Rapport d'élèves ingénieurs
Projet ZZ3
Filière F2 : Génie Logiciel et Systèmes Informatiques

Développement d'une IA capable de jouer à Pokémon Rouge

Présenté par :
Mathis ROUBILLE
Sami ABDUL-SALAM

Responsable filière : Loïc YON
Durée du projet : 120 heures

Date de rendu du rapport : 25/03/2025
Date de la soutenance : 27/03/2025

Campus des Cézeaux. 1 rue de la Chebarde. TSA 60125. 63178 Aubière CEDEX

Table des matières

Table des figures

Glossaire

A2C Advantage Actor-Critic, algorithme d'apprentissage par renforcement combinant deux approches : l'*Actor*, qui détermine les actions à prendre, et le *Critic*, qui évalue ces actions en estimant leur avantage par rapport à une ligne de base. A2C met à jour simultanément plusieurs environnements parallèles pour stabiliser l'apprentissage et améliorer son efficacité. Cet algorithme est largement utilisé pour résoudre des problèmes complexes, notamment dans les jeux vidéo et la robotique. 7

Assembleur Langage de programmation bas niveau proche du langage machine, utilisé pour modifier directement le comportement du jeu Pokémon Rouge à partir de son code source décompilé. 32

CNN Convolutional Neural Network (réseau neuronal convolutionnel). Ce type de réseau neuronal artificiel est spécialement conçu pour traiter des données visuelles telles que des images ou des vidéos. Il utilise des couches de convolution qui permettent d'extraire automatiquement les caractéristiques spatiales des images en détectant des motifs comme des contours, textures ou formes complexes. Cette capacité à identifier et généraliser ces caractéristiques visuelles rend les CNN particulièrement efficaces pour des tâches telles que la reconnaissance d'objets, la classification d'images et l'analyse visuelle complexe, comme dans notre projet où ils sont employés pour interpréter visuellement l'état du jeu Pokémon Rouge. 15

DQN Deep Q-Network (Réseau Q profond). Algorithme d'apprentissage par renforcement combinant le *Q-learning*, une méthode d'apprentissage fondée sur la valeur des actions, avec des réseaux de neurones profonds (deep learning). Le DQN apprend à associer à chaque action possible une valeur représentant la récompense future attendue dans un état donné. L'utilisation de réseaux neuronaux profonds permet de généraliser efficacement à partir d'observations complexes, telles que des images, et ainsi de résoudre des problèmes d'apprentissage où les espaces d'état sont de grande dimension, comme dans le cas du jeu Pokémon Rouge. 7

Fitness Mesure numérique indiquant la performance d'un réseau neuronal ou d'un individu dans un algorithme évolutionnaire. 12, 26, 27, 29

Génome Dans le contexte de l'algorithme d'apprentissage évolutif NEAT (*NeuroEvolution of Augmenting Topologies*), un génome représente l'ensemble complet des paramètres définissant un réseau neuronal spécifique. Cela inclut à la fois la topologie du réseau (c'est-à-dire la disposition et le nombre de neurones ainsi que les connexions entre eux) et les poids associés à chaque connexion. Chaque génome constitue un individu au sein de la population évolutive, et sa performance ou *fitness* détermine sa capacité à se reproduire et à transmettre ses caractéristiques aux générations suivantes. 11, 19, 21, 23, 26, 27, 29

PP Points de Pouvoir (*Power Points*). Dans l'univers des jeux Pokémon, les PP représentent le nombre de fois qu'une attaque spécifique peut être utilisée par un Pokémon. Chaque attaque possède son propre nombre maximal de PP, déterminant ainsi combien de fois elle peut être lancée au cours des combats. Lorsque les PP d'une attaque tombent à zéro, cette attaque ne peut plus être utilisée jusqu'à restauration. 18, 21

PPO Proximal Policy Optimization (optimisation de politique proximale). Algorithme d'apprentissage par renforcement particulièrement efficace et stable, conçu pour optimiser des politiques de prise de décision dans des environnements complexes. PPO

fonctionne en limitant les changements de la politique à chaque mise à jour, grâce à une contrainte dite « proximale » qui évite les modifications trop importantes en une seule étape, stabilisant ainsi l'apprentissage. Cet algorithme est particulièrement adapté aux situations nécessitant des interactions longues et complexes avec l'environnement, comme les jeux vidéo ou la robotique. 7

ROM Acronyme de *Read-Only Memory*, désigne ici un fichier informatique contenant les données extraites d'une cartouche de jeu vidéo. Une ROM peut être utilisée avec un émulateur pour exécuter le jeu original sur une autre plateforme ou être modifiée pour créer une version optimisée, spécifiquement adaptée à des entraînements en intelligence artificielle. 10, 28, 29, 32

SAC Soft Actor-Critic. Algorithme d'apprentissage par renforcement basé sur l'approche *Actor-Critic*, qui intègre explicitement une notion d'entropie dans l'objectif d'apprentissage. SAC encourage l'exploration en récompensant les politiques diversifiées, ce qui permet à l'agent de mieux explorer l'environnement et d'éviter de rester bloqué dans des stratégies sous-optimales. Cette approche est particulièrement efficace dans les environnements complexes à action continue ou avec de grandes dimensions d'état, comme en robotique et dans les jeux vidéo. 7

Table des types Tableau définissant l'efficacité des attaques selon le type du Pokémon attaquant et du Pokémon défenseur. 17

TRPO Trust Region Policy Optimization (optimisation de politique par région de confiance). Algorithme avancé d'apprentissage par renforcement conçu pour améliorer la stabilité des mises à jour de la politique. TRPO définit une « région de confiance » autour de la politique actuelle, limitant ainsi la taille des changements autorisés à chaque mise à jour afin d'éviter des modifications trop brusques et instables. Bien qu'efficace pour gérer les environnements complexes, TRPO est plus coûteux en calcul que PPO, son successeur direct, ce dernier étant généralement préféré pour sa simplicité d'implémentation. 7

Type Classification fondamentale dans les jeux Pokémon attribuée à chaque Pokémon et à leurs attaques. Chaque type (par exemple, Feu, Eau, Plante, Électrique, etc.) détermine l'efficacité des attaques durant les combats. La relation entre les types suit une logique complexe définissant leurs forces et faiblesses : certaines attaques seront super efficaces (causant davantage de dégâts), d'autres seront moins efficaces (causant moins de dégâts), ou n'auront aucun effet sur certains types spécifiques. La maîtrise des interactions entre types est essentielle pour élaborer des stratégies efficaces en combat. 4, 6, 17, 18

1 Introduction et Contexte

1.1 Présentation du jeu Pokémon Rouge

Pokémon Rouge est un jeu de rôle (RPG) développé par Game Freak et publié par Nintendo sur la Game Boy en 1996 au Japon, puis en 1999 en Europe et en Amérique du Nord. Il est l'un des premiers volets de la célèbre franchise Pokémon et a posé les bases du gameplay qui perdure dans les générations suivantes. Avec son succès international, il a contribué à l'essor du phénomène Pokémon et à son inscription durable dans la culture vidéoludique [3].

Dans ce jeu, le joueur incarne un jeune dresseur dont l'objectif est de capturer et entraîner une équipe de Pokémon afin de vaincre les huit champions d'arène et d'accéder à la Ligue Pokémon. L'aventure se déroule dans la région fictive de Kanto, une zone variée composée de villes, de forêts, de grottes et de routes interconnectées, où le joueur doit affronter d'autres dresseurs et résoudre des énigmes pour progresser.

Les principales caractéristiques du jeu sont :

— Un **monde semi-ouvert** avec une liberté d'exploration progressive à mesure que le joueur débloque de nouvelles capacités.

— Un **système de combats au tour par tour** stratégique, où chaque Pokémon dispose de statistiques, d'un Type et d'une sélection de capacités influençant l'issue des affrontements.

— Une **gestion des ressources et de l'équipe**, où le joueur doit composer une équipe équilibrée et utiliser intelligemment les objets disponibles.

— Des **événements aléatoires**, influençant la progression, notamment à travers les rencontres de Pokémon sauvages et certains effets de combat.

Figure 1 – Couverture du jeu Pokémon Rouge

Ces éléments rendent le jeu particulièrement intéressant pour une intelligence artificielle. En effet, celle-ci doit non seulement apprendre à naviguer dans un environnement semi-ouvert en tenant compte des obstacles et des transitions entre les zones, mais aussi maîtriser un système de combat où les décisions prises doivent s'adapter à la situation et aux spécificités des adversaires rencontrés. L'apprentissage d'une IA pour jouer efficacement à Pokémon Rouge implique donc une approche alliant exploration, prise de décision et adaptation dynamique aux contraintes du jeu.

1.2 Objectifs du projet

L'objectif principal de ce projet est de concevoir une intelligence artificielle capable de jouer à Pokémon Rouge de manière autonome. Contrairement aux approches classiques cherchant à optimiser le temps et le taux de complétion du jeu, notre projet vise avant tout à explorer les défis techniques liés à la conception d'une telle IA, en mettant l'accent sur la prise de décision et l'adaptation aux conditions du jeu.

Notre démarche ne consiste pas à développer une IA performante capable de terminer le jeu de manière optimale, mais plutôt à nous initier aux problématiques de l'intelligence artificielle appliquée à un environnement complexe et semi-ouvert. L'objectif est donc d'acquérir une première expérience dans ce domaine, d'expérimenter différentes méthodes, et d'obtenir des résultats initiaux permettant d'évaluer la faisabilité d'une telle approche. Bien entendu, si l'IA parvient à progresser de manière significative dans le jeu, ce serait un résultat intéressant, mais ce n'est pas notre priorité. Ce projet se veut avant tout une opportunité d'apprentissage et d'exploration, où chaque avancée, même modeste, apportera un éclairage sur les défis liés à l'automatisation du jeu.

Nos objectifs se déclinent en plusieurs axes de recherche et développement :

- **Comprendre les mécaniques internes du jeu** : analyser le fonctionnement de Pokémon Rouge en identifiant les règles, les interactions et les contraintes inhérentes à son environnement. Cela inclut l'étude des mécanismes de combat, la gestion des déplacements sur la carte, et l'utilisation des objets. L'objectif est d'acquérir une vision claire des éléments influençant la progression d'un joueur.
- **Explorer les méthodes d'interfaçage entre l'IA et le jeu** : identifier les moyens techniques permettant à l'IA d'interagir avec Pokémon Rouge. Cela inclut l'utilisation d'un émulateur pour capturer et interpréter l'état du jeu, l'analyse mémoire pour extraire des données précises, ou encore la vision par ordinateur pour simuler une perception humaine. Le choix de la méthode d'interfaçage aura un impact direct sur la flexibilité et la performance de l'IA.
- **Définir une stratégie de prise de décision** : concevoir un système permettant à l'IA de gérer à la fois la navigation dans le monde du jeu et la stratégie en combat. Il s'agit de choisir les algorithmes adaptés pour planifier les déplacements et sélectionner les meilleures actions en fonction des adversaires rencontrés. Un bon équilibre entre exploration et prise de risque sera essentiel pour éviter que l'IA ne se retrouve bloquée dans certaines situations.
- **Mettre en place un mécanisme d'apprentissage** : expérimenter des méthodes d'apprentissage comme l'apprentissage par renforcement afin que l'IA puisse ajuster son comportement en fonction des situations rencontrées et des résultats obtenus. L'objectif sera de permettre à l'agent d'améliorer ses performances progressivement, en apprenant à réagir aux événements de manière plus efficace.
- **Élaborer un système de récompense adapté** : définir des critères qui permettront de guider l'IA vers des comportements cohérents avec le jeu. Par exemple, encourager l'exploration de la carte, favoriser la victoire en combat, ou encore optimiser la gestion des ressources pour assurer une progression efficace. La conception du système de récompense sera un enjeu crucial, car elle influencera directement les décisions prises par l'IA.

L'ensemble de ces objectifs vise à nous initier à la conception d'une IA dans un cadre ludique mais complexe, tout en tenant compte des multiples paramètres influençant la prise de décision dans un RPG comme Pokémon Rouge. De plus, cette étude pourra servir de point de départ pour d'éventuelles explorations futures, que ce soit dans l'amélioration de l'IA développée ou dans l'application de ces concepts à d'autres jeux ou environnements interactifs.

1.3 Présentation des défis

Le développement d'une intelligence artificielle capable de jouer à *Pokémon Rouge* soulève plusieurs défis complexes, en raison des spécificités du jeu, de son **monde semi-ouvert** non linéaire et de son système de **combat stratégique**.

Tout d'abord, l'**exploration** du monde de *Pokémon Rouge* constitue un véritable défi pour l'IA. Le jeu se déroule dans un environnement semi-ouvert, où l'exploration n'est pas linéaire et comprend des zones variées telles que des forêts, des villes et des grottes. L'IA doit naviguer à travers ces différentes zones, en prenant en compte les obstacles et les transitions entre les environnements, tout en évitant les cul-de-sac et les zones inaccessibles. Cette tâche est rendue encore plus difficile par l'absence de cartes interactives dans le jeu, ce qui oblige l'IA à explorer de manière dynamique et à s'adapter constamment aux informations dont elle dispose.

Le système de **combat** de *Pokémon Rouge* représente un autre défi majeur. Il repose sur un mécanisme au tour par tour, dans lequel chaque décision prise peut influencer l'issue du combat. L'IA doit gérer une multitude de situations : choisir le bon Pokémon en fonction des Types, utiliser les bonnes attaques, gérer les ressources disponibles (objets, PV, etc.), et réagir en temps réel aux actions de l'adversaire. De plus, l'IA doit prendre en compte les particularités de chaque combat, telles que le changement de Pokémon en raison de désavantages de Type ou de points de vie faibles, ainsi que les événements aléatoires comme les états de paralyse, sommeil ou confusion.

Un autre défi réside dans la gestion des **événements aléatoires** durant les combats. Ces événements peuvent avoir un impact considérable sur les performances de l'IA, la forçant à ajuster ses stratégies en fonction des conditions imprévues. Par exemple, un coup critique ou un état altéré (paralysie, sommeil) peut rendre un combat plus difficile et nécessiter des décisions adaptatives de la part de l'IA.

En outre, il convient de noter qu'un projet similaire a été mené par Peter Whidden, qui a consacré **plusieurs années** et **plus de 1000 euros d'investissement** pour obtenir des résultats significatifs dans *Pokémon Rouge*. Bien que notre approche ne s'inspire pas directement de ses méthodes techniques, ce projet souligne la difficulté de la tâche et l'ampleur des efforts nécessaires pour atteindre des résultats notables dans ce domaine [10].

Au regard de ces défis techniques, notre objectif n'est pas de créer une IA capable de finir le jeu, mais plutôt d'explorer les aspects fondamentaux de l'intelligence artificielle dans un environnement de jeu complexe et de mieux comprendre les difficultés inhérentes à une telle tâche. L'IA que nous développons vise avant tout à s'initier aux défis d'**exploration**, de **prise de décision** et d'apprentissage dans un monde de jeu ouvert, tout en tenant compte des nombreux paramètres influençant la progression dans *Pokémon Rouge*.

1.4 Choix des outils et technologies

1.4.1 Langage de programmation

Pour ce projet de développement d'une IA Pokémon, nous avons choisi d'utiliser le langage de programmation Python. Python est largement reconnu pour sa simplicité et sa lisibilité, ce qui facilite le développement et la maintenance du code. De plus, Python est extrêmement populaire dans le domaine de l'intelligence artificielle et de l'apprentissage automatique, grâce à une vaste communauté et un écosystème riche en bibliothèques spécialisées.

Python est un langage interprété, ce qui signifie qu'il permet une exécution interactive et une détection rapide des erreurs. Cela est particulièrement utile dans le développement de projets complexes comme une IA de jeu, où des itérations rapides et des ajustements fréquents sont nécessaires. De plus, Python est multiplateforme, ce qui permet de développer et de tester le projet sur différents systèmes d'exploitation sans modifications majeures du code.

1.4.2 Bibliothèques utilisées

Stable Baselines Stable Baselines est une bibliothèque de renforcement de l'apprentissage qui fournit des implémentations fiables et bien documentées de divers algorithmes d'apprentissage par renforcement. Elle est particulièrement utile pour les projets nécessitant des agents intelligents capables d'apprendre à partir de l'interaction avec leur environnement. Stable Baselines offre une interface simple pour entraîner, évaluer et ajuster les modèles d'apprentissage par renforcement.

Les algorithmes disponibles dans Stable Baselines incluent DQN, PPO, A2C, TRPO, et SAC, parmi d'autres. Ces algorithmes sont largement utilisés dans la recherche et l'industrie pour résoudre des problèmes complexes d'apprentissage par renforcement. Par exemple, PPO (Proximal Policy Optimization) est connu pour sa stabilité et son efficacité dans les environnements à haute dimensionnalité [8].

Pygame Pygame est une bibliothèque Python utilisée pour développer des jeux vidéo. Elle offre des fonctionnalités pour gérer les graphiques, les sons et les interactions utilisateur. Dans le cadre de ce projet, Pygame est utilisé pour simuler l'environnement de jeu Pokémon, permettant à l'IA d'interagir avec le jeu de manière réaliste. Pygame facilite également la visualisation des actions de l'IA et l'évaluation de ses performances.

Pygame est basé sur la bibliothèque SDL (Simple DirectMedia Layer), qui est utilisée pour accéder au matériel graphique et sonore. Cela permet de créer des simulations de jeu fluides et interactives. Par exemple, Pygame peut être utilisé pour afficher les sprites des Pokémon, jouer des effets sonores lors des combats, et gérer les entrées de l'utilisateur pour contrôler le jeu.

NEAT (NeuroEvolution of Augmenting Topologies) NEAT est une méthode d'évolution des réseaux de neurones artificiels qui optimise à la fois la topologie et les poids des réseaux. Contrairement aux réseaux de neurones classiques, qui ont une structure fixe, NEAT commence avec des réseaux simples et les complexifie progressivement en ajoutant des neurones et des connexions. Cela permet de découvrir des architectures de réseaux optimales pour une tâche donnée.

Les concepts clés de NEAT incluent :

— **Gènes** : Les gènes représentent les neurones et les connexions dans le réseau. Chaque gène a des attributs tels que le poids de la connexion, l'activation du neurone, etc.

— **Topologie dynamique** : NEAT permet l'évolution de la structure du réseau, ajoutant ou supprimant des neurones et des connexions au fil du temps.

— **Innovation** : NEAT utilise un mécanisme d'innovation pour suivre les modifications apportées aux réseaux, facilitant ainsi la recombinaison des réseaux sans perturber les structures existantes.

NEAT se distingue des algorithmes d'apprentissage par renforcement traditionnels par sa capacité à évoluer non seulement les poids des connexions, mais aussi la structure même du réseau. Cela permet de trouver des solutions plus efficaces et adaptées à des problèmes complexes. Par exemple, NEAT a été utilisé avec succès pour résoudre des problèmes de contrôle de robots et de jeux vidéo [9].

1.4.3 Technologies d'émulation

Pour entraîner l'IA à jouer à Pokémon, nous utilisons des technologies d'émulation. L'émulation permet de recréer l'environnement de jeu original sur un ordinateur moderne, offrant ainsi un contrôle total sur le déroulement du jeu. Cela inclut la possibilité de modifier la vitesse du jeu, de sauvegarder et de charger des états de jeu, et d'accéder directement à la mémoire du jeu pour extraire des informations pertinentes.

Modification du déroulement temporel L'une des principales avantages de l'émulation est la capacité de modifier le déroulement temporel du jeu. En accélérant le jeu, nous pouvons effectuer des entraînements beaucoup plus rapidement que si nous devions attendre que le jeu se déroule en temps réel. Cela permet d'itérer rapidement sur les modèles d'IA et d'effectuer des ajustements en fonction des performances observées.

Par exemple, en utilisant un émulateur comme PyBoy, nous pouvons augmenter la vitesse d'exécution du jeu jusqu'à 1 000 000 fois la vitesse normale. Cela permet de simuler des milliers de combats Pokémon en quelques minutes, ce qui serait impossible à réaliser en temps réel.

Accès mémoire L'accès direct à la mémoire du jeu est une fonctionnalité cruciale pour ce projet. En accédant à la mémoire, nous pouvons extraire des informations précises sur l'état du jeu, telles que la position des personnages, les statistiques des Pokémon, et les actions disponibles. Cela permet à l'IA de prendre des décisions éclairées basées sur l'état actuel du jeu, améliorant ainsi ses performances.

Grâce aux travaux de reverse engineering réalisés par la communauté, notamment par des contributeurs comme ceux du projet [6], l'accès à la mémoire de Pokémon Rouge a été grandement facilité. Ces efforts ont permis de documenter et de comprendre en profondeur les structures de données et les mécanismes internes du jeu. En conséquence, il est désormais possible pour les développeurs et les chercheurs intéressés de manipuler directement la mémoire du jeu pour extraire des informations précises et interagir avec le jeu de manière plus efficace. Ces avancées ont ouvert de nouvelles perspectives pour la création d'outils et d'IA capables de jouer à Pokémon Rouge.

Par exemple, en accédant à la mémoire du jeu, nous pouvons déterminer les points de vie actuels des Pokémon, les attaques disponibles, et les effets de statut. Ces informations sont essentielles pour que l'IA puisse planifier ses actions de manière optimale et remporter les combats.

En résumé, le choix de Python et des bibliothèques spécialisées telles que Stable Baselines, Pygame et NEAT, combiné à l'utilisation de technologies d'émulation, offre une plateforme permettant de développer une IA capable de jouer à Pokémon. Ces outils et technologies permettent de créer un environnement d'entraînement et de tirer parti des avancées récentes en matière d'apprentissage automatique et d'évolution des réseaux de neurones.

2 Mise en œuvre

Dans cette section, nous détaillons la mise en place et le développement de notre intelligence artificielle pour jouer à Pokémon Rouge. Nous présentons les différentes étapes qui ont marqué notre travail, depuis la simulation du jeu jusqu'à l'interprétation des résultats obtenus. Nous débutons par la simulation du jeu, où nous décrivons l'environnement utilisé et les choix techniques. Ensuite, nous abordons la configuration de l'environnement de l'IA, en expliquant les outils et bibliothèques utilisés pour permettre à notre agent d'interagir avec le jeu. Afin de mieux comprendre les implémentations techniques évoquées, l'intégralité du code source est disponible sur un dépôt GitHub [1], dont le lien est fourni en bas de page. Nous définissons ensuite le champ d'observation de l'IA, un élément essentiel qui détermine quelles informations du jeu sont accessibles à notre modèle et comment elles sont exploitées. La section suivante, consacrée à la gestion des entrées et interactions avec le jeu, illustre comment l'IA prend des décisions et les applique sous forme de commandes valides dans l'environnement simulé. Nous décrivons ensuite la phase d'entraînement, où nous expliquons les approches d'apprentissage utilisées pour optimiser les performances de l'agent. Enfin, nous présentons les résultats et leur interprétation, afin d'évaluer l'efficacité de notre approche et d'en tirer des conclusions sur les forces et limites du système développé.

2.1 Simulation du jeu

La simulation de l'environnement de jeu est une composante essentielle du projet d'IA jouant à Pokémon Rouge. Cette simulation est réalisée à l'aide de la bibliothèque PyBoy, qui permet d'émuler le jeu Game Boy. Bien que cette partie ne constitue pas le cœur du projet en ce qui concerne l'IA, elle est intéressante afin de poser le contexte et définir les bases du décor avant d'aborder les aspects plus avancés de l'apprentissage et de la prise de décision. Cette section présente la mise en place de l'émulateur, l'intérêt de mettre en place un usage manuel avec le mapping des touches, ainsi que la gestion des entrées utilisateur.

1. https://github.com/xerneas02/AiPokemonRed

2.1.1 Mise en place de l'émulateur

L'émulateur est initialisé avec une ROM de Pokémon Rouge, qui constitue le fichier exécutable du jeu. Cette ROM est essentielle pour exécuter le jeu et interagir avec lui.

```python
from pyboy import PyBoy

ROM_PATH = "Rom/Pokemon Red.gb"
SHOW_DISPLAY = True

# Initialisation de l'émulateur
pyboy = PyBoy(ROM_PATH, window_type="SDL2" if SHOW_DISPLAY else "null")

# Définition de la vitesse de l'émulation
pyboy.set_emulation_speed(0 if SHOW_DISPLAY else 1_000_000)
```

La vitesse de l'émulation est ajustée en fonction du mode d'affichage. Si l'affichage est activé, le jeu s'exécute en temps réel, sinon il est accéléré pour optimiser l'entraînement de l'IA. Lorsqu'on a notre instance PyBoy, on peut choisir de charger un état de jeu par l'usage de `load_state(pyboy, chemin_vers_state)`.

2.1.2 Intérêt du mode manuel

L'ajout d'un mode de jeu manuel offre plusieurs avantages. Tout d'abord, il permet de vérifier que la ROM utilisée est bien l'originale et que l'émulation fonctionne correctement. Ensuite, en jouant manuellement, il est possible de choisir précisément un lieu et un moment du jeu pour sauvegarder un *state* (un état du jeu), afin d'obtenir des situations plus adaptées à l'entraînement de l'IA. Par ailleurs, en créant plusieurs *states*, il devient possible de diversifier les scénarios d'entraînement, ce qui pourrait améliorer la capacité générale de l'IA à s'adapter à différentes configurations.

2.1.3 Mapping des touches

Afin de permettre un contrôle manuel du jeu, un *mapping* des touches du clavier est défini :

```python
from pynput import keyboard

key_mapping = {
    'z': 'up',
    'q': 'left',
    's': 'down',
    'd': 'right',
    'a': 'a',
    'shift': 'b',
    'f': 'start',
    'e': 'select',
    'r': 'save',  # Sauvegarde de l'état du jeu
    'p': 'screen',  # Capture d'écran
    't': 'attack',  # Attaque en combat
}
```

Les touches du clavier sont associées aux boutons de la Game Boy afin de permettre une interaction avec le jeu.

Il est important de noter que le *mapping* de touche n'est utile que pour un usage manuel. Pour que cela serve à l'IA, c'est lors de la configuration de l'environnement que l'on définit les *inputs* de l'IA pour qu'elle puisse interagir avec le jeu. Ces *inputs* représentent des fonctions particulières que nous présenterons ultérieurement.

2.1.4 Gestion des entrées utilisateur

Un gestionnaire d'événements clavier est mis en place pour capturer les actions du joueur.

```python
def on_press(key):
    try:
        if key.char in key_mapping:
            keys_pressed.add(key_mapping[key.char])
    except AttributeError:
        if key == keyboard.Key.shift:
            keys_pressed.add('b')
```

```python
def on_release(key):
    try:
        if key.char in key_mapping:
            keys_pressed.discard(key_mapping[key.char])
    except AttributeError:
        if key == keyboard.Key.shift:
            keys_pressed.discard('b')
```

Ces fonctions capturent les touches pressées et relâchées pour simuler les interactions d'un joueur.

2.2 Configuration de l'environnement de l'IA

Dans ce projet, nous travaillons sur deux environnements distincts, chacun dédié à un aspect spécifique du jeu Pokémon Rouge. Le premier environnement est axé sur l'exploration et le déplacement sur la carte du jeu, utilisant les bibliothèques stable-baselines3 et gym. Le second environnement est consacré aux phases de combat du jeu, utilisant la bibliothèque NEAT. Bien que le vocabulaire diffère entre ces bibliothèques, l'idée d'un environnement d'IA reste similaire dans la forme. Chaque environnement configure son état initial, définit ce qu'il fait à chaque étape lors d'un entraînement, détermine son champ d'observation (les entrées, c'est-à-dire les valeurs acquises grâce à la consultation de la mémoire du jeu dans l'émulateur), et définit ses sorties (comment il interagit avec le jeu en fonction de la décision prise par le réseau de neurones ou le Génome dans le cas de NEAT). Avant de détailler l'environnement, il est important de comprendre les principales méthodes utilisées pour interagir avec celui-ci. Voici un aperçu des principales méthodes et de leur rôle dans l'environnement de l'IA :

— **‘__init__‘** : Cette méthode initialise l'environnement. Elle définit les espaces d'action et d'observation, ainsi que les variables d'état nécessaires pour l'environnement. C'est ici que l'état de départ et d'autres paramètres essentiels sont configurés.
— **‘reset‘** : La méthode `reset` réinitialise l'environnement à son état initial. Elle est appelée au début de chaque épisode pour garantir que l'environnement est dans une configuration cohérente avant l'interaction avec l'agent.
— **‘step‘** : La méthode `step` applique l'action de l'agent à l'environnement, puis renvoie la nouvelle observation, la récompense obtenue, une indication de fin d'épisode (booléenne), ainsi que des informations supplémentaires sous forme de dictionnaire.
— **‘_apply_action‘** : Cette méthode applique une action spécifique à l'environnement, modifiant ainsi son état. Elle définit la logique derrière chaque type d'action et son impact sur l'environnement.
— **‘_calculate_reward‘** : La méthode `_calculate_reward` calcule la récompense obtenue par l'agent pour une action donnée. Cela peut inclure des récompenses basées sur la progression de l'agent ou des pénalités pour des comportements non désirés.
— **‘_is_done‘** : Cette méthode détermine si l'épisode est terminé, soit en fonction du nombre de pas effectués, soit lorsqu'un objectif spécifique a été atteint par l'agent.
— **‘close‘** : La méthode `close` est utilisée pour nettoyer les ressources utilisées par l'environnement. Elle est appelée à la fin de l'entraînement ou lorsque l'environnement n'est plus nécessaire.

Pour le cas de NEAT, le principe est similaire, mais pour notre cas, seules les méthodes `step`, `reset` et les calculs de récompenses (appelés *Fitness*) sont pertinentes. Une configuration spécifique concernant les entrées et sorties de l'algorithme NEAT est nécessaire. Cela sera montré ultérieurement lors de la présentation des entraînements.

2.2.1 Définition du champ d'observation de l'IA

Pour définir le champ d'observation de l'IA, nous devons déterminer quelles informations du jeu seront accessibles à l'agent pour prendre des décisions. Dans le cas de l'exploration, cela inclut des informations telles que la position du joueur sur la carte, les niveaux des Pokémons dans l'équipe, et d'autres états pertinents du jeu. Pour les phases de combat, l'agent doit observer l'état du combat ; cela peut être les états des Pokémons actifs (alliés et ennemis), les mouvements disponibles, et les effets de statut.

Exploration Son environnement se fait via la librairie *gym*. Tout environnement est représenté par une classe que l'on code en Python. Au sein de cette classe, on y définit ses attributs, c'est-à-dire les informations pertinentes à retenir en mémoire lors du processus d'entraînement. Puis, on définit les différentes méthodes constituant un environnement pour une IA (2.2). Parmi ces attributs, on y retrouve ceux concernant les entrées qu'auront les réseaux de neurones, ce qui définit l'observation de l'IA. Voici l'initialisation de notre environnement `PokemonRedEnv` :

```python
from gym import spaces

def __init__(self, rom_path, show_display=False):
    super(PokemonRedEnv, self).__init__()

    # ... (Initialisation de l'environnement PyBoy)

    self.observation_space = spaces.Box(low=0, high=255, shape=(84, 84, 1), dtype=np.uint8)
    self.observation_space.dtype = np.uint8

    # ... (Autres attributs et configurations)
```

L'espace d'observation est configuré pour fournir à l'agent des images en niveaux de gris de taille 84x84 pixels, où chaque pixel peut avoir une valeur entre 0 et 255. Cela permet à l'agent de percevoir visuellement l'état actuel du jeu et de prendre des décisions basées sur ces observations.

```python
def reset(self):
        # Charger l'état de jeu de départ
        with open(get_random_state(), "rb") as state:
            self.pyboy.load_state(state)

        # Récupérer l'observation initiale
        observation = self._get_observation()
        self.steps = 0

        self.visited_positions.clear()

        # Reset action tracking
        self.last_action = None
        self.repeat_action_count = 0

        # Reset position tracking
        self.last_position = None
        self.steps_without_new_position = 0


        self.current_goal_index = {map_id: 0 for map_id in self.goals}

        x, y ,map_id = get_pos(self.pyboy)

        return observation
```

Dans cette méthode, on réinitialise les valeurs de l'environnement pour préparer l'agent à l'entraînement. On commence par charger un état de jeu aléatoire, ce qui permet de démarrer chaque session avec un environnement différent. Ensuite, l'observation initiale est récupérée pour offrir à l'agent une vue de l'environnement. Divers compteurs et mémoires sont également réinitialisés, comme le suivi des positions visitées et les actions précédentes, permettant à l'agent de repartir sur des bases propres à chaque épisode.

```
def _get_observation(self):
     # Prend un screenshot de l'écran actuel et le prétraite pour qu'il
     soit compatible avec ton cnn
    frame = self.pyboy.screen.ndarray
    return preprocess_frame(frame)
```

```
def preprocess_frame(frame, levels=4):
    # Convertir l'image en niveaux de gris
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

    # Redimensionner l'image à (84, 84)
    resized_frame = cv2.resize(gray_frame, (ImageSize, ImageSize),
    interpolation=cv2.INTER_AREA)

    # Normaliser l'image entre 0 et 1
    normalized_frame = resized_frame / 255.0

    # Quantifier l'image en niveaux de gris
    quantized_frame = np.floor(normalized_frame * levels) / levels  #
    Divise l'image en 'levels' nuances

    # Redimensionner la forme pour correspondre à l'entrée du CNN
    preprocessed_frame = quantized_frame.reshape(ImageSize, ImageSize,
    1)

    return preprocessed_frame
```

Cette fonction `preprocess_frame` prépare les images du jeu avant de les utiliser dans l'algorithme d'apprentissage. Elle commence par convertir l'image en niveaux de gris pour simplifier les données, puis la redimensionne en (84, 84) pour une taille uniforme. Ensuite, elle normalise les pixels entre 0 et 1 et réduit le nombre de nuances de gris pour limiter la complexité des entrées. Enfin, elle reformate l'image pour être compatible avec le modèle d'apprentissage.

Puis, on doit pouvoir gérer l'espace d'observation de l'IA lors de l'entraînement, pas à pas (à chaque *step*). Cela se configure dans la méthode `step`, appliquée à chaque "pas temporel" de l'entraînement.

```python
def step(self, action):
    # Convertir l'action en une action de bouton sur PyBoy
    self._apply_action(action)

    # Avancer d'un tick dans PyBoy
    self.pyboy.tick()

    # Obtenir l'observation suivante
    observation = self._get_observation()

    self.steps += 1

    # Vérifier si l'agent a découvert une nouvelle position
    x, y, map_id = get_pos(self.pyboy)
    current_position = (x, y, map_id)
    if current_position not in self.visited_positions:
        #self.visited_positions.add(current_position)
        self.steps_without_new_position = 0
    else:
        self.steps_without_new_position += 1

    # Calculer la récompense
    reward = self._calculate_reward(action)

    # Vérifier si l'épisode est terminé
    done = self._is_done()

    return observation, reward, done, {}
```

À chaque étape (`step`), l'agent récupère une nouvelle observation de l'environnement grâce à la fonction `_get_observation()`. Cette observation représente l'état actuel du jeu et sera utilisée par l'agent pour prendre ses décisions. En parallèle, la position de l'agent est suivie à l'aide de `get_pos(self.pyboy)`, qui extrait les coordonnées (x, y) et l'identifiant de la carte (map_id). Si l'agent découvre une nouvelle position, un suivi est mis à jour pour refléter cette progression. Dans le cas contraire, un compteur `steps_without_new_position` est incrémenté. L'évaluation des récompenses et des conditions de fin d'épisode sera détaillée ultérieurement.

Pour traiter efficacement ces observations visuelles complexes, le modèle d'apprentissage que nous avons utilisé est un réseau neuronal convolutionnel (CNN). Ce type de réseau est particulièrement adapté à la reconnaissance et à l'analyse d'images, ce qui lui permet d'extraire automatiquement les caractéristiques importantes des captures d'écran du jeu fournies en entrée. Grâce à ses couches convolutionnelles, le CNN est en mesure de reconnaître des motifs spatiaux, tels que le personnage, les murs, les portes ou autres éléments essentiels à la navigation dans Pokémon Rouge.

Combat Concernant l'environnement dédié au traitement des combats Pokémon de l'IA, on reprend les mêmes méthodes avec une classe ayant une structure similaire. On dispose de la classe `PokemonRedBattleEnv` :

```python
class PokemonRedBattleEnv:
    def __init__(self, rom_path, state_path='State/battle/',
    show_display=False, progress_counter=None):
        # Initialize PyBoy
        window_type = "SDL2" if show_display else "null"
        self.pyboy = PyBoy(rom_path, window=window_type)

        if show_display:
            self.pyboy.set_emulation_speed(0)
        else:
            self.pyboy.set_emulation_speed(1_000_000)


        self.progress_counter = progress_counter
        self.state_path = state_path
        self.state = None
        self.reset()
        self.steps = 0
        self.nb_battles = 0
        self.state_file = ""
        self.starting_party_hp = 0
```

Attributs de la classe `PokemonRedBattleEnv` :

— `pyboy` : Instance de l'émulateur PyBoy.
— `progress_counter` : Compteur de progression.
— `state_path` : Chemin vers les états de sauvegarde.
— `state` : État actuel de la bataille.
— `steps` : Nombre de pas effectués.
— `nb_battles` : Nombre de batailles effectuées.
— `state_file` : Nom du fichier d'état chargé.
— `starting_party_hp` : Points de vie initiaux de l'équipe.
— `total_reward` : Récompense totale accumulée.

Ensuite, on a besoin de pouvoir réinitialiser cet environnement, voici la méthode `reset` le permettant :

```python
def reset(self):
    self.state_file = load_random_state(self.pyboy, self.state_path)
    self.state = get_battle_state(self.pyboy)
    self.starting_party_hp = self.state[3] + self.state[42] +
    self.state[49] + self.state[56] + self.state[63] + self.state[70]
    self.total_reward = 0.0
    self.steps = 0
    return self.state
```

La méthode `reset` charge un état de bataille aléatoire, récupère l'état actuel de la bataille, et calcule les points de vie initiaux de l'équipe en fonction des Pokémon présents. Elle réinitialise également la récompense totale et le compteur de pas, puis retourne l'état de la bataille. Cela permet de préparer l'environnement pour un nouvel épisode d'entraînement.

La méthode `get_battle_state` extrait et retourne l'état actuel de la bataille Pokémon sous forme de tableau NumPy en récupérant des valeurs issues de la mémoire du jeu. Elle commence par obtenir les informations des Pokémon actifs et ennemis, puis récupère les statistiques de base de ces Pokémon. Ensuite, elle récupère les informations des mouvements du Pokémon actif et convertit les Types et effets des mouvements en identifiants numériques. La méthode calcule ensuite l'efficacité des mouvements contre le Pokémon ennemi. Elle récupère également les informations des Pokémon de l'équipe du joueur en vérifiant dans la mémoire du jeu. L'entièreté de la méthode est trouvable en annexe.

Enfin, la méthode crée un tableau NumPy représentant l'état de la bataille, incluant les informations sur le Pokémon actif du joueur et ses attaques, les membres de l'équipe du joueur, ainsi que le Pokémon actif adverse. Ce tableau regroupe des valeurs telles que les identifiants des Pokémon, leurs Types, leurs points de vie actuels et maximums, leurs niveaux et leurs statuts. Ces informations sont extraites directement de la mémoire du jeu, en s'appuyant sur des dictionnaires définis dans le fichier `Constante.py`, qui permettent notamment de déterminer les Types des Pokémon et des attaques, ainsi que la Table des types. La méthode retourne finalement cet état sous forme d'un vecteur de 79 valeurs, destiné à être utilisé par d'autres parties du programme.

TABLE 1 – Structure de l'état de combat retourné à l'IA (vecteur de 79 valeurs)

Index	Description
Pokémon actif du joueur	
0	ID du Pokémon
1	Type 1
2	Type 2
3	Points de vie actuels
4	Points de vie maximum
5	Niveau
6	Statut
Attaques du Pokémon actif (4 attaques × 8 infos)	
7–14	Attaque 1 : ID, efficacité, effet, puissance, Type, précision, PP max, PP actuel
15–22	Attaque 2 : ID, efficacité, effet, puissance, Type, précision, PP max, PP actuel
23–30	Attaque 3 : ID, efficacité, effet, puissance, Type, précision, PP max, PP actuel
31–38	Attaque 4 : ID, efficacité, effet, puissance, Type, précision, PP max, PP actuel
Autres Pokémon de l'équipe du joueur (5 × 7 infos)	
39–45	Pokémon 1 : ID, Type1, Type2, PV actuels, PV max, niveau, statut
46–52	Pokémon 2 : ID, Type1, Type2, PV actuels, PV max, niveau, statut
53–59	Pokémon 3 : ID, Type1, Type2, PV actuels, PV max, niveau, statut
60–66	Pokémon 4 : ID, Type1, Type2, PV actuels, PV max, niveau, statut
67–73	Pokémon 5 : ID, Type1, Type2, PV actuels, PV max, niveau, statut
Pokémon adverse	
74	ID du Pokémon
75	Type 1
76	Type 2
77	Points de vie actuels
78	Points de vie maximum

Concernant la méthode `step`, on récupère l'état du combat en tant qu'observation de l'IA via la méthode `get_battle_state` après que l'IA ait effectué une action. Avec cet état, on pourra évaluer les conséquences de cette action. Cela sera détaillé ultérieurement, dans la section dédiée aux interactions de l'IA avec le jeu.

2.2.2 Gestion des entrées et interactions avec le jeu

Dans cette section, nous aborderons la manière dont les entrées sont gérées et comment elles interagissent avec le jeu Pokémon Rouge via l'émulateur PyBoy. Nous expliquerons comment les actions sont traduites en commandes pour l'émulateur, permettant ainsi de simuler des interactions telles que les attaques, les changements de Pokémon, et d'autres actions de jeu. La gestion des entrées et des interactions est cruciale pour le bon fonctionnement de l'agent intelligent, lui permettant de prendre des décisions basées sur l'état actuel du jeu et d'interagir efficacement avec son environnement.

Un environnement est constitué d'un espace dédié aux actions que peut réaliser notre IA au sein de celui-ci. Dans notre cas, ces actions forment un espace discret, c'est-à-dire que l'IA effectue des actions distinctes et indépendantes les unes des autres. Au sein de cet espace, une valeur sera reliée à une succession d'*inputs* saisie dans l'émulateur PyBoy afin de simuler une action "humaine".

Exploration Premièrement, dans la classe `PokemonRedEnv`, on dispose d'un attribut

```
self.action_space = spaces.Discrete(6)
```

permettant de définir notre espace d'actions. Puis, l'action est traité dans la méthode `step` ayant en paramètre la valeur associé à la décision d'action prise par le réseau de neurone. C'est à nous de traduire cette valeur en une action concrète dans l'émulateur PyGame. Pour cela, nous disposons de la méthode `_apply_action`

```python
def step(self, action):
    # Convertir l'action en une action de bouton sur PyBoy
    self._apply_action(action)

    # Avancer d'un tick dans PyBoy
    self.pyboy.tick()

    # ... (Obtention des observations et gestion des récompenses)
```

```python
def _apply_action(self, action):
        actions = ["up", "down", "left", "right", "a", "b"]
        self.pyboy.button(actions[action])
```

Dans le cadre de l'exploration, on se contente uniquement de se déplacer de haut en bas, de gauche à droite, et on peut interagir ou annuler une action avec les boutons 'a' et 'b'. L'accès aux menus via les boutons 'start' et 'select' n'a pas été fourni à l'IA, car il n'est pas nécessaire à la progression dans le jeu et complexifie fortement les interactions. En effet, permettre l'ouverture des menus pourrait entraîner l'IA à s'y perdre ou à y rester bloquée. En contrepartie, cela signifie que certaines actions, comme soigner ses Pokémon avec des objets ou modifier l'ordre de l'équipe, ne seront pas possibles pour l'IA.

Combat Concernant l'environnement dédié aux combats, on traduit la valeur de l'action donné par le Génome (équivalent du réseau de neurone) par le biais de la méthode `play_action` utilisée dans la méthode `step` de l'environnement.

Fonctionnement de `play_action` La fonction `play_action` est responsable d'exécuter une action spécifique dans l'environnement de bataille Pokémon en utilisant l'émulateur PyBoy. Voici une explication détaillée de son fonctionnement :

1. **Déterminer les actions possibles** :

   ```python
   possible_inputs = input_possible(pyboy)
   ```

   Cette ligne appelle la fonction `input_possible` pour obtenir un dictionnaire indiquant si les actions `"attack"` (attaquer) et `"switch"` (changer de Pokémon) sont possibles dans l'état actuel du jeu.

2. **Exécuter une action d'attaque ou de changement de Pokémon** :

```python
if 0 <= action <= 3:
    if possible_inputs["attack"]:
        attack(pyboy, action)
    else:
        switch(pyboy, random.randint(0, 5))
elif 4 <= action <= 9:
    if possible_inputs["switch"]:
        switch(pyboy, action - 4)
    else:
        attack(pyboy, random.randint(0, 3))
```

— **Actions d'attaque (0 à 3)** :
  — Si l'action est comprise entre 0 et 3, la fonction vérifie si l'état actuel permet d'attaquer (`possible_inputs["attack"]`).
  — Si l'attaque est possible, la fonction appelle `attack` avec l'index de l'action.
  — Sinon, elle appelle la fonction `switch` pour changer de Pokémon de manière aléatoire.
— **Actions de changement de Pokémon (4 à 9)** :
  — Si l'action est comprise entre 4 et 9, la fonction vérifie si l'état actuel permet de changer de Pokémon (`possible_inputs["switch"]`).
  — Si le changement est possible, elle appelle la fonction `switch` avec l'index ajusté (`action - 4`).
  — Sinon, elle appelle la fonction `attack` avec une attaque choisie aléatoirement.

Dans tous les cas, l'IA peut soit attaquer, soit changer de Pokémon.

— Une attaque est considérée comme possible si le Pokémon actif n'est pas K.O.
— Le changement de Pokémon est considéré comme possible si il existe au moins un autre Pokémon vivant dans l'équipe autre que le pokemon actif.

Le code complet des méthodes `attack` et `switch` est disponible dans le dépôt GitHub.

Voici l'explication des méthodes permettant de traduire la valeur donné par le Génome en une action `in-game` :

Méthode `attack`

La méthode `attack` exécute une attaque en utilisant l'attaque spécifiée par l'index (0 - 3). Elle commence par vérifier si l'attaque a des points de pouvoir (PP) disponibles. Si l'attaque spécifiée n'a pas de PP, elle passe à l'attaque suivante jusqu'à trouver un mouvement avec des PP disponibles (il existe au maximum 4 attaques par Pokémon). Ensuite, la méthode navigue dans le menu de combat pour sélectionner et exécuter l'attaque. Si aucune attaque n'a de PP disponibles, le jeu lance automatiquement l'attaque "Lutte" (Struggle), une attaque basique. Elle utilise des conditions basées sur les couleurs (ou détection de présence) des pixels noirs et blancs à des positions spécifiques de l'écran pour déterminer l'état actuel du menu et appuyer sur les boutons appropriés (comme `'a'` pour confirmer l'attaque ou les boutons directionnels pour naviguer dans le menu). La méthode continue de vérifier l'état du jeu jusqu'à ce que l'attaque soit exécutée avec succès.

Méthode `switch`

La méthode `switch` change le Pokémon actif en utilisant l'index spécifié (4 - 9). Elle vérifie d'abord si le Pokémon à l'index spécifié est vivant et n'est pas déjà le Pokémon actif. Si le Pokémon est déjà en bataille ou est évanoui, elle passe au Pokémon suivant. Ensuite, la méthode navigue dans le menu de combat pour sélectionner et changer de Pokémon. Comme pour la méthode `attack`, elle utilise des conditions basées sur les couleurs des pixels à des positions spécifiques de l'écran pour déterminer l'état actuel du menu et appuyer sur les boutons appropriés (comme `'a'` pour confirmer le changement ou `'up'` et `'down'` pour naviguer dans le menu). La méthode continue de vérifier l'état du jeu jusqu'à ce que le changement de Pokémon soit effectué avec succès.

Capture d'Écran et Détection de Pixels

Dans notre implémentation, l'état visuel du jeu est obtenu à l'aide de l'attribut `pyboy.screen.ndarray`, qui retourne directement une image de l'écran sous forme d'un tableau NumPy en niveaux de gris. Ce tableau est ensuite utilisé pour la détection de pixels à des positions spécifiques.

Afin de comprendre l'état du jeu pendant un combat Pokémon, plusieurs fonctions s'appuient sur l'analyse de ces pixels pour déterminer le menu actuellement affiché. Par exemple, une fonction vérifie que les pixels à certaines coordonnées correspondent aux teintes attendues pour identifier le menu d'attaque. De manière similaire, d'autres fonctions analysent les pixels pour détecter les menus de fuite, de sélection de Pokémon ou d'objets.

Cette approche permet à l'agent de déterminer dans quel menu il se trouve et de choisir une action adaptée. Toutefois, nous avons rencontré certaines difficultés liées aux animations de combat : en effet, celles-ci peuvent temporairement modifier les couleurs des pixels, rendant leur détection moins fiable durant ces périodes.

Voici une capture d'écran d'une phase de combat dans Pokémon, illustrant l'interface et les éléments clés du jeu. On y distingue notamment la flèche noire du menu, un repère essentiel pour notre IA afin de comprendre son état actuel et réagir en conséquence :

FIGURE 2 – *Pokémon Red Battle Phase*

FIGURE 3 – *Pixel vérifier pour trouver l'état actuel*

Après avoir défini les entrées et sorties de l'IA, il est maintenant essentiel de s'intéresser à la manière dont celle-ci apprend à jouer. L'entraînement constitue une étape clé du développement, car c'est à ce moment que le modèle ajuste ses paramètres pour améliorer ses performances. Que ce soit à travers l'optimisation d'un réseau de neurones classique ou l'évolution d'un ensemble de Génomes dans le cas de l'algorithme NEAT, cette phase vise à produire un agent capable de prendre des décisions optimales en fonction des situations rencontrées. Nous allons donc examiner les différentes stratégies de formation du modèle et leur application à notre IA.

2.3 Phase d'entraînement

Dans le cadre de l'apprentissage par renforcement, il est crucial de mettre en place un système de récompense afin de guider l'IA vers les comportements souhaités et d'éviter ceux qui sont contre-productifs. Ce système repose sur la définition de fonctions d'évaluation permettant d'analyser les conséquences des actions effectuées par l'agent. En attribuant un score à chaque action en fonction de son impact sur l'objectif global, on établit un mécanisme d'apprentissage qui pousse l'IA à privilégier les stratégies les plus efficaces. Ainsi, la conception d'une fonction de récompense adaptée est une étape déterminante pour assurer un entraînement pertinent et l'émergence d'un comportement optimal.

Cependant, l'un des défis majeurs réside dans le choix et l'ajustement des récompenses et des pénalités. Une mauvaise calibration peut conduire l'IA à adopter des comportements sous-optimaux, voire à exploiter involontairement des failles dans le système. Il ne s'agit plus simplement d'écrire du code, mais d'affiner progressivement les paramètres en interprétant les résultats des entraînements. Cette phase demande de la patience, car les améliorations ne viennent qu'en observant les performances de l'IA sur de longues sessions d'apprentissage et en ajustant les configurations en conséquence.

2.3.1 Système de récompense

Dans tout environnement impliquant une intelligence artificielle, la mise en place d'un système de récompense est essentielle. Ce dernier permet d'évaluer les conséquences des actions réalisées par l'IA et d'attribuer un score reflétant la pertinence de ses choix. En s'appuyant sur cette évaluation, il devient possible de mesurer les performances de l'agent tout au long de son entraînement et d'orienter son apprentissage vers des comportements plus efficaces.

Exploration Concernant l'environnement d'exploration, dans la classe PokemonRedEnv, plusieurs attributs sont utilisés pour suivre les positions visitées, les objectifs à atteindre, et les actions répétées. Voici une explication détaillée de ces attributs :

```python
def __init__(self, rom_path, show_display=False):
    super(PokemonRedEnv, self).__init__()

    # ... (Initialisation de PyGame)

    # .. (Initialisation des espaces d'observation et d'action)

    self.visited_positions = set()

    self.goals = {
```

```
        38: [(7, 1, 38)],
        37: [(2, 8, 37)]
    }
    self.coefficients = {
        38: 0.1,
        37: 1.0
    }
    self.current_goal_index = {map_id: 0 for map_id in self.goals}

    # Track the last action
    self.last_action = None
    self.repeat_action_count = 0

    # Track the last position and steps without discovering new
    positions
    self.last_position = None
    self.steps_without_new_position = 0
    self.max_steps_without_new_position = 100
```

visited_positions : Cet ensemble (`set`) stocke les positions déjà visitées par l'agent. Cela permet de suivre les nouvelles positions découvertes et d'éviter les répétitions inutiles.

goals : Ce dictionnaire contient les objectifs à atteindre pour chaque carte (`map_id`). Chaque objectif est représenté par une liste de tuples contenant les coordonnées (x, y) et l'identifiant de la carte (`map_id`). Par exemple, pour la carte 38, l'objectif est d'atteindre la position $(7, 1, 38)$.

coefficients : Ce dictionnaire associe un coefficient de récompense à chaque carte (`map_id`). Ces coefficients sont utilisés pour pondérer les récompenses en fonction de l'importance des objectifs sur chaque carte. Par exemple, la carte 38 a un coefficient de 0.1, tandis que la carte 37 a un coefficient de 1.0.

current_goal_index : Ce dictionnaire garde une trace de l'index de l'objectif actuel pour chaque carte (`map_id`). Cela permet de suivre la progression de l'agent vers les différents objectifs définis dans le dictionnaire `goals`.

last_action : Cet attribut stocke la dernière action effectuée par l'agent. Il est utilisé pour détecter les actions répétées et appliquer des pénalités si nécessaire.

repeat_action_count : Cet attribut compte le nombre de fois que l'agent a répété la même action consécutivement. Une pénalité est appliquée en fonction de ce compteur pour encourager l'exploration et éviter les comportements répétitifs.

last_position : Cet attribut stocke la dernière position de l'agent. Il est utilisé pour suivre les mouvements de l'agent et détecter les situations où l'agent reste bloqué sans découvrir de nouvelles positions.

steps_without_new_position : Cet attribut compte le nombre de pas effectués par l'agent sans découvrir de nouvelles positions. Si ce compteur dépasse un certain seuil donné par `max_steps_without_new_position`, une pénalité peut être appliquée pour encourager l'exploration.

max_steps_without_new_position : Cet attribut définit le nombre maximum de pas que l'agent peut effectuer sans découvrir de nouvelles positions avant qu'une pénalité ne soit appliquée. Dans cet exemple, la valeur est fixée à 100.

Ensuite, lors de l'entraînement, on calcule la récompense à attribuer à l'IA à chaque `step` en fonction de l'action réalisée. Pour ce faire, on dispose d'une méthode `_calculate_reward` prenant en paramètre l'action réalisée.

```python
def _calculate_reward(self, action):
    x, y, map_id = get_pos(self.pyboy)
    exploration_reward = self._calculate_exploration_reward(x, y,
    map_id)
    goal_reward = self._calculate_goal_reward(x, y, map_id)
    penalty_for_repeating_action =
    self._calculate_penalty_for_repeating_action(action)

    # Combine rewards with coefficients
    total_reward = (
        0.1  * exploration_reward +
        1.0  * goal_reward +
        0.0  * penalty_for_repeating_action +
        10.0 * (map_id == 0)  # Bonus for reaching the goal map
    )

    return total_reward
```

La méthode calcule la récompense totale pour une action donnée en fonction de plusieurs critères. Elle commence par récupérer la position actuelle de l'agent dans le jeu, incluant les coordonnées x, y ainsi que l'identifiant de la carte. Ensuite, elle calcule trois types de récompenses distinctes : la récompense d'exploration, la récompense d'objectif et la pénalité pour répétition d'action.

La récompense d'exploration est attribuée si l'agent découvre une nouvelle position. Pour cela, la méthode vérifie si la position actuelle $(x, y, \texttt{map\_id})$ existe déjà dans l'ensemble (`set`) `visited_positions`. Si la position est nouvelle, une récompense est attribuée et la position est ajoutée à l'ensemble des positions visitées. Si l'agent atteint une position déjà visitée, aucune récompense n'est donnée.

La récompense d'objectif est déterminée par la proximité de l'agent par rapport aux objectifs définis pour chaque carte. Chaque carte a une liste d'objectifs sous forme de coordonnées $(x, y, \texttt{map\_id})$. Si l'agent atteint un objectif, une récompense maximale est attribuée. Sinon, la récompense est proportionnelle à la distance par rapport à l'objectif. Si l'agent est sur une carte sans objectifs restants, une pénalité est appliquée.

La pénalité pour répétition d'action est appliquée si l'agent répète la même action consécutivement. La méthode compare l'action actuelle avec la dernière action effectuée (`last_action`). Si l'action est répétée, un compteur (`repeat_action_count`) est incrémenté, et une pénalité croissante est appliquée pour encourager l'exploration et éviter les comportements répétitifs.

La méthode combine ensuite ces récompenses et pénalités en utilisant des coefficients spécifiques : 0.1 pour la récompense d'exploration, 1.0 pour la récompense d'objectif et 0.0 pour la pénalité de répétition d'action. Un bonus supplémentaire de 10.0 est attribué si l'agent atteint la carte cible. La récompense totale est ensuite retournée pour être utilisée dans la mise à jour de l'état de l'agent.

Ces coefficients ne sont pas fixes et doivent être ajustés en fonction des observations faites après plusieurs entraînements. En effet, le comportement de l'IA évolue en fonction des récompenses attribuées, et un mauvais équilibrage peut conduire à des stratégies sous-optimales, voire contre-productives. Ces valeurs sont définies manuellement en fonction de l'importance accordée à chaque catégorie de récompense. Ainsi, un ajustement progressif est nécessaire pour guider efficacement l'apprentissage de l'agent et assurer un comportement optimal dans l'environnement du jeu.

Le détail des fonctions

`self._calculate_exploration_reward`,
`self._calculate_goal_reward`
et `self._calculate_penalty_for_repeating_action`

est disponible dans le fichier `PokemonRedEnv.py`. Nous invitons les intéressés à consulter notre dépôt GitHub pour en savoir plus et explorer leur implémentation.

Combat Avec l'algorithme NEAT, les récompenses sont qualifiées par le terme de "*Fitness*". Sans détailler l'algorithme de NEAT (cela sera fait lors de la présentation du déroulement des entraînements), on traite une population de Génomes évoluant au fur et à mesure que l'on itère dans l'entraînement. Ce qu'on appelle *Fitness* représente le score réalisé par un Génome lors d'une itération de l'entraînement, après d'avoir réalisé une action. Cela se met en œuvre en réalisant une méthode `Fitness` dans l'environnement.

```python
def fitness(self, genome, config):
    # Implement the fitness function for neat-python
    net = neat.nn.FeedForwardNetwork.create(genome, config)
    total_fitness = 0.0
    for battle_num in range(BATTLE_PER_GENOM):
        state_file = self.reset()
        print(f"Start - Genome {genome.key} - Battle {battle_num +
        1}/{BATTLE_PER_GENOM} - Loaded state: {self.state_file}")
        observation = self.state
        fitness = 0.0
        done = False
        while not done:
            action = net.activate(observation)
            action = max(action, key=abs)
            observation, reward, done, _ = self.step(int(action))
            fitness += reward
        fitness += self._end_reward()
        total_fitness += fitness
    return total_fitness / BATTLE_PER_GENOM
```

La méthode est utilisée pour évaluer la performance d'un Génome dans le cadre de l'algorithme d'apprentissage NEAT. Cette fonction implémente la fonction de Fitness pour *neat-python*, en utilisant un réseau neuronal *feedforward* créé à partir du Génome et de la configuration fournie.

La méthode commence par initialiser le réseau neuronal et une variable pour accumuler la Fitness totale. Pour chaque bataille, elle réinitialise l'état du jeu en chargeant un état aléatoire et initialise les variables d'observation et de Fitness pour la bataille en cours. Ensuite, elle entre dans une boucle où elle exécute des actions déterminées par le réseau neuronal jusqu'à ce que la bataille soit terminée. À chaque étape, l'observation actuelle est passée au réseau neuronal pour obtenir une action, qui est ensuite exécutée dans l'environnement. La récompense obtenue pour cette action est ajoutée à la Fitness de la bataille.

Une fois la bataille terminée, une récompense supplémentaire est ajoutée pour évaluer la performance globale de la bataille, et la Fitness totale est mise à jour. Ce processus est répété pour un nombre prédéfini de batailles, et la Fitness moyenne sur toutes les batailles est calculée et retournée.

Cette méthode permet d'évaluer la capacité d'un Génome à prendre des décisions efficaces dans l'environnement de bataille Pokémon, en utilisant les récompenses obtenues pour guider l'évolution des réseaux neuronaux dans l'algorithme NEAT.

La méthode `Fitness` décrite précédemment s'appuie sur plusieurs types de récompenses pour guider l'apprentissage et évaluer l'efficacité d'un Génome dans l'environnement de combat Pokémon. Ces récompenses sont divisées en deux catégories principales :

- **Récompenses immédiates :**
  - **Issue du combat** : une victoire rapporte $+1.0$, une défaite -1.0, et un combat dépassant la limite de temps (500 étapes) est pénalisé par une récompense négative de -0.5.
  - **Efficacité des attaques** : chaque attaque effectuée génère une récompense proportionnelle à son efficacité selon la formule :

$$\text{récompense attaque} = 0.1 \times \frac{\log_2(\text{efficacité de l'attaque})}{2}$$

    Ainsi, une attaque doublement super efficace génère une récompense de $+0.1$, tandis qu'une attaque doublement résistée pénalise de -0.1.
- **Récompenses finales (en fin de combat) :**
  - **Points de vie restants (HP)** : à la fin d'un combat, la récompense dépend du ratio des points de vie restants dans l'équipe du joueur par rapport à leur état initial. Cette récompense est calculée comme :

$$\text{récompense HP} = 0.5 \times (2 \times \text{ratio HP restant} - 1)$$

    Elle varie donc entre -0.5 (tous les Pokémon K.O.) et $+0.5$ (aucun dégât reçu).
  - **Rapidité du combat** : une récompense est attribuée selon le nombre de tours pris pour conclure le combat. Plus le combat est rapide, plus la récompense est élevée. La formule utilisée est une fonction hyperbolique pour assurer une transition douce :

$$\text{récompense rapidité} = 0.5 \times \tanh\left(\frac{\text{tours max} - \text{tours réalisés}}{\text{facteur de douceur}}\right)$$

    Avec des valeurs typiques telles que tours max $= 100$ et facteur de douceur $= 10$, cette récompense est positive pour un nombre de tours faible et devient négative lorsque le combat dure trop longtemps.

Enfin, ces récompenses sont combinées pour former la Fitness finale d'un Génome, ce qui permet à l'algorithme NEAT d'évoluer en privilégiant des stratégies à la fois efficaces et rapides tout en minimisant les pertes en points de vie.

2.3.2 Déroulement des entraînements

Pour entraîner efficacement nos IA d'exploration et de combat, nous avons créé des ROM custom à partir du code source décompilé de Pokémon Rouge disponible sur GitHub [2][6]. Cette étape en glsassembler représentait une première expérience significative pour nous. Le code modifié permettant ces adaptations est accessible publiquement via notre dépôt GitHub [3].

IA d'exploration : L'entraînement s'est déroulé en plusieurs phases progressives. Initialement, l'agent était entraîné à se déplacer uniquement dans la maison du joueur, avec des états sauvegardés à divers endroits stratégiques, l'objectif initial étant de sortir de la maison. Progressivement, l'environnement d'entraînement a été étendu en intégrant la zone extérieure, obligeant l'agent à se rendre dans les hautes herbes pour déclencher l'événement avec le professeur Chen. Afin d'améliorer l'efficacité de l'entraînement, tous les dialogues des PNJ et les combats ont été désactivés. Une fois dans le laboratoire du professeur, l'objectif de l'IA consistait à sortir du laboratoire, puis à sortir de la ville pour finalement atteindre la ville suivante. À chaque nouvelle zone introduite, de nombreux états ont été sauvegardés pour permettre à l'IA d'apprendre à se localiser et à naviguer efficacement vers ses objectifs.

Au fur et à mesure de ces phases, il était essentiel de fournir à l'agent des états initiaux variés pour éviter le surapprentissage à partir d'une position spécifique et favoriser une généralisation efficace. Des états intermédiaires dans des endroits clés ont ainsi été capturés, par exemple à l'entrée des hautes herbes, à l'intérieur du laboratoire du professeur Chen, à la sortie de la ville initiale et à l'entrée de la ville suivante. Cette méthodologie a permis à l'IA d'exploration de développer progressivement une meilleure compréhension de la géographie et des caractéristiques distinctives de chaque zone, rendant ses déplacements de plus en plus efficaces et rapides.

Par ailleurs, pour assurer une progression stable de l'agent, nous avons implémenté plusieurs mécanismes d'observation de l'évolution de ses performances. Des statistiques sur la rapidité de l'atteinte des objectifs et la fréquence des comportements indésirables (tels que les déplacements inutiles ou les blocages répétés dans des zones précises) ont été enregistrées et analysées après chaque phase d'entraînement. Ces données ont permis d'affiner continuellement l'entraînement et d'adapter les états initiaux en fonction des difficultés rencontrées.

IA de combat : Pour cette IA, une autre ROM modifiée spécifiquement pour les combats a été créée. L'IA démarrait directement contre les membres de la Ligue Pokémon et le champion, équipée d'une équipe équilibrée de Pokémon de niveau adapté aux adversaires :

1. Dracaufeu niveau 55
2. Raichu niveau 53
3. Staross niveau 54
4. Rhinoféros niveau 56
5. Noidkoko niveau 55
6. Roucarnage niveau 54

2. https://github.com/pret/pokered/tree/master
3. https://github.com/xerneas02/pokered-master

Cette configuration rendait les combats réalisables mais non triviaux. Chaque Génome de l'algorithme NEAT était évalué à travers un affrontement contre chacun des membres de la ligue ainsi que contre le champion, afin de mesurer précisément sa capacité à gérer des stratégies complexes et variées.

Les états initiaux de ces combats ont été précisément sauvegardés au début de chaque affrontement afin de garantir une reproductibilité totale des évaluations. Pour chaque évaluation, le Génome testé devait affronter l'ensemble des membres de la Ligue Pokémon dans un ordre fixe, assurant ainsi que toutes les stratégies d'équipe et de type soient testées de manière exhaustive. Chaque bataille était évaluée individuellement, permettant une analyse détaillée des décisions prises par l'IA.

Afin d'optimiser les résultats de l'apprentissage, nous avons spécifiquement ajusté les récompenses liées à la performance durant les combats. Des indicateurs tels que l'efficacité des attaques utilisées, la conservation des points de vie, et le nombre de tours nécessaires pour remporter la victoire ont été intégrés dans la fonction de Fitness. Cette approche permettait à l'IA de développer des stratégies efficaces sur le plan tactique tout en favorisant une gestion optimale des ressources de l'équipe.

Enfin, l'évolution des performances des Génomes a été systématiquement suivie au fil des générations grâce à des outils de suivi et d'analyse mis en place spécifiquement pour cet entraînement. Ces outils ont permis de détecter rapidement les limites des stratégies adoptées et d'effectuer des ajustements réguliers sur la configuration de l'algorithme NEAT, optimisant ainsi progressivement les performances globales de l'IA dans les combats.

2.4 Résultats, interprétation et perspectives

À l'état d'avancement actuel du projet, nous disposons de deux modèles distincts : l'un dédié à l'exploration de l'environnement, et l'autre focalisé exclusivement sur les combats Pokémon. Cette séparation en différents modèles spécialisés permet une gestion simplifiée des informations pertinentes à chaque tâche, réduisant ainsi la complexité de l'entraînement. Cette approche facilite l'apprentissage, puisque chaque modèle traite des données plus ciblées, correspondant uniquement à la tâche qu'il doit accomplir.

Le modèle d'exploration est actuellement capable, en utilisant uniquement l'écran du jeu en nuances de gris comme entrée, de partir de la chambre initiale du joueur, de traverser la maison, puis de naviguer efficacement jusqu'à la première ville du jeu. Il parvient ainsi à réaliser des tâches complexes telles que l'identification de portes ou de passages et à différencier efficacement les environnements. Le choix de désactiver les dialogues et les combats dans la ROM d'exploration a contribué à accélérer considérablement l'apprentissage, en permettant au modèle de se concentrer exclusivement sur la reconnaissance visuelle des décors et le déplacement optimal vers un objectif.

La figure 4 montre l'évolution du reward moyen obtenu par l'IA d'exploration lors d'un entraînement dédié spécifiquement à sortir de la maison du joueur depuis n'importe quelle pièce ou position initiale.

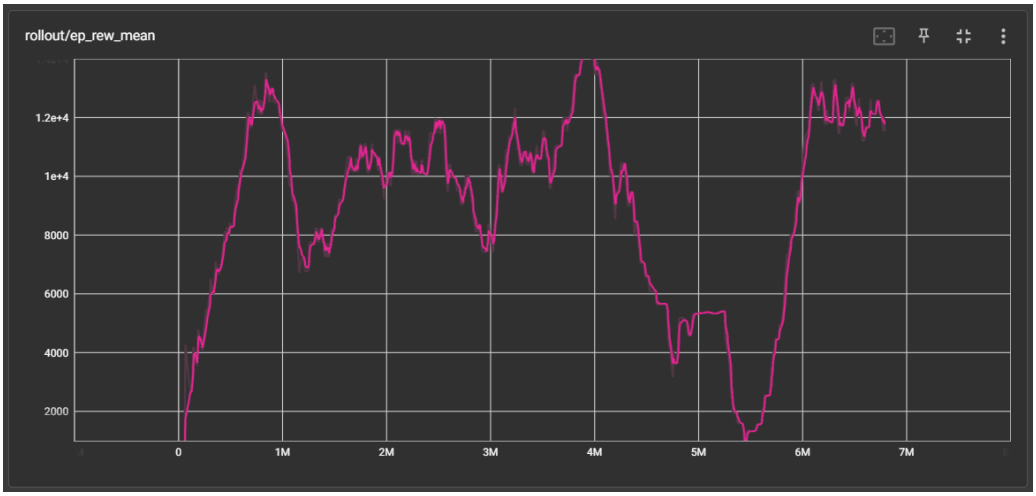

FIGURE 4 – Évolution du reward moyen de l'IA d'exploration lors de l'entraînement à la sortie de la maison.

Sur ce graphique, on observe une nette progression du reward durant les premières phases d'apprentissage, indiquant une rapide amélioration dans la capacité du modèle à identifier la sortie et à s'y diriger efficacement. Après une première période de stabilisation autour de 12 000 points, le modèle connaît une baisse temporaire des performances, possiblement liée à des stratégies exploratoires moins optimales découvertes durant l'apprentissage. Cependant, après environ 5 millions d'itérations, l'IA retrouve rapidement ses performances initiales élevées, démontrant ainsi sa robustesse et sa capacité à réapprendre rapidement après des périodes de performance moindre.

Cette courbe souligne à quel point le modèle est capable d'intégrer efficacement des informations visuelles complexes, validant notre choix d'utiliser uniquement les pixels de l'écran en nuances de gris comme entrées.

Le modèle de combat, quant à lui, présente également des résultats prometteurs. Évalué face à chacun des membres de la Ligue Pokémon ainsi qu'au Champion, le modèle obtient un taux de victoire global d'environ 81.6%. Il est important de noter que la majorité des défaites du modèle surviennent face au Champion, ce qui semble cohérent puisque ce dernier possède une équipe particulièrement équilibrée et de haut niveau. Ces résultats positifs démontrent que l'IA a acquis des stratégies suffisamment efficaces pour rivaliser avec des adversaires complexes malgré le temps limité d'entraînement disponible.

TABLE 2 – Performances du meilleur modèle contre les membres de la Ligue Pokémon (sur 100 combats)

Champion	**Taux de victoire (%)**
Lorelei	92 %
Bruno	94 %
Agatha	84 %
Lance	78 %
Blue (Maître)	60 %
Total moyen	**81,6 %**

Cependant, plusieurs limites importantes doivent être soulignées concernant ces résultats préliminaires. Le modèle de combat n'a été entraîné que sur une seule composition d'équipe et d'adversaire, ce qui restreint considérablement la variété des scénarios auxquels il a été confronté. Cet entraînement limité entraîne une spécialisation excessive du modèle, limitant potentiellement sa capacité à généraliser face à des compositions d'équipes différentes.

Quant au modèle d'exploration, bien qu'il ait réussi à atteindre la première ville, il n'a pas encore été confronté à la problématique du retour en arrière (*backtracking*). Cette capacité, consistant à revenir sur ses pas pour atteindre un objectif différent dans un environnement précédemment exploré, constitue pourtant une compétence essentielle dans un jeu tel que Pokémon Rouge, où les interactions avec l'environnement peuvent évoluer au fil de l'histoire. Implémenter cette compétence représenterait un défi supplémentaire particulièrement intéressant à explorer pour de futures phases d'entraînement.

Notre objectif initial, plus ambitieux, était d'explorer l'utilisation d'un réseau neuronal modulaire (*Modular Neural Network*), combinant plusieurs sous-modèles spécialisés dans des tâches spécifiques telles que l'exploration, les combats, et potentiellement d'autres sous-objectifs comme l'utilisation d'objets ou la gestion stratégique des Pokémon capturés.

Les réseaux neuronaux modulaires (MNN) constituent une approche intéressante en apprentissage automatique, permettant de diviser une tâche complexe en plusieurs sous-tâches distinctes et d'attribuer chaque sous-tâche à un module spécifique, entraîné indépendamment [2]. Un contrôleur central ou réseau d'aiguillage (*routing network*) se charge alors de sélectionner dynamiquement quel(s) sous-module(s) doivent être activés en fonction de l'état actuel de l'environnement ou de l'entrée fournie. Les avantages principaux des réseaux neuronaux modulaires incluent une réduction significative de la complexité computationnelle, une meilleure spécialisation des modules et une facilité accrue de généralisation par rapport à des modèles monolithiques classiques [5, 7]. Cette méthodologie a déjà démontré des résultats prometteurs dans divers domaines tels que la robotique, les jeux vidéo et la reconnaissance visuelle et textuelle [1, 4].

Malheureusement, en raison du temps limité disponible durant le projet, nous n'avons pu mettre en place et tester que deux modèles d'IA séparés (exploration et combat), sans pouvoir les intégrer dans une structure modulaire complète. Il serait particulièrement intéressant, dans le cadre de futures recherches, d'élargir cette approche modulaire pour bénéficier pleinement de ses avantages, notamment en évaluant la synergie potentielle entre différents modules spécialisés au sein d'une même partie complète de Pokémon Rouge.

3 Conclusion

Ce projet a représenté pour nous une aventure aussi ambitieuse qu'enrichissante, au carrefour de l'intelligence artificielle, du rétro-engineering et du développement logiciel. Notre objectif initial était de concevoir des intelligences artificielles capables de jouer à *Pokémon Rouge*, chacune spécialisée dans une tâche précise : l'exploration autonome de la carte et le combat stratégique contre des adversaires du jeu.

Si notre objectif semblait clair, sa réalisation s'est avérée bien plus complexe qu'initialement envisagée. La majeure partie de notre temps n'a pas été investie directement dans l'entraînement des IA, mais plutôt dans la compréhension approfondie et la modification du code source décompilé du jeu original. Ce travail en Assembleur, totalement inédit pour nous, a représenté une véritable épreuve technique, nécessitant de nombreuses heures d'étude et de documentation. Notre principal objectif était de créer des ROM personnalisées et optimisées spécifiquement pour faciliter l'entraînement des agents : en désactivant certains aspects non essentiels du jeu comme les dialogues et les combats pour l'agent d'exploration, ou en préparant une ROM avec un état initial précis avec une équipe équilibrée et un enchaînement direct des combats contre la Ligue Pokémon pour l'agent de combat. Ces modifications ont permis de rendre le processus d'apprentissage de nos modèles beaucoup plus efficace.

Grâce à ces efforts conséquents en amont, nous avons pu aboutir à deux modèles distincts : un modèle d'exploration capable de naviguer depuis la maison initiale du joueur jusqu'à la première ville, et un modèle de combat en mesure d'affronter la Ligue Pokémon avec un taux de victoire prometteur de 81.6%. Les résultats obtenus, bien que préliminaires, sont extrêmement encourageants au regard du temps limité consacré à la phase d'entraînement elle-même.

Cependant, il reste clair que nos modèles, malgré leurs succès initiaux, nécessitent davantage de tests et de diversification dans leurs scénarios d'entraînement. L'agent de combat, par exemple, manque de variété dans les équipes affrontées et l'agent d'exploration n'a pas encore eu à gérer des situations complexes de navigation impliquant du backtracking et des objectifs multiples.

À plus long terme, la perspective la plus fascinante serait d'aller au-delà des modèles individuels en implémentant un réseau neuronal modulaire (*Modular Neural Network*). Ce type de réseau permettrait à plusieurs agents spécialisés, entraînés indépendamment sur différentes tâches, de collaborer dynamiquement pour résoudre efficacement l'ensemble des défis proposés par le jeu. Une telle approche, documentée et validée par la littérature scientifique, pourrait ouvrir des portes vers des performances supérieures et une généralisation accrue de l'intelligence artificielle dans des environnements complexes et ouverts tels que Pokémon.

Ce projet nous aura donc apporté, au-delà des compétences techniques, une profonde compréhension des enjeux de l'intelligence artificielle appliquée aux jeux vidéo rétro, et nous laisse entrevoir des possibilités enthousiasmantes pour des développements futurs. Nous espérons ainsi pouvoir poursuivre cette exploration passionnante de l'IA appliquée, en affinant nos modèles et en relevant des défis toujours plus complexes.

Références

[1] Jacob Andreas et al. “Neural Module Networks”. In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), p. 39-48. doi : 10.1109/CVPR.2016.12.

[2] G. Auda et M. Kamel. *Modular Neural Networks : A Survey*. T. 9. 2. International Journal of Neural Systems, 1999, p. 129-151. doi : 10.1142/s0129065799000125.

[3] B. contributors. *Pokémon Red and Blue Versions*. Disponible : https://bulbapedia.bulbagarden.net/wiki/Pokmon_Red_and_Blue_Versions, consulté le 8 mars 2025.

[4] Chrisantha Fernando et al. “PathNet : Evolution Channels Gradient Descent in Super Neural Networks”. In : *arXiv preprint arXiv :1701.08734* (2017). url : https://arxiv.org/abs/1701.08734.

[5] Robert A. Jacobs et al. “Adaptive Mixtures of Local Experts”. In : *Neural Computation* 3.1 (1991), p. 79-87. doi : 10.1162/neco.1991.3.1.79.

[6] pret. *pokered*. Disponible : https://github.com/pret/pokered/blob/master/constants. consulté le 10 mars 2025.

[7] Clemens Rosenbaum, Tim Klinger et Matthew Riemer. “Routing Networks : Adaptive Selection of Non-linear Functions for Multi-task Learning”. In : *International Conference on Learning Representations (ICLR)*. 2018. url : https://openreview.net/forum?id=ry8dvM-R-.

[8] John Schulman et al. “Proximal Policy Optimization Algorithms”. In : *arXiv preprint arXiv :1707.06347* (2017).

[9] Kenneth O. Stanley et Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In : *Evolutionary Computation* 10.2 (2002), p. 99-127.

[10] Patrick Whiddy. *PokemonRedExperiments*. Disponible : https://github.com/PWhiddy/PokemonRedExperiments. consulté le 10 mars 2025.