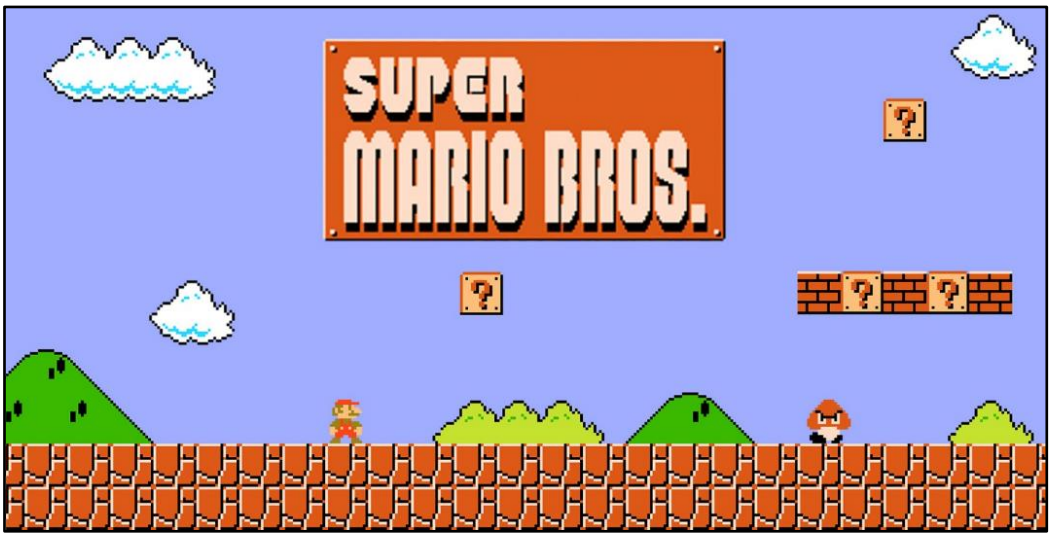
ESIEE PARIS

PROJET DE MACHINE LEARNING

Apprentissage par renforcement sur Super Mario Bros.



Par : Alex FOULON, Thomas LE MAGNY et Erwan GAUTIER
Le : 22 novembre 2023

Table des matières

Introduction

Le machine learning, ou apprentissage automatique, est un sous-domaine de l'intelligence artificielle qui permet aux machines d'apprendre à partir de données. En utilisant des algorithmes, les systèmes améliorent leurs performances dans une tâche spécifique avec l'expérience, sans être explicitement programmés pour chaque situation. Ils détectent des patterns ou des structures dans les données pour faire des prédictions ou prendre des décisions.

Notre objectif pour ce projet est de développer une intelligence artificielle basée sur le machine learning qui sera capable d'apprendre à jouer à un jeu par renforcement, c'est-à-dire que l'IA doit apprendre de ses succès et erreurs afin d'obtenir le meilleur score possible.

Initialement, nous avions décidés de développer un modèle d'IA pour le jeu « Enter the Gungeon ». Plus tard, à la suite d'une analyse des solutions possibles et des contraintes, nous avons finalement choisi le jeu « Super Mario Bros » qui était un très bon candidat pour nos expériences.

Dans l'état de l'art, nous allons parler de l'IA dans les jeux vidéo de façon générale avant de nous pencher sur le cas de Mario en particulier. Nous présenterons ensuite notre propre approche pour le développement d'un modèle basé sur le renforcement, ainsi que les résultats que nous avons obtenus.

État de l’art

Introduction

Définition : *L'intelligence artificielle est un ensemble de théories et de techniques visant à réaliser des machines capables de simuler l'intelligence humaine*.

Avant les années 1990, le domaine de l'informatique lié à l'intelligence artificielle (IA) a connu un véritable désintérêt de la part des investisseurs, ce qui a ralenti, voire fortement freiné, la recherche dans ce secteur. Cette période, qui a duré environ vingt ans, est surnommée "l'hiver de l'IA". Les progrès technologiques ont permis de contribuer à la renaissance de l’IA. Le premier est l'émergence du Web dans les années 90, l'afflux de données qu'il a engendré a permis de nourrir les algorithmes d'IA, notamment les nouveaux réseaux de neurones qui pallient le problème de l'explosion combinatoire. Et le second est l'importante augmentation de la puissance de calcul et de la mémoire des machines et la baisse du prix de ces dernières.

De nombreuses applications peuvent désormais intégrer des algorithmes d'intelligence artificielle. Un domaine de l'informatique qui connaît également un grand succès à cette époque, et qui s'est imposé dans de nombreux foyers, est celui du jeu vidéo.

L'intelligence artificielle et les jeux de plateau tels que les échecs ou le Go ont toujours été étroitement liés. Ces jeux offrent un cadre idéal pour tester diverses méthodes et algorithmes d'IA. C'est donc naturellement que l'intérêt s'est porté sur ces nouveaux supports.

Le principal objectif de l'IA tel que nous le connaissons aujourd'hui dans un jeu vidéo est de rendre l'univers du jeu le plus cohérent et le plus vivant possible, afin d'améliorer le plaisir et l'immersion du joueur.

Toutefois, il est courant de parler de l'IA dans les jeux vidéo, bien que beaucoup n'intègrent pas réellement d'algorithmes sophistiqués. Parmi les différentes formes d'IA que l'on retrouve, citons :

- Les IA adaptatives ajustent la difficulté en fonction des performances du joueur. Par exemple, dans certains jeux de tir, l'IA peut devenir plus agressive si le joueur réussit trop facilement.
- Les IA procédurales génèrent du contenu de manière aléatoire ou suivant des règles complexes, comme dans les jeux de survie où les environnements sont générés dynamiquement.
- Les IA d'apprentissage progressent au contact des joueurs, adaptant leurs stratégies en fonction des interactions.

Selon la définition stricte de l'IA énoncée précédemment, ces différentes IA trouvées dans les jeux vidéo correspondent bien à cette définition, puisqu'elles visent à donner vie aux PNJ au sein de l'univers où évolue le joueur. Cependant, souvent, elles ne font pas appel au Machine Learning ou au Deep Learning, mais s'appuient plutôt sur des algorithmes comme le pathfinding, MCTS, ou Minimax. Cela dit, l'IA continue de transformer le monde des jeux vidéo, offrant des expériences toujours plus immersives et engageantes.

Technologies clés

Dans le domaine des jeux vidéo, l'application de l'intelligence artificielle s'est diversifiée et perfectionnée avec le temps. De nombreuses méthodes et technologies ont été développées pour répondre aux besoins spécifiques des jeux, allant de la navigation simple des personnages à la génération de mondes dynamiques et interactifs. Voici un aperçu des technologies clés qui ont joué un rôle crucial dans l'intégration de l'IA dans les jeux vidéo.

Pathfinding et navigation :

- A* et Dijkstra : Ces algorithmes de pathfinding sont cruciaux pour la navigation des personnages non-joueurs (PNJ) dans les jeux. A* combine les meilleurs aspects de l'efficacité et de l'optimisation, permettant aux PNJ de trouver le chemin le plus court et le plus rapide dans des environnements complexes. Dijkstra, bien que plus simple et plus lent que A*, est utile dans des environnements où tous les mouvements ont le même coût (non pondéré).

Simulation de la foule :

- Comportement de groupe : La simulation de foule est utilisée pour modéliser le mouvement et le comportement de multiples personnages. Des jeux comme "Assassin's Creed" et "Hitman" utilisent des techniques avancées pour simuler des foules réalistes, où chaque individu réagit de manière cohérente aux joueurs et à l'environnement.

IA comportementale :

- Arbres de comportement et machines à états Finis : Ces structures sont essentielles pour modéliser le comportement des PNJ. Les arbres de comportement permettent une planification dynamique et contextuelle des actions des PNJ, tandis que les machines à états finis offrent un moyen plus simple mais efficace de gérer des comportements variés et conditionnels.

Apprentissage et adaptation :

- IA adaptative : Ces systèmes permettent aux PNJ d'adapter leur comportement en fonction des actions des joueurs. Ils peuvent apprendre de l'environnement et des joueurs, offrant une expérience de jeu plus dynamique et personnalisée.

Procédural et génération de Contenu :

- La Création de Mondes : La génération procédurale utilise des algorithmes pour créer automatiquement des contenus, comme des niveaux ou des mondes. Des jeux comme "No Man's Sky" ou "Minecraft" utilisent cette technologie pour offrir des expériences de jeu quasi infinies et uniques.

Dialogues et interaction :

- Des technologies avancées (tel que les NPC) sont utilisées pour rendre les dialogues et les interactions avec les PNJ plus réalistes et dynamiques. Ceci inclut le traitement du langage naturel et des systèmes capables de générer des réponses en temps réel basées sur le contexte du jeu.

Réseaux de neurones et apprentissage profond :

- Bien que leur utilisation soit encore limitée, les réseaux de neurones et l'apprentissage profond commencent à jouer un rôle dans le développement des jeux vidéo. Ils sont utilisés pour des tâches telles que l'amélioration des graphiques, la modélisation des comportements des PNJ, et même pour créer des éléments de jeu entiers.

Jeux emblématiques & développements historiques

Cette section met en lumière plusieurs études de cas et jeux emblématiques qui ont marqué l'histoire de l'intelligence artificielle dans l'industrie du jeu vidéo.

Pong et Space Invaders : Ces jeux classiques des années 70 ont introduit des formes rudimentaires d'IA. Dans Pong, l'IA se limitait à déplacer la raquette adverse de manière prévisible. Space Invaders a franchi un pas supplémentaire, avec des ennemis qui changeaient de vitesse et de direction, bien que toujours selon un schéma préprogrammé. Ces jeux symbolisent les premières étapes vers une interaction dynamique entre le joueur et l'ordinateur.

Avec l'avènement des jeux plus complexes dans les années 90, les développeurs ont commencé à intégrer des algorithmes de pathfinding tel que A* et Dijkstra. Ces algorithmes ont permis aux personnages non-joueurs (PNJ) de naviguer de manière plus intelligente dans des environnements plus complexes, marquant un progrès significatif dans la capacité des IA à interagir de manière crédible avec le joueur et l'environnement de jeu.

Le jeu Pac-Man a été assez important dans le développement de l'intérêt pour les jeux vidéo dans le milieu de la recherche en IA. Ainsi, Pac-Man a été considéré comme un problème intéressant pour l'IA dès le début des années 90, mais la recherche a réellement commencé dessus au début des années 2000.

Deep Blue, développé par IBM, a marqué un tournant historique en 1997 en battant le champion du monde d'échecs Garry Kasparov. Cet événement a non seulement démontré la capacité de l'IA à maîtriser des jeux complexes, mais a également mis en évidence le potentiel de l'IA dans le traitement de problèmes stratégiques et de prise de décisions.

AlphaGo, créé par DeepMind, a réalisé une percée dans l'usage de l'apprentissage profond en battant le champion du monde de Go, Lee Sedol, en 2016. Cette victoire a montré la capacité des réseaux de neurones profonds à apprendre et à exceller dans des jeux complexes, ouvrant la voie à des applications plus larges de l'IA dans les jeux vidéo.

Des titres récents comme "The Witcher 3", "Red Dead Redemption 2" et "The Last of Us" ont intégré des IA avancées pour créer des PNJ réalistes et des mondes dynamiques. Ces jeux illustrent comment l'IA peut améliorer l'immersion et la réactivité des mondes de jeu, offrant aux joueurs des expériences uniques et personnalisées.

Les jeux en monde ouvert comme "GTA V" et "Cyberpunk 2077" utilisent des IA pour gérer des éléments du jeu tels que le trafic, les comportements des foules, et les réactions des PNJ à l'environnement et aux actions du joueur. Ces systèmes d'IA contribuent à créer des mondes vivants et crédibles, où chaque élément réagit de manière cohérente et réaliste.

Étude du cas de Super Mario Bros

"Super Mario Bros", un jeu classique développé par Nintendo, s'est avéré être un excellent terrain d'expérimentation pour l'intelligence artificielle. Sa structure de jeu linéaire, couplée à des défis de navigation et de prise de décision, en fait un cas d'étude idéal pour l'application et le test de diverses techniques d'IA.

L'utilisation de l'IA dans "Super Mario Bros" a été explorée par de nombreux chercheurs et développeurs. Des événements tels que "The 2010 Mario AI Championship", successeur de "The 2009 Mario AI Competition", ont invité chercheurs et développeurs à mettre à l'épreuve leurs compétences en intelligence artificielle en créant des agents capables de jouer à "Super Mario Bros".

En plus de ces initiatives, on trouve également des individus relevant ce défi, avec des projets d'implémentation d'algorithmes de Machine Learning disponibles sur des plateformes comme GitHub et YouTube.

Voici quelques approches d'IA utilisées pour automatiser le contrôle de Mario :

- **Apprentissage par renforcement** : Dans cette approche, l'IA apprend en recevant des récompenses pour des actions positives, telles que terminer un niveau ou collecter des pièces, et des pénalités pour des actions négatives, comme mourir ou perdre du temps. Des algorithmes comme le Q-Learning et les Deep Q-Networks ont été utilisés pour former des agents à naviguer efficacement dans les niveaux.

- **Algorithmes évolutionnaires** : Ces algorithmes ont pour inspiration la théorie de l'évolution. Plusieurs générations d'agents sont testées, et les plus performants sont sélectionnés pour créer la génération suivante. Des techniques telles que les Algorithmes Génétiques et la Programmation Génétique ont été utilisées pour développer progressivement des agents capables de performances optimales.

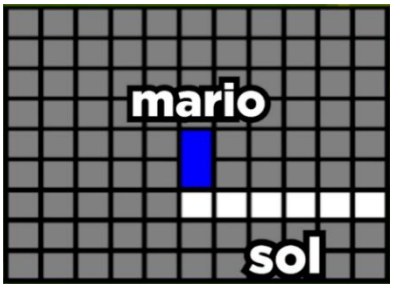
- **Réseaux de neurones et apprentissage profond** : Ces modèles, notamment ceux impliquant l'apprentissage profond, peuvent apprendre des stratégies de jeu complexes à partir de vastes ensembles de données. Des architectures comme les réseaux de neurones convolutifs (CNN) ont été explorées pour traiter les données visuelles du jeu et prendre des décisions en temps réel (YOLO par exemple).

Apprentissage par Renforcement avec NEAT

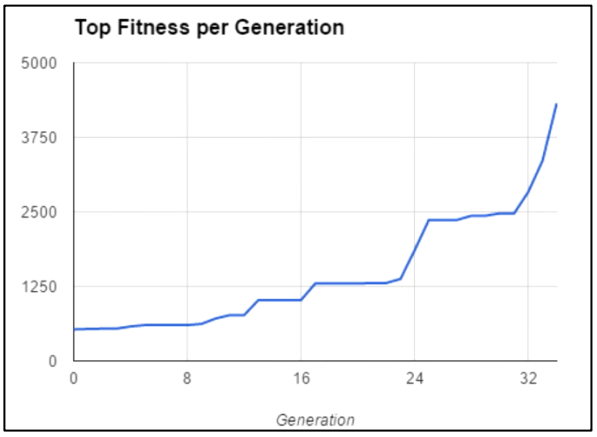
L'objectif est de créer un programme d'intelligence artificielle conçu pour jouer à "Super Mario World". En utilisant l'algorithme NEAT (Neuro Evolution of Augmenting Topologies), le programme fait évoluer un réseau de neurones en croisant les gènes des espèces les plus performantes. Notez qu'il n'est pas obligatoire d'utiliser des réseaux de neurones pour appliquer des algorithmes évolutionnaires.

Dans le programme, chaque réseau de neurones représente une solution potentielle au problème. L'algorithme NEAT ajuste et améliore ces réseaux au fil des générations, en sélectionnant ceux qui performent le mieux et en les utilisant pour créer la génération suivante. Au fur et à mesure le programme va ajouter des mutations au réseau de neurone, en essayant de relier une entrée à une sortie. Le succès d'un réseau est mesuré par la distance parcourue dans le niveau et le temps qui s'est écoulé, de sorte à encourager l'IA à finir le niveau rapidement.

Les réseaux prennent comme entrée une matrice représentant l'environnement autour de Mario. Les sorties du réseau correspondent aux touches de la manette :



Sur le graphique qui suit, on observe que Mario a réussi à compléter un niveau au bout de 34 générations, démontrant l'efficacité de l'apprentissage par renforcement et de l'évolution neuronale dans les jeux vidéo.



Sources :

- https://www.youtube.com/watch?v=qv6UVOQ0F44&t=209s
- https://www.youtube.com/watch?v=F63GNXGHVwM

Autres approches de deep learning pour Super Mario Bros

Des chercheurs ont exploré l'utilisation de l'apprentissage par renforcement profond pour jouer à "Super Mario Bros", en se basant sur des réseaux de neurones convolutifs et des techniques d'apprentissage par renforcement.

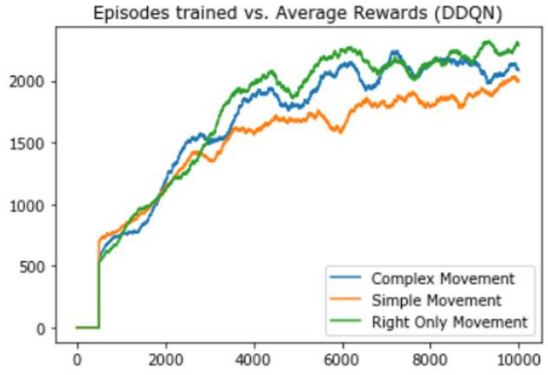
Initialement, l'expérimentation envisagée se basait sur l'apprentissage par Q-learning, qui utilise un tableau 2D pour stocker toutes les combinaisons possibles d'états et d'actions. Cependant, il s'est avéré impossible d'appliquer le Q-learning en raison de la nécessité de stocker une très grande Q-table.

Le projet a donc utilisé l'algorithme DQN comme modèle de base. Les algorithmes DQN utilisent le Q-learning pour apprendre la meilleure action dans un état donné et un réseau de neurones profonds pour estimer la fonction de valeur Q.

Le réseau de neurones utilisé était un réseau convolutionnel (CNN) à 3 couches, suivi de deux couches linéaires entièrement connectées, avec une sortie unique pour chaque action possible.

Pour obtenir une valeur Q plus précise, le réseau DDQN a été utilisé pour comparer les résultats expérimentaux avec le réseau DQN précédent. Pour atténuer la surestimation, le DDQN utilise deux réseaux Q : un pour obtenir les actions et l'autre pour mettre à jour les poids par rétropropagation. Le réseau Q^ copie simplement les valeurs de Q* à chaque n étapes.

Ci-dessous, deux graphiques, le DDQN (Double Deep Q-Network) et le DQN (Deep Q-Network), appliqués à "Super Mario Bros" avec trois mouvements : mouvement complexe, mouvement simple et mouvement uniquement vers la droite.



D'après les résultats ci-dessus, nous observons que le DDQN est plus efficace que le DQN pour ce type de tâche, et que des stratégies de mouvement plus complexes permettent généralement d'atteindre de meilleures performances dans "Super Mario Bros".

Sources :

- https://www.analyticsvidhya.com/blog/2021/06/playing-super-mario-bros-with-deep-reinforcement-learning/
- https://blog.paperspace.com/building-double-deep-q-network-super-mario-bros/

Enjeux et défis actuels

Dans le monde en constante évolution des jeux vidéo, l'IA continue de rencontrer des enjeux et des défis qui façonnent son développement et son application. Un des principaux défis est l'équilibrage et le design de l'IA, où les développeurs s'efforcent de créer une intelligence artificielle qui est à la fois défiante et divertissante. Une IA trop prévisible ou facile peut rendre un jeu ennuyeux, tandis qu'une IA trop difficile peut le rendre frustrant. Trouver le juste équilibre est une tâche compliquée qui nécessite une compréhension approfondie des mécanismes de jeu et du comportement des joueurs.

Un autre enjeu majeur réside dans la personnalisation de l'expérience de jeu. Avec l'émergence de techniques d'apprentissage automatique et de profiling, les jeux ont le potentiel de s'adapter de plus en plus aux préférences individuelles des joueurs, offrant une expérience personnalisée et unique. Cependant, cela soulève des questions concernant la confidentialité des données et la création de bulles de filtrage, où les joueurs pourraient se retrouver enfermés dans des expériences de jeu qui reflètent trop étroitement leurs habitudes passées. Ce principe est utilisé dans le magnifique jeu “Echo” développé et édité par le studio indépendant danois Ultra Ultra, dans lequel le joueur se bat contre des clones de lui-même, reproduisant le style de jeu du joueur au niveau précédent.

Alors que les jeux deviennent plus réalistes et immersifs, les implications de l'utilisation de l'IA peuvent créer des comportements addictifs. Les développeurs sont confrontés à la tâche de concevoir des jeux qui sont responsables et éthiques, tout en étant engageants et divertissants.

L'intégration continue de technologies d'IA plus avancées pose le défi de maintenir une accessibilité et une compréhension de ces systèmes pour les développeurs. Avec l'avènement de techniques telles que l'apprentissage profond, les développeurs doivent non seulement maîtriser ces technologies, mais aussi comprendre comment les intégrer de manière significative dans le gameplay.

Conclusion

En explorant le développement et l'évolution de l'intelligence artificielle dans le monde des jeux vidéo, il est clair que l'IA a joué un rôle important dans la transformation de l'expérience de jeu. De ses débuts modestes dans des jeux comme Pong et Space Invaders à ses applications complexes dans les mondes ouverts et dynamiques des jeux modernes, l'IA a constamment repoussé les limites de ce qui est possible dans le divertissement interactif.

Les avancées technologiques, notamment dans les domaines du pathfinding, de la simulation de foule, de l'IA comportementale, et de l'apprentissage automatique, ont permis de créer des expériences de jeu plus riches et plus immersives. Les jeux vidéo ne sont plus simplement des défis basés sur des règles fixes, mais des mondes vivants, peuplés de personnages qui peuvent apprendre, s'adapter et réagir de manière crédible aux actions des joueurs.

Cependant, avec ces avancées viennent des défis significatifs. L'équilibre entre une IA défiante et divertissante, la personnalisation des expériences de jeu tout en respectant la vie privée des joueurs, et la navigation dans les questions éthiques posées par des jeux de plus en plus réalistes et immersifs, sont autant de sujets qui nécessitent une attention continue. De plus, l'intégration des technologies d'IA avancées pose des questions sur l'accessibilité et la compréhension de ces systèmes pour les développeurs et les joueurs.

En regardant vers l'avenir, il est évident que l'IA continuera à jouer un rôle essentiel dans l'industrie du jeu vidéo. Les technologies émergentes, telles que l'apprentissage profond, la réalité virtuelle et augmentée, et les nouvelles formes de narration, offrent des possibilités passionnantes pour des expériences de jeu encore plus captivantes et personnalisées. L'impact futur de l'IA sur les jeux vidéo sera probablement aussi profond que les changements qu'elle a déjà apportés, promettant de nouvelles façons d'interagir, d'explorer, et de vivre des histoires au sein de mondes virtuels.

Enter the Gungeon

“Enter the Gungeon” est un jeu vidéo de type roguelike développé par Dodge Roll et édité par Devolver Digital, sorti en 2016.

Le but du jeu est de naviguer au sein de donjons générés procéduralement en combattant des ennemis pour finalement trouver un boss.

Gameplay du jeu, le héros bombarde les ennemis de roquettes

Au départ, nous avions choisis ce jeu pour nos expérimentations de machine learning. L’approche pour notre modèle était de l’entrainer à travers une série de parties de jeu, de sorte qu’il apprenne et s’adapte aux diverses situations pour réussir à avancer dans le donjon tout en combattant les ennemis.

Cependant, plusieurs problématiques ont influencé notre décision de ne pas poursuivre le développement sur "Enter the Gungeon". Les mouvements du personnage sont complexes et variés, cela exige une compréhension solide des combinaisons possibles et de leurs conséquences, ce qui représente un défi pour notre IA.

D’autre part, "Enter the Gungeon" présente des objectifs dynamiques qui ne sont pas toujours situés au même endroit ou visibles à l’écran. Cela rend difficile la définition d’une fonction de récompense, qui est essentielle pour l’apprentissage par renforcement.

Face à ces défis, nous avons reconsidéré notre décision. Bien que le jeu offre un terrain d'expérimentation intéressant pour une IA de machine learning, ses spécificités exigent des solutions technologiques trop avancées.

Super Mario Bros

Après avoir rencontré des défis significatifs dans la conception d'une IA pour "Enter the Gungeon", nous avons décidé de nous orienter vers un jeu différent : "Super Mario Bros". Ce jeu offre un gameplay plus linéaire et un objectif clair, ce qui le rend idéal pour nos expérimentations

Super Mario Bros est un jeu vidéo de plates-formes à défilement horizontal développé par Nintendo R&D4 et édité par Nintendo. Il est sorti sur Nintendo Entertainment System (NES) en 1987 en Europe. Il s'agit du premier jeu de la série Super Mario. Le jeu est composé de nombreux niveaux dans lesquels le joueur doit naviguer vers la droite de l'écran pour progresser. Dans le jeu, Mario est capable de sauter, courir et peut récupérer des power-ups.

Mario qui court vers la fin du niveau

"Super Mario Bros" est un excellent choix pour développer un algorithme de machine learning, et ce pour plusieurs raisons. D'abord, l'objectif est clair : il faut aller à droite. La fonction de récompense peut donc retourner un résultat qui varie selon les déplacements de Mario. D'autre part, les mouvements sont simples et l'état du jeu est relativement facile à définir.

Expérimentations de vision par ordinateur avec YOLO

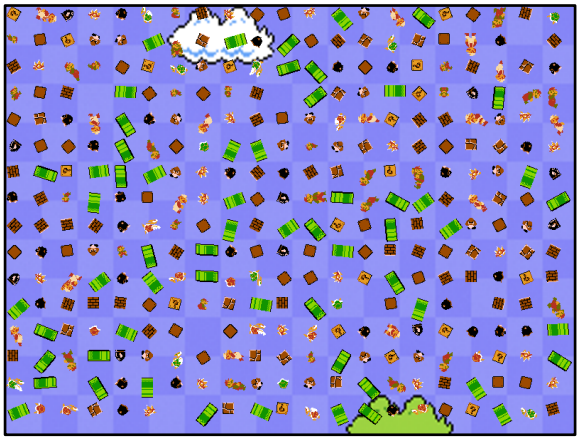
Pour récupérer les informations des jeux, nous avions envisagé d'utiliser YOLO (You Only Look Once), un modèle de Deep Learning avancé pour la détection d'objets. YOLO est réputé pour sa rapidité et son efficacité, capable d'identifier et de classifier différents objets dans une image en temps réel. Cette technologie devait permettre à notre IA de reconnaître les éléments clés du jeu, tels que les ennemis, les obstacles, et les objets.

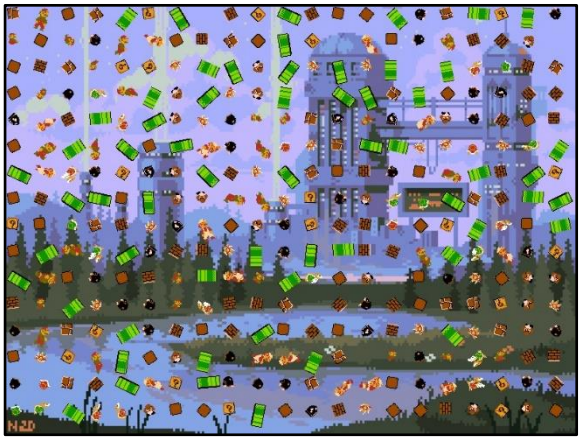
Pour intégrer la détection d'objets via YOLO dans le jeu, une approche méthodique a été adoptée pour l'entraînement de la reconnaissance d'objets. Le modèle nécessite un ensemble de données d'entraînement large et précis pour interpréter correctement les éléments du jeu.

Nous avons d'abord découpé les spritesheet de Super Mario Bros à l'aide d'un script Python personnalisé. Un autre script prenait ces images découpées et les plaçait aléatoirement sur des arrière-plans variés.

Pour améliorer la capacité de YOLO à généraliser et à reconnaître ces objets dans divers contextes de jeu, nous avons introduit des variations dans ces images, notamment en changeant le fond et la rotation des objets.

Voici quelques exemples d'images générées par le script python :






Le script génère avec chaque image un fichier texte, dans lequel sont stockées les coordonnées des objets relatives à l'image. Chaque ligne, correspondant à un objet se compose de plusieurs paramètres :

- La classe de l'objet (identifiant unique lié à l'objet)
- Les coordonnées normalisées du centre de l'objet (x_center, y_center)
- Sa largeur (width) et sa hauteur (height)

Cette méthode d'annotation est essentielle pour l'apprentissage supervisé, car elle fournit au modèle les informations nécessaires pour apprendre à localiser et identifier les objets dans des images.

Voici un aperçu d'un fichier .txt contenant des étiquettes, nous y trouvons les coordonnées d'un tuyau (classe 0), d'un Mario (classe 1), d'un bloc (classe 2) et d'un Goomba (classe 3) :

```
0 0.48875 0.7583333333333333 0.0275 0.03666666666666667
1 0.3425 0.13666666666666666 0.02 0.02666666666666667
2 0.34375 0.2 0.0225 0.03
3 0.96625 0.025 0.0325 0.05
```

Une fois cet ensemble de données préparé, l'étape suivante consistait à entraîner le modèle YOLO. Le processus d'entraînement impliquait de nourrir le modèle avec ces images et annotations, permettant à YOLO d'apprendre progressivement à reconnaître et à localiser les différents objets du jeu. Cette phase d'entraînement est cruciale, car elle détermine la précision avec laquelle le modèle pourra ensuite identifier et réagir aux éléments du jeu en temps réel.

Exemple de détection sur une vidéo du jeu (modèle sous entrainé)

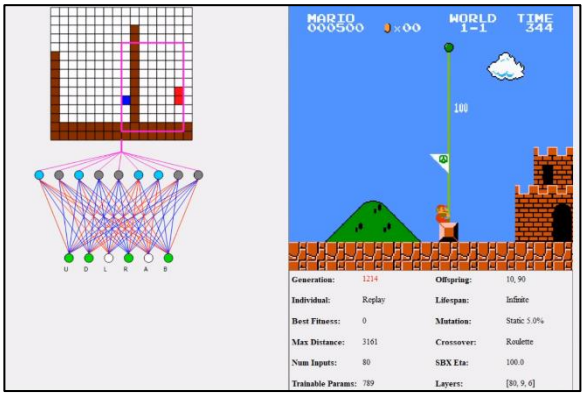
Après avoir testé la détection en temps réel sur le jeu, nous avons réalisés que cela n'était pas une solution viable pour plusieurs raisons :

- Avec nos données, YOLO a du mal à détecter et différencier les différents objets. Le travail de préparation des données devient trop conséquent et nous préférons concentrer nos efforts sur les techniques d'apprentissage automatique.
- Les performances de la détection ont beau être bonnes pour certaines utilisations, elles sont insuffisantes pour notre cas, on passe de 700 images par secondes à environ 24 avec la détection. Cette perte de vitesse n'est pas acceptable pour un entrainement intensif.

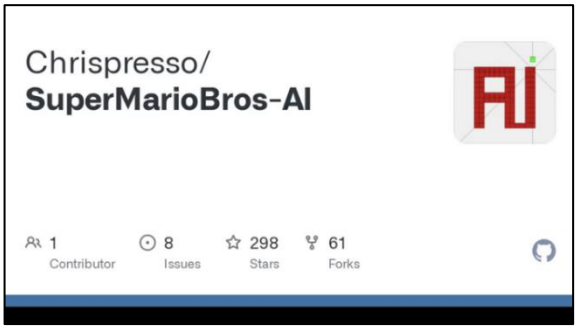
Mise en place de l'environnement final

En parallèle de l'extraction des données via YOLO, nous nous sommes mis à la recherche d'une autre façon de récupérer l'état du jeu. Nous avons finalement décidé d'émuler Super Mario Bros avec "nes-py", un émulateur possédant un API python qui nous donne accès à la RAM de la console.

A l'époque de la NES, les ingénieurs programmaient les jeux avec de fortes contraintes de mémoire. Pour un meilleur contrôle, chaque variable était allouée manuellement, en conséquence, les emplacements de mémoire qui nous intéressent se situent toujours au même endroit. En fait, des "cartes RAM" de Super Mario Bros existent et permettent de récupérer à coup sûr la donnée recherchée. Après quelques recherches sur YouTube, nous avons trouvé cette vidéo qui traite le même sujet et pointe vers un répertoire GitHub très intéressant.



La vidéo en question



Le répertoire GitHub

Dans ce répertoire se trouve un fichier nommée "utils.py". Il continent une carte RAM de Super Mario Bros ainsi qu'une classe contenant des dizaines de méthodes pour récupérer des informations précieuses, telles que la liste des ennemis visibles, la position de Mario et une grille de toutes les tuiles visibles à l'écran.

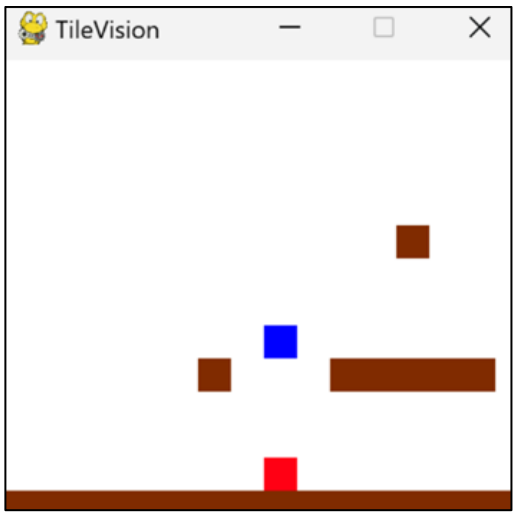
```
@classmethod
def get_enemy_locations(cls, ram: np.ndarray):···

@classmethod
def get_mario_state(cls, ram: np.ndarray):···

@classmethod
def get_mario_location_in_level(cls, ram: np.ndarray) -> Point:···

@classmethod
def set_mario_position(cls, ram: np.ndarray, x: int, y: int):···

@classmethod
def advance_screen_scrolling(cls, ram: np.ndarray, pixels: int):···

@classmethod
def get_mario_score(cls, ram: np.ndarray) -> int:···

@classmethod
def get_mario_location_on_screen(cls, ram: np.ndarray):···
```

Méthodes utilitaires

```
Enemy_Type = 0x16
Enemy_X_Position_In_Level = 0x6E
Enemy_X_Position_On_Screen = 0x87
Enemy_Y_Position_On_Screen = 0xCF

Player_X_Postion_In_Level = 0x06D
Player_X_Position_On_Screen = 0x086

Player_X_Position_Screen_Offset = 0x3AD
Player_Y_Position_Screen_Offset = 0x3B8
Enemy_X_Position_Screen_Offset = 0x3AE

Player_Y_Pos_On_Screen = 0xCE
Player_Vertical_Screen_Position = 0xB5
```

Carte RAM

Premièrement, nous avons développés une fenêtre de débogage pour vérifier que l'ordinateur perçoit correctement le jeu :



Vision via la RAM

Rendu original

Cette façon de récupérer les données est fiable, performante et facile à manipuler. C'est un gain de confort et de temps exceptionnel comparé à la détection via YOLO, au prix d'un code moins adaptable aux autres jeux Mario.

Les données à notre disposition, il faut maintenant mettre en place un environnement d'entrainement. Le cahier des charges pour celui-ci est le suivant :

- Détecter la fin d'une partie
- Recharger instantanément le début du niveau
- Savoir quand Mario avance, recule ou touche un ennemi

Ces simples fonctionnalités nous permettent de relancer un niveau en boucle à chaque fois que Mario meurt ou gagne. De plus les données sur Mario nous permettront d'associer une récompense à chaque action. A notre grand bonheur, la bibliothèque gym-super-mario-bros fournit un API basé sur nes-py qui remplit l'intégralité de ce cahier des charges.

Architecture et guide d'utilisation

Le programme est architecturé pour l'entrainement de la sorte :

- La classe Training contient tout ce qui est relatif à l'entrainement en cours
- Le fichier utils.py contient le code permettant de lire la mémoire de la NES.
- Le fichier settings.py contient tous les paramètres constants liés au programme
- Le fichier debug.py contient les outils de débogage (compteur de FPS, affichage de débug...)
- La classe State contient et récupère toutes les données que l'IA considèrera comme l'état du jeu.
- La classe QTable stocke les données d'apprentissage de l'IA et permet de les sauvegarder dans un fichier JSON

Pour exécuter le programme :

- Dans le répertoire, exécutez "pip install -r requirements.txt"
- Configurez l'apprentissage dans settings.py
- Par défaut, un modèle ayant 100% de réussite est chargé, si vous voulez commencer un nouvel entrainement, supprimez simplement qTable.json
- Lancez main.py

De nombreux paramètres sont disponibles afin de configurer l'apprentissage et l'affichage :

```
# Debug parameters
SHOW_FPS = True
AVERAGE_FPS_CALCULATION_TIME = 20
MAX_FRAMERATE = 60
SHOW_MINI_DISPLAY = True
RENDER_MODE = "human" # [human | none]

# Learning parameters
ENABLE_TRAINING = True
EPSILON_MIN = 0
EPSILON_START = 0
EPSILON_SCALING = 0.9999
GAMMA = 0.9
ALPHA = 0.1
VISION_RANGE = 6
MAX_STUCK_TIME = 1000
DEATH_PENALTY = -150
STAND_STILL_PENALTY = -2
FRAMES_BEFORE_UPDATE = 60//8
SPEEDRUN_ACTIONS = [3,4]
IDLE_ACTION = 0
```

Pour générer des graphiques :

- Assurez-vous que score_graph.json est bien rempli
- Exécutez Plot.py
- Un second graphique apparaitra en fermant le premier
- Quand vous changez de qTable ou la supprimez, réinitialisez les données de score_graph.json en remplaçant tout le contenu par ceci : “[]”. Cela vous évitera d’avoir des données incohérentes.

Bugs connus :

- Quand on quitte au moment où la QTable est sauvegardée (quand la partie se relance), il y a un risque de corruption des données JSON. Si cela arrive, pas de panique, un système est mis en place pour sauvegarder la QTable périodiquement, vous trouverez les sauvegardes dans le dossier “Backup”.
- Le même phénomène peut se produire pour les données liées aux graphiques. Il n’y a pas de mécanisme de sauvegarde pour celles-ci, si vous y tenez, sauvegardez-les avant de quitter l’entrainement.
- Si le fichier JSON contenant les données des graphiques est corrompu, effacez son contenu et laissez juste un tableau vide comme ceci : “[]”.

Notre solution basée sur le Machine Learning

Contexte

Dans le cadre de notre projet sur Super Mario Bros, l'implémentation d'une intelligence artificielle efficace et adaptative est cruciale pour progresser dans des délais raisonnables au sein des niveaux. Après une analyse approfondie des différentes techniques d'apprentissage automatique disponibles, notre équipe a choisi de mettre en œuvre le Q-Learning, une forme d'apprentissage par renforcement, pour développer l'IA de notre Mario.

Choix du Q-Learning

Le Q-Learning a été sélectionné pour plusieurs raisons pertinentes :

1. **Indépendance du Modèle :** Contrairement à d'autres méthodes, le Q-Learning n'a pas besoin d'un modèle prédictif de l'environnement. Cette caractéristique le rend idéal pour notre jeu "Mario", où les scénarios peuvent être très variés et imprévisibles.
2. **Simplicité et Efficacité :** Le Q-Learning est relativement simple à comprendre et à mettre en œuvre. Malgré sa simplicité, il est puissant et peut gérer des tâches complexes, ce qui est essentiel pour le comportement dynamique dans notre jeu.
3. **Flexibilité et Adaptabilité :** Cette méthode permet à l'IA de s'adapter et d'apprendre de nouvelles stratégies en fonction des interactions avec l'environnement, ce qui est crucial pour un jeu qui évolue comme "Mario".

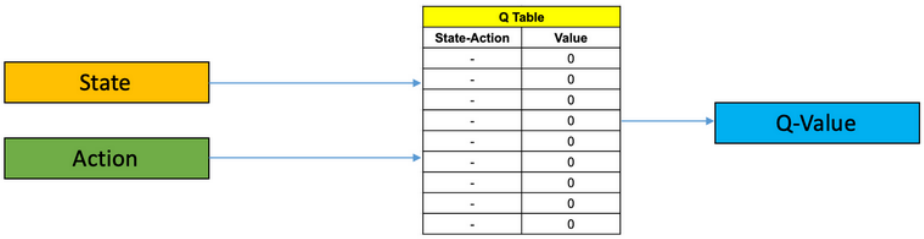
Fonctionnement du Q-Learning

Le Q-Learning fonctionne en utilisant une table de valeurs Q, connue sous le nom de Q-table. Cette table aide l'agent IA (Mario dans notre cas) à déterminer quelle action prendre dans un état donné.

A. États (States) : Dans "Mario", un état peut être défini par la position de Mario, l'état des ennemis, les obstacles présents, etc. Chaque combinaison unique de ces éléments constitue un état différent dans la Q-table.

B. Actions : Les actions représentent les différentes possibilités que Mario peut entreprendre à un moment donné, comme sauter, se déplacer vers la gauche ou la droite, etc. Chaque action a un impact potentiel sur l'état futur de Mario.

C. Q-table : La Q-table est une matrice où chaque ligne représente un état possible et chaque colonne une action possible. Les valeurs dans cette table (valeurs Q) sont estimées pour représenter la "qualité" ou l'utilité de prendre une action donnée dans un état donné.



D. Récompenses : Le système de récompenses est crucial dans le Q-Learning. Lorsque Mario effectue une action, il reçoit une récompense (positive ou négative) basée sur l'issue de cette action (par exemple, gagner des points pour avoir collecté une pièce ou être pénalisé pour avoir touché un ennemi).

E. Mise à Jour de la Q-table : La mise à jour de la Q-table se fait via la formule de mise à jour Q-learning qu'on étudiera plus tard.

Choix des actions possibles et détermination de la prochaine action :

Dans notre projet "Mario", l'objectif principal est de faire avancer le personnage le plus loin possible et le plus rapidement possible. Pour rester aligné avec cet objectif, nous avons pris la décision de limiter les actions du personnage à deux options fondamentales : courir et courir + sauter.

```
SPEEDRUN_ACTIONS = [3,4]
```

Cette simplification est basée sur l'idée que le recul n'est pas une option viable dans le jeu, car il contrecarre l'objectif d'avancement. En conséquence, Mario court en permanence, et l'IA doit choisir entre continuer à courir ou courir tout en sautant.

La détermination de la prochaine action par l'IA suit un processus spécifique :

1. **Vérification de l'Atterrissage :**
   - Si Mario vient d'atterrir d'un saut précédent, la seule action que nous lui permettons est de continuer à courir. Cette règle est basée sur la mécanique du jeu où un enchaînement immédiat de sauts n'est pas possible. Cette condition s'applique directement dans l'update de l'entraînement.

```
if self.just_hit_ground:
    action, action_index = IDLE_ACTION, -1 # Don't spam jump
```

2. **Décision Basée sur Epsilon :**
   - Pour décider de la prochaine action (continuer à courir ou courir et sauter), l'IA effectue un tirage aléatoire entre 0 et 1. Si cette valeur est inférieure à un certain seuil, appelé epsilon (ϵ), l'IA choisit une action au hasard.

```
if random.uniform(0, 1) < epsilon:
    index = random.randint(0, len(SPEEDRUN_ACTIONS) - 1)
    return SPEEDRUN_ACTIONS[index], index
```

   - Si la valeur est supérieure à epsilon, l'IA choisit l'action qui a la valeur Q la plus élevée pour l'état actuel, ce qui signifie qu'elle exploite les connaissances qu'elle a acquises jusqu'à présent.

```
else:
    # On exploite en choisissant l'action avec la valeur Q la plus élevée pour l'état donné.
    state_combination = self.q_table.Q[str(self.state.combination())]
    index = int(max(state_combination, key=state_combination.get))
    return SPEEDRUN_ACTIONS[index], index
```

Le rôle d'epsilon, bien qu'important, est relativement simple : il aide à équilibrer l'exploration de nouvelles actions et l'exploitation des stratégies connues. Au début du processus d'apprentissage, un epsilon plus élevé est utile pour encourager l'exploration. Avec le temps et l'apprentissage, cet epsilon est généralement réduit jusqu'à un minimum, permettant à l'IA de se concentrer davantage sur les actions qui se sont avérées efficaces. La valeur exacte d'epsilon et son ajustement sont définis en fonction des besoins spécifiques du jeu, en cherchant un équilibre entre l'apprentissage et la performance optimale.

```
EPSILON_MIN = 0
EPSILON_START = 0
EPSILON_SCALING = 0.9999
```

Choix des états

Pour l'implémentation de l'intelligence artificielle dans notre projet "Mario" via le Q-Learning, nous avons adopté une stratégie spécifique dans le choix des états, en mettant l'accent sur la gestion des distances. Étant donné que Mario ne peut pas reculer dans le jeu, nous avons décidé de ne prendre en compte que l'environnement qui est devant lui. Cette décision reflète la dynamique et les mouvements du gameplay. Les distances sont mesurées en unités de carreaux qui divisent l'écran, et nous nous concentrons sur les éléments situés à 6 carreaux ou moins de Mario. Cette approche permet de garder l'IA concentrée sur les obstacles immédiats et les défis pertinents. Ce nombre est d'ailleurs facilement modifiable dans les settings avec la variable *VISION_RANGE.*

Si un objet pertinent (comme un ennemi ou un obstacle) est situé à une distance supérieure à 6 carrés ou est inexistant dans l'environnement immédiat, nous attribuons par défaut la distance comme étant égale à la longueur de l'écran, qui est de 16 carrés. Cette standardisation assure que l'IA a une valeur de référence constante pour les situations où les défis immédiats ne sont pas présents, permettant une prise de décision cohérente et rationnelle.

En se basant sur ces critères, les états que nous avons sélectionnés pour l'IA sont les suivants :

1. **Distance à l'Obstacle le Plus Proche :** Calcule la distance horizontale à l'obstacle le plus proche (comme un bloc ou un mur) devant Mario.
2. **Distance au Trou le Plus Proche :** Évalue la distance horizontale entre Mario et le trou le plus proche sur son chemin.
3. **Distance en X à l'Ennemi le Plus Proche :** Mesure la distance horizontale à l'ennemi le plus proche devant Mario.
4. **Distance en Y à l'Ennemi le Plus Proche :** Mesure la distance verticale à l'ennemi le plus proche.
5. **Distance au Trou le Plus Proche au moment du saut :** Évalue la distance horizontale entre Mario et le trou le plus proche mesurée au moment où Mario à entamer son saut. La valeur redevient par défaut lorsque le saut est terminé.

Chaque état a été soigneusement choisi pour garantir que l'IA dispose des informations les plus pertinentes pour naviguer efficacement dans le monde de "Mario", en se concentrant sur les défis immédiats et en évitant les distractions inutiles, assurant ainsi un apprentissage performant.

Nous voulions initialement utiliser une partie de la grille comme état, mais les états possibles auraient explosé : pour une grille de 6x6 et 4 valeurs possibles par case (vide, mur, ennemi, joueur), on aurait 4^36 possibilités, soit 4,722,366,482,869,645,213,696 états différents.

Avec les états que nous avons définis, ce nombre est réduit à (VISION_RANGE+1)^5 états possibles, soit 16,807 possibilités. Ce nombre bien plus petit permet au modèle de généraliser beaucoup plus vite ses comportements à d'autres niveaux et situations, en plus d'économiser beaucoup de stockage et de mémoire vive.

Initialisation, mise à jour et sauvegarde de la Q-Table

Dans notre projet, nous avons choisi de structurer la Q-Table sous la forme d'un multi-dictionnaire pour faciliter le processus d'apprentissage par Q-Learning. Chaque clé de ce multi-dictionnaire est un string de tuple représentant un état spécifique :

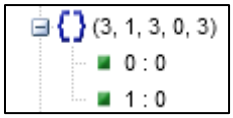
```
(5, 4, 16, 16, 16)
```

```
(self.enemy[0],self.enemy[1],self.obstacle,self.hole,self.last_jump_hole_dist)
```

Ce multi-dictionnaire a pour valeurs des dictionnaires qui ont pour clé une action qui est représentée par le string d'un entier (0 pour courir, 1 pour courir et sauter).

```
self.Q[str(combination)][str(action)]
```

Par exemple, un élément de la Q-Table pourrait ressembler à ceci :



Lors de l'initialisation, toutes les valeurs Q dans la table sont fixées à 0, à moins qu'une Q-Table préalablement sauvegardée ne soit chargée. Cette initialisation à zéro offre une toile vierge sur laquelle l'IA peut commencer à apprendre par essais et erreurs.

```
try:
    with open('qTable.json', 'r') as file:
        data = json.load(file)
    # Si le fichier est vide
    if not data:
        print("Could not load Q-Table, creating new one (JSON file was empty)")
        self.Q = self.initQ()
    else:
        print("Successfully loaded Q-Table from file")
        self.Q = data
```

La mise à jour de la Q-Table dans notre projet "Mario" se fait conformément à la formule standard du Q-Learning, qui est cruciale pour l'apprentissage et l'optimisation des stratégies de l'IA. La formule est la suivante :

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

```
next_max = max(self.Q[str(next_combination)].values())
self.Q[str(combination)][str(action)] += alpha*(reward+gamma*next_max-self.Q[str(combination)][str(action)])
```

Où :

- Q(s,a) est la valeur Q actuelle pour un état spécifique s et une action a.
- α (taux d'apprentissage), définit le poids des nouvelles informations comparées à celles acquises.
- R(s,a) est la récompense obtenue après avoir effectué l'action a dans l'état s.
- γ (facteur de remise) reflète l'importance accordée aux récompenses futures.
- max(Q(s',a')) représente la valeur Q maximale pour le prochain état possible s' et toutes les actions possibles a' depuis cet état. Cela indique la meilleure récompense attendue pour le prochain coup.

Dans cette formule, s et s’ représentent respectivement l'état actuel et l'état suivant. Les actions a et a’ correspondent à l'action actuelle et aux actions potentielles dans l'état suivant. Cette mise à jour se fait après chaque action entreprise par l'IA dans le jeu, permettant un ajustement continu des valeurs Q en fonction de l'expérience accumulée et des récompenses reçues.

Pour notre projet, nous avons défini :

```
GAMMA = 0.9
ALPHA = 0.1
```

- Taux d'apprentissage (α) à 0.1 : Ce choix assure un équilibre entre une intégration rapide des nouvelles données et la stabilité des connaissances déjà acquises. Un α plus élevé rendrait l'IA plus réactive aux nouvelles informations, mais au risque de perturber ce qu'elle a déjà appris. À l'inverse, un taux plus faible ralentirait l'apprentissage, mais augmenterait la stabilité.
- Facteur de remise (γ) à 0.9 : Ce choix souligne la valeur accordée aux récompenses à venir. Un γ élevé, proche de 1, indique que les gains futurs sont presque aussi pertinents que les récompenses immédiates, orientant ainsi l'IA vers des stratégies à long terme. Un γ plus bas aurait pour effet de limiter la vision de l'IA aux bénéfices immédiats. En choisissant 0.9, nous encourageons l'IA à envisager les conséquences à long terme de ses actions, favorisant ainsi des prises de décision plus stratégiques.

Ces paramètres ont été choisis pour favoriser un apprentissage efficace et équilibré, permettant à l'IA de s'adapter et de répondre de manière optimale aux défis du jeu Mario.

Pour la sauvegarde de la Q-Table, nous utilisons le format JSON, qui est à la fois léger et facile à manipuler. La Q-Table est périodiquement sauvegardée sous cette forme, permettant ainsi de conserver l'état d'apprentissage de l'IA entre différentes sessions de jeu. Cela est particulièrement utile pour les processus d'apprentissage prolongés, où l'IA peut continuer à évoluer sur plusieurs sessions sans perdre les progrès précédemment réalisés.

```
def saveQ(self, filename = "qTable.json"):
    with open('qTable.json', 'w') as file:
        json.dump(self.Q, file)
```

Ces procédures d'initialisation, de mise à jour et de sauvegarde sont cruciales pour l'évolution et l'efficacité de notre IA dans le jeu "Mario", permettant une adaptation stratégique et une évolution constante face aux défis du jeu.

Attribution de la récompense

Dans notre jeu, le système de récompenses est essentiel pour guider l'apprentissage de l'IA. Heureusement, une partie importante de ce mécanisme est déjà intégrée dans l'API que nous utilisons. Selon cette API, les récompenses sont attribuées principalement en fonction de la progression du joueur dans le niveau.

```
frame, reward, done, truncated, info = self.env.step(action)
```

La structure de base des récompenses est la suivante :

- Progression dans le niveau : L'IA reçoit des points pour chaque avancement réalisé dans le niveau.
- Achèvement du niveau : Une récompense importante de 5 est attribuée lorsque le joueur termine avec succès le niveau.
- Échec (Mort) : Si le joueur meurt, une pénalité de -15 est appliquée.

Pour affiner la stratégie d'apprentissage de notre IA et la rendre plus adaptée à nos objectifs spécifiques, nous avons introduit une fonction personnalisée, adjust_reward. Cette fonction modifie les récompenses de l'API de la manière suivante :

1. Décupler les Récompenses pour Gagner ou Perdre : Pour accentuer l'importance de gagner ou de perdre, nous multiplions par dix les récompenses et les pénalités associées. Cela signifie que finir la map rapporte désormais 50, tandis que mourir entraîne une pénalité significative de -150. Cette modification vise à renforcer l'apprentissage des actions qui conduisent à la réussite et à décourager celles qui mènent à l'échec. Cette pénalité est d'ailleurs ajustable avec la variable "DEATH_PENALTY".

```
if (reward == -15): reward = DEATH_PENALTY
if (reward == 5): reward *= 10 #TODO: back p
```

2. Pénalités pour l'Inaction ou le Mouvement Inefficace :
    - Pénalité pour Courir sans Avancer (STAND_STILL_PENALTY) : Si l'IA choisit de courir sans sauter et sans faire de progrès (c'est-à-dire sans avancer dans la map), nous appliquons une pénalité. Cette pénalité vise à décourager l'inaction ou les mouvements peu efficaces.

```
STAND_STILL_PENALTY = -2
```

    - Pénalité Réduite pour Sauter sans Avancer : Si l'IA saute mais ne progresse pas, une pénalité est également appliquée, mais celle-ci est moins sévère que la pénalité pour courir sans avancer. L'idée est d'encourager l'IA à sauter quand elle est coincée.

```
if (reward == 0):
    reward = STAND_STILL_PENALTY # Penalty when not moving right
    if (info["y_pos"] > self.last_y_pos):
        reward += 1 # Little bonus when not moving but jumping
```

En augmentant les enjeux pour les récompenses et les pénalités et en introduisant des pénalités pour des comportements spécifiques, nous pouvons guider l'IA vers des stratégies de jeu plus efficaces et alignées sur nos objectifs de performance.

Rétropropagation pour les Sauts

Nous avons mis en place un mécanisme de rétropropagation spécifique pour les sauts, dans le but de renforcer l'apprentissage de l'IA concernant les conséquences de ses actions. Ce processus est particulièrement important pour les situations où Mario meurt à la suite d'un saut, car il permet de pénaliser rétroactivement les décisions qui ont conduit à cet échec.

L'API que nous avons développée fournit deux états clés qui sont essentiels pour ce mécanisme : grounded (Mario est sur le sol) et floating (Mario est en l'air, typiquement en train de sauter). Nous utilisons ces états pour déterminer les moments où Mario commence un saut et les moments où il atterrit. À cet effet, deux attributs dans notre système de formation, self.just_hit_ground et self.just_jumped, sont utilisés pour marquer ces événements.

La logique de rétropropagation se divise en 3 étapes :

1. Détection du Saut et de l'Atterrissage :
    - Si l'état précédent était floating et que l'état actuel est grounded, cela signifie que Mario vient d'atterrir.
    - Inversement, si l'état précédent était grounded et que l'état actuel est floating, cela indique que Mario vient de sauter.
2. Gestion du Buffer de Saut :
    - À partir du moment où Mario saute (self.just_jumped activé), nous commençons à enregistrer dans un buffer toutes les combinaisons état-action jusqu'à ce qu'il atterrisse ou meure.
    - Si Mario atterrit avec succès (self.just_hit_ground activé), le buffer est entièrement vidé, car le saut a été réussi.
    - Si Mario meurt avant d'atterrir, les combinaisons état-action enregistrées dans le buffer sont utilisées pour la rétropropagation.
3. Application de la Pénalité en Cas de Mort :
    - Lorsque Mario meurt en sautant, une pénalité de -1 est appliquée rétroactivement à toutes les valeurs Q stockées dans le buffer. Cela signifie que pour chaque état-action enregistré depuis le début du saut, nous diminuons la valeur Q correspondante dans la Q-Table.
    - Cette pénalité vise à décourager les séquences d'actions qui ont mené à l'échec, guidant ainsi l'IA vers des stratégies de saut plus sûres et plus efficaces.

Ce système de rétropropagation pour les sauts est un élément clé de notre stratégie d'apprentissage pour l'IA dans "Mario". Il permet de rendre l'IA plus prudente et stratégique dans ses décisions de saut, en tenant compte non seulement des conséquences immédiates de ses actions, mais aussi des résultats à long terme.

Quand Mario retombe, le buffer se vide. Pourtant il est parfois déjà condamné à mort, car même s'il saute juste après, il est trop près d'un ennemi pour l'esquiver, et c'est ce tout petit buffer qui est utilisé pour la pénalité. Cela causait des boucles quasiment insurmontables de mauvais sauts menant à la mort, nous avons donc conçus un système de multi-buffer plus avancé :

- Un buffer indépendant est créé quand Mario saute.
- Une fois que Mario touche le sol, un compte à rebours de N frames est lancé, après lequel le buffer est supprimé.
- Si Mario resaute avant que le buffer soit expiré, un nouveau est créé en parallèle.
- Quand Mario meurt, un algorithme calcule le buffer le plus pertinent pour la pénalité.

```python
class LatestBufferTracker():
    def __init__(self, lifespan):
        self.buffers = []
        self.lifespan = lifespan
        self.last_update_frame = 0

    def create_buffer(self, jump_frame): ···

    def get_latest_buffer(self, frame): ···

    def reset(self): ···

    def update(self, frame, state_action, last_jump_frame, last_hit_ground_frame): ···
```

```python
class StateActionBuffer():
    def __init__(self, jump_frame):
        self.buffer = []
        self.jump_frame = jump_frame
        self.land_frame = -1

    def set_land_frame(self, land_frame): ···

    def append(self, state_action): ···

    def is_expired(self, frame, lifespan): ···

    def get_buffer(self): ···
```

Procédure d'entrainement

Pour rendre la mise à jour de la Q-Table plus pertinente et efficace, nous avons choisi de ne pas actualiser la Q-Table à chaque frame, mais plutôt toutes les n frames. Cette approche permet de s'assurer que les états varient suffisamment entre chaque mise à jour pour que les changements dans la Q-Table soient significatifs et reflètent une véritable progression ou régression dans le gameplay.

```
def should_train(self):
    return ENABLE_TRAINING and (self.done or self.frame % FRAMES_BEFORE_UPDATE == 0)
```

```
FRAMES_BEFORE_UPDATE = 60//4
```

Gestion des Actions et des Récompenses :

- Action Constante sur n Frames : Durant chaque bloc de n frames, l'action déterminée reste la même. Cela permet de mesurer l'impact d'une action spécifique sur une période plus longue, donnant ainsi un aperçu plus clair de son efficacité.

```
action, action_index = self.active_action[0], self.active_action[1]
if self.just_hit_ground:
    action, action_index = IDLE_ACTION, -1 # Don't spam jump
```

- Accumulation des Récompenses : Le reward total est calculé en accumulant les récompenses obtenues durant ces n frames. En cas de mort seule la pénalité finale est comptabilisée. Cela permet d'évaluer l'efficacité globale de l'action sur l'ensemble de la période.

```
reward = self.adjust_reward(reward, info)
if (reward == DEATH_PENALTY):
    self.active_reward = reward
else: self.active_reward += reward
```

Boucle d'Entraînement :

1. Mise à Jour de l'Environnement : À chaque frame, l'environnement est mis à jour en fonction de l'action en cours.

```
self.env.step(action)
```

2. Gestion du Reward : Si le jeu continue, le reward est incrémenté et ajouté au total.
3. Vérification de la fin du jeu : La variable done est vérifiée à chaque frame dans la fonction *should_train* (capture d'écran plus haut) pour déterminer si le jeu est terminé (soit parce que Mario est mort, soit parce qu'il a fini la map). Si done est vrai, l'environnement est réinitialisé et la Q-Table est sauvegardée dans le fichier JSON.

```
should_train = self.should_train()
```

```
if self.done:
    self.reset_env()
```

4. Actions à la nème Frame :

```
if self.should_train():
```

   - Epsilon est multiplié par un coefficient inférieur à 1 pour le réduire progressivement, encourageant ainsi l'IA à exploiter davantage ses connaissances acquises plutôt qu'à explorer de nouvelles actions.

```
self.epsilon *= EPSILON_SCALING
if (self.epsilon < EPSILON_MIN): self.epsilon = EPSILON_MIN
```

   - Le buffer pour la rétropropagation est rempli si nécessaire.

```
self.fill_buffers()
```

   - Le prochain état est récupéré depuis la RAM.

```
self.state.update(self.ram)
```

   - La Q-Table est mise à jour en utilisant le total des rewards accumulés, l'état actuel, l'action effectuée, et le nouvel état.

```
self.q_table.update(
    self.last_state,
    self.state,
    self.active_action[1],
    self.gamma,
    self.alpha,
    self.active_reward
)
```

   - La prochaine action est déterminée.

```
train_action, action_index = self.getNextAction(self.epsilon)
self.active_action = (train_action, action_index)
```

   - La logique de rétropropagation est appliquée si le joueur est mort après un saut.

```
if self.active_reward == DEATH_PENALTY:
    self.back_propagate_jump()
```

   - Le state est stocké en tant qu'ancien state pour être utilisé durant la prochaine update de la Q-table.

```
self.last_state = copy.copy(self.state)
```

   - Le total des rewards est réinitialisé pour la prochaine séquence de n frames.

```
self.active_reward = 0
```

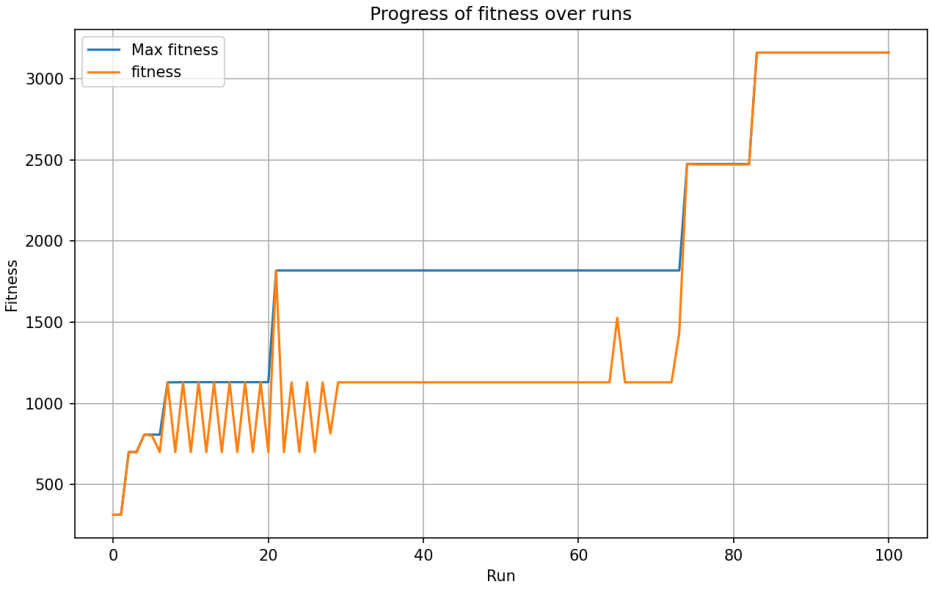
Ce processus d'entraînement permet une amélioration graduelle et structurée de l'IA dans le jeu "Mario". En actualisant la Q-Table toutes les n frames, nous assurons que les mises à jour sont basées sur des informations suffisamment significatives, reflétant de véritables progrès ou défis dans le gameplay. La diminution progressive d'epsilon garantit que, à mesure que l'IA apprend et s'adapte, elle se fie de plus en plus à son expérience plutôt qu'à des actions aléatoires.

Cette méthode d'entraînement est conçue pour développer une IA capable de naviguer efficacement et stratégiquement dans le monde de Mario, en s'améliorant constamment grâce à un apprentissage ciblé et adaptatif.

Résultats

Apprentissage déterministe

Lorsque l'on fixe epsilon à zéro pour l'entrainement de Mario, cela signifie qu'il va apprendre strictement à partir de ses succès et erreurs sans aucune intervention du hasard. Il est assez impressionnant de voir qu'il commence à éviter les ennemis en sautant après seulement une ou deux tentatives. De plus, il apprend à finir le niveau 1 à coup sûr au bout de 84 essais.



Graphique du score (fitness) en fonction du nombre de parties jouées (epsilon = 0)

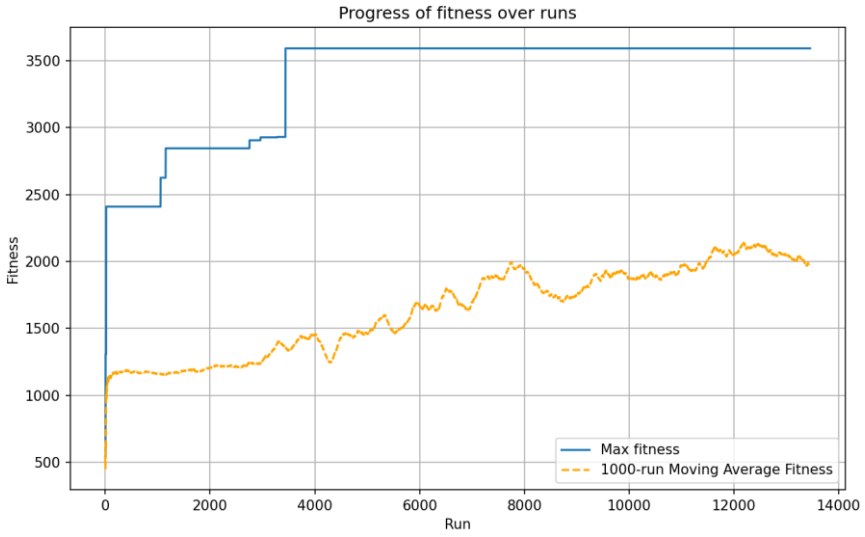
Cette progression quand epsilon est à zéro montre que le modèle est capable d'apprendre de ses actions, et prouve donc que la solution développée peut être viable avec le bon paramétrage et suffisamment d'entrainement.

Il est important de noter qu'un modèle entrainé avec un epsilon à 0 ne fait qu'apprendre les niveaux par cœur, et qu'à moins d'être exposé à de nombreux niveaux différents, il donnera de mauvais résultats quand il fera face à un environnement inconnu.

Cela dit, ces résultats sont un véritable succès comparé aux autres modèles existants, qui mettent beaucoup plus de temps à battre le premier niveau.

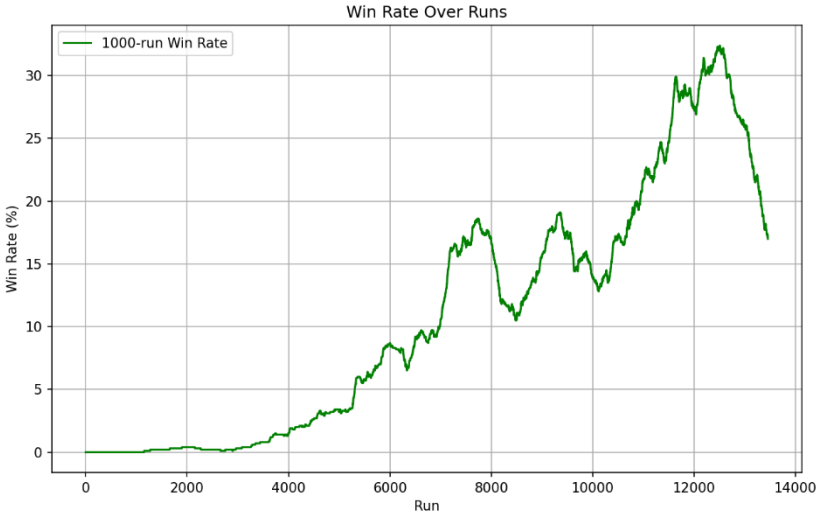
Apprentissage exploratoire

En augmentant EPSILON, on introduit plus de variations dans son apprentissage. Mario se retrouve face à des situations plus diverses et doit apprendre à s'adapter à des circonstances qui changent. Cela le rend plus apte à gérer les imprévus et à mieux réagir face aux défis du jeu. Un autre moyen d'améliorer ses capacités de généralisations sont de l'entrainer sur différents niveaux.



Dans cette configuration, chaque action a 2% de chance d'être aléatoire. C'est significatif car dans notre modèle, 4 actions sont décidées pour une seconde de jeu, ce qui implique que chaque seconde, Mario a ((2 * 4) * CHANCE_DE_SAUTER) = 4% de probabilité de faire un saut non-prévu initialement.

On observe non seulement que le record progresse jusqu'à atteindre le maximum (la fin du niveau), mais le score moyen glissant sur 1000 parties est également en hausse.



D'ailleurs, on constate avec joie que le taux de victoire étalé sur 1000 parties est lui aussi en forte hausse. Il tend parfois vers le bas quand la configuration de la Q-Table mène à certains obstacles, mais le modèle fini toujours par les surmonter.

Conclusion

La décision d'adopter le Q-Learning comme méthode d'intelligence artificielle pour notre jeu "Mario" s'est avérée être un choix judicieux et efficace. Cette technique d'apprentissage par renforcement offre plusieurs avantages significatifs qui se marient parfaitement avec les dynamiques et les défis uniques de "Mario".

Avantages Clés du Q-Learning dans Notre Contexte

1. Flexibilité et Adaptabilité : Le Q-Learning, avec sa capacité à apprendre de l'environnement sans nécessiter un modèle prédictif, s'est adapté de manière fluide à l'environnement dynamique de Mario. Il a permis à l'IA de réagir et de s'adapter aux situations variées et imprévisibles du jeu.
2. Équilibre entre Exploration et Exploitation : La mécanique de l'epsilon dans le Q-Learning a facilité un équilibre optimal entre l'exploration de nouvelles stratégies et l'exploitation des connaissances acquises, ce qui est crucial pour un apprentissage progressif et efficace.
3. Personnalisation et Réglage Fin : Grâce à notre capacité à ajuster les récompenses et à mettre en œuvre des mécanismes spécifiques comme la rétropropagation pour les sauts, nous avons pu affiner l'apprentissage de l'IA pour qu'elle réponde de manière plus précise et stratégique aux exigences du jeu.
4. Amélioration Continue et Apprentissage à Long Terme : Le système de mise à jour périodique de la Q-Table, conjugué à la sauvegarde et au chargement de l'état d'apprentissage, a assuré une amélioration continue et un apprentissage à long terme pour l'IA, lui permettant de devenir de plus en plus compétente au fil des sessions de jeu.
5. Dans le cas où l'objectif est de finir le niveau avec le moins d'essais possibles, l'approche déterministe (epsilon à 0) est encore plus efficace que les réseaux de neurones profonds et les modèles génétiques.

L'application du Q-Learning dans Mario a non seulement permis de développer une IA capable de naviguer de manière autonome et efficace dans le jeu, mais il a permis de battre le niveau en moins de 100 essais.

En résumé, le choix du Q-Learning pour notre projet s'est révélé être une décision stratégique qui a apporté profondeur, adaptabilité et sophistication à l'intelligence artificielle du jeu. Les résultats obtenus soulignent l'efficacité de cette méthode d'apprentissage par renforcement dans les environnements de jeu complexes et dynamiques.

Avec plus de temps, nous aurions aimés explorer les solutions liées à l'évolution génétique qui semblent très puissantes et généralisables. Une approche neuronale serait aussi intéressante, bien que les performances actuelles soient très satisfaisantes !


YOU WIN !

Annexe

https://www.thats-ai.org/fr-CH/units/l-histoire-de-l-ia

https://www.journaldunet.fr/intelligence-artificielle/guide-de-l-intelligence-artificielle/1501295-ai-winter/

https://www.datarockstars.ai/lintelligence-artificielle-et-les-jeux-video/

https://www.saagie.com/fr/blog/blog-l-intelligence-artificielle-dans-les-jeux-video/

https://www.lebigdata.fr/ia-bouleverser-jeux-videos

https://blent.ai/blog/a/detection-images-yolo-tensorflow

https://pypi.org/project/gym-super-mario-bros/

https://fr.wikipedia.org/wiki/R%C3%A9seaux_antagonistes_g%C3%A9n%C3%A9ratifs

https://fr.wikipedia.org/wiki/Enter_the_Gungeon

https://fr.wikipedia.org/wiki/Super_Mario_Bros.

https://fr.wikipedia.org/wiki/Deep_Blue

https://fr.wikipedia.org/wiki/AlphaGo

https://fr.wikipedia.org/wiki/Pong

https://fr.wikipedia.org/wiki/Space_Invaders

https://www.youtube.com/watch?v=qv6UVOQ0F44

https://www.youtube.com/watch?v=2eeYqJ0uBKE

https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique

https://fr.wikipedia.org/wiki/Algorithme_%C3%A9volutionniste

https://fr.wikipedia.org/wiki/Algorithme_NEAT