Optimisation des gains d'un contrôleur de type PID avec l'apprentissage par renforcement

Emile Dimas[*], Steve Regis Koalaga[*]

[*] Génie Mécanique, Polytechnique Mécanique
[**] INF8225 : Intelligence Artificielle technique probabiliste et d'apprentissage

Résumé

Dans le domaine industriel, les contrôleurs de type PID sont couramment utilisés pour asservir différents systèmes. L'implémentation de ces types de contrôleurs est assez aisée, mais leurs performances dépendent fortement de la qualité du réglage de leurs gains. Il existe des méthodes d'optimisation de ces gains en utilisant le diagramme de Bode et le lieu de racine. Cependant ces méthodes ne sont pas adéquates lorsque le système et ces conditions d'utilisations ne sont pas entièrement connus. L'apprentissage par renforcement propose une alternative à ces méthodes classiques en proposant une nouvelle approche permettant d'optimiser les gains du contrôleur sans pour autant connaître entièrement le système et ses conditions d'utilisions. Dans ce papier, 3 algorithmes d'apprentissages que sont le Q-learning, Deep Q-learning et le Deep Deterministic Policy Gradient ont été utilisés afin d'optimiser des gains d'un contrôleur servant à asservir l'angle d'un avion de type A320. Le logiciel Matlab a été utilisé pour simuler la réponse de notre système à une entrée unitaire pour caractériser la performance du contrôleur. Les résultats des expériences donnent une idée sur les avantages que pourrait avoir l'apprentissage par renforcement par rapport aux méthodes classiques d'optimisation des gains des contrôleurs.

Mots clés- apprentissage par renforcement, PID, contrôle,

I. INTRODUCTION

Les contrôleurs de type PID (Proportionnel, intégral, dérivé) sont couramment utilisés dans le domaine industriel pour asservir les systèmes. Ces contrôleurs sont faciles à concevoir, mais leur performance est fonction des gains qui régissent leur fonctionnement. Il existe plusieurs méthodes pour déterminer ces gains, mais ces méthodes nécessitent la connaissance parfaite du système à contrôler et de ses conditions d'utilisations.
Dans ce rapport, il est question d'utiliser l'apprentissage par renforcement pour optimiser les paramètres d'un contrôleur de type PID afin de contrôler l'angle d'attaque d'un avion de type A320. En effet, l'apprentissage par renforcement appliquer au domaine de la robotique/du contrôle permet aux agents d'apprendre une police optimale en interagissant avec leur environnement. Ainsi, cette méthode est adéquate pour résoudre le problème d'optimisation des paramètres du contrôleur en ne connaissant pas entièrement l'environnement où évolue le système ou les conditions d'utilisations. Trois algorithmes d'apprentissages par renforcement que sont le Q-Learning, Deep Q-Learning, et le Deep Deterministic Policy Gradient (DDPG) ont été utilisé.
Ce rapport présente dans la Section 1 l'introduction au sujet de notre papier. Dans la section 2, une mise en contexte pour le domaine du contrôle est proposée afin d'avoir une bonne compréhension de la problématique. Dans la Section 3, la démarche scientifique ainsi que les différents algorithmes utilisés sont présentés et analysés. La Section 3 présente les limites de notre démarche. La conclusion est faite dans la section 4, où un retour sur l'apprentissage ainsi que les travaux futurs sont présentés. Le lien suivant permet d'accéder à l'entièreté du projet. `https://github.com/SteveR21/Projet-INF8225.git`

II. MISE EN CONTEXTE

Figure 1: Angle d'attaque d'un avion

Les élévateurs sont des surfaces de contrôle (en jaune sur la figure) qui permettent de contrôler l'angle d'attaque d'un avion.
En effet, en modifiant l'angle de ces parties mécaniques d'un angle δ, on peut changer l'angle d'attaque de l'avion α.
La relation qui relie les deux angles δ et α est décrite par l'équation du mouvement suivante (obtenue après simplification):

$$I_{YY}\ddot{\alpha} = Q \cdot S \cdot \bar{c} \cdot (C_{m_\alpha}\alpha + C_{m_{\dot{\alpha}}}\frac{\dot{\alpha}\bar{c}}{2V} + C_{m_q}\frac{\dot{\alpha}\bar{c}}{2V} + C_{m_{\delta_e}}\delta_e)$$

On peut alors trouver la fonction de transfert de notre système mécanique en appliquant la transformée de Laplace pour obtenir :

$$G(s) = \frac{\Delta\alpha}{\Delta\delta_e}(s) = \frac{C_0}{C_1 \cdot s^2 + C_2 \cdot s + C_3}$$

Avec

$$C_0 = Q.S.\bar{c}.C_{m\delta_e}$$
$$C_1 = I_{yy}$$

$$C_2 = -Q.S.\bar{c}.C_{m\dot{\alpha}} - Q.S.\bar{c}.C_{mq}\frac{q\bar{c}}{2V}$$
$$C_3 = -Q.S.\bar{c}.C_{m\alpha}$$

Il est nécessaire d'étudier la réponse du système et voir si ce dernier offre les caractéristiques voulues. En général, il faut avoir une réponse adéquate caractérisée par les valeurs suivantes :

- Erreur en régime permanent nulle
- Temps de réponse < 1s
- Dépassement de l'ordre de 15%

La réponse du système peut être obtenue à l'aide de la fonction « step() » et « stepinfo() » de Matlab. Les valeurs suivantes sont obtenues pour une réponse en boucle ouverte :

- Erreur en régime permanent = 2.1375
- Temps de réponse= 26.37s
- Dépassement = 64%

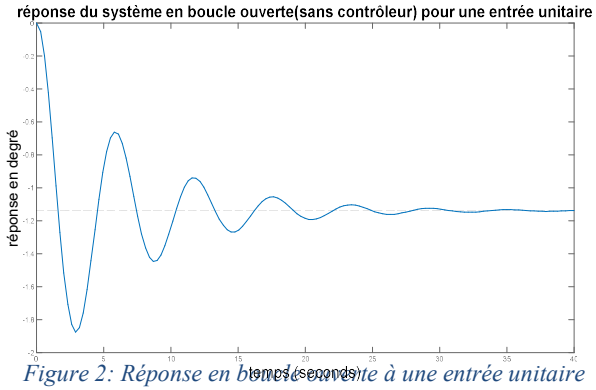

Figure 2: Réponse en boucle ouverte à une entrée unitaire

La figure ci-dessus représente la réponse α du système sans contrôleur pour une entrée à échelon unitaire. L'analyse de la figure montre que le système se stabilise à la mauvaise valeur et présente un très grand dépassement et temps de réponse.
Afin de pouvoir contrôler et corriger le système, un contrôleur très commun en robotique de type proportionnel, intégral, dérivé soit le contrôleur PID est implémenté. Ce type de correcteur est caractérisé par trois constantes appelées les gains : K_P, K_I et K_D et peut s'écrire sous la forme suivante :

$$C(s) = K_p + \frac{K_i}{s} + K_d.s$$

La fonction de transfert en boucle fermée est ainsi obtenue :

$$FTBF = \frac{C(s) * G(s)}{1 + C(s) * G(s)}$$

En sélectionnant ainsi les bons gains, le système peut être amené à avoir une réponse désirée respectant le cahier de charge.

Pour trouver ces gains, plusieurs méthodes sont utilisées. L'outil sisotool() de Matlab est souvent utilisé et permet d'ajuster les gains à la main afin de trouver une réponse qui respecte le cahier de charges. Cependant, vu qu'il existe une multitude de possibilités, cette méthode est souvent exhaustive. De plus, la complexité du modèle peut ajouter un degré de difficulté à cette méthode qui pourrait être optimisée.

C'est pour cela que l'apprentissage par renforcement est proposé. Ainsi, il serait juste demandé à l'opérateur de rentrer les caractéristiques voulues (temps de réponse, erreur en régime permanent,) et l'algorithme se chargerait de trouver les meilleurs gains possible du contrôleur.

III. APPRENTISSAGE PAR RENFORCEMENT

L'apprentissage par renforcement permet à un agent d'apprendre une police optimale en interagissant avec son environnement. Comme présenté sur la figure ci-dessous, l'agent prend une série d'actions A_t, pour lequel il reçoit une récompense R_t et un nouvel état S_{t+1} provenant de son environnement.

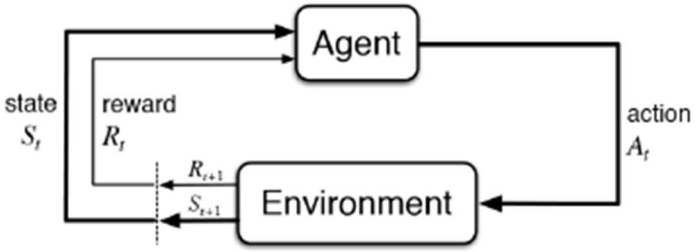

Figure 3: Architecture de l'apprentissage par renforcement

Pour l'étude suivante, l'agent choisit des valeurs de gains Kp, Kd et Ki aléatoirement. Ces valeurs sont envoyées à un programme Matlab qui simule le contrôle de l'angle d'attaque de l'avion à travers un « step response ». Après la simulation, les résultats les plus importants que sont : le temps de réponse, le dépassement(overshoot), et la réponse en régime permanent sont renvoyés aux algorithmes d'apprentissages.

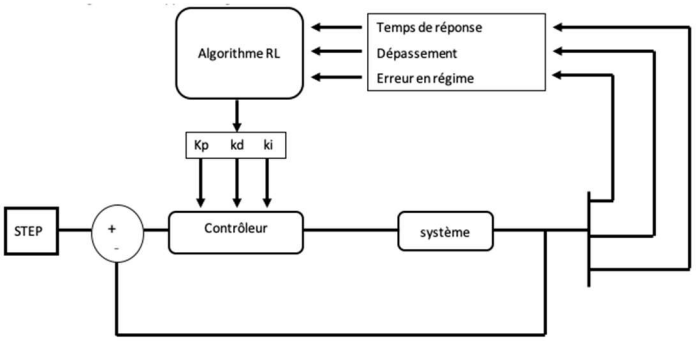

Figure 4 Structure du modèle de contrôle

Après avoir défini la structure du modèle de contrôle, trois algorithmes d'apprentissages différents ont été utilisés pour analyser leur performance quant à l'optimisation des gains du contrôleur.

A. Q-learning

Afin de pouvoir optimiser le contrôleur, il a été décidé d'implémenter tout d'abord un des plus simples algorithmes de renforcement soit le Q-learning. Cet algorithme d'apprentissage par renforcement basé sur les 'valeurs' est utilisé pour trouver la politique de sélection d'action optimale à l'aide d'une fonction Q. La première étape consiste à créer un tableau (la fameuse Q-table) où l'on trouve les « Q-value » selon l'action prise par l'agent à un état S_t. Les lignes représentent les états et les colonnes représentent les actions possibles. Ainsi, l'agent se trouvant dans un état S_t à un temps t effectue une action quelconque se retrouvant alors dans un état S_{t+1} et obtient donc une récompense. Le tableau est alors mis à jour au cours des itérations grâce à l'équation de Bellman :

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma * \max(Q'(s',a')) - Q(s,a)]$$

Avec:

- $Q(s,a)$: Q-value pour l'état s_t et l'action a
- α : Taux d'apprentissage
- $R(s,a)$: Récompense gagnée par l'agent à l'état s_t et l'action a
- γ : Facteur d'escompte
- $Q'(s',a')$: Q-value pour l'état s_{t+1} et l'action a'

Après avoir fini la mise à jour du tableau, il peut être utilisé afin de retrouver les meilleurs gains selon une initialisation aléatoire. La figure suivante résume l'algorithme de Q-Learning :

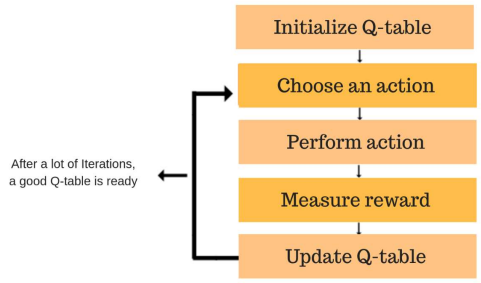

Figure 5 Algorithme de Q-learning

Configuration :

Vu que le « Q-learning », nécessite un environnement discontinu, une discrétisation des gains ainsi qu'une modification des actions est nécessaire.

Pour l'implémentation du Q-Learning, il a été décidé de représenter chaque état comme étant un vecteur des gains sous la forme [K_p , K_I , K_d]. Par exemple, la ligne 0 du tableau représente le vecteur [K_p =0, K_I=0, K_d=0]. Chacun des gains peut prendre une valeur entière entre (-25,25). En effet, une des solutions pour les gains est [-10, -10, -5]. L'intervalle des gains est restreint entre (-25,25) pour vérifier si l'algorithme converge vers une meilleure, pire ou même solution.

En outre, neuf actions sont possibles :

- Augmenter, diminuer ou garder constant K_P
- Augmenter, diminuer ou garder constant K_I
- Augmenter, diminuer ou garder constant K_D

La Q-table fait alors la taille suivante :
Taille = (51^3,9)=(132651,9)

La récompense gagnée par l'agent est le négatif de la valeur absolue de la différence entre les valeurs désirées et obtenues :

$$R = -|tr_{desiré} - tr_{obtenu}| - |ov_{desiré} - ov_{obtenu}| - |err_{desiré} - err_{obtenu}|$$

Avec :

- tr : Temps de réponse
- ov : Dépassement (overshoot)
- err : Erreur en régime permanent

Résultats :

Durant l'apprentissage, 10000000 d'itérations ont été effectué. Comme attendu, l'algorithme n'a pas abouti à de bons résultats. En effet, le tableau obtenu était à moitié vide. Cela est dû au nombre très important de possibilités par rapport au nombre d'itérations effectuées. Ainsi, on a remarqué qu'avec ce nombre d'itérations, le coût de calcul en temps était vraiment exhaustif. Augmenter le nombre d'itérations n'est donc pas envisageable. Une réduction de l'intervalle des gains a été envisagé. Cependant, le but initial était de pouvoir trouver les gains de n'importe quel système en utilisant une méthode simple, efficace et rapide. En diminuant l'intervalle des gains possibles, on ne peut pas généraliser la méthode. Une des autres limites du Q-Learning est la discrétisation de l'environnement. Ainsi, en discrétisant l'environnement, on saute des valeurs continues qui pourraient être optimales pour le système. On peut alors conclure que la méthode du Q-learning n'est pas le bon algorithme à appliquer pour cette application.

B. Deep Q-Learning

Le problème majeur avec le Q-Learning est la généralisation sur les états non vus par l'agent. Le Q-Learning peut être vu comme l'optimisation d'une matrice de dimensions deux (l'espace des états et des actions possibles) contenant les valeurs de Q pour chaque combinaison possible. Ainsi, l'agent du Q-learning ne peut pas estimer les valeurs Q des états qu'il n'a pas encore vues.

Avec le DQN, un réseau de neurones est utilisé pour estimer les valeurs Q à la place du tableau de deux dimensions. L'objectif qui est de minimiser la distance entre Q(s,a) et le TD-Target peut être représenté par la fonction de perte suivante :

$$L_i(\theta_i) = \mathbb{E}_{a\sim\mu}\left[(y_i - Q(s,a;\theta_i))^2\right]$$

$$\text{where } y_i := \mathbb{E}_{a'\sim\pi}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | S_t = s, A_t = a\right]$$

L'entrée du réseau est l'état actuel et la sortie les Q values correspondant à chacune des actions (discrètes) que peut prendre l'agent. Deux réseaux de neurones différents le Q-network dont les paramètres sont *θ(i) et* le TD-Network dont les paramètres

sont représentés par *θ(i-1)* sont utilisés. L'action de l'agent est choisie selon la police ***μ(a|s)*** et la police gloutonne ***π(a|s)***
Choisis l'action qui maximise la Q-value.

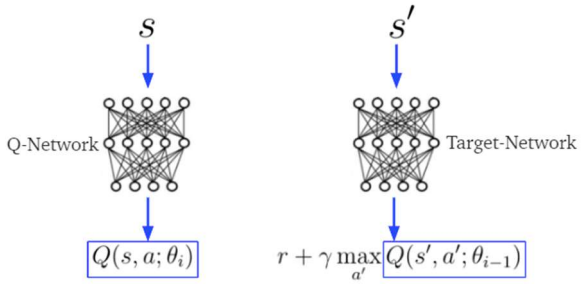

Figure 6: Réseaux de neurones Deep Q-learning

Les paramètres du « Q-network » sont optimiser par une *back-propagation* et celle du Target par une *soft update* suivant la formule suivante :

$$\theta(i\text{-}1)target = \tau*\theta(i) + (1 - \tau)*\theta(i\text{-}1)$$

L'utilisation de cette méthode permet de réduire le problème de surestimation des valeurs Q, permettant ainsi d'entraîner plus rapidement et d'avoir un apprentissage plus stable. La notion d'*Experience Replay est* utilisé afin de ne pas oublier les expériences précédentes et de réduire les corrélations entre les expériences.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Configuration :

Pour cet algorithme, la même modélisation que pour le Q-learning a été utilisé. Les états sont représentés par un vecteur des gains sous la forme [K_p, K_I , K_d] et il y a neuf actions discrètes possibles consistant à augmenter, dimère ou garder constants les gains individuellement. Nous avons également limité la valeur des gains dans un intervalle de (-25,25).

Table 1 : liste des 9 actions possibles

Actions	0	1	2	3	4	5	6	7	8
Kp	+1	-1	0	-	-	-	-	-	-
Ki	-	-	-	+1	-1	0	-	-	-
Kd	-	-	-	-	-	-	+1	-1	0

La fonction de récompense que nous avons implémentée est basée sur nos connaissances ainsi que les expériences que nous avons effectuées. Par exemple, il est très important en asservissement de système d'avoir une erreur en régime permanent nulle donc il est important de donner une récompense à l'agent seulement quand l'erreur en régime permanent est nulle. Pour les autres options de récompenses, différentes valeurs ont été testes et selon l'importance des paramètres du contrôleur nous en sommes arrivés aux résultats présentés dans le tableau ci-dessous.

Tableau 2 : liste des récompenses

Récompense	R1	R2	R3
+100	tr=tr_desiré	ov=ov_desire	err < 0.0006
+1	tr<tr_desiré	ov<ov_desire	-
-1	autre	autre	-
-10	-	-	err >= 0.0006

Ainsi, les récompenses sont sommés à travers les différentes itérations. La formule suivante est utilisée :

$$R = \Sigma_t R1 + R2 + R3$$

Résultats :

Tableau 3 : Configuration et paramètres des réseaux de neurones

1ère configuration	2ème configuration
30 neurones par couche	64 neurones par couche

- Nombre de couches cachées : 2 avec une fonction linéaire entre chaque couche
- Nombre d'épisodes : 2000
- Nombre de step : 1000
- Eps_start : 1.0
- Eps.end : 0.01
- Eps_decay : 0.995
- Replay buffer size : 100000
- Minibatch size : 64
- Discount factor : 0.99
- Soft update of target parameters: 0.001
- Learning rate : 0.0005

Les résultats pour deux configurations suivantes sont présentés dans la section suivante

Résultat 1ère configuration : 30 neurones par couches

Le graphique ci-dessous présente l'évolution du score qui représente l'accumulation de nos récompenses au cours des épisodes. Après 5 heures de temps de calcul, l'algorithme à déterminer les valeurs qui semblent répondre à nos critères au bout de 95 épisodes avec un moyenne de récompense de 222.86. Les valeurs des gains trouvés sont kp = -10, ki =14, et kd =8.

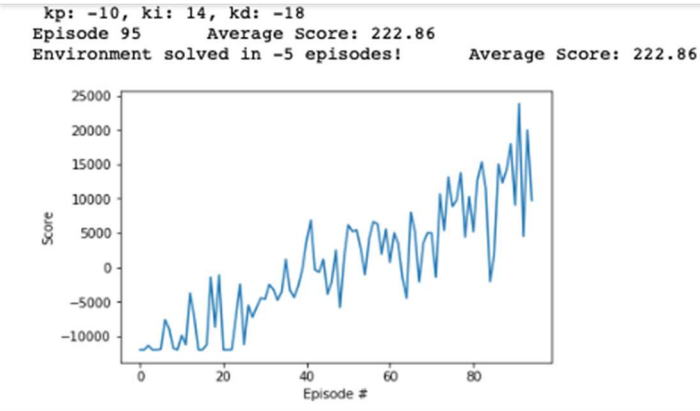

Figure 7 Accumulation des récompenses

Avec les valeurs de gains trouvés, le « step response » est obtenu grâce au logiciel Matlab afin de juger de qualité de la démarche. Le graphique du « step response » est présenté ci-dessous.

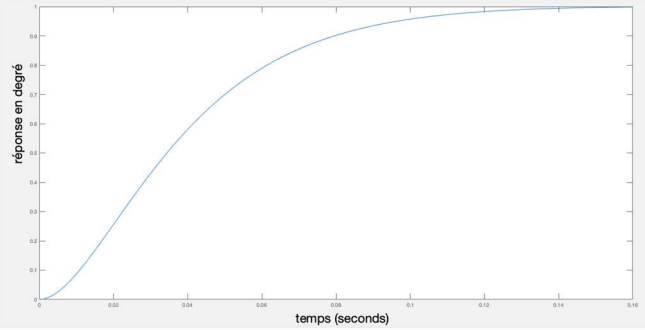

Figure 8: Réponse du système à une entrée unitaire

Les paramètre caractérisant la réponse sont les suivants :

- Temps de réponse : 0.1168 s
- Dépassement : 0%
- Erreur en régime permanent : 0

On remarque que l'algorithme a optimisé les valeurs des gains afin de converger vers les valeurs de temps de réponse, dépassement et erreur désirés. Cependant, on observe un dépassement de 0 ce qui n'est pas désirable dans le cas du contrôle, car les commandes ne sont pas amorties. Ces résultats peuvent s'expliquer par la fonction de récompense utilisée qui donne des récompenses à l'agent lorsque ce dernier arrive à avoir des valeurs de dépassement inférieur aux valeurs désirées.

Résultat 2ème configuration : 64 neurones par couches

Le nombre de neurones est doublé par couches pour la deuxième configuration en gardant les mêmes paramètres.
Le graphique ci-dessous présente l'évolution du score qui représente l'accumulation des récompenses au cours des épisodes. Après 8 heures de temps de calcul, l'algorithme à déterminer les valeurs qui semblent répondre aux critères desirés au bout de 96 épisodes avec une moyenne de récompense de 263.83. Cette moyenne de score est supérieure à la 1ère configuration ce qui est déjà encourageant. Les valeurs des gains trouvés sont
kp = -18, ki =-20, et kd =-7.

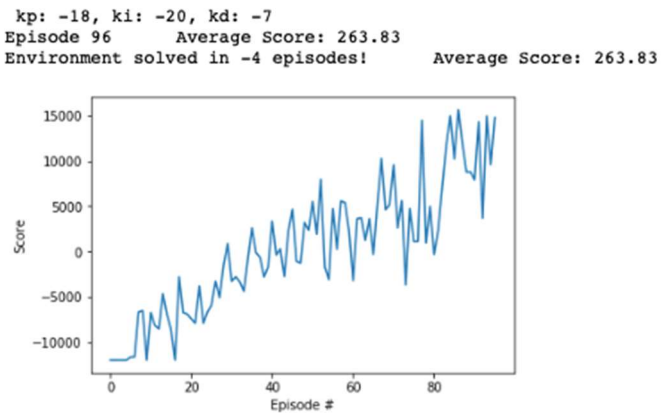

Figure 9: Accumulation des récompenses

Avec les valeurs de gains trouvés, le « step response » obtenu grâce au logiciel Matlab permet de juger de la qualité de la démarche. Le graphique du « step response » est présenté ci-dessous.

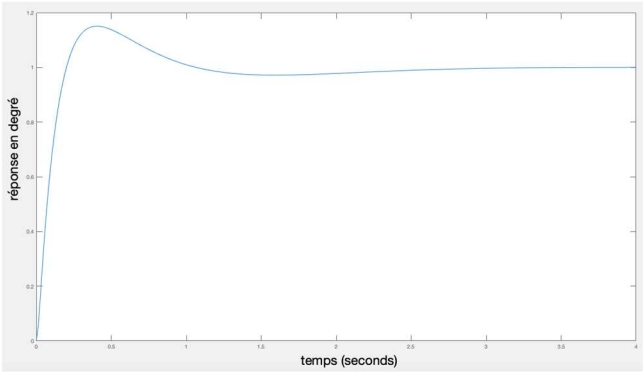

Figure 10: Réponse du système à une entrée unitaire

Les paramètres caractérisant la réponse sont les suivants :

- Temps de réponse : 2.096s
- Dépassement : 15%
- Erreur en régime permanent : 0

Une fois de plus on remarque que l'algorithme converge vers des valeurs de gains qui permettent de répondre partiellement à notre cahier de charge. L'augmentation du nombre de neurones par couches a augmenté le temps de calcul, mais a permis d'avoir d'assez bon résultat.
Le dépassement et l'erreur en régime permanent respectent notre cahier de charge, cependant le temps de réponse est supérieur de 1s par rapport à la valeur désirée. Bien que ce temps de réponse reste acceptable l'application, l'algorithme n'a pas convergé vers la limite qui lui a été imposée.
Pour conclure l'analyse, en utilisant le DQN on remarque que l'algorithme tend à optimiser les gains afin de répondre au cahier de charges. Cependant, il faut noter que l'algorithme et l'architecture ne sont pas optimaux.
La fonction de récompense n'est pas assez exhaustive pour représenter réellement l'évolution que notre contrôleur doit avoir. La liste des actions possible n'est pas elle aussi exhaustive. En outre, un paramètre important dans notre démarche est l'initialisation de nos états après chaque épisode. En effet, on utilise le même state de départ pour notre entraînement ce qui

n'est pas idéal pour la généralisation. On aurait pu aussi jouer avec les paramètres de notre modèle tel que le Learning rate, la taille de nos mini-batch, le « epsilon decay » pour optimiser notre algorithme. Par manque de temps et surtout de puissance de calcul, nous n'avons pas pu tester différentes approches afin d'aboutir à un algorithme optimal. Mais aux vues des résultats que nous avons obtenus avec une simple implémentation du DQN, on peut conclue que c'est algorithme est prometteur pour des réalisations futures malgré les actions discrètes.

C. Deep Deterministic Policy Gradient (DDPG)

Un des problèmes majeurs rencontrés avec les algorithmes précédents est la discrétisation de l'environnement (soit les valeurs des gains). Ainsi, le besoin de discrétisation est un facteur limitant :
Tout d'abord, les algorithmes précèdent demandent de préciser l'intervalle sur lequel on cherche les valeurs de gains. Il est impossible de choisir un intervalle très petit, car la valeur des gains désirée est inconnue. En effet, c'est à l'algorithme de trouver ces valeurs et non pas à l'utilisateur. Cependant, si une plage de valeur très grande est choisie, ceci résulte en un très grand coût de calcul. Il n'est donc pas évident de choisir l'intervalle sur lequel l'algorithme va effectuer son exploration, surtout que les gains peuvent prendre des valeurs tant positives que négatives.
Le deuxième problème correspond au niveau de discrétisation. Si on procède à une discrétisation très fine de l'environnement, l'algorithme trouvera une meilleure solution, mais le problème des coûts et de temps de calcul réémerge, alors que si une discrétisation plus grossière est utilisée, on risque de manquer une solution.
Pour régler ces deux problèmes, l'algorithme Deep Deterministic Policy Gradient (DDPG), développé par Google DeepMind,est utilisé. Ce dernier opère dans un environnement continu. De plus, cet algorithme est reconnu pour son utilisation dans le domaine de la robotique, ce qui est effectivement le cas de l'application étudiée.

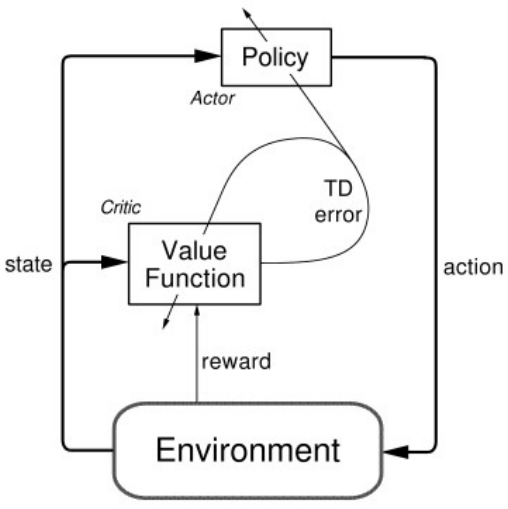

Figure 11: Diagramme Actor Critic

DDPG est formé de quatre réseaux de neurones :

- θ^Q : Réseau Q (Critique)
- θ^μ : Police déterministique (Acteur)
- $\theta^{Q'}$: Réseau Q cible
- $\theta^{\mu'}$: Police déterministique cible

L'acteur prend en entrée les états et rend une action plutôt qu'une probabilité sur les actions comme dans le cas de l'algorithme REINFORCE.

Le critique prend en entrée l'état dans lequel se trouve l'agent ainsi que l'action prise et renvoie une Q-value.

Les réseaux cibles sont des copies différées de leurs réseaux d'origine qui suivent lentement les réseaux appris. Ces copies permettent à l'algorithme de converger.
L'exploration de l'agent est faite grâce à l'ajout de bruit. On utilise souvent le processus d'Ornstein-Uhlenbeck.
DDPG utilise une astuce qui permet de rendre les données distribuées indépendamment. Pour cela, l'agent, lors de son exploration, stocke des séries $\{S_t , A_t , R_t , S_{t+1}\}$ et utilise des mini-batch aléatoires lors de son entraînement.

La figure ci-dessus présente le pseudocode de l'algorithme général DDPG :

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Configuration :

La configuration de l'acteur est présentée à la figure ci-dessous :

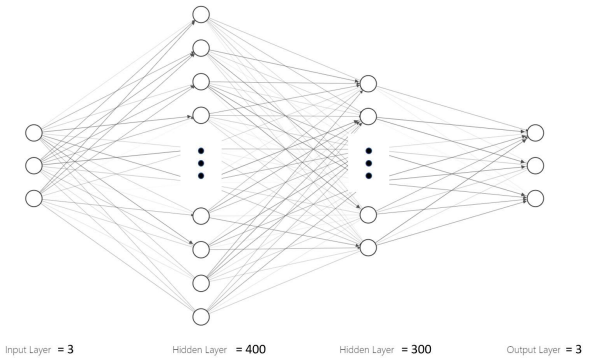

Figure 12 : Architecture de l'acteur

Les 3 entrées correspondent aux trois valeurs des gains, alors que les 3 sorties correspondent aux trois actions sur les gains. La sortie du réseau est alors sommée aux valeurs de gains pour atteindre les nouvelles valeurs (nouveaux états).
Il est important de noter que la fonction d'activation en sortie de l'acteur est tangente hyperbolique. Ainsi, les sorties seront toujours comprises entre -1 et 1 inclusivement.
En ce qui concerne le réseau du critique, celui prend en entrée aussi les valeurs des trois gains ainsi que les sorties de l'acteur pour donner la Q-value. Les deux couches cachées sont semblables au réseau de l'acteur.
La fonction de récompense est la même que celle qui est utilisée dans l'algorithme du Q-learning soit :

$$R = -|tr_{desiré} - tr_{obtenu}| - |ov_{desiré} - ov_{obtenu}| - |err_{desiré} - err_{obtenu}|$$

Avec :

- tr : Temps de réponse
- ov : Dépassement (overshoot)
- err : Erreur en régime permanent

Hyperparamètres :

- Taille du buffer = 10^5
- Taille du mini-batch =128
- γ=0.99 (facteur d'escompte)
- τ = 0.001 (pour la mise à jour des copies différées)
- Taux d'apprentissage de l'acteur=0.0001
- Taux d'apprentissage du critique=0.001
- Solveur : Adam
- Nombre d'épisodes= 2000
- Nombres de pas par épisode =300

Résultats :

Comme on peut le voir sur la figure à la page suivante, l'algorithme DDPG n'a pas convergé vers des valeurs désirables. En effet, la fonction devient instable en présence du contrôleur alors qu'elle est déjà stable sans ce dernier. Ce résultat ne correspond à nos attentes surtout après 2000 épisodes. En effet, une réponse meilleure que celle offerte par DQN est espérée, du fait que DDPG opère dans un environnement continu. Pourtant avec les gains obtenus soit [K_p =223, K_I=302, K_d=223], la réponse n'est même pas stable.

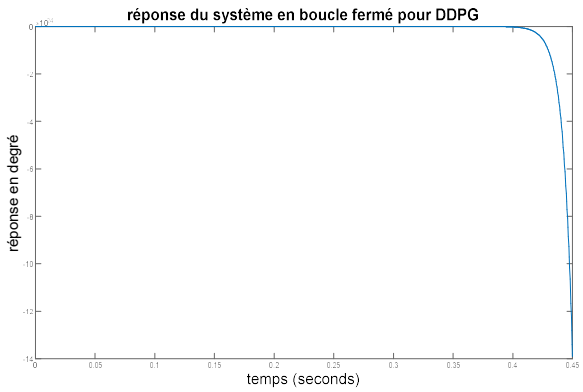

Figure 13: Réponse en boucle fermée (DDPG)

En observant l'évolution du score à la figure 14, on observe un résultat intriguant :

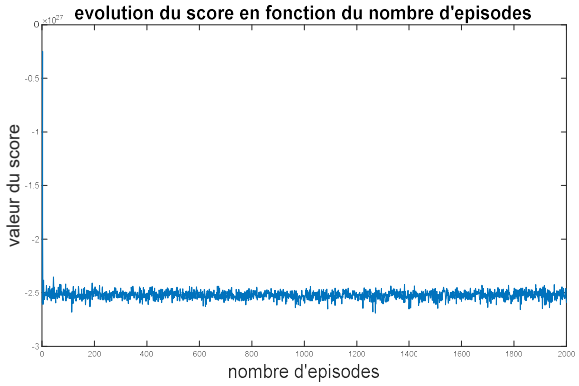

Figure 14: Score de l'algorithme DDPG

On peut voir que le score diminue énormément dès les premiers épisodes et ne varie que légèrement autour de cette position. L'agent n'arrive plus à explorer d'autres états. Les petites variations peuvent être expliquées par les bruits d'Ornstein-Uhlenbeck.
Nous pensons que cet échec à sortir de cet état est dû à la combinaison de trois facteurs :

- Mauvaise initialisation des gains
- Fonction Matlab
- Fonction de récompense

Nous avons par erreur gardé l'initialisation des gains à une valeur égale à 5. Cependant, à des fins de simplification, la fonction Matlab a été modifiée pour qu'elle renvoie des valeurs énormes et aberrantes lorsque les gains sont positifs. Ainsi, l'agent commence **toujours** avec un mauvais score. Pour commencer à avoir un meilleur score, il doit enchaîner cinq actions (-1) consécutives pour les trois gains simultanément. La fonction d'activation tanh() qui renvoie une valeur entre [-1 , 1] n'aide pas l'agent alors pour améliorer son score. Finalement, la fonction de récompense utilisée renvoie toujours des valeurs négatives. L'agent se 'décourage' rapidement et ne cherche plus à explorer une meilleure solution.

IV. Limites de la démarche scientifique

Le but initial de ce projet était de pouvoir contrôler un système mécanique dynamiquement (en direct). Pour effectuer l'entraînement, nous voulions initialement utiliser les microcontrôleurs standards comme Arduino ou Raspberry-Pi. Malheureusement, ces microcontrôleurs sont beaucoup trop faibles pour pouvoir supporter les calculs d'intelligence artificielle. Une solution était de communiquer par WIFI avec l'ordinateur qui recevrait les données du robot et effectuerait les calculs pour ensuite envoyer l'action au robot et ainsi de suite. Cependant, l'intégration d'un module pour la communication sans fil était bien trop exhaustive pour le temps alloué au projet.

Nous avons donc eu l'idée de concevoir un environnement virtuel sur Unity ou Mujoco afin d'effectuer l'entraînement dans cet environnement. Malheureusement, avec nos capacités limitées en matière de conception d'environnement virtuel, nous avons décidé de représenter l'environnement par une fonction sur Matlab qui prend en entrée le gain et renvoie alors le temps de réponse, le dépassement et l'erreur en régime permanent du système. L'algorithme est donc toujours en communication avec Matlab.

Concernant l'exhaustivité des tests, nous aurions aimé effectuer plus d'expérience, mais nous n'avions pas accès à des ordinateurs puissants. Par exemple, l'algorithme le plus rapide a pris 8 heures de temps de calculs pour terminer. Nous avons essayé d'effectuer nos entraînements sur le Cloud de Google Collab afin d'avoir accès à des calculateurs plus puissant, mais vu que notre algorithme est en constante communication avec Matlab qui travaille sur le disque local, il nous a été impossible de profiter de Collab.

V. Conclusion

Dans cet article, nous avons utilisé plusieurs algorithmes afin de contrôler l'angle d'attaque d'un avion. Nous sommes passés du domaine discret (avec Q-learning et DQN) au domaine continu avec DDPG. Nous espérions de meilleurs résultats avec ce dernier, mais malheureusement, nous avons effectué des erreurs dans nos implémentations et faute de temps, nous n'avons pas eu le temps d'appliquer les corrections nécessaires. Toutefois, nous sommes arrivés avoir des résultats très bons avec l'algorithme DQN. Nous sommes alors confiants que les prochaines implémentations de DDPG présenteront de meilleurs résultats. On espère ainsi effectuer ces changements pour la suite du projet afin d'avoir une méthode générale, efficace et simple qui permet de contrôler des systèmes mécaniques.

Remerciement

Nous faisons une mention spéciale à Udacity qui nous a permis d'avoir accès à un Github avec différentes implémentations d'algorithme pour l'apprentissage par renforcement. Nous nous sommes basés sur ces algorithmes afin de produire nos propres algorithmes pour notre cas d'étude.

Reference

[1]Hado van Hasselt, A. G. (2015). *Deep Reinforcement Learning with Double Q-learning.* Google DeepMind.

[2]Hamid Boubertakh, M. T.-Y. (2010). Tuning fuzzy PD and PI controllers using reinforcement learning. *ISA Transactions*, 543-551 .

[3]Mayank, M. (2018, 03 2). *itnext.io*. Retrieved from Reinforcement Learning with Q tables : https://itnext.io/reinforcement-learning-with-q-tables-5f11168862c8

[4]Opperman, A. (2018, 11 4). *towardsdatascience.com* Retrieved from Self Learning AI-Agents III:Deep (Double) Q-Learning: https://towardsdatascience.com/deep-double-q-learning-7fca410b193a

[5]Simoni, T. (2018, 4 11). *medium*. Retrieved from An introduction to Deep Q-Learning: let's play Doom: https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8

[6]Timothy P. Lillicrap, J. J. (2016). *Continuous control with deep reinforcement learning* . London UK: Google Deepmind.

[7]Weng, L. (2018, 4 8). *gihub*. Retrieved from Policy Gradient Algorithms : https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

[8]Xue-songWANG, Y.-h. C. (2007). A Proposal of Adaptive PID Controller Based on Reinforcement Learning. *Journal of China University of Mining and Technology*, 40-44.

[9]Yoon, C. (n.d.). *towardsdatascience.com*. Retrieved from Deep Deterministic Policy Gradients Explained: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

[10]Yuenan Hou Department of Control and Systems Engineering, S. o., Liu, L., Wei, Q., Xu, X., & Chen, C. (2017). *A novel DDPG method with prioritized experience replay* . Banff: IEEE .

Authors

Emile Dimas – Étudiant en dernière année au Baccalauréat en génie mécanique, Polytechnique Montréal.
Email : emiled16@gmail.com

Steve Regis Koalaga – Étudiant en dernière année au Baccalauréat en génie mécanique, Polytechnique Montréal.
Email : steve.regis21@gmail.com