Maîtrise du pilotage sur circuit par apprentissage par renforcement profond

Wissam AKRETCHE[1] , **Madina TRAORÉ**[2] , **Yasmine AIT MIMOUN**[3] , **Saliha OUDALI**[4]

[1]akretche.wissam@etu.upmc.fr
[2]madina.traore@etu.upmc.fr
[3]yasmine.aitmimoun@etu.upmc.fr
[4]salyha.oudali@etu.upmc.fr

Abstract

Depuis quelques années, de plus en plus de constructeurs automobiles se lancent dans la commercialisation de voitures sans conducteur. L'essor de l'intelligence artificielle a en effet permis l'émergence de nombreux systèmes autonomes dont a profité l'industrie automobile. Le travail que nous présentons ici porte en premier lieu sur la mise en œuvre d'algorithmes d'apprentissage par renforcement (*DDPG, TD3 et PPO*) permettant de concevoir un contrôleur capable de piloter une voiture sur un circuit. Nous analysons les performances enregistrées par les différents algorithmes et améliorons les méthodes d'apprentissage en fonction des résultats obtenus. La fiabilité des voitures autonomes faisant l'objet de controverses, nous avons également tenté d'analyser le contrôleur obtenu à l'aide de l'algorithme *t-SNE* afin d'être capable d'expliquer son comportement.

Introduction

L'apprentissage par renforcement profond s'est imposé ces dernières années comme une thématique incontournable de la recherche en intelligence artificielle. Il enregistre des performances spectaculaires dans plusieurs domaines: jeux[1] [2], robotique [3] et contrôle en continu [4]. Pour appliquer les différents outils qu'offre cette technique révolutionnaire aux situations complexes du monde réel, les agents doivent obtenir une représentation efficace des entrées sensorielles de grande dimension et utiliser ces fonctions pour généraliser les observations passées à un échantillon futur.
L'objectif de ce projet est d'explorer des méthodes d'apprentissage dans le but de former un agent capable de conduire une voiture sur des circuits offerts par le simulateur TORCS (*The Open Racing Car Simulator)*. Pour ce faire, nous nous intéressons particulièrement à trois algorithmes qui ont connu un grand succès dans le domaine: DDPG, TD3 et PPO.
Ces algorithmes se montrent en effet très efficaces et atteignent un niveau de performances comparable à celui d'un testeur de jeux professionnel.

1 Contexte

1.1 L'environnement TORCS

TORCS (The Open Racing Car Simultor) est l'un des environnements de course automobile les plus populaires. Il est écrit en C++ et disponible sous licence GPL à partir de sa page web. Pour des usages académiques, TORCS présente plusieurs avantages comparé à d'autres simulateurs:

- C'est un environnement entièrement personnalisable: il contient plusieurs pistes, adversaires et voitures.
- Il comporte un moteur physique sophistiqué (aérodynamique, consommation de carburant...), ainsi que des graphismes en 3 dimensions pour la visualisation des courses.
- Il n'a pas été conçu comme une alternative aux jeux de course commerciaux, mais spécifiquement conçu pour faciliter au maximum le développement de contrôleurs.

Chaque véhicule est contrôlé par un pilote automatique ou un "bot". À chaque étape de contrôle, on peut accéder à l'état actuel du jeu (pour extraire des informations concernant la voiture, le parcours, la vitesse, la position, etc.) pour pouvoir, après un processus de décision, gérer les actionneurs du véhicule (accélération, freinage, etc.). La distribution du jeu comprend plusieurs robots préprogrammés qui peuvent être personnalisés ou étendus pour en créer de nouveaux.

1.2 L'approche Acteur-Critique

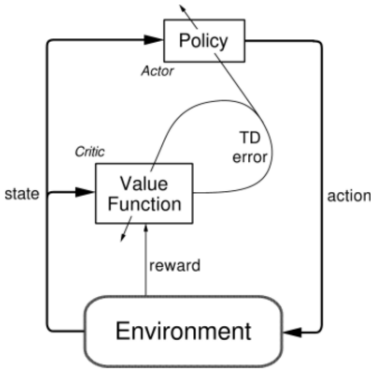

Figure 1: Architecture de l'approche Acteur-Critique

L'algorithme Actor-Critic est essentiellement une méthode hybride permettant de combiner la méthode de descente de gradient appliquée à la fonction de valeur pour améliorer l'acteur. La politique elle-même est dite acteur pendant que la fonction de valeur est appelée critique. Principalement, l'acteur produit l'action compte tenu de l'état actuel de l'environnement, tandis que le critique envoie un signal pour évaluer les actions de l'acteur. Cette évaluation représente la TD-erreur (*TD error*):

$$\delta_t = r_{t+1} + \gamma.V(s_{t+1}) - V(s_t)$$

2 Méthodes

Afin de réaliser un contrôleur qui apprend à piloter une voiture sur circuit, nous avons mis en oeuvre des algorithmes d'apprentissage par renforcement profond (*deep reinforcement learning*).

2.1 La fonction de récompense

Sous l'environnement TORCS, l'état du jeu à un instant donné est représenté par plusieurs composantes, parmi elles on peut distinguer:

- V_x: la vitesse de la voiture suivant l'axe longitudinal,
- V_y: la vitesse de la voiture suivant l'axe transversal,
- V_z: la vitesse de la voiture suivant l'axe Z de la piste,
- θ: l'angle entre l'axe longitudinal de la voiture et l'axe de la piste
- $trackPos$: la distance entre la voiture et l'axe de la piste, normalisé par rapport à la largeur de la piste,
- $distRaced$ la distance parcourue par la voiture,
- rpm: nombre de tours par minute effectués par la voiture,
- $wheelSpinVel$ Vecteur de 4 capteurs représentant la vitesse de rotation des roues.

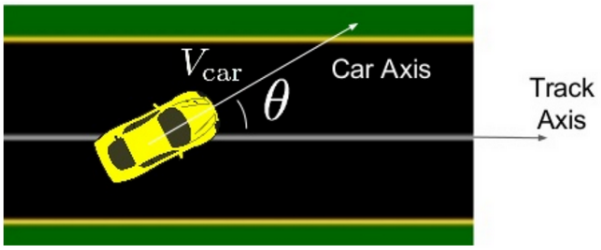

Figure 2: Position de la voiture par rapport à la piste

On considère les fonctions de récompense suivantes:

$$r = V_x Cos(\theta) \quad (1)$$

$$r = V_x Cos(\theta) - V_x Sin(\theta) - V_x|trackPos| \quad (2)$$

La première fonction, qui représente la vitesse de la voiture projetée sur l'axe de la piste aboutit à des résultats peu stables. En effet, le contrôleur a tendance à accélérer au maximum, afin d'obtenir la meilleure récompense, mais finit très vite par converger vers un très faible optimum local en se bloquant contre un mur. Ainsi, la deuxième fonction de récompense a été introduite dans le but de maximiser la vitesse longitudinale (premier terme), minimiser la vitesse transversale (deuxième terme) et éviter que la voiture s'éloigne du centre de la piste (troisième terme). Cette nouvelle fonction s'avère beaucoup plus efficace que la précédente (apprentissage plus stable et plus rapide).
En utilisant la fonction reward de l'équation (2), nous avons constaté que le contrôleur, au lieu d'avancer tout droit sur une piste, faisait parfois des oscillations inutiles. Ainsi, afin de pallier ce comportement indésirable, nous avons, dans un premier temps, essayé de remplacer le $V_x Cos(\theta)$ par $V_x Cos(2\theta)$ dans le but d'empêcher le contrôleur de tourner à plus de 45 degrés mais également de garder la voiture alignée avec la direction de la piste. Cependant, cette fonction s'est avérée moins efficace que la précédente. En effet, les expériences montrent que le contrôleur a, non seulement du mal à apprendre à tourner correctement, mais également à rester au centre de la piste. Ainsi, on introduit un nouveau terme à notre fonction de récompense pour faire en sorte que le mouvement suivant l'axe transversal soit sanctionné. La nouvelle fonction de récompense obtenue est la suivante:

$$r = V_x.Cos\theta - V_x.Sin\theta - 2|V_x.trackPos| - V_y.Cos\theta \quad (3)$$

Cette nouvelle fonction s'avère effectivement meilleure que les deux précédentes. C'est ce que nous verrons un peu plus tard dans la section consacrée aux expériences.

2.2 L'architecture des réseaux de neurones

L'acteur

L'acteur est composé de deux couches cachées comportant respectivement 300 et 600 neurones dotés de la fonction d'activation ReLu. Le réseau prend en entrée un vecteur représentant l'état du jeu (composantes de la vitesse, distance par rapport au centre de la piste, etc.) et produit trois actions continues en sortie: Accélération et Freinage, deux neurones dotés tous les deux de la fonction d'activation $sigmoide$ pour faire en sorte que le résultat soit compris entre 0 et 1 (1 étant l'état où on appuie au maximum sur la pédale); Rotation du volant, neurone doté de la fonction d'activation $tanh$ qui donne des résultats compris entre -1 et 1 (-1 étant la rotation maximale dans le sens anti-horaire et 1 la rotation maximale dans le sens horaire).

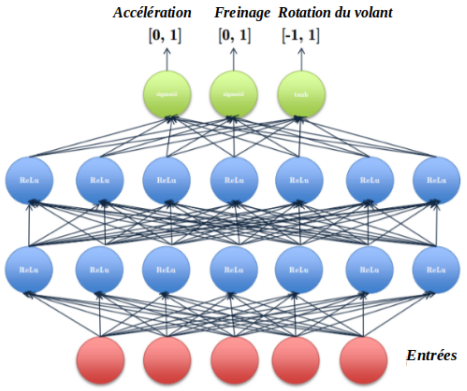

Figure 3: Architecture du réseau de neurones de l'acteur

Le critique

Le critique est composé de trois couches cachées. La première et la dernière couche comportent respectivement 300 et 600 neurones dotés de la fonction d'activation ReLu, tandis que la deuxième couche comporte 300 neurones dotés de fonctions linéaires. Le réseau prend en entrée les sorties du réseau précédent Accélération, Freinage et Rotation du volant ainsi que le vecteur représentant l'état du jeu, et produit les Q-valeurs en sortie.

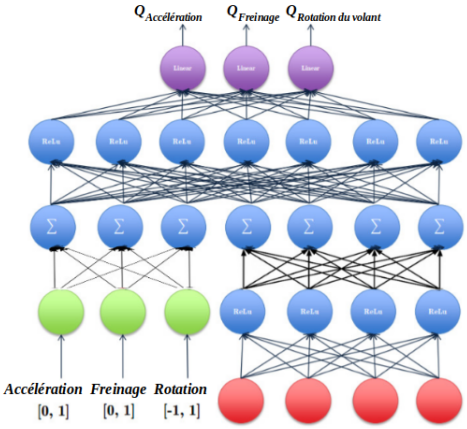

Figure 4: Architecture du réseau de neurones du critique

2.3 Le Replay Buffer

Afin d'augmenter la robustesse de nos modèles (DDPG et TD3) et diminuer les effets de données d'apprentissage fortement corrélées, on entraîne nos réseaux de neurones sur un *Replay Buffer* de 100000 échantillons. Pendant l'entraînement, les paramètres sont mis à jour suivant des échantillons (*mini-batches*) de l'expérience, tirés uniformément et de manière aléatoire $(s, a, r, s') \sim U(D)$.

2.4 Algorithme Deep Deterministic Policy Gradients (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

Deep Deterministic Policy Gradient (DDPG) [4] est un algorithme qui apprend simultanément une Q-fonction et une politique. Il utilise les données *off-policy* et l'équation de Bellman pour apprendre la Q-fonction, et se sert de celle-ci pour apprendre la politique.

Cette approche est étroitement liée au Q-apprentissage (*Q-learning*) et est motivée de la même manière: si on connaissait la fonction $Q^*(s,a)$ qui associe à chaque couple État-action une valeur de manière optimale, alors à n'importe quel état s, l'action optimale $a^*(s)$ pourrait être trouvée de la manière suivante:

$$a^*(s) = \arg\max_a Q^*(s,a).$$

Lorsque l'espace est continu, il n'est pas évident de définir la fonction $Q^*(s,a)$, DDPG mélange l'apprentissage d'un approximateur à $Q^*(s,a)$ avec l'apprentissage d'un approximateur à un $a^*(s)$, et ce d'une manière spécifiquement adaptée aux environnements comportant des espaces d'actions continus.
L'espace d'actions étant continu, la fonction $Q^*(s,a)$ est présumée différentiable par rapport à l'action. Cela nous permet de mettre en place une règle d'apprentissage efficace, basée sur le gradient, pour une politique qui exploite ce fait. Ensuite, au lieu d'exécuter un sous-programme d'optimisation coûteux chaque fois que nous souhaitons calculer $\max_a Q(s,a)$, nous pouvons l'approximer avec $\max_a Q(s,a) \approx Q(s,\mu(s))$.
L'équation de Bellman décrivant la fonction $Q^*(s,a)$ est la suivante:

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

où s' représente l'état suivant dans l'environnement, suivant la probabilité $P(\cdot|s,a)$.
Ainsi, afin d'approximer la fonction $Q^*(s,a)$, une fonction d'erreur de Bellman (MSBE) est mise en place:

$$L(\phi,\mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - (r+\gamma(1-d)\max_{a'} Q_\phi(s',a'))\right)^2\right]$$

Les algorithmes Q-learning, tels que DQN [9](et toutes ses variantes) et DDPG, reposent en grande partie sur la minimisation de cette fonction de perte MSBE et tous utilisent des *replay buffers* et des *réseaux cibles*.

Les réseaux cibles

Le terme:

$$r + \gamma(1-d)\max_{a'} Q_\phi(s',a')$$

est dit cible, car lorsque nous minimisons la perte MSBE, nous essayons de faire en sorte que la Q-fonction soit proche de cette cible. Or, celle-ci dépend des mêmes paramètres que nous essayons de construire: ϕ. Cela rend la minimisation MSBE instable. La solution consiste alors à utiliser un ensemble de paramètres qui s'approchent de ϕ , mais avec un "retard", c'est-à-dire un deuxième réseau, appelé réseau cible, qui est en retard sur le premier. Les paramètres du réseau cible seront notés ϕ_{targ}.

Dans les algorithmes basés sur DQN [9], le réseau cible est simplement copié du réseau principal toutes les k étapes, où k est un nombre fixé préalablement. DDPG met à jour son réseau cible suivant l'équation suivante:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$

où ρ est un hyper-paramètre compris entre 0 et 1.

Ainsi, l'apprentissage en DDPG s'effectue en minimisant la fonction de perte suivante:

$$L(\phi,\mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - (r+\gamma(1-d)Q_{\phi_{\text{targ}}}(s',\mu_{\theta_{\text{targ}}}(s')))\right)^2\right]$$

Quant à l'apprentissage de la politique en DDPG, on effectue simplement une montée de gradient (par rapport aux paramètres de la politique) afin de résoudre l'équation suivante:

$$\max_\theta \mathop{\mathrm{E}}_{s\sim\mathcal{D}}\left[Q_\phi(s,\mu_\theta(s))\right]$$

La stratégie d'exploration

L'exploration de l'environnement se fait en ajoutant du bruit aux actions proposées par l'acteur. Pour ce faire, nous utilisons le processus d'Ornstein-Uhlenbeck, aussi connu sous le nom de *mean-reverting process*, qui est un processus stochastique décrit par l'équation différentielle stochastique:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

où θ, μ et σ sont des paramètres déterministes et W_t le processus de Wiener [10]:

- θ représente la rapidité avec laquelle la variable retourne à la moyenne,
- μ représente la moyenne,
- σ le degré de volatilité du processus.

2.5 Algorithme Twin Delayed DDPG (TD3)

Algorithm 1 TD3

Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

Bien que DDPG puisse parfois permettre d'obtenir d'excellents résultats, il est souvent peu robuste en ce qui concerne les hyper-paramètres et autres types de réglages.
Un des points faibles de DDPG est que la *Q-fonction* apprise a tendance à surestimer considérablement les *Q-valeurs*, ce qui conduit à la rupture de la politique, puisque celle-ci exploite les erreurs de la *Q-fonction*. Twin Delayed DDPG (TD3) [7], qui est lui aussi un algorithme *off-policy*, permet de résoudre ce problème en introduisant les trois astuces suivantes:

Double Q-apprentissage (*Clipped Double-Q Learning)*

TD3 apprend deux *Q-fonctions*, Q_{ϕ_1} et Q_{ϕ_1}, au lieu d'une (donc "jumelles" ou *"twin"* en anglais) en minimisant l'erreur moyenne quadratique de Bellman. La plus petite des deux *Q-valeurs* est choisie pour former les cibles dans les fonctions de perte d'erreur de Bellman:

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s'))$$

Les deux *Q-fonctions* sont apprises par régression par rapport à la cible suivante:

$$L(\phi_1, \mathcal{D}) = (s, a, r, s', d) \sim \mathcal{D}\left(Q_{\phi_1}(s, a) - y(r, s', d)\right)^2$$

$$L(\phi_2, \mathcal{D}) = (s, a, r, s', d) \sim \mathcal{D}\left(Q_{\phi_2}(s, a) - y(r, s', d)\right)^2$$

Enfin, la stratégie est apprise simplement en maximisant Q_{phi_1}:

$$\max_\theta \mathop{\text{E}}_{s\sim\mathcal{D}}[Q_{\phi_1}(s, \mu_\theta(s))]$$

Ce qui est quasiment inchangé de DDPG.

Mises à jour de la politique *"retardées"*

TD3 met à jour la politique et les réseaux de neurones *"cibles"* moins souvent que la *Q-fonction* (il est souvent recommandé de mettre à jour la stratégie pour chaque deux mises à jour de la *Q-fonction* [7]). Cela aide à atténuer la volatilité qui apparaît normalement dans DDPG en raison de la façon dont une mise à jour de politique modifie la cible.

Lissage de la politique cible

TD3 ajoute du bruit à l'action cible, afin de rendre plus difficile l'exploitation des erreurs de la *Q-fonction* par la politique.
Les actions utilisées pour former la cible du *Q-apprentissage* sont basées sur la politique cible, $\mu_{\theta_{\text{targ}}}$, mais avec du bruit ajouté à chaque dimension de l'action. Après avoir ajouté le bruit, l'action cible est alors tronquée dans la plage d'actions valides (toutes les actions valides, a, satisfont $a_{Faible} \leq a \leq a_{\text{Élevée}}$). Ainsi, les actions cibles sont donc:

$$a'(s') = \text{clip}(\mu_{\theta\ texttt{targ}}(s') + \text{clip}(\epsilon, -c, c), a_{Bas}, a_{Haut})$$

où $\epsilon \sim \mathcal{N}(0, \sigma)$

Le lissage de la politique cible sert essentiellement à régulariser l'algorithme. Elle adresse un mode de défaillance particulier pouvant survenir dans DDPG: si l'approximateur de la *Q-fonction* présente un pic net incorrect pour certaines actions, la politique adoptera un comportement peu robuste ou incorrect. Cela peut être évité en lissant la *Q-fonction* par rapport à des actions similaires, pour lesquelles le lissage des politiques ciblées est conçu.

Exploration vs. Exploitation

TD3 est un algorithme *off-policy* qui construit sa politique de manière déterministe. Étant donné que la politique est déterministe, si l'agent explorait de manière *on-policy*, au début, il n'essaierait probablement pas une large gamme d'actions qui aurait permis de trouver des signaux d'apprentissage utiles. Pour améliorer les stratégies d'explorations de TD3, nous ajoutons du bruit aux actions au moment de la formation, (bruit gaussien non corrélé).

2.6 Algorithme Proximal Policy Optimization (PPO)

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Contrairement aux deux algorithmes précédents, l'algorithme *Proximal Policy Optimization (PPO)* est un algorithme on-policy. Bien qu'il soit moins *sample efficient* (car, contrairement aux deux algorithmes précédents, il n'utilise pas de *replay buffer*), il s'avère être plus stable que les précédents.
Il existe deux variantes principales de PPO : *PPO-Penalty* et *PPO-Clip*. [5]

PPO-Penalty

Cette variante pénalise la divergence de Kullback-Leibler (*KL-divergence*) dans la fonction objectif et ajuste le coefficient de pénalité au cours de l'entraînnement.

PPO-Clip

Cette variante ne comporte pas de terme spécifique à la KL-divergence et ne comporte pas de contraintes (contrairement à son prédécesseur TRPO [6]) mais s'appuie sur sa fonction objectif conçue de manière spécialisée (nous aborderons ce point de manière plus détaillée dans les paragraphes qui suivent) pour éliminer les incitations à l'éloignement de la nouvelle politique.
Dans ce qui suit, nous allons nous focaliser sur cette deuxième variante (la variante principale utilisée dans OpenAI [8]).
L'algorithme PPO-clip met à jour les politiques selon l'équation suivante:

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s, a, \theta_k, \theta) \right],$$

La fonction de perte L est donnée par:

$$L(s, a, \theta_k, \theta) = \min\Big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),$$

$$\text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\Big)$$

où ϵ est un (petit) hyper-paramètre qui indique à peu près à quelle distance la nouvelle politique peut s'éloigner de l'ancienne.
C'est une expression assez complexe et il est difficile au premier abord de dire ce qu'elle fait ou comment elle aide à maintenir la nouvelle politique proche de l'ancienne. En fait, il existe une version considérablement simplifiée de cet objectif, qui est un peu plus facile à comprendre (et qui est également la version que nous utilisons dans notre code):

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\quad g(\epsilon, A^{\pi_{\theta_k}}(s, a))\right)$$

où

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

Étant donné un couple état-action (s, a), l'intuition derrière cette équation est la suivante:

Dans le cas où l'avantage est positif

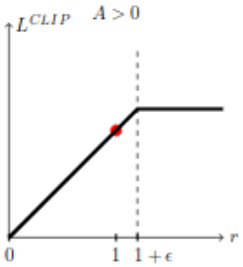

Figure 5: La figure montre un terme (c'est-à-dire un seul pas de temps) de la fonction de substitution L_{Clip} en fonction du taux de probabilité, pour des avantages positifs. Le cercle rouge sur chaque graphique indique le point de départ de l'optimisation, c'est-à-dire r=1. [5]

Si l'avantage relatif au couple état-action (s, a) est positif, sa contribution à l'objectif se réduit à:

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(s, a).$$

En effet, lorsque l'avantage est positif, l'objectif augmentera si l'action devient plus probable, c'est-à-dire si $\pi_\theta(a|s)$ augmente. Mais le min dans ce terme limite l'augmentation de l'objectif. Lorsque $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$, le min entre en action et ce terme atteint une limite de $(1+\epsilon)A^{\pi_{\theta_k}}(s, a)$. Par conséquent, la nouvelle politique ne s'améliore pas en s'éloignant de l'ancienne politique.

Dans le cas où l'avantage est négatif

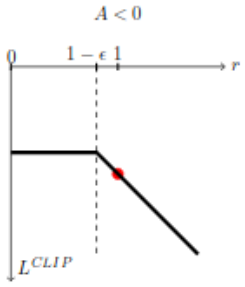

Figure 6: La figure montre un terme (c'est-à-dire un seul pas de temps) de la fonction de substitution L_{Clip} en fonction du taux de probabilité, pour des avantages négatifs. Le cercle rouge sur chaque graphique indique le point de départ de l'optimisation, c'est-à-dire r=1. [5]

Si l'avantage relatif au couple état-action (s, a) est négatif, sa contribution à l'objectif se réduit à:

$$L(s, a, \theta_k, \theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(s, a).$$

En effet, lorsque l'avantage est négatif, l'objectif augmentera si l'action devient moins probable, c'est-à-dire si $\pi_\theta(a|s)$ diminue. Mais le max dans ce terme limite l'augmentation de l'objectif. Lorsque $\pi_\theta(a|s) < (1-\epsilon)\pi_{\theta_k}(a|s)$, le max entre en action et ce terme atteint une limite de $(1-\epsilon)A^{\pi_{\theta_k}}(s, a)$. Par conséquent, la nouvelle politique ne s'améliore pas en s'éloignant de l'ancienne politique.

Ainsi, nous pouvons constater que cette méthode est un moyen de régularisation qui élimine toute incitation à un changement radical de la politique, et l'hyperparamètre ϵ correspond à la distance à laquelle la nouvelle politique peut s'éloigner de l'ancienne tout en améliorant l'objectif.

Exploration vs. Exploitation

PPO est un algorithme *on-policy* qui construit sa politique de manière stochastique. Cela signifie qu'il explore en échantillonnant les actions en fonction de la dernière version de sa politique stochastique. La quantité de hasard dans la sélection d'action dépend à la fois des conditions initiales et de la procédure d'entraînement. Au cours de la formation, la stratégie devient progressivement de moins en moins aléatoire, car la règle de mise à jour l'encourage à exploiter les récompenses qu'elle a déjà trouvées. Cela peut entraîner le blocage de la stratégie dans les optima locaux.

2.7 Algorithme t-SNE

Algorithm 1: Simple version of t-Distributed Stochastic Neighbor Embedding.

Data: data set $\mathcal{X} = \{x_1, x_2, ..., x_n\}$,
cost function parameters: perplexity *Perp*,
optimization parameters: number of iterations T, learning rate η, momentum $\alpha(t)$.
Result: low-dimensional data representation $\mathcal{Y}^{(T)} = \{y_1, y_2, ..., y_n\}$.
begin
  compute pairwise affinities $p_{j|i}$ with perplexity *Perp*
  set $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$
  sample initial solution $\mathcal{Y}^{(0)} = \{y_1, y_2, ..., y_n\}$ from $\mathcal{N}(0, 10^{-4}I)$
  for t=1 **to** T **do**
    compute low-dimensional affinities q_{ij}
    compute gradient $\frac{\delta C}{\delta \mathcal{Y}}$
    set $\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta\frac{\delta C}{\delta \mathcal{Y}} + \alpha(t)\left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)}\right)$
  end
end

L'algorithme de visualisation *t-distributed stochastic neighbor embedding* (t-SNE) est un outil permettant de projeter des données issues d'un espace de grande dimension dans un espace de plus petite dimension. Nous l'avons utilisé afin de tenter d'analyser le contrôleur obtenu avec l'algorithme DDPG présenté plus haut.

L'algorithme fonctionne de la manière suivante :

1. On calcule la distribution de probabilité exprimant la similarité de chaque paire de points (x_i, x_j) dans l'espace de grande dimension. Cette similarité est définie par la probabilité conditionnelle $p_{j|i}$ selon laquelle x_i choisirait x_j comme voisin si les voisins étaient choisis proportionnellement à la densité de probabilité d'une Gaussienne centrée en x_i.
2. On recrée cette distribution de probabilité dans l'espace de faible dimension correspondant en calculant de la même manière la similarité $q_{j|i}$ pour chaque paire (y_i, y_j) de points dans l'espace de faible dimension.
3. On tente de minimiser la différence entre $p_{j|i}$ et $q_{j|i}$ en effectuant une descente de gradient

De cette manière, en fournissant à l'algorithme une perplexité $Perp$ (pouvant être interprétée comme le nombre de voisins de chaque point), une borne T pour le nombre d'itérations de la descente de gradient et un taux d'apprentissage η, on obtient une représentation de nos données initiales dans un espace de faible dimension (généralement 2D ou 3D).

3 Expériences et résultats

3.1 Circuits d'apprentissage, de validation et de test

Pour entraîner notre IA, nous avons choisi une piste relativement difficile appelée "Aalborg" comme jeu de données d'entraînement. La figure ci-dessous montre la structure de la piste:

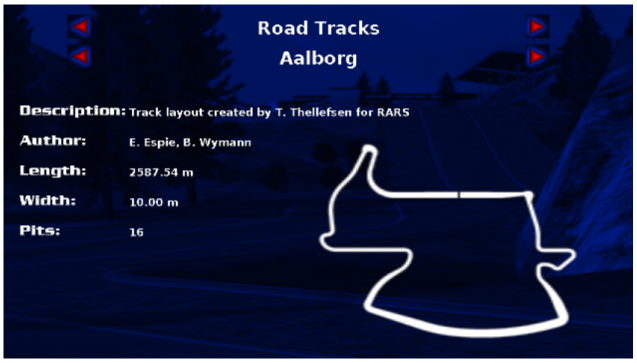

Figure 7: Figure montrant la structure de la piste d'entraînement "Aalborg"

Pour la validation de nos hyperparamètres, nous avons choisi un circuit un peu plus compliqué que le précédent, dans le sens où il est un peu plus long et comporte plus de virages. La figure ci-dessous montre la structure de la piste:

Figure 8: Figure montrant la structure de la piste de validation "E-track 2"

Enfin, nous testons notre algorithme sur un troisème circuit *CG track 3* dont la structure est présentée dans la figure suivante:

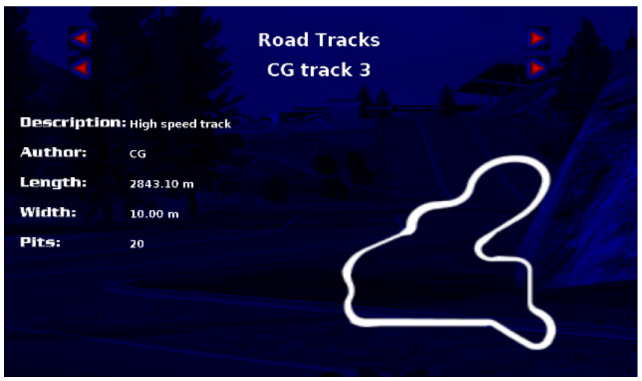

Figure 9: Figure montrant la structure de la piste de test "CG track 3"

3.2 Conditions d'arrêt d'un épisode

Afin de permettre à l'agent d'apprendre de manière efficace, les conditions d'arrêt d'un épisode sont définies de la manière suivante:

- Afin d'éviter que la voiture apprenne à conduire en dehors de la piste du jeu, un épisode d'apprentissage s'arrête aussitôt que la voiture quitte la piste,
- De la même manière, si l'agent se retrouve à rouler dans le sens inverse, l'épisode s'arrête.
- Pour optimiser le temps d'apprentissage, on arrête l'épisode dans le cas où la récompense augmente de manière très lente (c'est le cas par exemple lorsque l'agent se bloque contre un mur et atteint un optimum local)

3.3 Le freinage stochastique

Il s'avère que demander à l'IA d'apprendre à freiner est beaucoup plus difficile que de diriger ou d'accélérer. Ceci s'explique par le fait que la voiture ralentit lorsque le frein est appuyé. Par conséquent, la valeur de la fonction de récompense devient plus petite, l'agent a donc tendance à ne pas freiner du tout. De plus, si l'agent est autorisé à appuyer simultanément sur le frein et l'accélération pendant la phase d'exploration, l'agent appuiera souvent fort sur le frein. On se retrouve donc coincé dans un très mauvais optimum local (car la voiture ne bouge pas et il n'y a donc pas de récompense).

Pour pallier ce problème, on introduit l'idée de freinage stochastique qui consiste à freiner 10% du temps lors de la phase d'exploration tandis qu'on ne freine pas 90% du temps. Comme le contrôleur n'appuie que 10 % du temps sur les freins, la voiture peut prendre de la vitesse et ne restera donc pas coincée dans un optimum local insuffisant tout en apprenant à freiner.

3.4 Analyse des performances

Comparaison entre les différentes fonctions de récompense

Dans cette section, nous comparons les performances obtenues en entraînant notre contrôleur en se basant sur les deux fonctions de récompenses (2) et (3) évoquées précédemment.

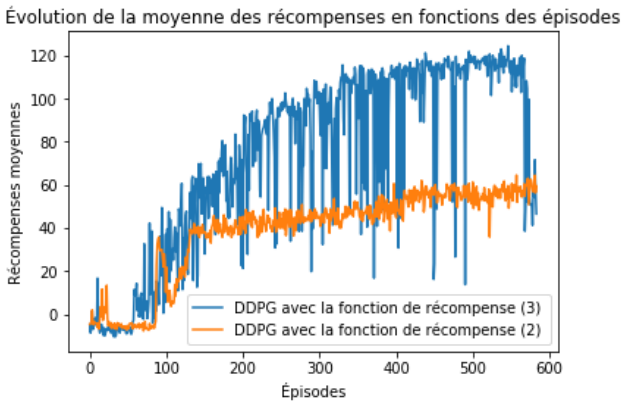

Figure 10: Comparaison entre les récompenses obtenues par DDPG avec les fonctions de récompense (2) et (3)

La figure ci-dessus montre que, bien que l'on ait ajouté un terme négatif (pénalisant), les récompenses obtenues avec notre nouvelle fonction de récompense se montrent plus élevées que celles obtenues avec la première fonction de récompense. Ceci se traduit par de plus grandes distances parcourues pendant un épisode donné (avant que la voiture ne quitte la piste, avance très lentement ou marche à contre-sens). C'est ce que montre la figure ci-dessous, représentant l'évolution des distances parcourues au cours des épisodes.

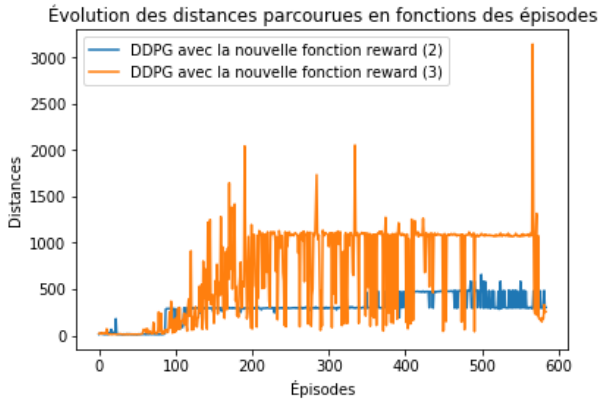

Figure 11: Comparaison entre les distances parcourues par DDPG avec les fonctions de récompense (2) et (3)

Comparaison entre les deux algorithmes DDPG et TD3

Nous avons vu précédemment que l'algorithme DDPG avait tendance à surestimer la Q-valeur. Pour pallier ce problème, nous avons appliqué l'algorithme TD3, en gardant les mêmes architectures des réseaux de neurones acteur-critique ainsi que les paramètres du *replay buffer*, afin de s'assurer de ne pas biaiser les résultats. Les résultats obtenus peuvent être observés dans les figures ci-dessous.

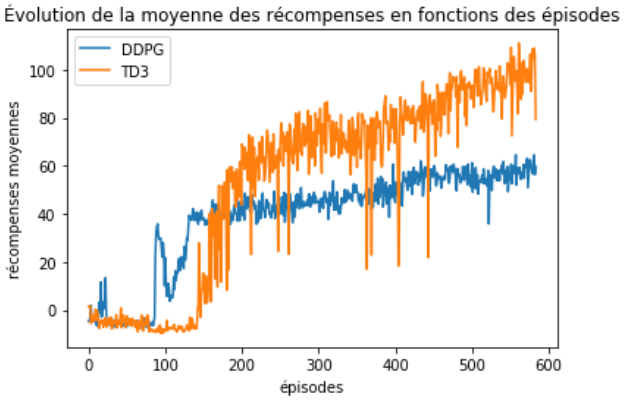

Figure 12: Comparaison entre la moyenne des récompenses obtenues par TD3 et celles obtenues par DDPG

On constate que l'algorithme TD3 est bien plus performant que DDPG à long terme. Bien qu'aux premiers épisodes, DDPG ait l'air d'être meilleur, TD3 le domine après environ 150 épisodes et enregistre des distances parcourues bien plus grandes. C'est ce que nous pouvons constater sur la figure ci-dessous.

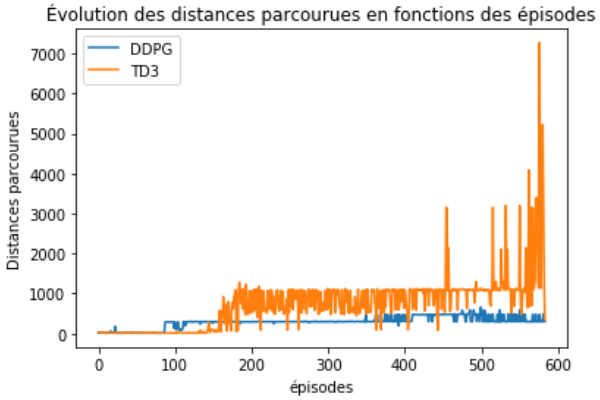

Figure 13: Comparaison entre les distances parcourues par TD3 et celles parcourues par DDPG

Comparaison entre les trois algorithmes DDPG, TD3 et PPO

Dans cette section, nous faisons une comparaison entre les performances obtenues par l'algorithme *on-policy* PPO et les deux algorithmes *off-policy* TD3 et DDPG.

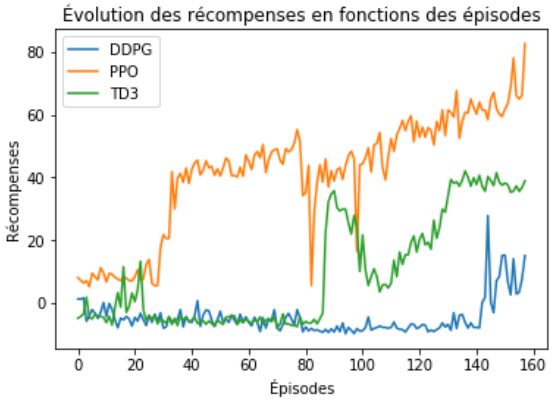

Figure 14: Comparaison entre les récompenses obtenues par DDPG, TD3 et PPO)

On constate que les résultats obtenus par PPO sont bien meilleurs que ceux obtenus par les deux algorithmes DDPG et TD3 et ceci se voit encore plus dans la figure ci-dessous, montrant clairement que les distances parcourues par PPO sont nettement plus grandes que celles parcourues par DDPG et TD3 durant les premiers épisodes. En effet, au bout de 160 épisodes, le contrôleur appris par PPO parvient à effectuer des tours de circuits complets alors que DDPG et TD3 nécessitent au moins 600 épisodes avant d'apprendre à faire un tour complet sur un circuit.

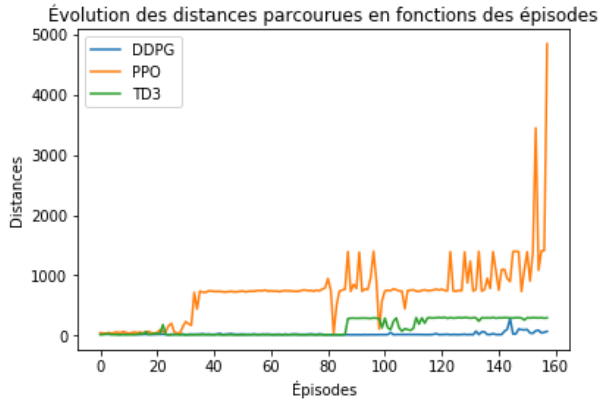

Figure 15: Comparaison entre les récompenses obtenues par DDPG, TD3 et PPO

L'algorithme PPO qui est *on-policy*, stable, mais a priori pas très sample efficient, puisqu'il n'utilise pas de *replay buffer*, s'avère plus performant que les deux autres algorithmes *off-policy*, instables, mais potentiellement beaucoup plus sample efficient quand il sont stables.
Dans notre cas, PPO se montre plus performant, d'autant plus qu'il reste toujours plus performant, même dans les circuits qui lui sont inconnus (circuit de test). C'est ce que nous pouvons constater dans la figure ci-dessous qui compare l'évolution des récompenses obtenues par PPO et TD3 sur le circuit de test "CG track 3" au cours des étapes du jeu.

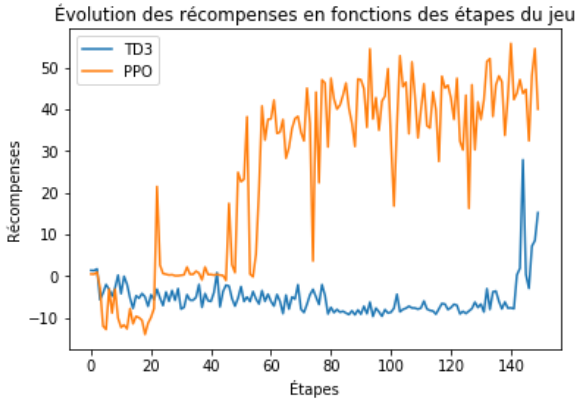

Figure 16: Comparaison entre les récompenses obtenues par TD3 et PPO sur le circuit de test

3.5 Analyse des contrôleurs

Application de l'algorithme t-SNE sur le contrôleur obtenu avec DDPG

Disposer d'un contrôleur performant est essentiel pour envisager une conduite complètement autonome mais cela n'est pas suffisant. Il est, en effet, nécessaire d'être capable d'assurer la sécurité d'éventuels passagers et piétons.

Dans cette optique, nous avons tenté d'analyser le contrôleur obtenu avec DDPG à l'aide de l'algorithme t-SNE. Le but premier était de comprendre les décisions prises par le contrôleur en fonction des différentes situations rencontrées, les objectifs secondaires étaient d'être en mesure :

- d'améliorer le contrôleur en détectant des situations où celui-ci aurait pu avoir un meilleur comportement
- d'évaluer sa flexibilité (capacité d'adaptation sur de nouveaux circuits)
- d'écrire un contrôleur à la main aussi voire plus performant que ceux obtenus avec les algorithmes décrits plus haut (DDPG, TD3 et PPO)

Pour réaliser ce début d'analyse nous avons utilisé une méthode similaire à celle décrite dans l'article *Graying the black box: Understanding DQNs* [12]. Nous avons alors commencé par entraîner notre contrôleur avec l'algorithme DDPG. Une fois que la voiture était capable de réaliser plusieurs tours de circuits sur différentes pistes sans faire de sortie de route, nous avons récupéré les activations de la dernière couche cachée du réseau de neurones de l'acteur (Figure 1) puis nous avons appliqué l'algorithme t-SNE sur celles-ci. Le résultat brut qui a été obtenu est représenté sur la figure 17.

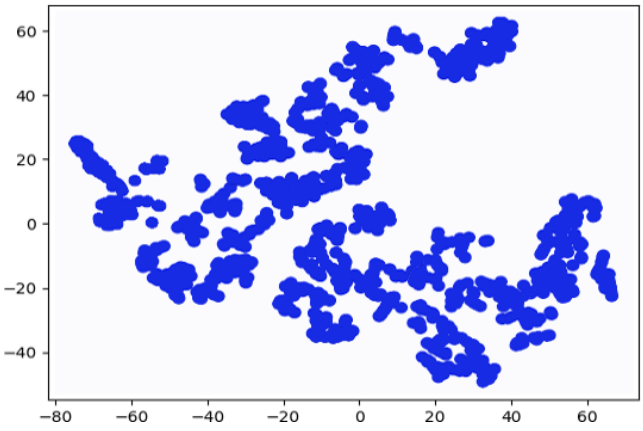

Figure 17: Application de t-SNE sur les données d'activations obtenues sur les circuits Aalborg et ETrack2

Comme première approche d'interprétation de ces résultats nous avons voulu mettre en évidence la présence des différents clusters observables a priori. L'idée était de faire émerger des types de comportements adoptés par le contrôleur (e.g. accélérer en allant tout droit, ralentir et tourner à droite ou à gauche) en fonction des différentes situations rencontrées sur le circuit. Pour ce faire, nous avons appliqué la méthode des k-moyennes (avec $k = 7$) sur notre ensemble de données d'activation (Figure 18).

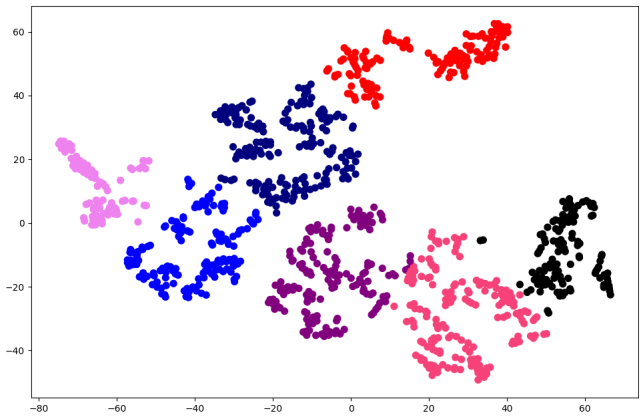

Figure 18: Application de k-means (avec $k = 7$) sur les données d'activations obtenues sur les circuits Aalborg et ETrack2

Nous avons ensuite tenté de déterminer manuellement la signification de ces clusters afin de comprendre le mode de fonctionnement du contrôleur appris. Pour ce faire, nous avons découpé les circuits de test (Aalborg et ETrack2) en plusieurs portions (montée, ligne droite, virage serré à droite, virage serré à gauche, léger virage à droite, léger virage à gauche) puis nous avons mis en correspondance ces portions de circuits avec les données d'activation utilisées pour t-SNE. L'objectif était de pouvoir déterminer si, dans des situations visuellement similaires pour un être humain, le contrôleur répondait par des actions similaires, autrement dit s'il décidait d'accélérer, freiner, orienter le volant de la même manière pour des situations semblables. Le graphique obtenu après affectation de chaque point à sa portion de circuit est représenté sur la figure 19.

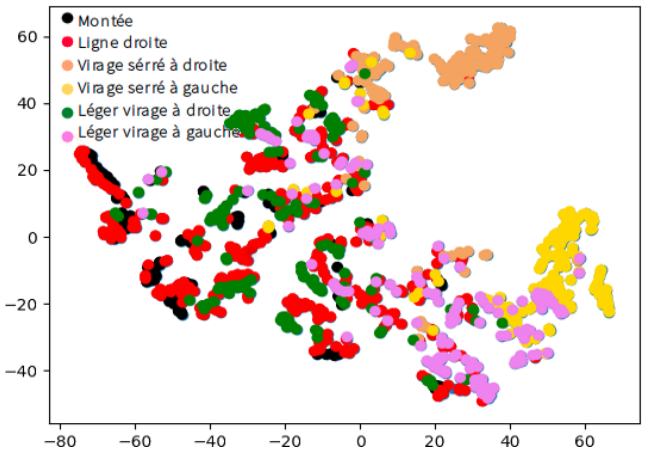

Figure 19: Étiquetage par appartenance à un type de portion de circuit (montée, ligne droite, virage serré à droite, virage serré à gauche, léger virage à droite, léger virage à gauche) des données d'activation obtenues sur les circuits Aalborg et ETrack2

Sur le graphique obtenu on peut remarquer que les virages serrés à gauche et à droite forment deux clusters correspondant à quelques points près aux clusters rouge et noir (les deux clusters les plus à droite) de la figure 18. Dans notre démarche d'identification de comportements type adoptés par le contrôleur, on peut faire l'hypothèse que ces deux clusters correspondent à un ralentissement de la voiture et un virage à gauche (pour le cluster noir), à droite (pour le cluster rouge). De manière légèrement moins évidente, on observe que la majorité des points appartenant à un léger virage à gauche forment un cluster que l'on peut assimiler au cluster rose situé en bas à droite de la figure 18 : il s'agirait alors d'un cluster correspondant à une accélération plus importante et un virage à gauche plus léger de la voiture par comparaison au cluster noir. Néanmoins les autres classes choisies ne sont pas bien séparées et ne correspondent pas aux clusters identifiés.

Si l'on se concentre maintenant uniquement sur la figure 19, on peut constater que les points associés à une montée et ceux associés à une ligne droite sont toujours proches. On peut alors imaginer que les points tels que la voiture peut accélérer autant qu'elle le souhaite sont similaires pour le contrôleur. Par ailleurs, on s'aperçoit que les points liés à un léger virage à gauche (respectivement à droite) sont plus nombreux à proximité des points associés à un virage serré à gauche (respectivement à droite).

On peut déduire de ces observations que le contrôleur comprend de la même manière qu'un être humain qu'il doit tourner à gauche ou à droite lorsqu'il se retrouve face à un virage mais les résultats obtenus ne suffisent pas à mettre en évidence l'ensemble des comportements adoptés par celui-ci.

Pour cela il faudrait être en mesure de déterminer précisément sur quels critères il se base pour choisir les actions à réaliser.

Nous avons alors pensé qu'il serait intéressant de générer une matrice de corrélation entre certaines variables du jeu utilisées par le contrôleur lors de son apprentissage et les actions réalisées par celui-ci (*steering*, *accelarating* et *braking*). Les variables du jeu prises en compte ici sont les suivantes :

- $angle$: angle entre l'axe longitudinal de la voiture et l'axe de la piste
- $trackPos$: distance entre la voiture et l'axe de la piste, normalisé par rapport à la largeur de la piste
- $speedX$: vitesse de la voiture suivant l'axe longitudinal
- $speedY$: vitesse de la voiture suivant l'axe transversal
- $speedZ$: vitesse de la voiture suivant l'axe Z de la piste
- $dist$: distance parcourue par la voiture

Nous avons également intégré à cette matrice la variable *reward* correspondant à la récompense attribué au contrôleur en fonction de l'état du jeu dans lequel il se trouve (défini à la section 2.1).

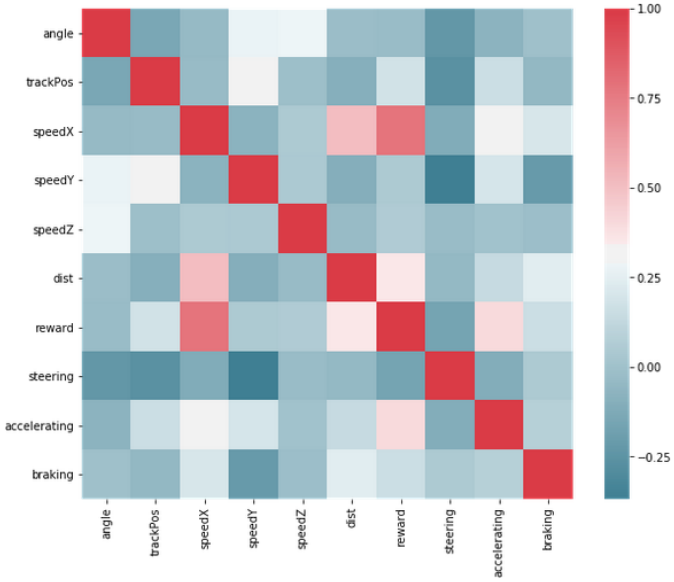

Figure 20: Matrice de corrélation entre les différentes variables du jeu utilisées par le contrôleur lors de son apprentissage et les actions réalisées par celui-ci

La première chose que l'on remarque est la forte corrélation entre les variables *accelerating* et *reward* : cela peut traduire l'idée que lorsque la voiture réalise des actions correctes, celle-ci "prend confiance" et peut alors accélérer en suivant la même dynamique.

On observe également une corrélation évidente entre les variables *accelerating* et *speedX* et entre les variables *braking* et *speedX* : plus on accélère (*accelerating*) plus la vitesse suivant l'axe longitudinal augmente, plus on ralentit (*braking*), plus elle diminue.

Par ailleurs, la distance entre la voiture et l'axe de la piste ($trackPos$) ainsi que la vitesse de la voiture suivant l'axe transversal ($speedY$) semblent jouer un rôle important dans l'accélération. La fonction de récompense ayant été écrite de telle sorte que le contrôleur soit pénalisé lorsqu'il dévie de l'axe de la piste, il serait naturel que celui-ci décide de ralentir lorsqu'il s'en éloigne et d'accélérer lorsqu'il s'en approche.

À partir de cette matrice on comprend difficilement comment le contrôleur détermine s'il doit tourner ou non. En effet, la variable *steering* ne semble être corrélée à aucune des variables présentées ici. Plus particulièrement, contrairement à ce que l'on pensait en observant les résultats obtenus avec t-SNE, on n'observe pas de forte corrélation entre l'action de tourner le volant (*steering*) et le fait de freiner (*braking*). Cependant, on peut se douter que *steering* aurait été fortement corrélé à *wheelSpinVel* (2.1) s'il avait été représenté ici.

On constate enfin une corrélation relativement forte entre le freinage et la distance parcourue par la voiture ($dist$). Une explication à cela pourrait être que le contrôleur apprend "par coeur" quand freiner sur un circuit donné en fonction de la distance parcourue.

Au vu de ces observations, une piste à explorer serait de tenter d'étiqueter les données d'activation avec des classes basées sur ces variables de jeu afin de mieux comprendre leur rôle dans les décisions prises par le contrôleur.

Conclusion

L'apprentissage par renforcement permet à un agent d'apprendre un comportement qui n'a pas été préalablement défini par l'homme et la méthode acteur-critique que nous avons utilisée réduit l'expérience nécessaire au contrôleur pour atteindre un comportement acceptable, à savoir conduire efficacement une voiture. Nous avons pu constater que quelques algorithmes de cette approche sont plus performants que d'autres, notamment le contrôleur appris par PPO qui prend significativement moins de temps pour faire des tours de circuits que ceux appris par DDPG et TD3.
Par ailleurs, nous nous sommes rapidement aperçu qu'analyser un contrôleur de ce type se révélait être une tâche complexe. En effet, cette mission nécessite de se détacher plus ou moins du point de vue humain (circuit tel que l'humain le voit) pour emprunter celui de la machine contrôlant la voiture (paramètres du circuit observés par le contrôleur).

Ouverture

Nous aurions aimé appliquer la méthode décrite dans l'article *World Models* [13] qui a connu un grand succès auprès de la communauté des chercheurs en Intelligence Artificielle. Malheureusement, cette méthode nécessite beaucoup de ressources computationnelles que nous n'avons pas actuellement.
Dans un futur proche, nous souhaitons explorer d'autres algorithmes tels que SAC (Soft Actor-Critic) [14] et améliorer les

techniques d'apprentissage (configuration automatique des hyper-paramètres).

Concernant la poursuite du travail d'analyse du contrôleur, nous aimerions tenter de classifier les données d'activation obtenues en nous basant sur les paramètres du jeu utilisés par le celui-ci. Si nous parvenons à identifier les différentes classes de comportements qu'il adopte au cours de sa course, nous pourrons alors écrire un contrôleur expert à la main dont nous testerons les performances.

Remerciements

Avant tout, nous tenons à remercier notre encadrant Monsieur SIGAUD pour ses précieux conseils, sa patience et son support tout au long de ce projet. Nous remercions également nos professeurs et nos familles pour leur soutien et leurs encouragements qui nous ont permis de mener à bien ce projet.

Références

[1] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, and et al *"Human-level control through deep reinforcementlearning"*. In: Nature, 518:529–533, 2015.)

[2] Silver, David, Schrittwieser, Julian, Simonyan, Karen, and et al. *"Mastering the game of go without human knowl-edge"*. In: Nature, 550:354359, 2017.)

[3] Levine, Sergey, Finn, Chelsea, Darrell, Trevor, and Abbeel,Pieter. *"End-to-end training of deep visuomotor policies"*. The Journal of Machine Learning Research, pp. 1334–1373, 2016.)

[4] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver and Daan Wierstra *"Continuous control with deep reinforcement learning"*. In: arXiv:1802.09477 (2015)

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov *"Proximal Policy Optimization Algorithms"*. In: arXiv:1509.02971 (2017)

[6] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordana and Pieter Abbeel *"Trust Region Policy Optimization"*. In: arXiv:1502.05477 (2015)

[7] Scott Fujimoto, Herke van Hoof and David Meger *"Addressing Function Approximation Error in Actor-Critic Methods"*. In: arXiv:1802.09477 (2018)

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang and Wojciech Zaremba *"OpenAI Gym"*. In: arXiv:1606.01540 (2016) https://www.openai.com/

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller *"Playing Atari with Deep Reinforcement Learning"*. In: arXiv:1312.5602 (2013)

[10] Yin Sun, Yury Polyanskiy and Elif Uysal-Biyikoglu *"Remote Estimation of the Wiener Process over a Channel with Random Delay"*. In: arXiv:1701.06734 (2017)

[11] Laurens van der Maaten and Geoffrey Hinton *"Visualizing Data using t-SNE"*. In: Journal of Machine Learning, Nov. 2008. http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf

[12] Tom Zahavy, Nir Ben Zrihem and Shie Mannor *"Graying the black box: Understanding DQNs"*. In: arXiv:1602.02658 (2016)

[13] David Ha and Jürgen Schmidhuber *"World Models"*. In: arXiv:1803.10122 (2018)

[14] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel and Sergey Levine *"Soft Actor-Critic Algorithms and Applications"*. In: arXiv:1812.05905 (2018)