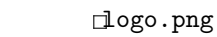

PROJ104

Rapport de projet

2024

Ma première IA jouant à un jeu vidéo

Besancenot Henri
Daboust Axel
Gilli Clément
Khoutaibi Iliass

Enseignant
Pascal Bianchi

Contents

1 Introduction

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

Pour cela, nous sommes passé par plusieurs étapes progressives en difficulté que nous allons vous présenter.

2 Formalisation du problème

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.

Définition 1 (Chaîne de Markov)
(X_n) une suite de VA est une chaîne de Markov si $P(X_{k+1}|X_k, X_{k-1}...X_0) = P(X_{k+1}|X_k)$

Définition 2 (Markov Decision Process)
Un MDP est constitué :

1. *d'un agent $A_t \in A = \{1, 2, ..., |A|\}$*
2. *d'un environnement $S_t \in S = \{1, 2, ..., |S|\}$*
3. *d'un noyau de transition $p(s'|s, a) = P(S_{t+1} = s'|S_t = s, A_t = a)$*
4. *d'une politique $\pi(a|s) = P(A_t = a|S_t = s)$*

On peut ajouter à cela dans notre cas une fonction de récompense $r(S_t, A_t) = E[R_{t+1}|S_t, A_t]$ où R_t est une VA représentant la récompense.

L'objectif est de maximiser J définit par $J(\pi) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \beta^t R_{t+1}]$ où $0 < \beta < 1$

On peut montrer que $(S_t, A_t, R_t)_{t=0,1,...}$ est une chaîne de Markov. De plus, on a

Corollaire 1
$P(S_{t+1} = s', A_{t+1} = a', R_{t+1} = r'|S_t = s, A_t = a, R_t = r)$
$= P(R_{t+1} = r'|S_{t+1} = s', A_{t+1} = a', S_t = s, A_t = a, R_t = r)$
$= P(A_{t+1} = a'|S_{t+1} = s', R_{t+1} = r', S_t = s, A_t = a, R_t = r)$
$= P(S_{t+1} = s'|A_{t+1} = a', R_{t+1} = r', S_t = s, A_t = a, R_t = r)$
$= r(S_t, A_t)\pi(a|s)p(s'|s, a)$

Pour trouver comment maximiser J, on va décomposer J : on pose

$Q_\pi(s, a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \beta^t R_{t+1}|S_0 = s, A_0 = a]$

On a donc $J(\pi) = \sum_{s,a} Q_\pi(s, a).\mathbb{P}(S_0 = s).\pi(a|s)$

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

Théorème 1 (Équation de Bellman)
$Q_\pi(s,a) = r(s,a) + \beta \sum_{s',a'} Q_\pi(s',a').\pi(a'|s').p(s'|s,a)$

Le nouveau but est de trouver une politique π qui maximise Q_π. Pour cela, il faut d'abord utiliser une méthode du point fixe pour trouver Q (il existe).

On pose à s fixé, $\pi'(a|s) = 1$ si $a = argmax_{a'} Q_\pi(s,a')$ et 0 sinon

Il est alors évident que $\forall s, a, Q_{\pi'}(s,a) \geq Q_\pi(s,a)$. On itère ce processus plein de fois jusqu'à converger (on peut montrer que cela converge en temps fini) vers un $Q^\star(s,a)$ avec une politique $\pi^\star$ optimale :

$$\pi^\star(a|s) = \begin{cases} 1 & \text{si } a = argmax_{a'} Q_\pi(s,a') \\ 0 & \text{sinon.} \end{cases}$$

À partir de ce $Q^\star$ optimal, on peut définir une nouvelle équation de Bellman :

Théorème 2 (Équation de Bellman$\star$)
$Q^\star(s,a) = r(s,a) + \beta \sum_{s'} max_{a'} Q^\star(s',a').p(s'|s,a)$
On remarque que le terme $\pi(a'|s')$ s'est simplifié.

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 ou bien en choisissant une loi (uniforme par exemple). Puis
$Q_{k+1}(s,a) = \alpha[r(s,a) + \beta \sum_{s'} max_{a'} Q_k(s',a').p(s'|s,a)] + (1-\alpha)Q_k(s,a)$ avec $0 < \alpha < 1$
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 :
$Q_{k+1}(s_t,a) = Q_k(s_t,a) + \alpha(r(s_t,a) + \beta \mathbb{E}[max_{a'} Q_k(s_{t+1},a')|S_t = s_t, A_t = a] - Q_k(s_t,a))$
et avec un peu de magie on peut faire disparaître l'espérance et on obtient :

Théorème 3 (Algorithme Q-learning)
$Q_{k+1}(s_t,a) = Q_k(s_t,a) + \alpha[r(s_t,a) + \beta max_{a'} Q_k(s_{t+1},a') - Q_k(s_t,a)]\mathbb{1}_{(S_t=s_t, A_t=a)}$
où s_{t+1} correspond à l'état du système après l'action a.

On a donc notre algorithme de Q-learning.

3 Q-learning

Let us now regroup the previous **Markov chain** results in order to train our agents to properly play various and (potentially) complex video games.

3.1 L'algorithme

This part serves as a ***primary*** implementation of the **Q-learning** algorithm that the PROJ104 team came up with to solve very simple environments, notably Cartpole and Mountain Car.

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 = argmax_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
```

Ensuite, nous nous sommes attaqué à des environnements où cette fois-ci les actions étaient continues et non plus discrètes. Nous sommes restés sur des environnements très simple : Mountain Car Continuous et Pendulum. Pour cela, nous avons commencé par discrétiser les actions et nous avons gardé exactement le même algorithme, mais au lieu de stocker directement les actions dans la matrice Q, nous avons stocké les indices des actions associées.

3.2 Résultats

En implémentant cet algorithme, nous avons très rapidement eu des résultats corrects. Par exemple, pour le jeu du pendule, il arrive à se stabiliser en haut directement et y reste sans jamais perdre. 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.

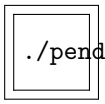

Figure 1: Pendulum

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 voulons 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.

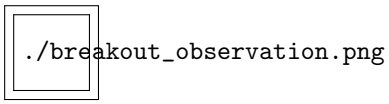

Figure 2: Breakout observation space

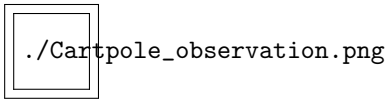

Figure 3: Cartpole observation space

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 comme Data Flowr notamment). 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**.

4.1 From Q-learning to 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:

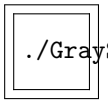

Figure 4: Exemple du preprocessing appliqué au jeu Breakout

- **Grayscaling**: L'image reçue devient grise.
- **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.
- **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.
- **Frame-Stacking**: pour la même valeur n du Frame-SKipping, on stack les n-dernieres frames pour obternir un tensor de taille **4x84x84**. 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.

4.3 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:

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