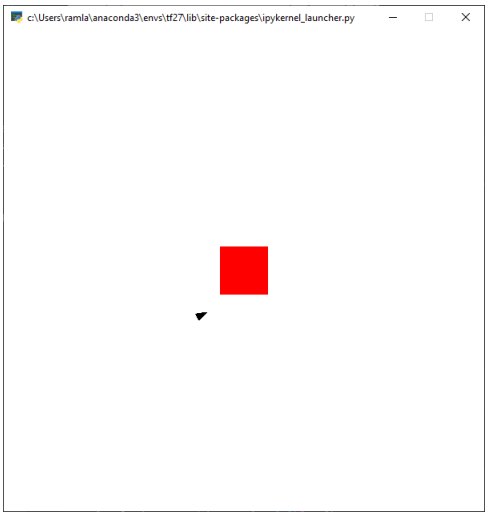


FIGURE 10 – Environnement Dubins Car

$$F : \begin{cases} x(k+1) & = x(k) + u_1(k)\cos(\theta(k)) \\ y(k+1) & = y(k) + u_1(k)\sin(\theta(k)) \\ \theta(k+1) & = \theta(k) + u_2(k) \end{cases} \tag{8}$$

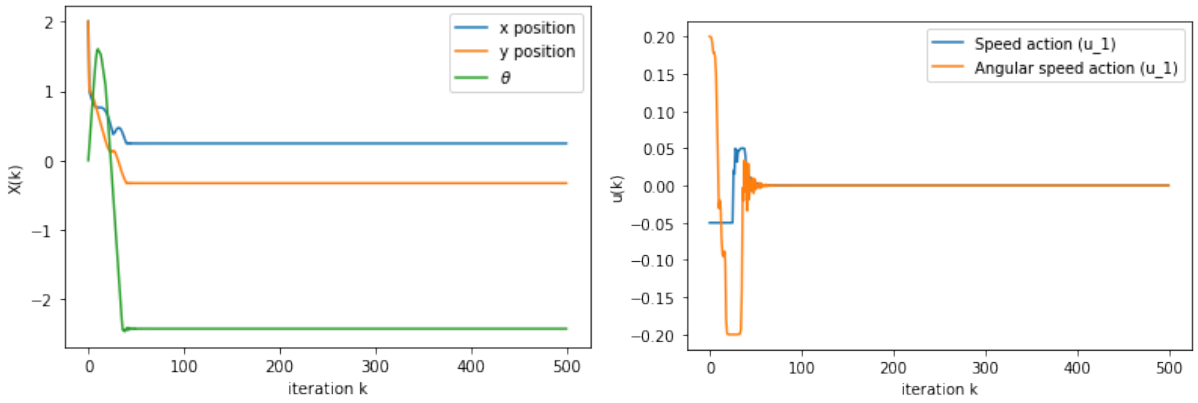
où $u_1 \in [-0.05, 0.05]$, $u_2 \in [-0.2, 0.2]$, $x, y \in [-2, 2]$, $\theta \in [-\pi, \pi]$

On se désigne d'abord comme objectif de stabiliser le véhicule à l'origine. L'objectif est alors de tester la résilience du contrôleur DDPG face à des problèmes simples comme celui-ci (il suffit juste de diriger le véhicule vers l'origine, puis d'appliquer une vitesse appropriée), mais avec deux degrés de libertés au lieu d'un. Nous choisissons alors une fonction de coût simple (opposée de la fonction de récompense), mais dont les paramètres sont à calibrer, prenant en compte la distance à l'origine $d_{goal}(k)$, la différence entre la direction actuelle du véhicule et la direction qu'il devrait prendre pour se diriger vers l'origine $\Delta\theta(k)$, ainsi que l'intensité des actions prises à l'itération actuelle.

$$C(k) = d_{goal}(k) + a\Delta\theta(k) + b||u(k)||^2 \tag{9}$$

où a, b sont des paramètres à rechercher. Pour trouver les valeurs appropriées de a et b, un *GridSearch* peut être effectué, où on fait prendre à ces paramètres des valeurs aléatoires entre 0 et 1, ainsi que certains paramètres de l'agent DDPG (*learning rate*, τ). 200 tests de paramètres aléatoires ont été effectués, chacun durant 40 épisodes (epochs). On trace ensuite toutes les courbes des récompenses pour chaque épisode, et on garde les paramètres donnant les agents les mieux performants (récompense croissante et convergente avec le nombre d'épisodes joués).

Après avoir choisi les meilleurs paramètres, on se rend compte (figure 11) que l'agent n'arrive pas à stabiliser le véhicule à l'origine, mais y arrive presque. Cette recherche de paramètres montre qu'il n'est pas toujours évident de résoudre tout type de problèmes (mêmes simples) avec les méthodes de Reinforcement Learning.



(a) Variables d'état

(b) Actions $u(k)$

FIGURE 11 – Dubins Car - Performances du meilleur contrôleur

3 Entraînement d'un contrôleur MPC explicit

3.1 Double intégrateur

Nous avons également considéré, au cours de ce projet, un système dynamique linéaire qui permet d'appliquer des méthodes plus conventionnelles de contrôle. Il s'agit du double intégrateur, que l'on représente en une dimension (donc une composante pour la position x_1 et une composante pour la vitesse x_2). L'action fournie correspond alors à l'accélération du système.

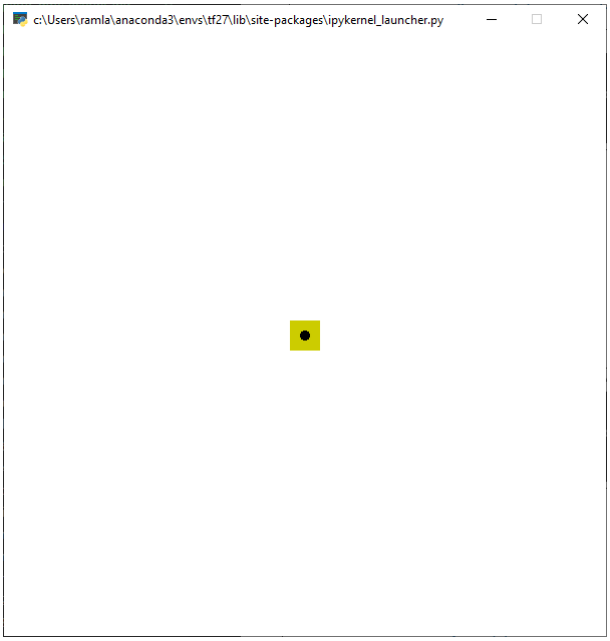


FIGURE 12 – Environnement du double intégrateur (1D horizontal)

$$F : \begin{cases} x(k+1) & = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(k) \\ y(k) & = \begin{bmatrix} 1 & 0 \end{bmatrix} x(k) \end{cases}$$

où $u(k) \in [-1, 1]$ et $x(k) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, x(k) \in [-\infty, +\infty] \times [-\infty, +\infty]$.
La fonction coût associée à ce problème est la suivante :

$$\sum_{t=0}^{\infty} x^{'}(t)Qx(t) + Ru^2(t)$$

que l'on calcule avec un horizon 2, avec Q la *weight matrix* et R la constante de Ricatti. L'objectif est de stabiliser le système à l'origine.

3.2 Entraînement d'un contrôleur MPC (Model Predictive Control)

Le MPC est devenu la norme acceptée pour les problèmes complexes de contrôle multivariable avec contraintes dans les industries de transformation. Ici, à chaque instant d'échantillonnage, en partant de l'état actuel, un problème de contrôle optimal en boucle ouverte est résolu sur un horizon fini de deux. Au pas de temps suivant, le calcul est répété en partant du nouvel état et sur un horizon décalé, ce qui conduit à une politique à horizon mobile. La solution s'appuie sur un modèle dynamique linéaire, respecte toutes les contraintes d'entrée et de sortie, et optimise un indice de performance quadratique. Ainsi, dans la mesure où un indice de performance quadratique associé à diverses contraintes peut être utilisé pour exprimer les véritables objectifs de performance, la performance du contrôleur MPC est excellente. Au cours de la dernière décennie, une base théorique solide pour le MPC a émergé, de sorte que dans les applications MIMO (Multiple Inputs, Multiple Ouputs) à grande échelle, les contrôleurs avec des garanties de stabilité non conservatives peuvent être conçus de manière routinière et facile.
On utilise alors la Toolbox d'Alberto Bemporad[6] pour résoudre le problème du double intégrateur. Les régions obtenues sont représentées dans la figure 13.

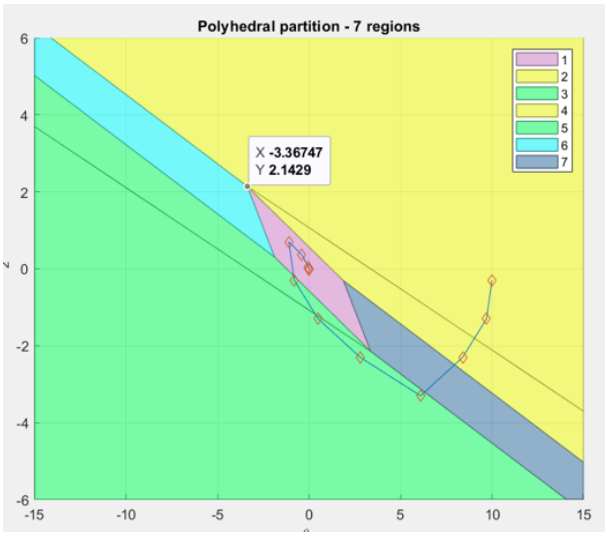
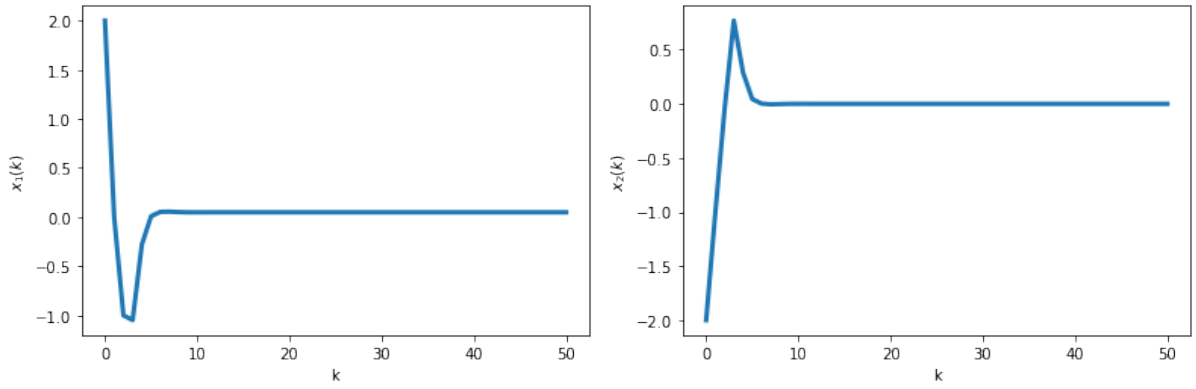


FIGURE 13 – Régions MPC issues de [6]

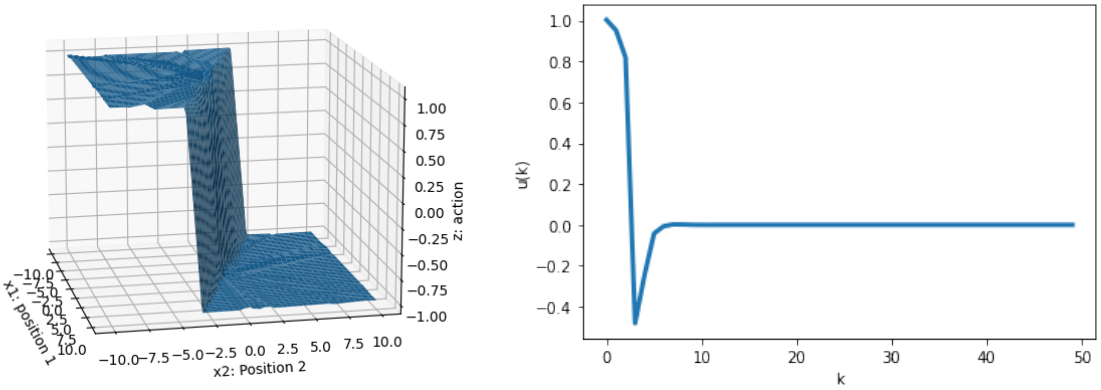
Mais le problème avec le MPC explicite est un problème de stockage d'informations, surtout si le système possède plusieurs régions. Ce que nous allons faire est donc d'approcher ce contrôleur MPC explicite par un réseau neuronal entraîné avec 10000 données générées de manière aléatoire en utilisant ce MPC. Voici quelques résultats du réseau de neurones après avoir appris le contrôleur MPC :

L'objectif est désormais de vérifier que cette approximation est bien légitime en prouvant que le contrôleur neuronale qui approche le contrôleur MPC (ainsi que les contrôleurs DDPG présentés précédemment) est bien correct et stabilise le système en fonction de l'intervalle de départ dans les parties suivantes du projet.



(a) Composante x_1 (b) Composante x_2

FIGURE 14 – Double intégrateur - Performances du réseau de neurones



(a) Actions en fonctions des états (b) Sur un épisode

FIGURE 15 – Actions du contrôleur MPC dans le cas du double intégrateur

4 Atteignabilité du réseau de neurones

4.1 Principe de la démarche

L'idée de cette section est de montrer comment approcher l'image d'un intervalle par un réseau de neurones suivant l'approche de W. Xiang, et al [7]. Les réseaux de neurones considérés dans ce projet seront tous constitués de couches denses uniquement et de fonctions d'activation croissantes (sigmoïde, ReLU, tanh...). On représente un réseau de neurone Φ à L couches intermédiaires par les dimensions $(n_0, n_1, \ldots, n_L) \in \mathbb{N}^{L+1}$ de chaque couches, les poids $W^l \in \mathbb{R}^{n_l \times n_{l-1}}$, les biais associés $b^l \in \mathbb{R}^{n_l}$ et enfin les fonctions d'activations $\sigma_l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$. On considère alors un intervalle de départ $\mathcal{I} \subset \mathbb{R}^{n_0}$. On cherche à trouver un intervalle contenant $\mathcal{I}_{\Phi(\mathcal{I})}$, aussi petit que possible. Une manière simple de le faire consiste à approcher l'image par chaque couche intermédiaire de l'intervalle d'entrée et de l'approcher elle aussi par un intervalle, puis répéter la procédure jusqu'à la traversée de tout le réseau de neurones. On se doute qu'en appliquant cette méthode, les erreurs d'approximation vont se propager à travers toutes les couches. Nous allons essayer de mesurer cette erreur dans cette partie. Il paraît quand même beaucoup plus intéressant de considérer le moins de couches possibles dans le réseau de neurones afin de garder des sur-approximation plus précises et mieux vérifier les spécifications tout en réduisant la complexité de la preuve de garantie.

Définition 4.1. *Pour $l \in [\![1, L]\!]$ on désigne par $\Phi_l : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$ la fonction sortie de la couche*

l, $\forall x \in \mathbb{R}^{n_{l_1}}, \Phi_l(x) = \sigma_l(W^l x + b^l)$. Notons $[\Phi_l]$, l'extension aux intervalles optimale de Φ_l. Alors

Proposition 4.1 (Image optimale d'un intervalle par une application affine)**.** *Pour $l \in [\![1, L]\!]$, l'extension aux intervalles optimale de la fonction $x \mapsto W^l x + b^l$ est la fonction $[x^{l-1}] \mapsto [x^l]$, où pour $i \in [\![1, n_l]\!]$,*

$$\begin{aligned} \underline{x}_i^l &= \sum_{j=1}^{n_{l-1}} \underline{p}_{i,j} + b_i^l \\ \overline{x}_i^l &= \sum_{j=1}^{n_{l-1}} \overline{p}_{i,j} + b_i^l \end{aligned} \tag{10}$$

où

$$\begin{aligned} \underline{p}_{i,j} &= \begin{cases} w_{i,j}\underline{x}_j \text{ si } w_{i,j} \geq 0 \\ w_{i,j}\overline{x}_j \text{ si } w_{i,j} < 0 \end{cases} \\ \overline{p}_{i,j} &= \begin{cases} w_{i,j}\overline{x}_j \text{ si } w_{i,j} \geq 0 \\ w_{i,j}\underline{x}_j \text{ si } w_{i,j} < 0 \end{cases} \end{aligned} \tag{11}$$

Démonstration. L'application affine étant continue, il s'agit de montrer d'après la proposition 1.3 que

$$[x_i^l] = \left[\min_{u \in [x]} \left(\sum_{j=1}^{n_{l-1}} w_{i,j} u_j + b_i^l \right), \max_{u \in [x]} \left(\sum_{j=1}^{n_{l-1}} w_{i,j} u_j + b_i^l \right) \right]$$

Nous allons le montrer pour le min, l'idée étant la même avec le max. Montrons d'abord que pour $u \in [x]$, $j \in [\![1, n_{l-1}]\!]$, si $u_j \notin \{\underline{x}_j, \overline{x}_j\}$, on peut trouver une solution plus optimale pour le minimum. En effet, si $w_{i,j} > 0$, il suffit de prendre $u_j = \underline{x}_j$ pour obtenir une solution plus optimale et si $w_{i,j} < 0$, on peut prendre $u_j = \overline{x}_j$ pour améliorer la solution. On en déduit alors que

$$u_j = \begin{cases} \underline{x}_j \text{ si } w_{i,j} \geq 0 \\ \overline{x}_j \text{ si } w_{i,j} < 0 \end{cases}$$

est la solution du problème de minimisation. De même, la solution pour le problème de maximisation est

$$u_j = \begin{cases} \overline{x}_j \text{ si } w_{i,j} \geq 0 \\ \underline{x}_j \text{ si } w_{i,j} < 0 \end{cases}$$

d'où le résultat. □

Nous allons désormais composer cette extension aux intervalles à l'extension aux intervalles de la fonction d'activation afin d'obtenir une extension aux intervalles de la fonction Φ_l. Cette dernière n'est pas nécessairement optimale.

Proposition 4.2. *La fonction*

$$\begin{aligned} [\Phi_l] : \mathbb{IR}^{n_{l-1}} &\to \mathbb{IR}^{n_l} \\ [x] &\mapsto [x^l] \end{aligned} \tag{12}$$

où

$$\begin{aligned} \underline{x}^l &= \sigma_l\left(\sum_{j=1}^{n_{l-1}} \underline{p}_{i,j} + b_i^l \right) \\ \overline{x}^l &= \sigma_l\left(\sum_{j=1}^{n_{l-1}} \overline{p}_{i,j} + b_i^l \right) \end{aligned} \tag{13}$$

est une extension aux intervalles de Φ_l.

Démonstration. En combinant les propositions 1.2 et 4.1 et le fait que l'extension aux intervalles d'une composition de fonctions est la composition des extensions aux intervalles, on obtient le résultat. □

Nous cherchons désormais à mesurer l'erreur que l'on effectue sur la taille de l'intervalle obtenue par l'extension aux intervalles du réseau de neurones, obtenues par compositions de toutes les extensions aux intervalles de chaque couche, car elle n'est pas supposée optimale.

Proposition 4.3 (Erreur de sur-approximation)**.** *Soit* $[x], [y]$ *deux intervalles tels que* $[x] \subset [y]$, *ie.* $\rho([x]) \leq \rho[y]$. *Alors* $[\Phi]([x]) \leq [\Phi]([y])$. *De plus, si* $[x]$ *est réduit à un point, ie.* $\underline{x} = \overline{x} = x$, *alors* $[\Phi]([x]) = \{x\}$.

Démonstration. En utilisant la proposition 4.2, la croissance des fonctions d'activation, et les formules de l'équation 11, on constate que $\underline{x}^l \geq \underline{y}^l$ et $\overline{x}^l \leq \overline{y}^l$, pour $l = 0$, puis pour toutes les autres couches par récurrence triviale. Lorsque $[x]$ est réduit à un point, alors $\underline{x} = \overline{x}$, donc pour tout $l, \underline{x}^l = \overline{x}^l$. □

L'idée de l'algorithme est alors d'utiliser les résultats de la proposition 4.2 et 4.1 afin de calculer l'ensemble atteignable d'un ensemble donné en entrée. Ainsi, pour tout ensemble $\mathcal{H}$ d'entrée, on peut obtenir un intervalle $[u] = [\Phi](\mathcal{I}_{\mathcal{H}})$.
Comme les ensembles atteignables ne sont pas nécessairement des intervalles, on peut comprendre que les sur-approximations effectuées ne permettent pas d'approcher très précisément des ensembles qui ne ressemblent pas à des hyper-rectangles, d'autant plus que $[\Phi]$ n'est pas forcément l'extension aux intervalles optimales.
Pour résoudre ce problème, on utilise la proposition 4.3, qui indique que plus l'intervalle d'entrée est de longueur petite, plus l'erreur effectuée est aussi petite. Nous pouvons alors partitionner notre intervalle de départ en plusieurs sous-intervalles de sorte que l'on puisse calculer l'ensemble d'atteignabilité pour chacun de ces nouveaux sous-intervalles, et l'ensemble atteignable finale serait juste la réunion des intervalles obtenus en sortie. On voit bien que plus la taille de ces sous-intervalles est petite, plus la précision sera meilleure, et mieux on pourra approcher des ensembles de forme géométrique complexe par une union d'hyper-rectangles. Il s'agit alors de contrôler cette taille d'intervalles, de sorte à ne pas faire exploser la complexité des calculs.

4.2 Algorithme d'atteignabilité

Afin d'éviter de faire des subdivisions inutiles, et d'accélérer l'exécution de l'algorithme, l'idée est de contrôler la manière dont on effectue les subdivisions. Pour ce faire, on génère des points aléatoires de l'ensemble d'entrée, et on calcule les sorties du réseau de neurones appliqué à ces points aléatoires. On obtient alors une sous-estimation de l'ensemble de sortie, et on calcule la sur-approximation par un intervalle de cet ensemble.
Étant donné que cet ensemble est une sous-estimation de l'ensemble atteignable, si une subdivision du grand intervalle de départ donne une image par les formules 4.2, alors l'ensemble atteignable estimé pour cette subdivision est déjà optimal, il est inutile de découper d'avantage cet intervalle.
Nous commençons ainsi avec l'intervalle initial, et à chaque fois que les formules 4.2 ne permettent pas l'estimation d'entrer dans l'intervalle simulé, on le découpe en deux en effectuant une bissection selon l'axe de plus grande longueur, et on recalcule les nouveaux intervalles d'atteignabilité des deux sous-intervalles obtenus et on réitère.
Nous avons désormais tous les outils pour présenter l'algorithme d'atteignabilité du réseau de neurones.

Algorithm 1 Atteignabilité du réseau de neurones

Input : Réseau de neurone Φ, l'ensemble de départ $\mathcal{H}$ dont on cherche l'atteignabilité, ϵ la précision de l'approximation souhaitée, N le nombre de simulations à effectuer

Output : Sur-approximation $\mathcal{U}$ de l'ensemble atteignable à partir de $\mathcal{H}$

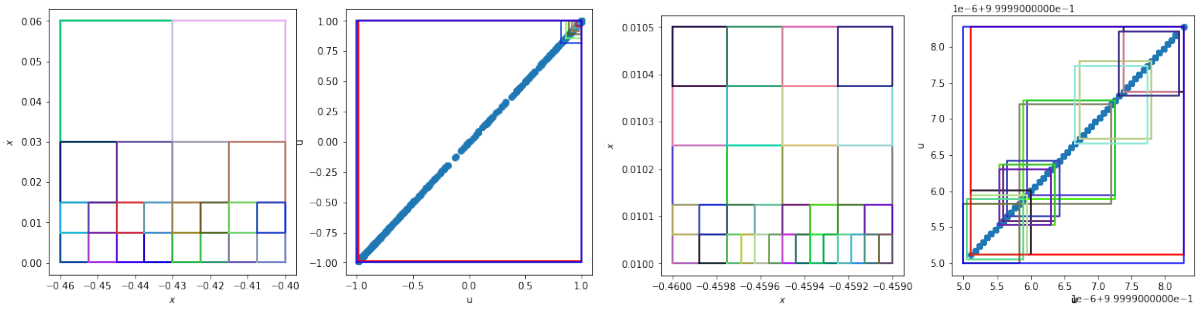
Calculer le plus petit $[x^0]$ tel que $\mathcal{H} \subset [x^0]$
Calculer $[u]$, l'ensemble atteignable à partir de $[x^0]$
Initialiser la file de couples $\mathcal{M} = \{([x^0], [u])\}$
Générer N simulations à partir de N vecteurs aléatoires issues de $\mathcal{H}$ et sur-approximer l'ensemble obtenu par l'intervalle $[u_{sim}]$
$\mathcal{U} \leftarrow \emptyset$ ▷ On initialise l'ensemble atteignable que l'on va construire par unions
while $\mathcal{M}$ non vide **do**
    Pop $([x^0], [u])$ from $\mathcal{M}$
    if $[u] \subset [u_{sim}]$ **then** ▷ Meilleure approximation réalisée
        $U \leftarrow U \cup [u]$
    else
        if $\rho([x^0]) > \epsilon$ **then**
            Générer $[x_1^0]$ et $[x_2^0]$ de la bissection de $[x^0]$ selon l'axe de plus grande longueur
            Calculer les sur-approximation par Φ $[u_1]$ et $[u_2]$
            Rajouter les nouveaux couples $([x_1^0, u_1])$ et $([x_2^0, u_2])$ à la queue $\mathcal{M}$
        else
            $\mathcal{U} \leftarrow \mathcal{U} \cup [u]$
            Break ▷ Sortir complètement de la boucle
        end if
    end if
end while
return $\mathcal{U} \cup (\cup_{([x^0,u])}[u])$ ▷ Rajouter les éléments manquants

4.3 Premiers résultats et performances

On veut observer les résultats de l'algorithme sur les problèmes mountain car et le pendule, en faisant varier les intervalles de départ et la constante ϵ. L'espace d'arrivée est unidimensionnel dans les deux cas car le contrôleur ne peut agir que sur une dimension. On décide quand même de réprésenter les intervalles en deux dimensions afin de mieux pouvoir interpréter les résultats (il s'agit juste de dédoubler la dimension).
Nous allons représenter à droite, l'intervalle d'action renvoyé par le contrôleur. Le plus grand intervalle tracé et représenté en bleu correspond à la sur-approximation par un intervalle de la sortie de l'algorithme. Les points correspondent aux valeurs obtenues par simulations sur des points aléatoires et l'intervalle en rouge correspond à la sur-approximation de l'ensemble des points simulés. L'objectif est alors d'avoir un intervalle bleu qui se rapproche le plus possible de l'intervalle rouge. Plus la différence est petite, plus l'algorithme est performant. Nous représentons à gauche l'intervalle des variables d'états ainsi que la subdivision qui a été effectuée par l'algorithme pour résoudre le problème et améliorer la précision. Avec la même couleur que ces subdivisions, on trace également les sous-intervalles données en sortie de la proposition 4.2.

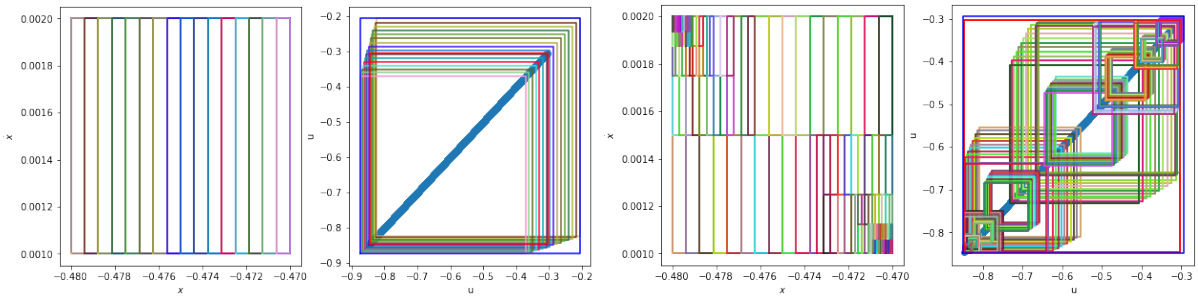
La figure 16 montre les sorties de l'algorithme dans le cas de la voiture, pour deux intervalles de taille différente. Pour l'intervalle plus grand, on constate que les sorties du contrôleur sont plus écartées et occupent une plus grande partie de l'espace des actions alors que lorsque l'intervalle est petit, le scénario est beaucoup plus spécifique et l'intervalle des actions est quasi-nul. Dans les deux cas, la précision de l'algorithme est très satisfaisante.

Afin d'observer l'importance de la valeur de la précision ϵ dans cet algorithme, on représente



(a) Intervalle large (b) Intervalle petit

FIGURE 16 – Atteignabilité du réseau pour la voiture



(a) $\epsilon = 0.001$ (b) $\epsilon = 0.0001$

FIGURE 17 – Effet d'ϵ sur la précision

les sorties de l'algorithme pour le même intervalle d'entrée, mais avec ϵ qui change. Dans la figure 17a, $\epsilon = 0.001$ et la sur-approximation est trop conservative, avec un écart d'environ 0.1 par rapport à l'intervalle simulé en rouge. Dans la figure 17b, on diminue cette valeur, et la prédiction est quasiment la même que l'intervalle réelle. L'idée lors de l'utilisation de cet algorithme sera alors de trouver le meilleur compromis entre le temps d'exécution et la précision voulue.

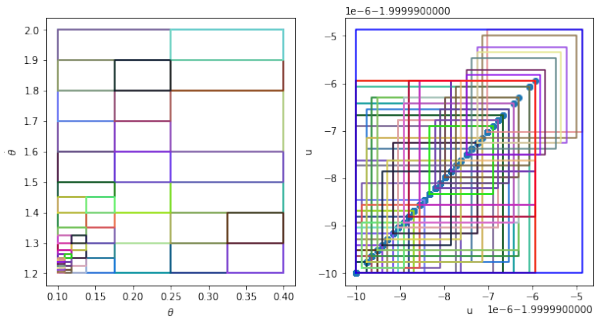


FIGURE 18 – Atteignabilité du réseau pour le pendule

Nous représentons également les résultats obtenus dans le problème du pendule à la figure 18 afin de montrer que les deux problèmes sont résolus de manière satisfaisante par cet algorithme.

4.4 Algorithmes alternatifs

La version de l'algorithme présentée ci-dessus permet de contrôler la taille de division que l'on se permet sur l'intervalle des états. Cela signifie que l'on ne contrôle l'erreur de sur-approximation qu'indirectement. Une solution est de modifier légèrement l'algorithme et de remplacer ϵ par une constate δ qui représente l'erreur de sur-approximation. Ainsi, la condition pour vider la boucle est d'obtenir des intervalles dont l'erreur avec l'intervalle simulé est inférieure à δ, sinon on continue à diviser l'intervalle des états. Cette erreur est calculée en mesurant la taille de l'intervalle obtenu par intersection entre l'intervalle simulée et l'intervalle calculée (convention : longueur nulle pour l'ensemble vide). Si toutes les subdivisions de l'intervalle de départ permettent d'obtenir une erreur inférieure à cette valeur, alors nécessairement l'union des intervalles images aura également une erreur inférieure à δ.

Cela permet directement de prévoir l'erreur que l'on aura, sans connaître les paramètres exacts du problème (taille des intervalles d'état classiques etc). En s'appuyant sur les résultats observés avec l'influence d'ϵ sur cette erreur, nous savons que l'algorithme terminera (l'erreur étant proportionnelle à la taille de l'intervalle).

Nous pouvons également, afin de garantir que l'algorithme ne passe pas trop de temps à résoudre le problème ajouter des conditions de time-out, où il suffit juste de renvoyer l'intervalle approché au bout de t secondes si le calcul n'est toujours pas terminé. On peut également renvoyer directement le premier résultat si l'intervalle simulée possède une taille plus grande qu'une tolérance $\epsilon_{actions}$ donnée en entrée. On verra plus loin que ces conditions permetteront d'accélérer le calcul de l'atteignabilité en boucle fermée.

5 Atteignabilité du système dynamique

5.1 Arithmétique d'intervalles

Il existe plusieurs manières d'approcher l'ensemble atteignable d'un système. Nous allons employer une méthode basée toujours sur des ensembles d'entrée et de sortie qui sont des intervalles. L'intérêt et de pouvoir y intégrer les résultats de l'atteignabilité du réseau de neurones lors du calcul et de garder une méthode simple et rapide du point de vue de la computation. Nous allons alors utiliser ici la méthode d'arithmétique d'intervalle [8].
L'arithmétique d'intervalle consiste à définir des opérations sur des intervalles unidimensionnels, puis de généraliser ces opérations à des intervalles de plus grande dimension en effectuant les opérations composante par composante.
Pour des intervalles unidimensionnels $[\underline{x}, \overline{x}]$, $\left[\underline{y}, \overline{y}\right]$ on définit

$$\lambda\,[\underline{x}, \overline{x}] = \begin{cases} [\lambda\underline{x}, \overline{x}] & \text{si } \lambda \geq 0 \\ [\lambda\overline{x}, \underline{x}] & \text{si } \lambda < 0 \end{cases} \tag{14}$$

$$[\underline{x}, \overline{x}] + \left[\underline{y}, \overline{y}\right] = \left[\underline{x} + \underline{y}, \overline{x} + \overline{y}\right] \tag{15}$$

$$[\underline{x}, \overline{x}] - \left[\underline{y}, \overline{y}\right] = \left[\underline{x} - \overline{y}, \overline{x} - \underline{y}\right] \tag{16}$$

$$[\underline{x}, \overline{x}] \cdot \left[\underline{y}, \overline{y}\right] = \left[\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})\right] \tag{17}$$

$$[\underline{x},\overline{x}]^{-1} = \begin{cases} [-\infty,\infty] & \text{si } \underline{x} < 0 \text{ et } \overline{x} > 0 \\ \left[-\infty, \frac{1}{\underline{x}}\right] & \text{si } \overline{x} = 0 \\ \left[\frac{1}{\overline{x}}, \infty\right] & \text{si } \underline{x} = 0 \\ \left[\frac{1}{\overline{x}}, \frac{1}{\underline{x}}\right] & \text{si } \underline{x}\overline{x} > 0 \\ \emptyset & \text{si } \underline{x} = \overline{x} = 0 \end{cases} \tag{18}$$

$$[\underline{x},\overline{x}] \,/\, \left[\underline{y},\overline{y}\right] = [\underline{x},\overline{x}] \cdot \left[\underline{y},\overline{y}\right]^{-1} \tag{19}$$

$$\begin{aligned} \cos([\underline{x},\overline{x}]) = [\underline{a},\overline{a}]\,, \text{ où } \underline{a} &= \begin{cases} -1 & \text{si } \exists k \in \mathbb{Z} | (2k\pi + \pi) \in [\underline{x},\overline{x}] \\ \min(\cos(\underline{a}), \cos(\overline{a})) & \text{sinon} \end{cases} \\ \overline{a} &= \begin{cases} 1 & \text{si } \exists k \in \mathbb{Z} | (2k\pi) \in [\underline{x},\overline{x}] \\ \max(\cos(\underline{a}), \cos(\overline{a})) & \text{sinon} \end{cases} \end{aligned} \tag{20}$$

Pour $\phi : \mathbb{R} \to \mathbb{R}$ monotone et continue,

$$\phi([\underline{x},\overline{x}]) = \begin{cases} [\phi(\underline{x}), \phi(\overline{x})] & \text{si } \phi \text{ est croissante} \\ [\phi(\overline{x}), \phi(\underline{x})] & \text{si } \phi \text{ est décroissante} \end{cases} \tag{21}$$

De même pour la fonction sinus, puissance etc. Nous pouvons maintenant calculer l'intervalle de sortie pour toute série d'opérations invoquant un nombre fini de fois ces opérateurs. L'avantage de cette méthode est quelle permet de rapidement obtenir une extension aux intervalles optimale, mais elle ne prend pas en compte les corrélations entre plusieurs variables. Par exemple dans le cas où $x \in [0,1]$, et la fonction $f : x \mapsto x - x$, l'arithmétique d'intervalle renvoie $[-1,1]$ au lieu de $\{0\}$.
Pour des fonctions à valeurs dans $\mathbb{R}^m$ séparables, pour calculer l'intervalle de sortie, on calcule l'intervalle pour chaque dimension $\left[\underline{a_i}, \overline{a_i}\right]$ puis on concatène pour obtenir $\left[\underline{a_1}, \overline{a_1}\right] \times \ldots \times \left[\underline{a_n}, \overline{a_n}\right]$.

5.2 Arithmétique d'intervalles appliquée au système

Nous rappelons la forme de la représentation d'état du système en équation 23.

$$\begin{aligned} F : \mathbb{R}^n \times \mathbb{R}^m &\to \mathbb{R}^n \\ (x(k), u(k)) &\mapsto F(x(k), u(k)) = x(k+1) \end{aligned}$$

Appliquons cette méthode à un système. La fonction F de sa dynamique étant séparable, on définit une extension aux intervalles

$$\begin{aligned} [F] : \mathbb{IR}^n \times \mathbb{IR}^k &\to \mathbb{IR}^n \\ ([x(k)], [u(k)]) &\mapsto [x(k+1)] \end{aligned}$$

où $\forall i \in [\![1, n]\!]$, $[\underline{x}(k+1), \overline{x}(k+1)] = [F_i]([x(k)], [u(k)])$ est obtenue par arithmétique d'intervalles.
Notons que nous présentons tous les algorithmes en supposant que le vecteur des observations du système est le même que le vecteur d'état, mais il est possible d'adapter les algorithmes lorsqu'ils sont différents.

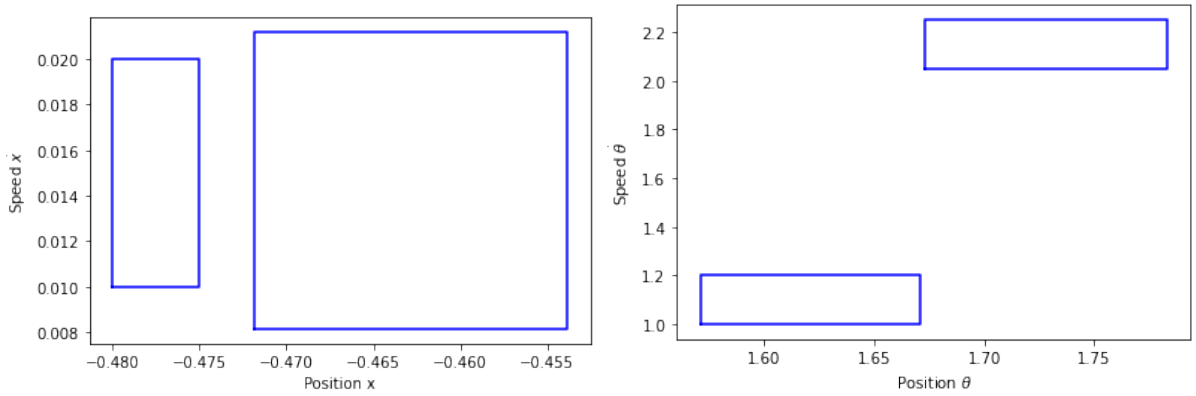
Algorithm 2 Atteignabilité d'un système discret contrôlé par un réseau de neurones

Input : Réseau de neurone Φ, Extension aux intervalles de la dynamique du système $[F]$, l'ensemble de départ du vecteur d'état $\mathcal{A}$, ϵ la précision de l'atteignabilité du réseau de neurone, N le nombre de simulations à effectuer pour le réseau de neurones

Output : Ensemble atteignable au bout d'une itération

1: Calculer le plus petit $[x(0)]$ tel que $\mathcal{A} \subset [x(0)]$
2: Calculer $[u(0)]$, une sur-approximation de l'ensemble atteignable du réseau de neurones à partir de $[x(0)]$
3: Calculer $[x(1)] = [F]([x(0)], [u(0)])$
4: **return** $[x(1)]$

On analysera les performances de cette méthode dans la partie suivante. En effet, il est difficile de conclure sur la précision de l'approximation en faisant des tests individuels sur une itération du système dynamique car il risque de ne pas se passer grand chose de visible. On cherchera à voir si la méthode permet d'observer des changements cohérents sur plusieurs itérations, c'est-à-dire que l'angle du pendule se stabilise bien pour plusieurs itérations par exemple dans le cas où les résultats de l'atteignabilité du réseau de neurones sont eux aussi satisfaisants. Voici par exemple deux exemples de déplacement des systèmes pour la voiture et pour le pendule.
Notons que l'on utilise dans cet algorithme un intervalle issu de la sur-approximation de l'ensemble renvoyé par l'algorithme d'atteignabilité du réseau de neurones. L'objectif est de pouvoir l'utiliser et effectuer l'arithmétique d'intervalle. Dans le cas où cette solution n'est pas satisfaisante, il faudrait trouver une alternative à l'arithmétique d'intervalle. On peut également récupérer une réunion d'intervalles en sortie de l'algorithme 1 plutôt qu'un ensemble et ainsi itérer l'algorithme 2 sur chacun de ses intervalles (la complexité risque cependant d'exploser).



(a) Mountain car

(b) Pendule

FIGURE 19 – Intervalle de départ et intervalle d'arrivée au bout d'une itération

On représente alors les intervalles obtenus par arithmétique d'intervalle appliquée à un intervalle de départ spécifié en entrée et l'intervalle des actions prises par le réseau de neurones. On constate bien que l'on est capable, au bout d'une itération, de prévoir que l'on va totalement sortir de l'intervalle de départ dans les deux problèmes. Dans le cas de la voiture, elle se trouve à gauche et possède une vitesse positive, et le contrôleur ne modifie pas suffisamment sa vitesse, ce qui explique que le nouvel intervalle se situe à la droite du premier. Dans le cas du pendule, c'est à la fois la vitesse angulaire et l'angle qui augmentent à l'itération suivante, ce qui peut s'expliquer par le fait que $\frac{3}{2}\pi \geq \theta \geq \pi/2$, et donc la gravitation accélère le pendule, en plus de l'action du contrôleur qui consiste à légèrement augmenter la vitesse autour de cette position.

6 Contrôle garanti du système

6.1 Contrôle garanti

Plusieurs stratégies peuvent désormais être adaptées, que l'on pourra choisir en fonction du problème à résoudre et des spécifications du système à vérifier. Une manière naïve serait par exemple de donner tout l'intervalle sur lequel le système a été entraîné en entrée de l'algorithme 2, et d'itérer cette algorithme sur plusieurs itérations en espérant que l'estimation soit incluse dans l'ensemble vérifiant les spécifications. Cependant, cette méthode ne risque pas de fonctionner pour la majorité des problèmes. En effet, l'intervalle de départ étant généralement trop large, en fonction du point de départ, le contrôleur adoptera des stratégies différentes que l'on ne pourra jamais projeter sur un intervalle mobile à chaque itération. Pour mieux illustrer ce problème, on peut penser à la voiture avec un intervalle de départ contenant un point où la voiture est à droite de la pente avec une vitesse vers la gauche et un point où elle garde la même vitesse est à gauche de la pente. Le contrôleur voudra renvoyer une valeur -1 dans le premier cas et $+1$ dans le deuxième cas, ce qui donne un intervalle d'estimation bien plus grand que celui de départ. La conclusion est que notre intervalle ne risque pas de converger. Ce problème est à la fois dû à la nature du problème, mais également à la naïveté de l'algorithme. Elle est à différencier des erreurs de précision qui interviennent lorsque ϵ n'est pas assez grand où lorsque des corrélations entre variables augmentent l'imprécision plus que nécessaire.

L'idée pour le résoudre est de séparer notre problème en différents scénarios en divisant l'intervalle de départ et le rendant aussi petit que nécessaire. En effet, on cherche les valeurs de l'intervalle de départ qui font que le contrôleur adopte des stratégies similaires, c'est-à-dire pour lesquelles les valeurs, à chaque itération, renvoyé par le contrôleur sur tout l'intervalle sont incluses dans un intervalle bien plus petit que l'espace des actions. Ensuite, il suffit de partitionner l'intervalle sur lequel nous souhaitons vérifier la spécification et la vérifier sur chaque partie de cette partition.

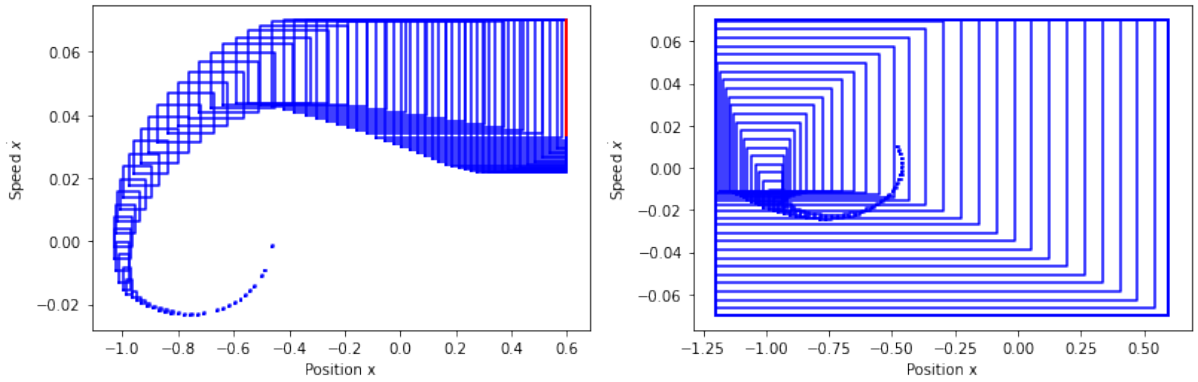
Nous pouvons également essayer d'exploiter, comme expliqué plus haut, le fait que l'ensemble d'atteignabilité du réseau de neurones n'est pas un intervalle mais une union d'intervalles et accélérer la dynamique du système pour éviter de faire exploser la dynamique du système.

6.2 Résultats

Nous exposerons ici les valeurs des d'intervalles ayant permis de prouver la capacité du contrôleur à vérifier les objectifs du problème, c'est à dire dépasser le drapeau pour le chariot et se stabiliser à la verticale ascendante pour le pendule. L'idée est d'exposer les limites du modèle et les difficultés que l'on a observé.

Chaque rectangle correspond à l'intervalle estimé d'atteignabilité à une itération k. La taille de l'intervalle a tendance a grossir avec le nombre d'itérations effectués car le nombre de sorties possibles par le réseau de neurone augmente et l'incertitude sur les sorties le fait également grossir. On constate dans le cas de la voiture que la taille de l'intervalle permettant de prouver le problème est très fortement réduite 20a (la ligne rouge correspond à l'accomplissement de la spécification qui est que la position dépasse 0.60 à une certaine itération), car dès que l'on fait bouger légèrement cet intervalle, les erreurs de sur-approximation de l'atteignabilité du réseau de neurones augmente la taille de l'intervalle jusqu'à la faire exploser et perdre totalement la précision. La solution est alors soit de réduire ϵ, soit de séparer l'intervalle en sous-intervalles ayant environ la même taille que le plus grand intervalle permettant de vérifier la spécification dans le cas où la première solution n'est pas satisfaisante ou est trop lente.

Pour le problème du pendule (figure 21), même en prenant des intervalles de départ très petits, l'algorithme prévoit des intervalles de taille raisonnable au début mais qui augmente



(a) $x(0) \in [-0.48, -0.4795], \dot{x}(0) \in [0.01, 0.0101]$ (b) $x(0) \in [-0.48, -0.475], \dot{x}(0) \in [0.01, 0.0101]$

FIGURE 20 – Contrôle garanti de la voiture sur 100 itérations

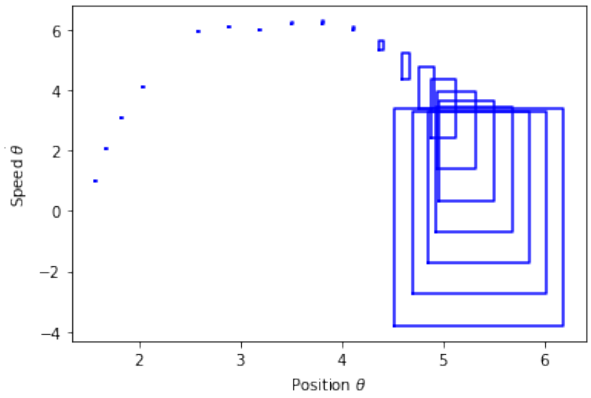


FIGURE 21 – Intervalles atteignables par le pendule au bout de 15 itérations

légèrement, jusqu'à commencer à exploser exponentiellement et remplir tout l'espace de la variable d'état. Même en modifiant la précision de l'atteignabilité du réseau de neurones, le même problème est visible. Il ne semble donc pas possible de garantir la performance du contrôleur dans ce cas.

6.3 Algorithme d'atteignabilité

En observant les résultats ci-dessus, on comprend que cette manière d'itérer sur les intervalles n'aboutira jamais à une preuve de stabilité car la taille de l'intervalle ne fait qu'augmenter de manière exponentielle à cause de la propagation des incertitudes. Nous proposons une solution pour remédier à ce problème. Constatons d'abord que ces incertitudes sont dues au fait que les méthodes d'atteignabilité du contrôleur et du système dynamique sont indépendantes. Ainsi, quelles que soit les méthodes employées dans ces deux étapes, lorsque l'on calcule l'intervalle des états à l'itération suivante à partir d'un intervalle des états précédents et un intervalle d'actions, on l'obtient en supposant que toutes ces actions peuvent être appliquées partout dans l'intervalle des états. Or ce n'est pas le cas en réalité. Considérons le système suivant :

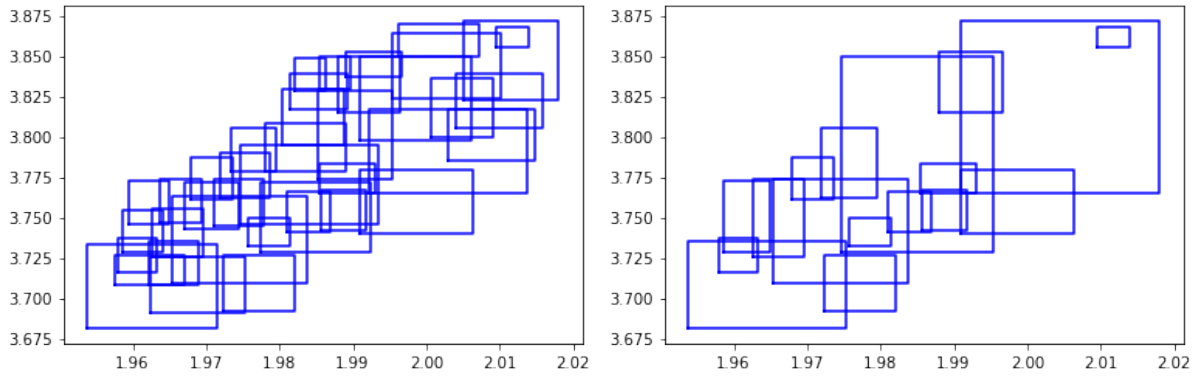
$$x(k+1) = x(k) + u(k)$$

On prend l'exemple d'un contrôleur qui renvoie une action de $+1$ lorsque le système est dans un état $x \leq 0$ et -1 sinon. Dans ce cas, si on considère notre système à l'état $x \in [-1, 1]$, on dira que le système sera à l'état $x \in [-2, 2]$ en calculant les atteignabilités indépendamment alors qu'en réalité, le système sera toujours à l'état $x \in [-1, 1]$ à l'itération suivante.

L'idée est alors d'imposer une taille maximale d'introduire une constante $\epsilon_{actions}$ qui définit une taille maximale que l'on permet à l'intervalle des actions de prendre. Si la condition n'est pas satisfaite, alors on divise l'intervalle des états en deux et on répète indépendamment l'analyse. Cela permet ainsi de contrôler la taille de l'erreur obtenue par le phénomène décrit ci-dessus. En effet, en imposant que l'intervalle des actions soit petit, on s'efforce à garder des intervalles d'état donc les actions sont proches. C'est la raison pour laquelle nous avons besoin de la condition de régularisation introduite en sous-section 2.6. En effet, cela nous permet de nous assurer qu'en subdivisant notre intervalle des états, nous allons parvenir à trouver des intervalles d'actions suffisamment petits.

Dans l'exemple présenté ci-dessus, en prenant $\epsilon_{actions} = 0.5$, on arrive à la subdivision $[-1, 0]$ et $[0, 1]$ associée aux actions $\{+1\}$ et $[-1, 1]$ respectivement, ce qui permet déjà de mieux approcher l'atteignabilité du système si l'opération est répétée. On se doute qu'en utilisant la méthode ci-dessus, l'algorithme ne terminera jamais du fait de la discontinuité. On rajoute alors une condition de terminaison imposée sur la taille de l'intervalle des états qui permet de stopper la subdivision lorsqu'il atteint une taille suffisamment petite.

Il s'agit ensuite de traiter les petits intervalles obtenus par la suite. Nous proposons de les réunir partiellement en utilisant le critère d'Intersection over Union (IoU). À partir d'un seuil défini entre 0 et 1, on réalise l'union de tous les intervalles ayant un IoU supérieur à ce seuil. Cela permet de fortement réduire la complexité de l'algorithme et réunir les divisions ayant menés à des intervalles proches. La figure 22 montre dans le cas d'une itération ce que l'on obtient en appliquant l'algorithme présenté ci-dessus puis la manière dont cette réunification permet de simplifier les calculs.



(a) Avant le critère

(b) Avec le critère (seuil = 0.15)

FIGURE 22 – Intervalles obtenus après divisions de l'intervalle de départ (critère IoU)

Algorithm 3 Atteignabilité en boucle fermée

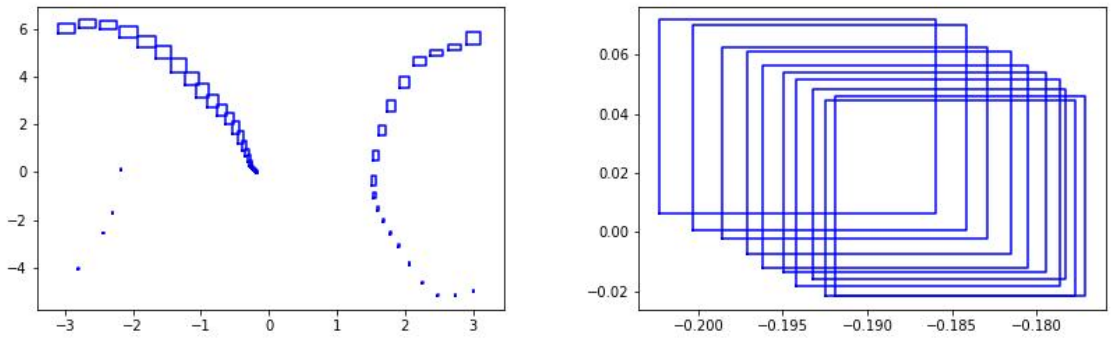
Input : Nombre d'itérations T, Réseau de neurones (contrôleur) Φ, Fonction de dynamique du systèmes par arithmétique d'intervalles $F = [f]$, Intervalle de départ η, δ erreur maximale tolérée pour l'atteignabilité du réseau de neurones, $\epsilon_{actions}$

Output : u, ensemble d'intervalles dont l'union donne la sur-approximation de l'ensemble atteignable par le système au bout de T itérations

Initialiser $\Lambda(0) = [\eta]$
for t in $[1, T]$ **do**
    Initialiser $\Lambda(t)$, liste vide
    while $\Lambda(t-1)$ non vide **do**
        Pop η de $\Lambda(t-1)$
        if $\rho([\Phi(\eta)]) \leq \epsilon_{actions}$ **then**
            Ajouter $[\Phi](\eta)$ à $\Lambda(t)$
        else
            Diviser η en η_1 et η_2
            Ajouter η_1 et η_2 à $\Lambda(t)$
        end if
    end while
    Réunir les intervalles proches dans $\Lambda(t)$ en utilisant le critère IoU avec un seuil prédéfini
end for
return $\Lambda(T)$

6.4 Résultats finaux

Nous appliquons finalement l'algorithme précédent aux problèmes de stabilité qu'il était impossible de résoudre avec l'algorithme précédent. La partie la plus difficile ici est de trouver les paramètres optimaux à choisir afin de ne pas faire exploser la taille des intervalles estimés, tout en garantissant un temps satisfaisant pour la preuve d'atteignabilité et éviter des calculs inutiles (ne pas prendre un $\epsilon_{actions}$ et un δ trop petits). Nous utilisons, pour le pendule, le contrôleur entraîné avec la condition de continuité.

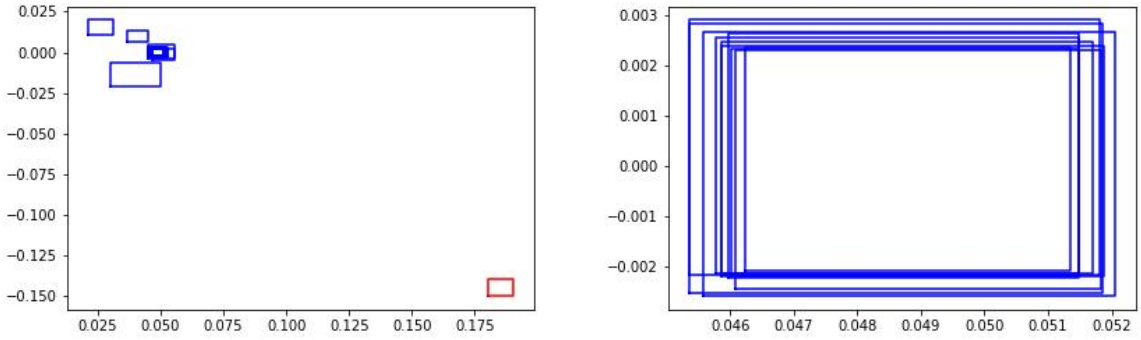


(a) Itérations 1 à 80

(b) Itérations 70 à 80

FIGURE 23 – Preuve de stabilité du pendule pour $\theta(0) \in \left[\pi, \pi + 10^{-4}\right], \dot{\theta}(0) \in \left[1, 1 + 10^{-4}\right]$ ($\epsilon = 0.2$)

La figure 23 prouve alors que le contrôleur DDPG entraîné sur le pendule est bien correct sur le petit intervalle mentionné. Étant donné que cette preuve à eu besoin d'une dizaine de minutes, il est difficile de vérifier la spécification sur un intervalle plus grand.



(a) Itérations 1 à 30

(b) Itérations 20 à 30

FIGURE 24 – Double intégrateur - $x_1(0) \in [0.18, 0.19], x_2(0) \in [-0.15, -0.14]$ ($\epsilon = 0.001$)

Pour le double intégrateur, il est plus facile de prouver la stabilité sur un intervalle plus grand avec cet algorithme. La preuve dans ce cas ce fait en moins d'une minute. Cette performance montre que pour des contrôleurs suffisamment continus, et un choix des paramètres optimaux, il est possible d'améliorer les performances obtenues avec cet algorithme.

6.5 Stabilité par analyse de Lyapunov

Dans cette partie, nous cherchons à mettre en oeuvre une nouvelle méthode permettant de prouver la stabilité d'un système linéaire à temps discret en utilisant l'approche présentée par A. Saoud [9]. Cette dernière est basée sur la théorie de stabilité de Lyapunov, qui consiste à chercher une fonction quadratique V agissant sur l'espace des états et étant décroissante. La difficulté dans le cas des systèmes introduits dans ce projet vient du fait que l'entrée est déterminée par un réseau de neurone rendant la recherche d'une telle fonction plus difficile. Prenons alors un système dynamique caractérisé par la dynamique :

$$x(k+1) = Ax(k) + B\Phi(x(k)), \text{ où } \Phi \text{ est le contrôleur réseau de neurones} \tag{22}$$

Proposition 6.1 (Stabilité de Lyapunov)**.** *Pour que le système soit stable (converge vers l'état nul lorsque $t \to \infty$), il suffit qu'il existe $P \in \mathbb{R}^{n\times n}$, symétrique, telle que la fonction V définie par :*

$$\begin{aligned} V : X &\to \mathbb{R} \\ x &\mapsto x^T P x \end{aligned} \tag{23}$$

où X est l'ensemble des états atteignables, est positive et strictement décroissante.

$$\begin{aligned} V(x(k+1)) - V(x(k)) &= (x(k)^T A^T + u(k)^T B^T) P(Ax(k) + Bu(k)) - x(k)^T P x(k) \\ &= x(k)^T (A^T P A - P) x(k) + u(k)^T B^T P A x(k) \\ &+ x(k)^T A^T P B u(k) + u(k)^T B^T P B u(k) \end{aligned}$$

Nous cherchons ainsi $P \in \mathbb{R}^{n\times n}$, matrice négative (ie $\forall x \in \mathbb{R}^n, x^T P x < 0$), telle que

$$x^T (A^T P A - P) x + \Phi(x)^T B^T P A x + x^T A^T P B \Phi(x) + \Phi(x)^T B^T P B \Phi(x) < 0$$

En utilisant les formules de la proposition 4.2, et en prenant la sur-approximation par un intervalle $\mathcal{H}_X$ de l'ensemble X, on désigne l'intervalle $[\gamma] = \left[\underline{\gamma}, \overline{\gamma}\right]$, l'intervalle obtenu tel que

$\mathcal{H}_X \subset \left[\underline{\gamma}, \overline{\gamma}\right]$ Nous voulons ainsi à simplifier cette expression en perdant la dépendance à $\Phi(x)$, pour pouvoir se ramener à une inégalité matricielle. Notons $\xi = x^T A^T PB$, et $\mu = B^T PB$, pour tout $\in X$. On définit les fonctions :

$$f : (\xi, [\gamma])_i = \begin{cases} \underline{\gamma}_i & \text{si } \xi_i > 0 \\ \overline{\gamma}_i & \text{sinon} \end{cases}$$

$$g : (\mu_{i,j}, \gamma) = \begin{cases} \mu_{i,j} \max(\underline{\gamma}_i \overline{\gamma}_j, \underline{\gamma}_i \underline{\gamma}_j, \overline{\gamma}_i \overline{\gamma}_j, \overline{\gamma}_i \underline{\gamma}) & \text{si } \xi_i > 0 \\ \mu_{i,j} \min(\underline{\gamma}_i \overline{\gamma}_j, \underline{\gamma}_i \underline{\gamma}, \overline{\gamma}_i \overline{\gamma}_j, \overline{\gamma}_i \underline{\gamma}) & \text{sinon} \end{cases}$$

On a alors :

$$x^T (A^T PA - P)x + \Phi(x)^T B^T PAx + x^T A^T PB\Phi(x) + \Phi(x)^T B^T PB\Phi(x) < 0$$

$$x^T (A^T PA - P)x + f(\xi, [\gamma])^T B^T PAx + x^T A^T PBf(\xi, [\gamma]) + \sum_{i=1}^{n} \sum_{j=1}^{n} g(\mu_{i,j}, \gamma) < 0$$

Comme il est nécessaire de connaître la matrice P pour déterminer les valeurs des fonctions f et g, on résout le problème en essayant toutes les combinaisons des valeurs possibles pour γ prises dans ces fonctions. Cela fait au total (choix des valeurs dans f, puis dans g) $2^m \times 2^{2m}$ possibilités. Dans le cas de problèmes simples où l'espace des actions ne possède qu'une dimension, il est facile d'appliquer cette méthode, cependant, la difficulté explose rapidement avec m. Voici le système final à résoudre : Trouver $P \in \mathbb{R}^{n \times n}$, symétrique, telle que, en posant $Y^T = \begin{bmatrix} x^T & 1 \end{bmatrix}$,

$$\forall x \in [x], Y \begin{bmatrix} A^T PA - P & A^T PBf(\xi, [\gamma]) \\ f(\xi, [\gamma])^T B^T PA & \sum_{i=1}^{n} \sum_{j=1}^{n} g(\mu_{i,j}, [\gamma]) \end{bmatrix} Y^T < 0 \tag{24}$$

Conclusion

Pour conclure, ce projet a permis de montrer qu'il est possible de garantir le contrôle de systèmes dynamiques représentés par des réseaux de neurones, que ce soit pour des problèmes d'atteignabilité, ou plus difficilement, pour des problèmes de stabilité. Nous avons montré, qu'en rajoutant une régularisation continue au cours de l'entraînement du réseau de neurones DDPG, nous parvenons à faciliter les preuves de garanties. Cependant, il est toujours difficile de garantir certains contrôleur sur des intervalles assez larges, malgré l'algorithme d'atteignabilité en boucle fermé que nous avons proposé. On peut penser à faire varier les valeurs de $\epsilon_{actions}$ en fonction des itérations, ou des intervalles considérés à un instant précis pour accélérer le temps de calcul des preuves. Cela créerait cependant d'avantage de paramètres à calibrer, spécifiques à chaque problème. Ce projet a également permis d'approcher un contrôleur MPC explicit par un réseau de neurones, tout en prouvant que cette approximation est correcte. Nous avons implémenté l'ensemble des techniques présentés dans ce rapport sur Python, et avons abouti à la création d'une librairie d'entraînement de contrôleurs DDPG, et de vérification de l'atteignabilité de systèmes contrôlés par des réseaux de neurones présente sur GitHub : https://github.com/Ramlaoui/guaranteed-nn-control.

A Appendix

Algorithme DDPG

Initialisation :

1. On initialise une mémoire qui va constituer les données d'entraînement, avec une taille de 20000 actions, et qui se vide avec le principe du premier arrivé, premier sorti. On va la remplir à chaque nouvelle transition.
2. Nous construisons un réseau de neurones pour le modèle Acteur et un réseau de neurones pour le *target actor*.
3. Nous construisons un réseau de neurones pour le modèle Critique et un réseau de neurones pour la *target critic*.

Processus de formation - Nous exécutons un épisode complet avec 10 000 premières actions jouées au hasard, puis avec des actions jouées par le modèle Acteur. Puis nous répétons les étapes suivantes :

4. Nous échantillonnons un lot de transitions (s, s', a, r) à partir de la mémoire. Puis pour chaque élément du batch :
5. A partir de l'état suivant s', la cible Acteur joue l'action suivante a'
6. Nous ajoutons le bruit d'Orstein-Uhlenbeck à cette action suivante a' et nous le calons dans une plage de valeurs supportées par l'environnement.
7. La cible critique prend le couple (s', a') en entrée et renvoie la valeur Qt1(s',a') en sortie.
8. Nous obtenons la cible finale du modèle critique, qui est : Qt=r + γ*Qt1, où γ est le facteur d'actualisation.
9. Le modèle critique prend le couple (s, a) en entrée et renvoie la valeur Q Q1(s, a) en sortie.
10. Nous calculons la perte issue du modèle critique : critique Loss = MSELoss(Q1 (s,a), Qt)
11. On fait de la backpropagation du réseau critique et on met à jour les paramètres du modèle critique avec un optimiseur SGD.
12. Nous mettons à jour notre modèle Actor en effectuant une ascension de gradient sur la sortie du modèle critique : $\nabla_{\Phi} J(\Phi) = N^{-1} \sum \nabla_a Q_{\Theta 1}(s,a) \mid_{a=\Pi_{\Phi}(s)} \nabla_{\Phi} \Pi_{\Phi}(s)$ Où ϕ et $\theta 1$ sont resp. les poids de l'Acteur et du Critique.
13. Nous mettons à jour les pondérations du *target actor* en faisant la moyenne polyak :$\Theta_i^{'} \longleftarrow \tau\Theta_i + (1-\tau)\,\Theta_{i'}$.
14. Nous mettons à jour les pondérations du *target critic* en faisant la moyenne des polyak :$\Phi i^{'} \longleftarrow \tau\Phi i + (1-\tau)\,\Phi i^{'}$.

Références

[1] A. Corso and M. J. Kochenderfer, "Interpretable safety validation for autonomous vehicles," *CoRR*, vol. abs/2004.06805, 2020. [Online]. Available : https://arxiv.org/abs/2004.06805

[2] T. Lecomte, T. Servat, and G. Pouzancre, "Formal methods in safety-critical railway systems," 08 2007.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. [Online]. Available : https://arxiv.org/abs/1606.01540

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015. [Online]. Available : https://arxiv.org/abs/1509.02971

[5] Q. Shen, Y. Li, H. Jiang, Z. Wang, and T. Zhao, "Deep reinforcement learning with smooth policy," *CoRR*, vol. abs/2003.09534, 2020. [Online]. Available : https://arxiv.org/abs/2003.09534

[6] A. Bemporad, "Hybrid toolbox for matlab - user's guide," 2003.

[7] W. Xiang, H.-D. Tran, X. Yang, and T. T. Johnson, "Reachable set estimation for neural network control systems : A simulation-guided approach," 2020. [Online]. Available : https://arxiv.org/abs/2004.12273

[8] P. Meyer, A. Devonport, and M. Arcak, *Interval Reachability Analysis : Bounding Trajectories of Uncertain Systems with Boxes for Control and Verification*, ser. SpringerBriefs in Electrical and Computer Engineering. Springer International Publishing, 2021. [Online]. Available : https://books.google.fr/books?id=YG8WEAAAQBAJ

[9] A. Saoud, "Stability analysis using lmis for systems with neural network controllers."