Apprentissage par renforcement appliqué à Super Mario Land

Jules Bobeuf
Université du Littoral Côte d'Opale
bobeuf.jules@gmail.com

Valentin Devisme
Université du Littoral Côte d'Opale
valentin.devisme@etu.univ-littoral.fr

Juin 2025

Mots-clés : Super Mario Land, Intelligence Artificielle, NEAT, Q-Learning

1 Abstract

Dans cet article, nous comparons deux approches d'intelligence artificielle appliquées à un jeu de plateforme emblématique : Super Mario Land. La première repose sur l'apprentissage par renforcement (Q-Learning), tandis que la seconde utilise NEAT (NeuroEvolution of Augmenting Topologies), une méthode évolutive permettant de générer des réseaux de neurones artificiels. Nous évaluons les performances des deux techniques en termes de progression dans le jeu et de régularité. Nos résultats montrent que les deux algorithmes sont capables de jouer de manière autonome, malgré plusieurs limitations liées à la complexité du jeu.

2 Introduction

Ces dernières années, les jeux vidéo sont devenus un bon terrain pour tester des approches en intelligence artificielle (IA). Ils offrent un environnement interactif, dynamique et contrôlé, qui permet d'observer comment différents algorithmes se comportent dans des situations variées. Les jeux de plateforme, comme *Super Mario Land*, sont souvent utilisés car leurs règles sont simples, mais ils restent assez difficiles pour poser de vrais défis aux agents.

La plupart des travaux en IA appliquée aux jeux utilisent l'apprentissage supervisé. Cette méthode consiste à entraîner un modèle à partir de données récoltées auprès de joueurs humains, pour qu'il apprenne à imiter leur comportement. Même si cette approche peut donner de bons résultats, elle dépend fortement de la qualité et de la quantité des données, et elle a du mal à s'adapter à des situations nouvelles que le modèle n'a jamais rencontrées.

Dans cet article, nous nous intéressons à deux méthodes qui ne nécessitent pas de données humaines : le Q-Learning, un algorithme d'apprentissage par renforcement, et NEAT (*NeuroEvolution of Augmenting Topologies*), une approche basée sur des algorithmes évolutionnaires. Ces techniques permettent à un agent d'apprendre en jouant lui-même, en testant différentes actions et en s'adaptant à ce qu'il observe dans l'environnement.

Nous appliquons ces deux méthodes à *Super Mario Land*, un jeu emblématique de la Game Boy, pour comparer leurs performances, leur stabilité et leur capacité à progresser de manière autonome dans un environnement varié et imprévisible.

2.1 État de l'art

Plusieurs travaux ont exploré l'utilisation du Q-Learning ou des algorithmes évolutionnaires dans les jeux vidéo. Bien que ces approches aient été testées sur divers types de jeux, peu d'articles les appliquent spécifiquement à *Super Mario Land*.

2.1.1 Apprentissage par renforcement (Q-Learning)

Lample et Chaplot [2] ont appliqué le Q-Learning, associé à un réseau de neurones, à un jeu de tir à la première personne (*Doom*). Leur agent apprend à jouer en explorant le jeu et réussit à surpasser les bots internes ainsi que certains joueurs humains.

2.1.2 Algorithmes évolutionnaires (Genetic Algorithms)

Baldominos et al. [1] ont utilisé un algorithme génétique pour apprendre à jouer à *Infinite Mario Bros.*. Leur agent parvient à générer des séquences d'actions efficaces qui lui permettent de finir des niveaux complexes sans supervision humaine.

3 Environnement de développement

Tous les tests ont été réalisés sur des ordinateurs portables classiques (entre 8 et 16 Go de RAM), sans recours à des serveurs ou GPU spécifiques. Le jeu *Super Mario Land* a été émulé à l'aide de la librairie `PyBoy`, qui permet d'interagir avec des jeux Game Boy en Python. Les données générées pendant les sessions d'apprentissage (comme les scores, états et actions) ont été stockées dans une base de données `MongoDB`, afin de faciliter le suivi des agents et l'analyse des performances. Le code a été écrit en Python, avec l'aide de librairies comme `NumPy` et `matplotlib` pour les calculs et les visualisations.

4 Algorithmes

Comme spécifié dans l'introduction, cet article traite deux algorithmes différents :

4.1 Q-Learning

Le Q-Learning est un algorithme d'apprentissage par renforcement qui permet à un agent d'apprendre à prendre de bonnes décisions en interagissant avec un environnement, sans avoir besoin d'un modèle de celui-ci. Le principe est de maximiser une récompense cumulée à long terme en apprenant une fonction $Q(s, a)$, qui estime la valeur d'une action a à effectuer dans un état s.

Lors de chaque étape, l'agent observe l'état courant s, choisit une action a, puis observe la récompense r et le nouvel état s'. La fonction Q est ensuite mise à jour selon la formule suivante :

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a) \right]$$

Où :
- s correspond à un état
- a correspond à une action
- α est le taux d'apprentissage (learning rate), entre 0 et 1 ;
- γ est le facteur de réduction (discount factor), qui détermine l'importance des récompenses futures ;
- r est la récompense reçue après avoir effectué l'action a ;
- s' est l'état suivant ;
- $\max_{a'} Q(s', a')$ représente la meilleure valeur estimée pour le nouvel état.

4.1.1 Exemple d'application de l'algorithme à Super Mario Land

Dans le contexte du jeu *Super Mario Land*, un état est représenté par une portion de la carte autour de Mario, sous forme d'une matrice de taille $x \times y$. Chaque élément de cette matrice (blocs, ennemis, objets, etc.) est encodé par un identifiant unique permettant à l'algorithme de reconnaître la situation actuelle du jeu.

Une action correspond à l'entrée que Mario effectue sur la console, comme appuyer sur un bouton. Les actions principales utilisées dans notre algorithme sont les déplacements (droite, gauche) et le saut.

Enfin, la récompense est une valeur numérique que l'agent reçoit après une action, selon son utilité. Par exemple, ramasser une pièce peut rapporter une récompense de 1 point, tandis qu'avancer sans se faire toucher peut aussi être valorisé. Le but est d'encourager les comportements efficaces pour progresser dans le niveau.

4.1.2 Exploitation vs Exploration

Un aspect central de l'apprentissage par renforcement est le dilemme entre **exploitation** et **exploration**.

— **L'exploitation** consiste à choisir l'action qui semble la meilleure selon ce que l'agent a déjà appris.
— **L'exploration**, au contraire, pousse l'agent à essayer de nouvelles actions, même si elles paraissent moins bonnes, pour découvrir éventuellement de meilleures stratégies.

Une méthode classique pour gérer ce compromis est l'algorithme ε**-greedy**. L'idée est simple :

— Avec une probabilité ε, l'agent choisit une action au hasard (exploration).
— Sinon, il choisit l'action qui maximise la valeur Q connue (exploitation).

Dans le cadre de nos recherches, **nous n'avons pas utilisé la stratégie** ε**-greedy**, qui consiste à baisser l'exploration au fur et à mesure de l'entraînement. À la place, nous avons privilégié une exploration constante.

4.2 NEAT

La méthode NEAT (*NeuroEvolution of Augmenting Topologies*) est un algorithme évolutionnaire développé par Ken Stanley pour concevoir automatiquement des réseaux de neurones artificiels. Contrairement aux approches traditionnelles qui fixent la topologie du réseau à l'avance, NEAT fait évoluer à la fois la structure (neurones et connexions) et les poids des réseaux, ce qui permet de découvrir des architectures adaptées à des problèmes complexes comme les jeux vidéo.

4.2.1 Principe de fonctionnement

NEAT s'inspire des principes de l'évolution biologique pour optimiser les réseaux de neurones. L'algorithme repose sur trois mécanismes principaux :

— **Initialisation minimaliste** : Les réseaux commencent avec une structure simple, composée uniquement de neurones d'entrée (représentant l'état du jeu) et de sortie (correspondant aux actions possibles). De nouvelles connexions et neurones sont ajoutés progressivement par mutation.
— **Mutation et complexification** : À chaque génération, les réseaux subissent des mutations qui peuvent ajouter ou supprimer des neurones et des connexions, ou modifier les poids existants. Cela permet d'explorer des topologies variées tout en évitant des structures inutilement complexes.
— **Sélection et spéciation** : Les réseaux sont évalués selon une fonction de fitness, qui mesure leur performance dans l'environnement. Pour préserver la diversité, NEAT divise la population en espèces basées sur la similarité de leurs topologies. Une sélection par tournoi favorise les meilleurs individus de chaque espèce, tout en protégeant les innovations récentes contre une élimination prématurée.

La fonction de fitness est cruciale pour guider l'évolution. Dans le contexte de *Super Mario Land*, elle peut être définie comme suit :

$$\text{Fitness} = w_1 \cdot \text{progression} + w_2 \cdot \text{survie} - w_3 \cdot \text{stagnation}$$

où :

— progression représente l'avancement dans le niveau (par exemple, mesuré par `mario.level_progress` dans PyBoy) ;

— survie reflète la durée de vie de Mario (par exemple, le nombre de frames avant une mort) ;

— stagnation pénalise l'absence de mouvement (par exemple, si Mario reste immobile) ;

— w_1, w_2, w_3 sont des poids ajustés expérimentalement.

Pour éviter la surcomplexification, on peut appliquer à NEAT une pénalité proportionnelle à la complexité du réseau (nombre de neurones et connexions), intégrée dans une fonction de fitness ajustée :

$$\text{Fitness}_{\text{ajustée}} = \text{Fitness} - c \cdot \text{complexité}$$

où c est un coefficient de pénalité.

4.2.2 Application à Super Mario Land

Dans notre implémentation, l'objectif était d'utiliser NEAT pour entraîner un agent capable de jouer à *Super Mario Land* de manière autonome. Les neurones d'entrée du réseau correspondent à une représentation compressée de la carte de jeu, obtenue via la fonction `game_area()` de PyBoy. Cette carte est une matrice où chaque élément représente un type de tuile (blocs, ennemis, bonus, etc.), encodé par un identifiant unique défini dans un dictionnaire de compression (`mapping_compressed`).

Chaque neurone est conçu pour détecter des motifs spécifiques dans la carte de jeu, à une position relative par rapport à Mario. Par exemple, un neurone peut s'activer si une tuile correspondant à un ennemi (identifiée par un ensemble de valeurs dans `mapping_compressed`) est détectée à des coordonnées relatives (`relative_x`, `relative_y`). Contrairement à un algorithme NEAT traditionnel, où les poids des connexions sont des valeurs numériques ajustées par mutation, notre approche associe à chaque neurone un ensemble de paramètres évolutifs : les motifs attendus (`expected_values`), les coordonnées relatives, et un type d'activation (*match* ou *inverse*). Les connexions entre neurones déterminent des actions de jeu (déplacement à droite, à gauche, ou saut) et leur durée, également sujettes à mutation. Ainsi, les mutations modifient non seulement la topologie du réseau (ajout/suppression de neurones ou connexions), mais aussi les motifs reconnus, les positions relatives, les types d'activation, et les paramètres des actions, permettant une adaptation fine à l'environnement dynamique de *Super Mario Land*. Les neurones de sortie traduisent ces activations en commandes directes pour l'émulateur PyBoy.

Les réseaux évoluent sur plusieurs générations, chaque génération comprenant une population de génomes (représentant des réseaux distincts). À chaque génération :

1. Les génomes sont évalués dans l'émulateur PyBoy pendant un nombre fixe de frames (par exemple, 100).
2. Une fonction de fitness récompense la progression dans le niveau et pénalise la stagnation ou la mort.
3. Les meilleurs génomes sont sélectionnés via un tournoi, puis mutés pour former la génération suivante.

5 Approche et résultats

5.1 Q-Learning - Récompenses

Après avoir implémenté l'algorithme, nous nous sommes penchés sur notre premier sujet : quelles récompenses donner à Mario afin de maximiser son avancement dans le niveau.

Nous avons d'abord donné une récompense à chaque fois qu'il avançait. C'est une bonne approche, car c'est exactement ce que l'on souhaite : qu'il aille le plus loin possible. Pour aller plus loin, nous avons aussi testé un malus lorsqu'il reculait. Mais ce n'était pas une bonne solution : Mario avançait puis reculait sans arrêt, ce qui le faisait stagner. Pas idéal pour notre objectif.

Pour résoudre ce problème, nous avons donné une récompense uniquement lorsqu'il atteignait une position plus lointaine que la maximale précédente. Cela élimine l'effet de sur-place.

Après quelques tests, nous avons aussi supprimé les malus, car cela n'améliorait pas les performances du modèle.

Nous avons testé plusieurs autres systèmes de récompenses, basés sur le temps ou l'élimination des ennemis présents sur la carte, mais ceux-ci ne se sont pas révélés concluants. D'après nos différents tests, plus il y avait de contraintes pour contrôler Mario, moins le modèle fonctionnait efficacement. Nous avons donc choisi de nous arrêter là pour les récompenses et de nous concentrer davantage sur les hyperparamètres.

5.2 Q-Learning - Hyperparamètres

L'étape suivante a été de déterminer quels hyperparamètres donnaient les meilleurs résultats. Pour cela, nous avons défini quatre modèles différents, chacun avec un couple *(taux d'apprentissage* et *facteur de réduction* spécifique. Chacun de ces modèles a été entraîné pendant 1500 époques.

— **Modèle 1** : learning rate = 0.5, discount factor = 0.1
— **Modèle 2** : learning rate = 0.001, discount factor = 0.99
— **Modèle 3** : learning rate = 0.001, discount factor = 0.1
— **Modèle 4** : learning rate = 0.99, discount factor = 0.99

Ces valeurs ont été choisies pour représenter des comportements très différents : certains modèles apprennent très vite mais oublient vite le passé (comme le modèle 1), tandis que d'autres apprennent lentement mais gardent en mémoire des événements lointains (comme le modèle 2). L'idée était de voir lequel de ces équilibres donnait de meilleurs résultats dans un environnement aussi instable et dynamique que *Super Mario Land*.

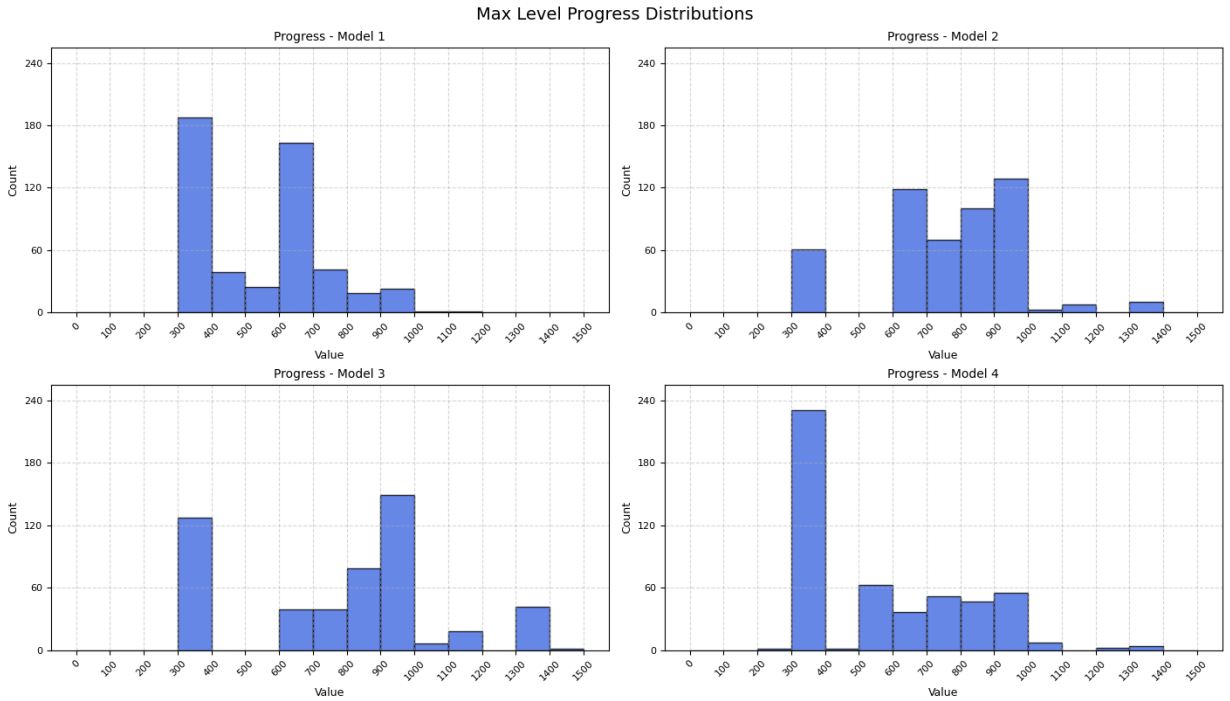

FIGURE 1 – Distribution du niveau maximum atteint sur 500 itérations pour les modèles 1, 2, 3 et 4

Les modèles 1 et 4 ont montré de mauvais résultats et beaucoup d'instabilité. On voit qu'ils échouent souvent entre les positions 300 et 400, une zone assez simple du niveau. Cela montre qu'ils ne sont pas fiables. Les modèles 2 et 3 ont mieux fonctionné, avec des résultats plus réguliers. Le modèle 2 était le plus stable, donc c'est celui qu'on a choisi pour continuer. Chacun de ces modèles a été entraîné pendant 500 époques. Nous avions initialement prévu un entraînement sur 1500 époques afin d'obtenir des résultats équivalents aux modèles précédents, mais des limitations matérielles nous ont empêchés de poursuivre l'entraînement.

Nous avons ensuite étudié l'effet de la taille de l'état sur les performances de l'agent. Étant donné que Mario n'est pas entraîné sur toute la surface du jeu, il est important de choisir une fenêtre d'observation adaptée. Pour cela, nous avons testé quatre combinaisons différentes de dimensions (`state_width` × `state_height`) :

— Modèle 5 : 20 × 10
— Modèle 6 : 20 × 20
— Modèle 7 : 10 × 20
— Modèle 8 : 10 × 10

L'objectif était de déterminer si un état plus large ou plus haut permettait à Mario d'avoir une meilleure vision de son environnement, et donc de mieux anticiper les obstacles.

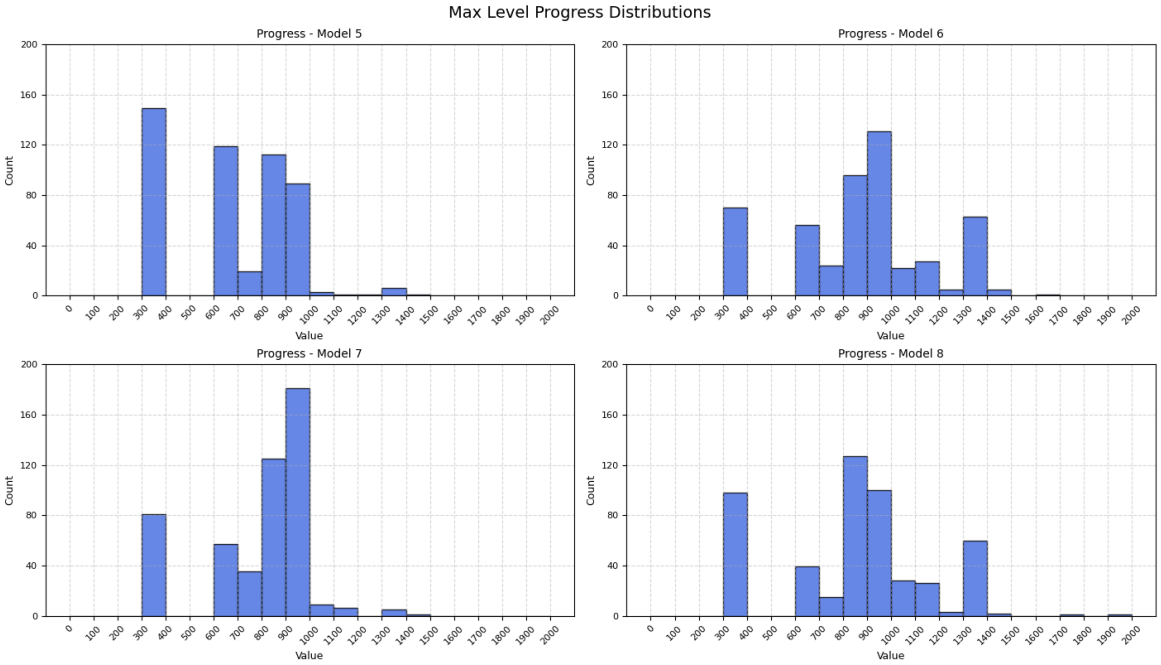

FIGURE 2 – Distribution du niveau maximum atteint sur 500 itérations pour les modèles 5, 6, 7 et 8

Le modèle 5 obtenait parfois de bons résultats, mais restait très irrégulier. Le modèle 7, quant à lui, avait beaucoup de mal à franchir certains obstacles du niveau, probablement à cause d'un manque d'informations visuelles suffisantes sur l'environnement.

Les modèles 6 et 8 ont montré de meilleures performances globales. Le modèle 8 (10 × 10) atteignait parfois des niveaux plus élevés, mais de manière instable. En revanche, le modèle 6 (20 × 20) était plus constant et montrait une progression plus fiable.

Nous avons donc poursuivi nos tests avec le modèle 6, que l'on utilisera pour comparer les performances du Q-Learning et de NEAT.

5.3 NEAT

Concernant la méthode NEAT, nous n'avons malheureusement pas pu obtenir de résultats exploitables. En effet, des difficultés techniques lors de l'implémentation de l'algorithme nous ont empêchés de finaliser son intégration et de lancer des entraînements dans les délais impartis pour cet article.

Parmi les problèmes rencontrés :

— **Incohérence des fitness** : Certains génomes obtenaient des fitness élevés lors de l'évaluation initiale, mais ne reproduisaient pas ces performances lors de tests ultérieurs, suggérant des anomalies dans la lecture de `mario.level_progress` ou une corruption des génomes lors de la sérialisation.

— **Stagnation de l'évolution** : L'algorithme tendait à converger vers des génomes simples mais peu performants, probablement à cause d'une pénalité de complexité trop forte ou de mutations destructrices.

— **Performance de l'émulateur** : L'utilisation de multiples instances de PyBoy en parallèle surchargeait les ressources matérielles, rendant l'entraînement instable.

Ces limitations nous ont empêchés de comparer directement NEAT avec Q-Learning dans ce travail. Cependant, NEAT reste une approche prometteuse pour *Super Mario Land*, car sa capacité à faire évoluer des topologies complexes pourrait permettre de découvrir des stratégies adaptatives face aux environnements dynamiques du jeu.

6 Comparaisons des deux approches

Afin de comparer nos algorithmes, nous avons entraîné le modèle 6 afin d'atteindre les 1500 époques. Ensuite, nous avons relancé ce modèle 500 fois afin d'évaluer ses performances finales et sa régularité.

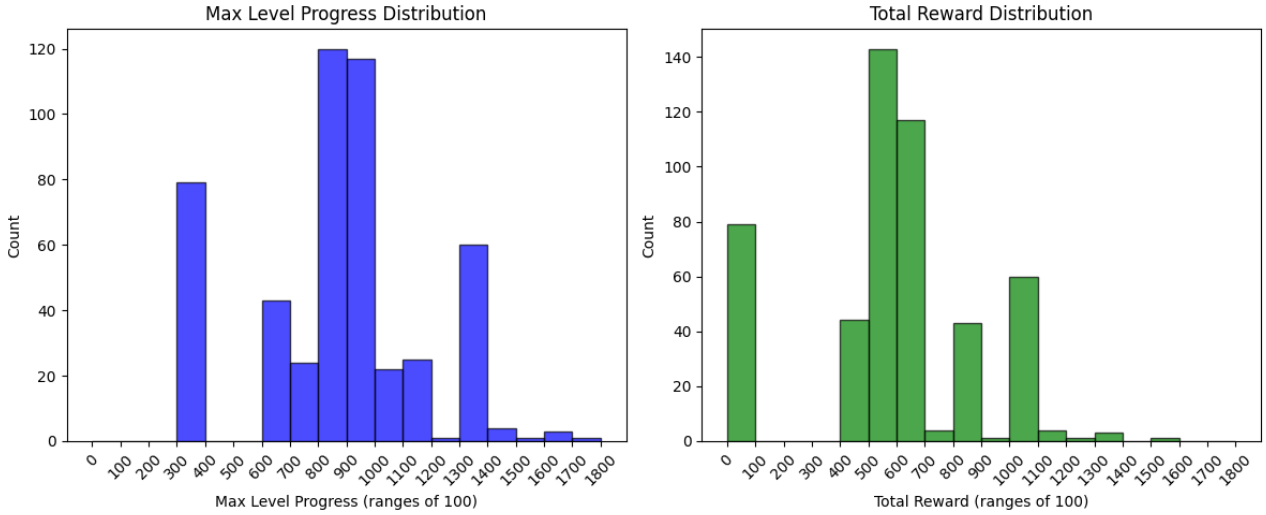

FIGURE 3 – Distribution du niveau maximum atteint et des récompenses totales sur 500 itérations pour le modèle 6

Le modèle 6 entraîné sur 1500 itérations montre une nette amélioration : la majorité des agents atteignent une progression modérée (autour de 900–1100), avec moins de cas bloqués au début du niveau. De plus, plusieurs agents atteignent des positions avancées (1600-1800), ce qui reflète un apprentissage plus stable et efficace.

7 Conclusion et Ouverture

Nous avons exploré l'efficacité de deux approches d'intelligence artificielle appliquées à un jeu de plateforme rétro : Super Mario Land. Malgré des efforts d'implémentation, la méthode NEAT n'a pas pu être exploitée à cause de problèmes techniques. Cependant, l'algorithme de Q-Learning a montré des résultats prometteurs.

Parmi les différents modèles testés, certains couples de paramètres (comme un taux d'apprentissage faible avec un facteur de réduction élevé) ont permis une progression plus régulière et plus stable du joueur. Le modèle 2, avec un learning rate de 0.001 et un discount de 0.99, a été retenu pour ses performances constantes. Par la suite, l'ajustement de la taille de l'état a également eu un impact : le modèle 6, utilisant une fenêtre de 20×20 autour de Mario, a donné de meilleurs résultats sur 1500 itérations, avec une capacité accrue à franchir des obstacles complexes.

Pour aller plus loin, plusieurs pistes peuvent être envisagées. D'abord, affiner davantage les hyperparamètres permettrait peut-être d'améliorer encore les performances. Ensuite, augmenter le temps d'entraînement au-delà de 1500 itérations pourrait conduire à un apprentissage plus abouti et une progression plus profonde dans le jeu.

Pour finir, ce travail montre qu'un agent peut apprendre à jouer de manière autonome à un jeu comme Super Mario Land grâce au Q-Learning, tout en laissant la porte ouverte à de nombreuses améliorations futures.

Références

[1] Alejandro Baldominos, Yago Saez, and Pedro Isasi. Learning levels of super mario bros with genetic algorithms. *Proceedings of the 2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 2412–2419, 2015.

[2] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.