Rapport de projet

2024

IA jouant à un jeu vidéo

Henri Besancenot
Axel Daboust
Clément Gilli
Iliass Khoutaibi

Contents

1 Introduction **2**

2 Formalisation du problème **2**

3 Q-learning **5**

3.1 Algorithme 5

3.2 Résultats 5

3.3 Limites du Q-learning 6

4 Deep Q-learning **7**

4.1 Du Q-learning au Deep Q-learning 7

4.2 Preprocessing 7

4.3 Algorithme 9

4.4 Double Deep Q-learning 9

4.5 Résultats 10

4.6 Limites du Deep Q-learning 11

5 REINFORCE **12**

5.1 Fonction Objectif 12

5.2 Théorème de Policy Gradient 12

5.3 Algorithme 13

5.4 Résultats 13

5.5 Limites de REINFORCE 13

5.6 Fonction Advantage 14

5.7 Algorithme REINFORCE with baseline 14

5.8 De REINFORCE à A2C 14

6 Advantage Actor-Critic (A2C) **14**

6.1 Fonction Advantage 14

6.2 Policy Gradient avec Advantage 15

6.3 Algorithme 15

7 Asynchronous Advantage Actor-Critic (A3C) **16**

8 Conclusion **17**

9 Bibliographie **18**

1 Introduction

Dans le cadre de ce projet, nous avons eu l'opportunité de travailler dans le domaine de l'IA appliquée aux jeux vidéos. L'objectif était d'entraîner des IA à jouer à des jeux, en commençant par des jeux simples comme Cartpole, puis complexifier progressivement les jeux en passant par des jeux Atari et finalement arriver jusqu'à des jeux comme Mario Bros où les environnements sont très complexes.

Pour cela, nous nous sommes basés sur l'apprentissage par renforcement, en passant par plusieurs étapes progressives en difficulté que nous allons vous présenter.

2 Formalisation du problème

Le principe de l'apprentissage par renforcement est très simple, il imite le comportement des humains, plus précisément celui des enfants dans leur processus d'apprentissage : un agent agit dans un environnement donné, faisant des actions qui lui permettent d'obtenir des récompenses.

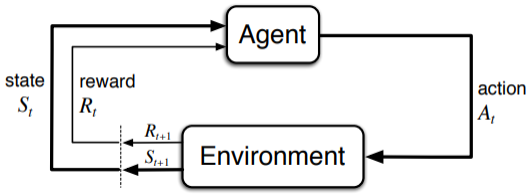


Figure 1: Schéma RL

Pour entraîner notre IA, il faut trouver un algorithme qui lui permette d'apprendre d'elle-même en jouant un grand nombre de fois. Nous allons voir comment formaliser notre problème, c'est-à-dire comment modéliser le jeu de manière mathématique afin d'en tirer des propriétés et ainsi trouver un algorithme. [1]

Définition 1 (Chaîne de Markov) *Une suite* $(X_n)\in \mathbb{N}$ *de* $(\Omega, \mathcal{A}, \mathbf{P})$ *est dite une chaîne de Markov si* $\forall k \in \mathbb{N}, P(X_{k+1} = x_{k+1}|X_k = x_k, X_{k-1} = x_{k-1}...X_0 = x_0) = P(X_{k+1} = x_{k+1}|X_k = x_k)$

Définition 2 (Markov Decision Process)
Un MDP est un quadruplet (S,A,P,R) caractérisé par $\forall t \in \mathbb{N}$*:*

1. *d'un action* $A_t \in A = \{1, 2, ..., |A|\}$
2. *d'un état* $S_t \in S = \{1, 2, ..., |S|\}$
3. *d'un noyau de transition stationnaire* $p(s_{t+1}|s_t, a_t) = P(S_{t+1} = s'|S_t = s, A_t = a)$
4. *d'une fonction R de récompense d'une transition d'un état* s_t *à un état* s_{t+1} *vérifiant* $R(S_0 = s_0, A_0 = a_0, S_1 = s_1, a_1 = a_1, ..., S_{t+1} = s_{t+1}) = R(s_0, a_0, ..., s_{t+1}) = R(s_t, a_t, s_{t+1}) = r_{t+1}$

On définit aussi une politique $\pi(a|s) = P(A_t = a|S_t = s)$

On écrit aussi $\pi \sim X_t$ *la variable aléatoires en paramètres est* X_t
Le triplet $(S_t, A_t, R_t)_{t=0,1,...}$ *forme une chaîne de Markov.*

Objectif

L'objectif est de maximiser J définit par

$$J(\pi) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_{t+1}], i.e \max_{\pi policy} \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_{t+1}],$$

où $0 < \gamma < 1$ est un discount factor.

Pour maximiser J, on va le décomposer grâce aux quantités suivantes:

$$V_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{k+t+1} | S_t = s]$$

$$Q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{k+t+1} | S_t = s, A_t = a] = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

avec

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{k+t+1}$$

Après, plusieurs étapes et grâce au propriété des chaînes de Markov, on arrive à l'équation de Bellman.

Théorème 1 (Équation de Bellman)

$$Q_\pi(s,a) = \sum_{s'} p(s'|s,a).(R(s,a,s') + \gamma V_\pi(s'))$$

$$Q_\pi(s,a) = \sum_{s'} p(s'|s,a).(R(s,a,s') + \gamma \sum_{a'} Q_\pi(s',a').\pi(a'|s'))$$

Le nouveau but est de trouver $Q^\star = \max_\pi Q_\pi$ et $V^\star = \max_\pi V_\pi$.

Admettons un théorème fondamental des MDP.

Théorème 2 (Existence d'une politique optimale) *Reprenons la définition d'un MDP*

1. *Il existe une politique optimale $\pi^\star$ tel que $\forall \pi, s$, $max_a \pi^\star(a|s) \geq max_a \pi(a|s)$*
2. *Il existe plusieurs politiques optimales $\pi^\star$ mais il vérifient toutes $V^\star(s) = V^{\pi^\star}(s)$ $\forall s$*
3. *Il existe plusieurs politiques optimales $\pi^\star$ mais il vérifient toutes $Q^\star(s,a) = Q^{\pi^\star}(s,a)$ $\forall s, a$*
4. $\pi(a|s) = \begin{cases} 1 & si\ a = argmax_{a'} Q*(s,a') \\ 0 & sinon. \end{cases}$ *est une politique optimale sous le nom de* ***Greedy Policy***

Ainsi on trouve les équations fondamentales $\star$ suivantes.

Théorème 3 (Équations de Bellman⋆) *D'après le théorème précèdent.*

$$V^\star(s) = \max_{a \in A} Q^\star(s,a)$$

$$Q^\star(s,a) = \sum_{s'} p(s'|s,a).(R(s,a,s') + \gamma V^\star(s'))$$

$$Q^\star(s,a) = \sum_{s'} p(s'|s,a).(R(s,a,s') + \gamma \max_{a'} Q^\star(s',a'))$$

Remarque 2.1 *On remarque que le terme $\pi(a'|s')$ s'est simplifié et ces équations ressemblent à celles d'un point fixe !*

Finalement, avec cette deuxième équation de Belmann, on a juste à trouver $Q^\star$ avec une méthode du point fixe puis passer à l'espérance pour retrouver J qui sera ainsi maximisé !

Comment trouver $Q^\star$ efficacement ? On pose de manière aléatoire Q_0. Puis on pose la suite:

$$Q_{k+1}(s,a) = \sum_{s'} p(s'|s,a)(R(s,a,s') + \gamma \max_{a'} Q_k(s',a'))$$

Ainsi, cette suite converge vers $Q^\star$.

Tout cela est très beau, mais en pratique on ne connaît pas le noyau de transition $p(s'|s,a)$. On réécrit notre suite comme :

$$Q_{k+1}(s_t,a) = \sum_{s'} R(s_t,a,s')p(s'|s,a) + \gamma \mathbb{E}[\max_{a'} Q_k(.,a')|S_t = s_t, A_t = a]$$

et avec un peu de magie i.e Temporal Difference Methods[1] [2] on peut faire disparaître l'espérance et on obtient :

Algorithme Q-learning/Temporal Difference Method

$$Q_{k+1}(s_t,a_t) \leftarrow (1-\alpha)Q_k(s_t,a_t) + \alpha[R(s_t,a_t,s_{t+1}) + \gamma \max_{a'} Q_k(s_{t+1},a')]$$

où s_{t+1} correspond à l'état du système après l'action a_t et α le learning rate.

On a donc notre algorithme Q-learning.

[1] mise à jour à chaque pas au lieu de calculer une espérence

3 Q-learning

Regroupons maintenant les résultats précédents sur les **chaînes de Markov** afin d'entraîner nos agents à jouer correctement à des jeux vidéo variés et (potentiellement) complexes [3].

3.1 Algorithme

Cette partie sert d'implémentation ***naive*** de l'algorithme **Q-learning** pour résoudre des environnements très simples, notamment Cartpole.

Algorithm 1 Q-Learning simple

Initialiser l'environnement, les variables d'environnement et le nombre d'épisode M
for i allant de 1 à M **do**
    Initialiser l'état s
    while l'agent n'a pas perdu **do**
        Executer l'action a qui vérifie $a_t = arg\max_a Q(s_t, a)$
        Récupération des dans s_{t+1} observations et du la récompense
        Actualisation des variables d'environnement
        Actualisation de la matrice en appliquant Bellman
        Actualisation de s_t avec s_{t+1}
    end while
end for

3.2 Résultats

En implémentant cet algorithme par nous même, nous avons très rapidement eu des résultats corrects. Pour Cartpole, il arrive à faire tenir le bâton droit sans jamais perdre également, c'est vraiment impressionnant. Il faut noter que l'algorithme converge très rapidement vers une solution optimale, c'est-à-dire que l'agent ne fait plus d'erreur et va droit au but.

L'agent continue d'accumuler progressivement des rewards jusqu'à atteindre **1000 points** en moyenne ! On obtient cependant ce score après plus de 100 000 épisodes, ce qui est assez conséquent pour un environnement si basique.

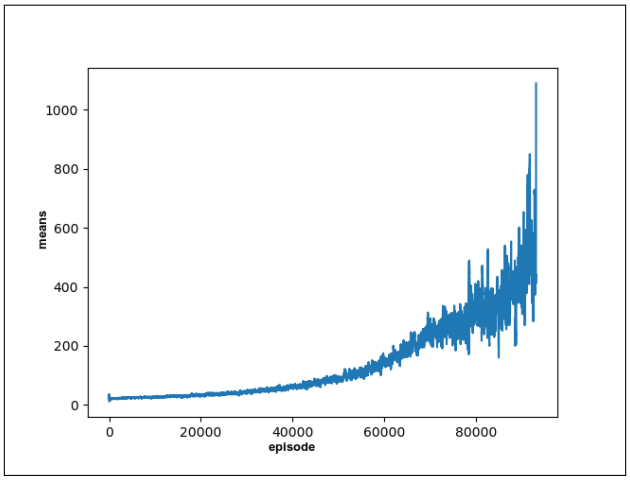


Figure 2: Rewards en fonction du nombre d'épisodes (Cartpole)

3.3 Limites du Q-learning

Le Q-learning nous a donc permis d'entraîner notre IA sur tous ces environnements assez simples. Le problème est que, comme l'on peut voir, l'algorithme repose sur la manipulation d'une matrice de taille proportionnelle au nombre d'actions possible ainsi que de la taille de l'environnement. Typiquement, dans le cas du Cartpole, il n'y avait que 2 actions possibles (droite ou gauche) et l'environnement était constitué de 4 variables, ainsi la matrice Q était déjà assez grande mais cela restait raisonnable.

Or, nous avons voulu nous attaquer à des jeux plus complexes, par exemple le Lunar Lander où il y a 4 actions possibles et 8 variables d'environnement : c'est déjà trop pour notre simple algorithme de Q-learning qui met beaucoup trop de temps à faire seulement une itération.

De plus, l'algorithme du Q-learning ne nous permet pas de traiter des environnements de type RGB comme Pong et Breakout puisque une action prise à chaque itération est liée à un analyse d'une image. Il faut donc améliorer drastiquement notre algorithme.

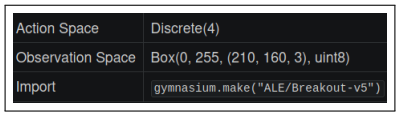


Figure 3: Breakout observation space

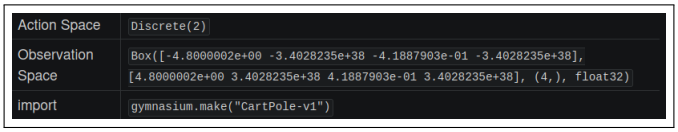


Figure 4: Cartpole observation space

Heureusement, le domaine de l'intelligence artificielle sut plusieurs développement ces dernières années notamment en 2015 avec la publication de recherches scientifiques importantes comme Playing Atari with Deep Reinforcement Learning où ils s'intéressent à la méthode du **Deep Q-learning**.

Notre prochaine objectif est donc de comprendre et d'implémenter cette méthode.

4 Deep Q-learning

Nous avons donc commencé par nous intéresser au deep learning en général, en commençant simplement par créer un multilayer perceptron afin de classifier des chiffres écrits à la main (MNIST). Nous avons appris comment un MLP marche, ainsi que plus précisément les différentes loss functions et optimizers (à l'aide de ressources en ligne Dataflowr). Nous sommes ensuite passés à l'étape supérieure en se formant sur les CNN. Ainsi, nous avons compris la base du deep learning et comment est formé un réseau de neurones avec ses différentes couches et leurs fonctions.

Nous pouvons ainsi commencer ce qui nous intéresse réellement : **le Deep Q-learning** [3].

4.1 Du Q-learning au Deep Q-learning

Comme nous l'avons dit plus tôt, nous voulons à présent nous attaquer à des environnements plus complexes et donc l'algorithme de base de Q-learning avec la matrice Q ne fonctionne plus. L'idée générale est donc de remplacer cette matrice Q par un réseau de neurones renvoyant "virtuellement" les coefficients d'une matrice selon un environnement donné.

En effet, à un état donné, la matrice Q permettait de voir quelle était la meilleure action à faire. Maintenant, nous voulons donner en entrée de notre réseau de neurones un état, et qu'il nous renvoie la meilleure action à faire. Les poids du réseau serait alors mis à jour grâce à l'équation de Bellman.

4.2 Preprocessing

Les environnements les plus complexes nécessitent une analyse d'images comme cité dans la partie précédente. Cependant, un traitement directe des images de l'environnement s'avère lourde et complexe pour les réseaux de neurones. Il est donc indispensable de modifier ces images.

Après de recherches extensives, nous avons trouvé **quatres** étapes importantes du preprocessing à suivre afin d'accélérer la convergence de notre Q matrice:

- **Grayscaling**: L'image reçue devient grise. L'image reçue devient grise.

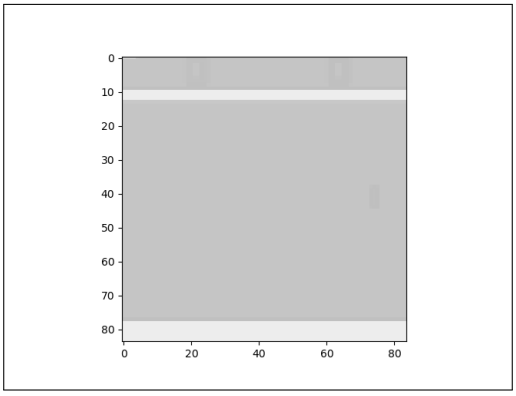
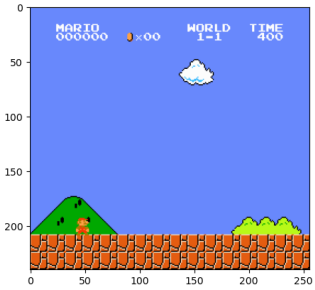
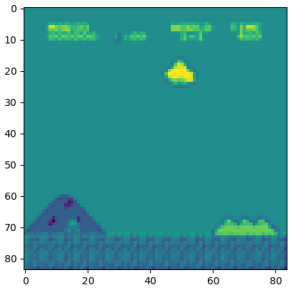


Figure 5: Exemple de Grayscaled Pong

- **Resizing**: On modifie la taille de notre matrice image, on change la taille souvent à **84x84**, il s'agit d'une valeur convenable qu'on a trouvé expérimentalement aprés plusieurs essaies et itérations.



(a) Mario sans resize



(b) Mario avec du resize

- **Frame-Skipping**: pour une valeur n qui représente le Frame-skipping, il s'agit de répéter une action n fois, afin d'explorer le maximum de possibilité et de nouveaux espaces d'observatrion.

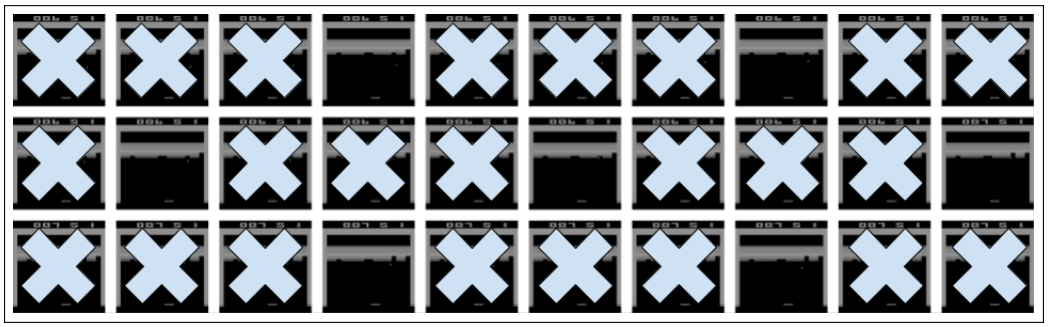
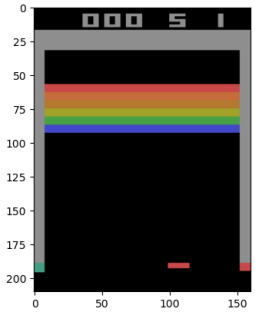


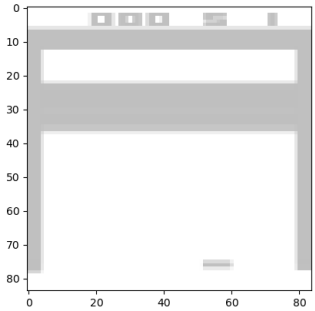
Figure 6: Frame-skipping pour n = 4

- **Frame-Stacking**: pour la même valeur n du Frame-Skipping, on stack les n-dernieres frames pour obternir un tensor de taille **nx84x84**. Le Frame-Stacking permet à l'agent d'analyser n actions et n espaces d'observation à chaque itération au lieu d'une seule action et un seul espace d'observation.

Voici les résultats du preprocessing en entier appliqué à l'environnement Breakouts :



(a) Breakout sans preprocessing



(b) Breakout avec du preprocessing

4.3 Algorithme

L'implémentations de l'algorithme nécessite quelques méthodes fondamentales qui permettent d'accélerer la convergence.

- **Approche ϵ-greedy**
  Nous suivons l'algorithme ϵ-greedy utilisé dans DQL, cette approche est utilisée pour que l'agent explore le maximum de possibilités dans un certain environnement. Cela permet d'éviter les situations où un agent trouve un minimum local de perte et, bien sûr, y reste. Cela s'est produit lorsque l'agent a constaté qu'il obtenait les récompenses maximales en conservant une certaine position dans les jeux breakout et pong. De plus, ce ϵ décroît avec le temps, ce qui signifie que l'agent devient de plus en plus indépendant et doit calculer la meilleure action dans son environnement en fonction de ses valeurs Q

- **Memory buffer**
  Nous commençons par remplir la mémoire tampon afin d'obtenir au moins le minimum d'environnement aléatoire, d'actions, de récompenses, d'états terminaux et de nouveaux états d'observation. Une fois la mémoire tampon remplie, nous choisissons des nombres aléatoires et des états aléatoires (états, nouveaux états, récompenses, états terminaux, états d'actions) qui seront utilisés pour calculer le gradient dans notre situation actuelle afin que Q converge.

Voici l'implémentation algorithmique du Deep Q-Learning [4] :

Algorithm 2 Deep Q-Learning

Initialiser un replay Memory D de capacité N
Initialiser un réseau de neurones Q
for i allant de 1 à M **do**
    Initialiser l'état $s_1 = x_1$ et l'état préprocessé $\phi = \phi(s_1)$
    while l'agent n'a pas perdu **do**
        Choisir une action a :
            avec une probabilité ϵ, choisir une action aléatoire a
            sinon, $a = \arg\max_{a'} Q(s, a')$
        Exécuter l'action a, observer la récompense r et la nouvelle image $\tilde{x}$
        Actualiser $\tilde{s} = (s, a, \tilde{x})$ et préprocesser $\tilde{\phi} = \phi(\tilde{s})$
        Enregistrer $(\phi, a, r, \tilde{\phi})$ dans D
        Prélever aléatoirement une transition $(\phi, a, r, \tilde{\phi})$ de D
        Calculer la cible y :

$$y = \begin{cases} R(s,a) & \text{si } \tilde{\phi} \text{ est terminale} \\ R(s,a) + \gamma \cdot \max_{a'} Q(\tilde{s}, a') & \text{sinon} \end{cases}$$

        Appliquer la descente de gradient à $(y - Q(\phi, a))^2$ par rapport aux paramètres du réseau de neurones Q
        Mettre à jour l'état actuel $s \leftarrow \tilde{s}$
    end while
end for

Pour l'implémentation, nous nous sommes inspirés de plusieurs github permettant de trouver les hyperparamètres optimaux pour les différents jeux, notamment pour les couches de convolution du réseau de neurones afin de gagner du temps.

4.4 Double Deep Q-learning

Une variante a été utilisée ici, qui est d'utiliser un deuxième réseau de neurones lors de l'apprentissage. Sa principale motivation est un critère de stabilité et de convergence. Dans l'équation de Bellman, les paramètres du réseau apparaissent des deux côtés de l'équation. Le deuxième réseau,

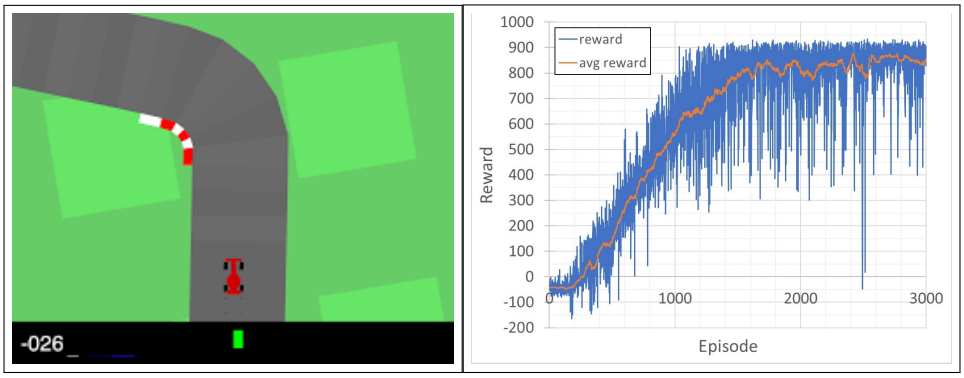
souvent appelé "target", servira à la recherche de la valeur maximale accessible depuis le nouvel état. Il sera d'ailleurs actualisé avec les poids du réseau principal au bout d'un certain nombre d'épisodes d'entraînement.

Pour optimiser la convergence, nous nous sommes à nouveau appuyés sur le papier de 2015 qui se limite seulement à l'assignation de la fonction de récompense R dans le cas où un épisode est terminé. L'équation de Bellman devient:

$$Q(s,a) = R(s,a) + \gamma \cdot (1 - terminated) \cdot \max_{a'} Q'(s',a')$$

4.5 Résultats

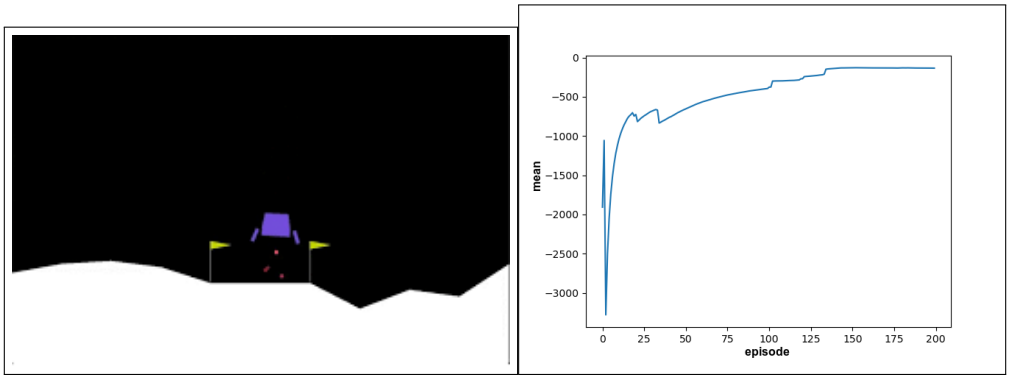
Voici les résultats de l'entraînement de l'environnement **Car-racing**.



(a) (b)

Figure 7: Rewards de Car-racing

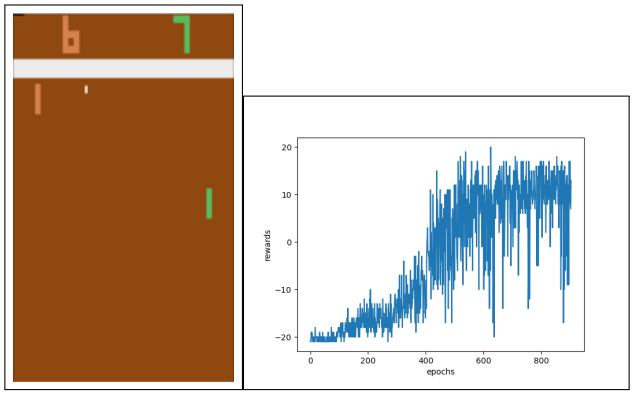
Voici les résultats de l'entraînement de l'environnement **Lunar Lander**.



(a) (b)

Figure 8: Rewards de Lunar Lander

Voici les résultats de l'entraînement de l'environnement **Pong**.



(a) (b)

Figure 9: Rewards de Pong

4.6 Limites du Deep Q-learning

Les algorithmes vus jusqu'ici sont dits "value-based". Cela signifie qu'en sortie du réseau de neurones, pour chaque élément du domaine d'action, une Q-value lui était associée et nous cherchions à maximiser cette dernière pour le choix. Nous allons à présent voir des algorithmes dits "policy-based", qui donnent eux en sortie une distribution de probabilité, correspondant à la politique de jeu en cours, et nous prendrons un *sample* de cette loi comme décision. Le but est *in fine* d'avoir une "cloche" pour chaque état, nous permettant de converger vers une politique optimale. Nous commencerons par la technique la plus naïve de cette catégorie: REINFORCE.

5 REINFORCE

5.1 Fonction Objectif

[5] L'objectif des méthodes fondées sur les politiques est de maximiser le rendement attendu :

Objectif

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_{t+1}\right], i.e \max_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_{t+1}\right]$$

Où θ représente les paramètres de la politique π_θ et r_t les récompenses à l'instant t.

Redonnons une définition de la fonction de *State-value* et de la fonction Q:

$$Q_\pi(a,s) = \mathbb{E}_{\pi_\theta}\left[\sum_{k=0}^{\infty}\gamma^k r_{k+t+1}|S_t = s, A_t = a\right]$$

$$Q_\pi(A_t,S_t) = \mathbb{E}_{\pi_\theta}\left[\sum_{k=0}^{\infty}\gamma^k r_{k+t+1}|S_t, A_t\right]$$

$$V_\pi(s) = \mathbb{E}_{\pi_\theta}\left[\sum_{k=0}^{\infty}\gamma^k r_{k+t+1}|S_t = s\right],$$

5.2 Théorème de Policy Gradient

Théorème 4 (Policy Gradient Theorem) *Ce théorème est central dans la théorie des méthodes par Policy Gradient car il permet d'avoir une expression simple du gradient de J.*

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_{t+1}\right] \propto \mathbb{E}_{\pi\sim S_t}\left[\sum_a Q_\pi(S_t,a)\nabla_\theta \pi_\theta(a|S_t)\right]$$

Le théorème est admis.

Corollaire 1 *On peut à partir de ce théorème en déduire une expression plus simple qui sera utile pour notre algorithme REINFORCE.*

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi\sim S_t}\left[G_t \nabla_\theta \log \pi_\theta(A_t|S_t)\right] = \mathbb{E}_{\pi\sim S_t, A_t=a}\left[Q_\pi(S_t,a)\nabla_\theta \log \pi_\theta(a|S_t)\right]$$

avec $G_t = \sum_{k=0}^{\infty}\gamma^k r_{k+t+1}$

Preuve: 1 *En utilisant les notations précédentes*

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi\sim S_t}\left[\sum_a Q_\pi(S_t,a)\nabla_\theta \pi_\theta(a|S_t)\right] \\
&= \mathbb{E}_{\pi\sim S_t}\left[\sum_a \pi(a|S_t) Q_\pi(S_t,a)\frac{\nabla_\theta \pi_\theta(a|S_t)}{\pi(a|S_t)}\right] \\
&= \mathbb{E}_{\pi\sim S_t}\left[\mathbb{E}_{\pi\sim A_t}\left[Q_\pi(S_t,A_t)\nabla_\theta \log \pi_\theta(A_t|S_t)\right]\right] \\
&= \mathbb{E}_{\pi\sim S_t}\left[\mathbb{E}_{\pi\sim A_t}\left[\mathbb{E}_\pi\left[G_t|A_t,S_t\right]\nabla_\theta \log \pi_\theta(A_t|S_t)\right]\right] \\
&= \mathbb{E}_{\pi\sim S_t}\left[G_t \nabla_\theta \log \pi_\theta(A_t|S_t)\right]
\end{aligned}$$

en rappelant que $Q_\pi(S_t,A_t) = \mathbb{E}_\pi\left[G_t|A_t,S_t\right]$

5.3 Algorithme

En utilisant la méthode TD (temporal difference), on trouve la formule d'update suivante :

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

où α est le learning rate.

5.4 Résultats

Nous avons entraîné Cartpole avec cet algorithme et le résultat est vraiment bluffant : en seulement 250 épisodes, on obtient le même résultat qu'avec le q-learning standard (plus de 100000 épisodes avec celui-ci !).

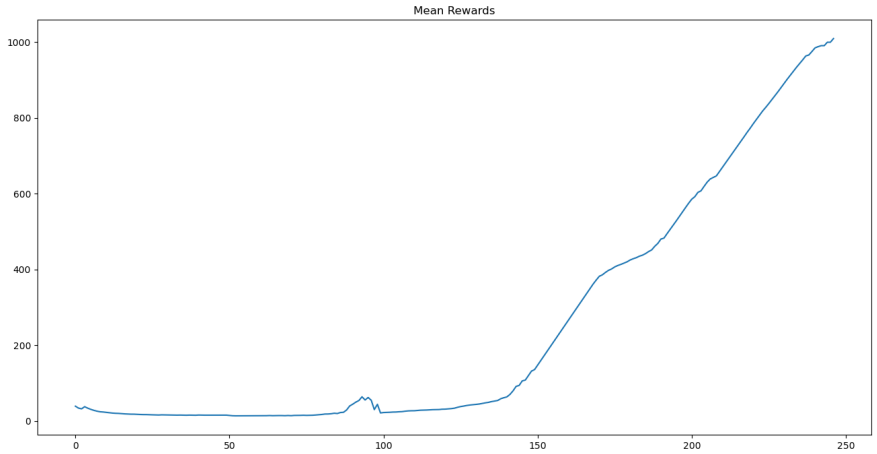


Figure 10: Rewards moyennes pour Cartpole

5.5 Limites de REINFORCE

REINFORCE possède deux principaux problèmes :

- **Variance** : la variances des gradients est élevée, ce qui nous empêche de tirer les meilleures actions possibles. Pour Cartpole cela marchait bien car il y a seulement 2 actions possibles, sinon on est très limité en performance...
- **Exploration** : on peut tomber dans un minimum local même avec une *policy method*, la politique devient donc déterministe ce qui empêche l'exploration.

Pour le problème d'exploration, une solution est de punir l'agent lorsqu'il est trop sûr de lui. Cela peut se traduire en ajoutant un terme d'entropie dans la fonction de perte :

$$H(\pi) = -\sum_a \pi(a|s) \log \pi(a|s)$$

Concernant le problème de variance, la solution trouvée est de soustraire une certaine valeur au terme $q_\pi(a, s)$. Nous allons appeler cette valeur *baseline*, et à partir de cela, nous allons obtenir un nouvel algorithme : REINFORCE with baseline.

5.6 Fonction Advantage

On introduit *adv* la fonction dites *advantage* définie comme :

$$adv(s,a) = Q_\pi(s,a) - b(s),$$

Avec $q_\pi(s,a)$ la fonction d'*action-value* et $b(s)$ la fonction de *baseline*.

5.7 Algorithme REINFORCE with baseline

En utilisant le REINFORCE with baseline, on trouve la formule d'update suivante en considérant la méthode de différence finie et avec $G_t = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$:

$$\theta \leftarrow \theta + \alpha(G_t - b(t))\nabla_\theta \log \pi_\theta(a_t|s_t)$$

5.8 De REINFORCE à A2C

Toute la question est maintenant de savoir comment choisir $b(s)$. Une approche naïve est de choisir la moyenne des $Q_\pi(a,s)$ des derniers épisodes. Cependant, pour des environnements complexes, cela n'est pas suffisant.

En remarquant que $Q_\pi(a,s) = V_\pi(s) + A_\pi(s,a)$, comme $V_\pi(s)$ ne dépend pas de a, c'est une bonne idée de choisir $b(s) = V_\pi(s)$.

Cependant, on ne connaît pas $V_\pi(s)$ (tout comme $Q_\pi(a,s)$) : il va falloir l'estimer !

6 Advantage Actor-Critic (A2C)

Pour estimer V_π, l'idée est donc d'utiliser un autre réseau de neurones. La méthode obtenue s'appelle Advantage Actor-Critic (A2C)[6].

Nous allons étudier les méthodes Actor-Critic, une architecture hybride combinant une méthode value-based et une méthode policy-based permettant de stabiliser l'apprentissage en réduisant la variance. Les deux réseaux de neurones sont :

- Un Actor qui contrôle le comportement de notre agent (policy-based method)
- Un Critic qui mesure la qualité de l'action choisie (value-based method)

6.1 Fonction Advantage

La fonction d'avantage mesures la valeur relative d'une action en comparaison à l'action moyenne dans un certain état :

$$adv(s,a) = Q_\pi(s,a) - V_\pi(s),$$

Avec $Q_\pi(s,a)$ la fonction d'*action-value* et $V_\pi(s)$ la fonction de *valeur*.

6.2 Policy Gradient avec Advantage

L'intérêt principale de l'*Advantage function* est qu'il ne modifie pas $\nabla_\theta J(\theta)$.

Théorème 5 (Policy Gradient avec Advantage)

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[Q_\pi(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t|S_t) \right] = \mathbb{E}_\pi \left[adv(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t|S_t) \right]$$

Preuve: 2 *On a*

$$\begin{aligned}
\mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(A_t|S_t) V_\pi(S_t) \right] &= \mathbb{E}_\pi \left[\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} V_\pi(S_t) \right] \\
&= \sum_{s,a} \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} V_\pi(s) \\
&= \sum_s V_\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) \\
&= \sum_s V_\pi(s) \nabla_\theta (\sum_a \pi_\theta(a|.))(s) \\
&= \sum_s V_\pi(s) \nabla_\theta (s' \mapsto 1)(s) \\
&= 0
\end{aligned}$$

Par linéarité de l'espérance, on obtient le résultat voulu.

6.3 Algorithme

L'algorithme A2C met à jour les paramètres de la politique et ceux de la fonction de valeur selon les équations suivantes :

$$\begin{aligned}
\theta &\leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) adv(s_t, a_t) \\
\phi &\leftarrow \phi - \beta \nabla_\phi \left(V_\pi(s_t) - G_t \right)^2
\end{aligned}$$

Avec α et β les *learning rates* de la politique et de la fonction de valeur.

7 Asynchronous Advantage Actor-Critic (A3C)

Cet algorithme [6] reprend A2C en exploitant les résultats de plusieurs agents, ayants chacun leur réseau, améliorant ainsi le réseau global. Cela correspond à faire du multiprocessing.
En termes d'implémentation, une nouvelle classe d'agent Worker est créée, où chaque agent actualise son réseau par rapport au global, qui bénéficie d'un gradient plus élargi en champ d'action dans l'environnement.

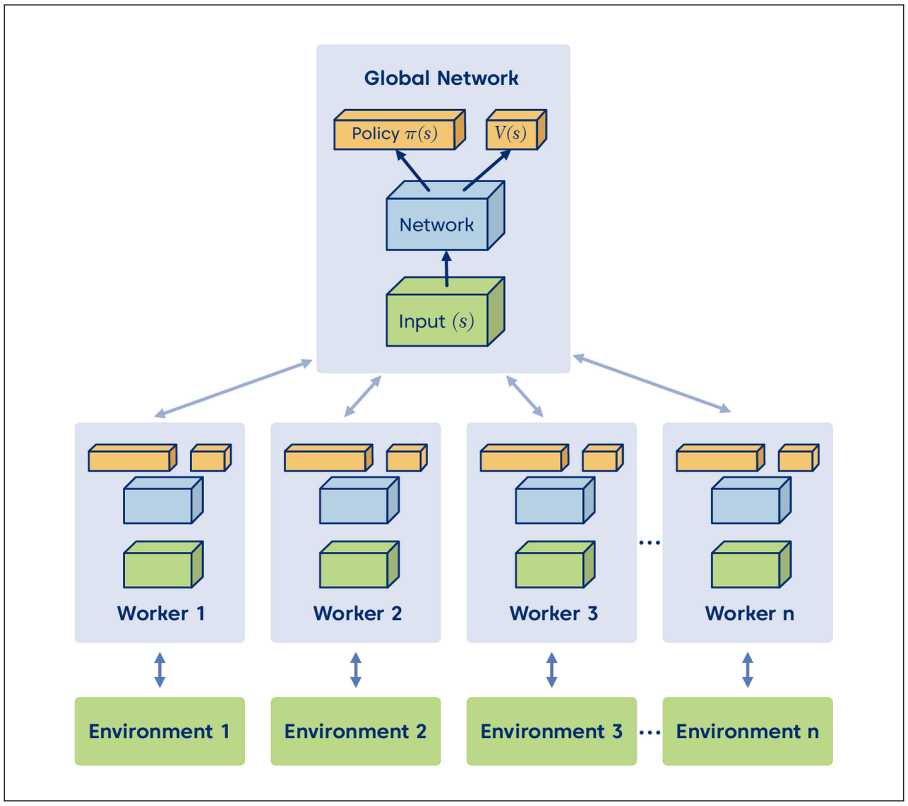


Figure 11: Schéma A3C

Cette méthode est de loin la plus puissante vue jusque maintenant : elle permet d'avoir une convergence beaucoup plus rapides pour les environnements déjà vus, et donc permet d'attaquer des environnements encore plus complexes tel que Mario.

Pour implémenter cette méthode, nous nous sommes aidés de ChatGPT afin de paralléliser notre code A2C car nous ne savions pas vraiment comment marchait le multiprocessing avec pytorch.

8 Conclusion

Ce rapport a exploré diverses méthodes pour l'apprentissage par renforcement, en commençant par les approches traditionnelles telles que le Q-learning, et en évoluant vers des techniques plus avancées comme le Deep Q-learning, les variantes d'Actor-Critic (A2C et A3C), ainsi que le Proximal Policy Optimization (PPO).

Le document a débuté par la formalisation du problème et des fondements théoriques du Q-learning, y compris ses algorithmes, ses résultats et ses limitations. Ensuite, le Deep Q-learning a été étudié, en mettant en évidence les améliorations apportées par le double Deep Q-learning et les techniques de prétraitement des données.

Nous avons également étudié les algorithmes policy-based. L'algorithme REINFORCE et ses variantes ont été analysés, avec un accent sur l'utilisation de fonctions d'avantage et de baselines pour améliorer la stabilité et la performance de l'apprentissage. Les méthodes Actor-Critic ont été explorées, en soulignant les avantages de l'approche asynchrone (A3C) et les améliorations apportées par A2C.

L'étude comparative de ces différentes approches a révélé leurs forces et faiblesses respectives, offrant ainsi une vue d'ensemble des techniques actuelles en apprentissage par renforcement. Les résultats obtenus montrent que chaque méthode a des contextes d'application spécifiques où elle excelle, soulignant l'importance de choisir l'algorithme approprié en fonction du problème à résoudre.

En conclusion, l'apprentissage par renforcement continue d'évoluer rapidement, et les innovations dans ce domaine ouvrent la voie à des applications toujours plus complexes et performantes. Les méthodes étudiées dans ce rapport constituent des fondations solides pour le développement de systèmes intelligents capables de s'adapter et d'apprendre de manière autonome dans des environnements dynamiques et incertains.

9 Bibliographie

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[2] J. Ma, "Discerning temporal difference learning," 2024.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[4] M. Lapan, *Deep Reinforcement Learning Hands-On*. Birmingham, UK: Packt Publishing, 2018.

[5] J. Zhang, J. Kim, B. O'Donoghue, and S. P. Boyd, "Sample efficient reinforcement learning with REINFORCE," *CoRR*, vol. abs/2010.11364, 2020.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016.