Rapport de Stage M1 Informatique

2025

Simon Baudron

Développement d’une IA coopérative pour un jeu 2D à espace continu

Encadré par M. Sylvain Lamprier,

réalisé à l'Université d’Angers.

Sommaire

1. Remerciements

Je tiens tout d’abord à remercier chaleureusement mon encadrant, M. Sylvain Lamprier, pour ses précieux conseils et encouragements, et pour m’avoir permis de réaliser ce stage.

Je remercie également M. Benoît Da Mota, responsable du Master 1, pour sa disponibilité et son écoute tout au long de l'année, sans qui ce stage n’aurait pas pu avoir lieu.

Je souhaite aussi remercier le laboratoire LERIA de l’Université d’Angers pour son accueil et les conditions de travail mises à disposition.

Enfin, je remercie l’ensemble des enseignants du Master 1 Informatique ainsi que ceux de la licence pour leur encadrement et leur investissement tout au long de mon parcours à l'Université d’Angers.

2. Introduction

2.1 Introduction personnelle

Dans le cadre de ma première année de master informatique à l'Université d'Angers, j'ai eu l'opportunité d'effectuer un stage de recherche portant sur le développement d'une intelligence artificielle appliquée à un environnement de jeu vidéo.

L'objectif était d'explorer des techniques d'IA permettant d'améliorer le comportement d'agents autonomes dans un environnement interactif, tout en garantissant une expérience de jeu cohérente et engageante pour le joueur. Ce sujet mêle à la fois des problématiques liées à l'apprentissage automatique, à la planification, et à l'adaptation du comportement des agents dans un univers dynamique.

Ce stage m'a permis de m'initier à la méthodologie de recherche scientifique : revue de l'état de l'art, mise en œuvre expérimentale, et évaluation des résultats. Il m'a également donné l'occasion de contribuer à un projet complexe et concret, dans des domaines qui me passionnent particulièrement : le jeu vidéo et l'intelligence artificielle.

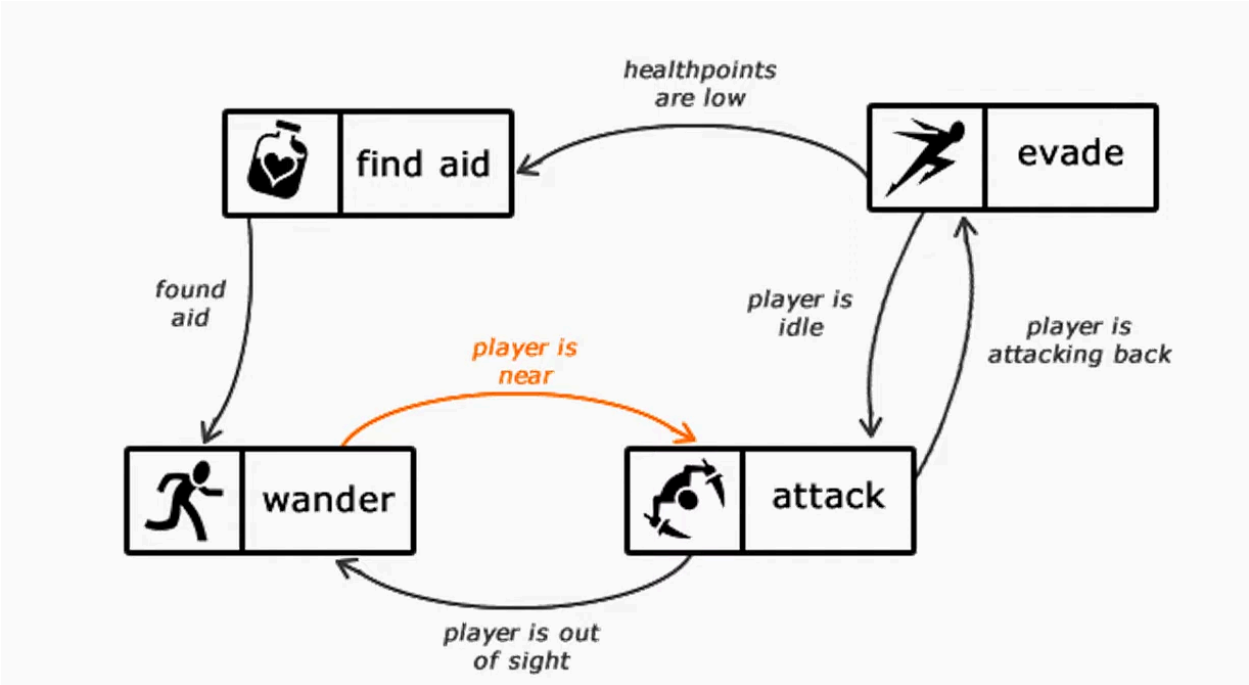
2.2 Problématique générale

L'intelligence artificielle dans les jeux vidéo n'est pas apparue récemment, elle existe en vérité depuis les débuts du jeu vidéo, principalement dans l'ajout d'ennemis tels que les fantômes dans Pac-Man, sorti en 1980, ou Donkey Kong dans le jeu du même nom sorti en 1981. Avec l'évolution de la complexité des jeux et des attentes des joueurs, il a fallu changer d'approche et recourir à des solutions permettant à des personnages non joueurs (PNJ) de s'adapter à des situations variées et d'agir différemment selon ces dernières.

2.3 Technologies existantes / utilisées

2.3.1 Final State Machines

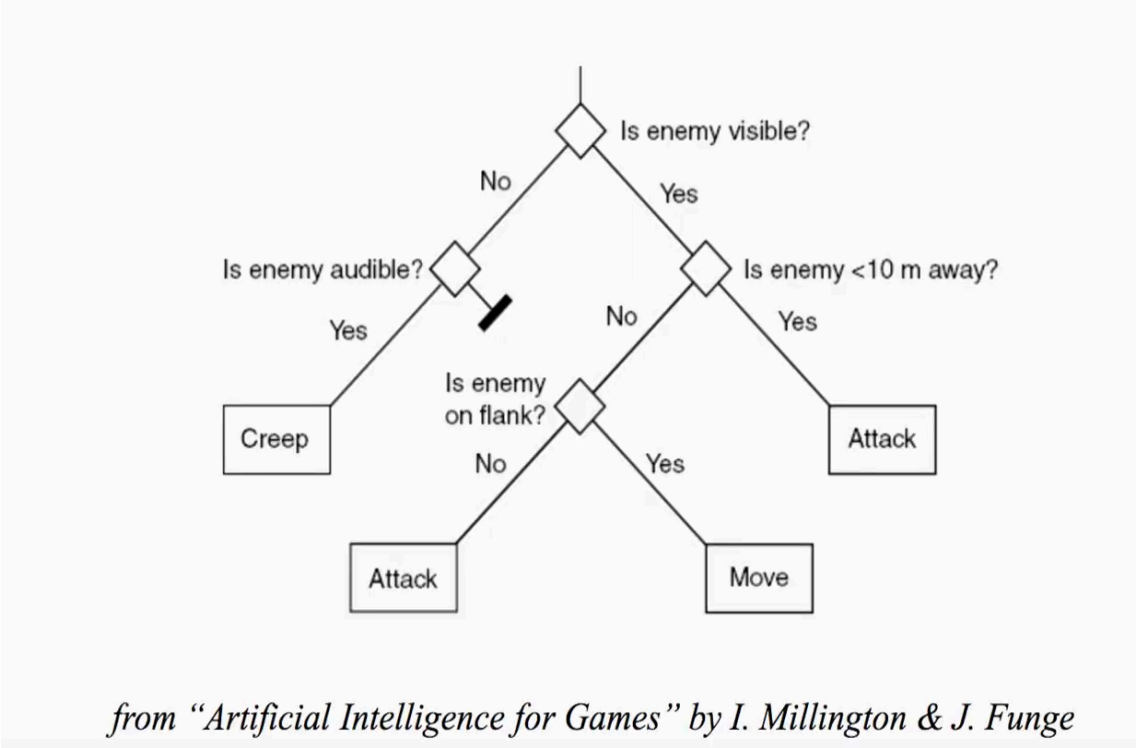
Les Finite State Machines (FSM) ont été utilisées pour permettre à des IA de changer de comportements (aussi appelés *states* ou état en français) comme par exemple attaquer un joueur, fuir, se cacher, etc. Les changements d'états sont provoqués par des *events* (événement) le nouvel état aura lui aussi des *events* qui vont permettre au PNJ de changer à nouveau d'état. Une des premières apparitions des FSM fut dans Half-Life, sorti en 1998, permettant aux ennemis de se comporter différemment en fonction de l'environnement dans lequel ils se trouvent. Par exemple, si un ennemi est dans un état d'attente, qu'un *event* coup de feu se produit, alors il passera dans un état défensif. Les FSM permettent de définir simplement des réactions à des changements d'environnement pour un PNJ avec très peu de puissance de calcul nécessaire, cependant le rendu est un comportement peu naturel dû à des décisions peu nuancées. Les FSM deviennent complexes à maintenir lorsque leur complexité augmente, du fait qu'il est nécessaire, pour chaque nouvel état, de définir toutes les transitions possibles, ce qui peut grandement nuire à la lisibilité à terme.



2.3.2 Arbres de Décisions

Une meilleure alternative aux FSM sont les Arbres de Décision, présents dans une majorité des jeux actuels (Far Cry Primal, Halo 3, Tom Clancy's The Division pour n'en citer que quelques-uns). Leur fonctionnement se découpe en plusieurs parties : la première (la racine de l'arbre) représentant le début de l'exécution, viennent ensuite les sélecteurs (les branches), ce sont des étapes intermédiaires permettant de savoir quel état exécuter en fonction d'informations sur le jeu, et ensuite les feuilles de l'arbre représentant les comportements devant être exécutés par le PNJ. Les feuilles peuvent être regroupées dans une séquence, ce qui va permettre aux PNJ de réaliser plusieurs tâches à la suite (par exemple se mettre à couvert puis se soigner). Une fois que le comportement du PNJ a été terminé, un nouveau parcours de l'arbre de décision sera exécuté.

Contrairement aux FSM, les Arbres de Décision permettent de définir des réactions plus précises plus facilement, de définir des séquences d'actions, et sont moins sujets à souffrir de leur grande taille, car chaque branche de l'arbre étant indépendante des autres, cela les rend plus faciles à déboguer.



from "Artificial Intelligence for Games" by I. Millington & J. Funge

2.3.3 Goal Oriented Action Planning

En 2002, l'équipe de FEAR (First Encounter Assault Recon) introduit pour la première fois dans un jeu vidéo l'Automated Planning, déjà présent dans l'industrie, permettant à une machine de déduire une liste d'actions permettant d'arriver à un objectif. L'implémentation de cette technologie dans les jeux vidéo sera nommée GOAP (Goal Oriented Action Planning). L'idée étant qu'un agent (le plus souvent un PNJ) possède une liste d'objectifs, qui sont divisibles en actions, une action pouvant elle-même nécessiter d'autres actions avant d'être réalisable.

Prenons par exemple un soldat voulant se soigner, si ce soldat n'a pas de kit de soins, il doit alors aller chercher un kit de soins avant de pouvoir réaliser la première tâche, la tâche principale étant maintenant d'aller récupérer un kit de soins.

Comme un agent peut posséder plusieurs objectifs, il est nécessaire d'attribuer une importance à chacun de ces objectifs, cette importance pouvant changer selon l'état dans lequel se trouve l'agent (eg. si le soldat possède tous ses points de vie, alors se soigner ne sera pas important). Comme un objectif peut parfois être satisfait en prenant différentes suites d'actions, il devient également nécessaire de préciser le coût d'une action (eg. est-ce qu'il est plus intéressant de se soigner avec le kit de soins que le soldat possède ou d'aller en chercher un à la base ?). Une fois qu'un objectif est divisé en plusieurs sous-ensembles d'actions, il suffit de choisir la liste d'actions ayant le coût cumulé le plus petit. Toutefois, si lors de l'accomplissement d'un objectif, un autre objectif devient plus important, il faudra alors changer d'objectif et tout recalculer. Cependant, si le temps est une métrique importante dans le jeu, il faudra alors simuler chaque suite d'action pour en déduire la plus intéressante, une méthode qui peut avoir un impact sur les performances du jeu. GOAP est une approche plus complexe et moins répandue que les Arbres de Décision, même si utilisée dans quelques grands titres tels que Just Cause 2, Deus Ex: Human Revolution, Tomb Raider (2013).

GOAP permet d'avoir des comportements ayant plus de sens, permettant une meilleure immersion, au prix d'une implémentation plus complexe et de performances moindres, principalement dues à la simulation d'actions.

2.4 Deep Learning dans les Jeux Vidéos

Après l'explosion du domaine dans les années 2012, avec AlexNet, un réseau de reconnaissance d'image, Alpha GO qui viendra triompher du champion du monde d'échecs en 2016, et plus récemment les IA génératives telles que Llama ou Chat-GPT, il ne serait pas surprenant de voir les PNJ de nos jeux vidéo contrôlés par des intelligences artificielles issues du Deep Learning ?

Des projets comme MineRL, ayant pour but de faire apprendre à un bot à progresser dans le jeu Minecraft à partir simplement de l'affichage du jeu à l'aide de l'apprentissage par renforcement. Les comportements restent encore bien loin de ceux d'un joueur humain, dû à la complexité de l'environnement, bien qu'il réussisse à faire des tâches primaires telles que récupérer des ressources faciles d'accès comme le bois ou la pierre. Un autre exemple plus concluant étant celui d'AlphaStar, développé par DeepMind, cette IA entraînée par apprentissage par renforcement et via l'imitation de parties réalisées par des humains, a réussi à vaincre des joueurs professionnels en 1 contre 1 sur le jeu StarCraft II. Encore plus impressionnant cette fois-ci : OpenAI Five contrôlant les 5 personnages d'une équipe dans le jeu Dota 2 réussira à l'emporter contre une équipe de professionnels en 2019.

Bien que ces prouesses reflètent une avancée notable, très peu de créateurs de jeux vidéo viennent à utiliser des PNJ contrôlés par des technologies issues du Deep Learning.

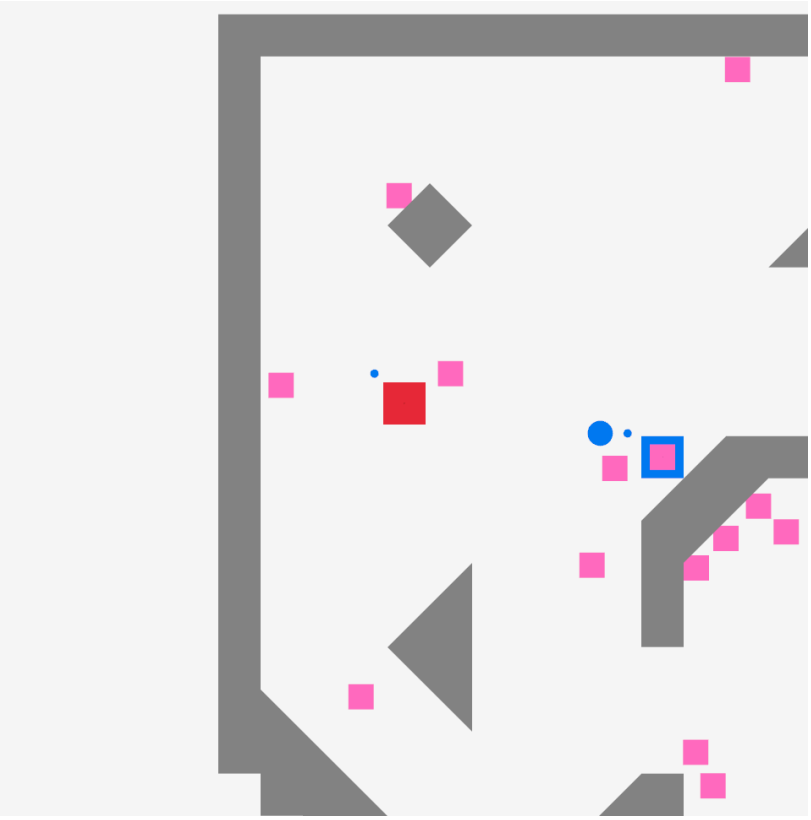
C'est pourquoi j'ai tenté de l'implémenter moi-même dans un jeu que je co-développe avec Léonard Gomez en parallèle de ma première année de Master 1, pour pouvoir mieux identifier quels sont les défis de l'implémentation du Deep Learning dans les PNJ d'un jeu vidéo.
Plus particulièrement, le développement d'une IA pouvant coopérer avec un joueur humain.

3. Outils utilisés

langage de programmation *C*, librairie de création de jeu (principalement dédiée à l'affichage) *Raylib*, *Git* pour le version control.

4. Amenagement et Developpement du jeu

Petite mise en contexte, l'idée du jeu est d'éliminer des monstres qui vous pourchassent, en leur tirant dessus à l'aide de la balle. Lorsqu'un joueur entre en collision avec un ennemi, il perd des points de vie. Quand ses points de vie atteignent 0, il meurt. Le jeu est perdu lorsqu'un des deux joueurs est éliminé, et gagné lorsque tous les ennemis sont éliminés. L'essence du jeu repose sur les passes entre coéquipiers, tout en essayant de toucher les ennemis sans perdre la balle. L'IA devra donc être capable de faire preuve de coopération, tout en ayant un comportement relativement naturel pour offrir la meilleure expérience de jeu possible.



Ci-dessus, une capture d'écran du jeu : en gris, les murs ; en rouge et bleu, les deux joueurs ; la balle est représentée par le disque situé entre les deux joueurs ; en rose, les ennemis. Les petits points bleus en haut à gauche des joueurs indiquent la direction dans laquelle ils visent.

Avant d'ajouter l'IA, il a fallu améliorer le jeu pour le rendre plus intéressant à jouer.

Trois points essentiels du gameplay ont été modifiés :

4.1 La Balle

L'idée était de donner une sensation de momentum au jeu via la balle.

- Premièrement, lorsque les joueurs effectuent des passes consécutives, la vitesse de la balle augmente (à noter que la vitesse est conservée entre chaque passe).

- Deuxièmement, si le joueur se fait des passes à lui-même (en utilisant les rebonds), alors la balle ralentit.

- Pour punir le joueur lorsqu'il perd la balle, si celle-ci rebondit plusieurs fois sans être attrapée, elle sera ralentie.

- Pour encourager l'utilisation du terrain, si la balle rebondit une seule fois, elle accélère.

- Pour donner un réel intérêt à faire accélérer la balle, celle-ci inflige désormais des dégâts proportionnels à sa vitesse.

- L'étape finale a été d'ajouter l'effet de repoussement des ennemis lors d'un contact avec la balle.

4.2 Les Ennemis

Le deuxième point majeur a été d'ajouter du dynamisme aux ennemis, qui étaient jusque-là statiques.

Désormais, lorsqu'un ennemi est proche d'un joueur, il se dirige vers lui (en ciblant toujours le joueur le plus proche). Si aucun joueur n'est suffisamment proche, l'ennemi se déplace alors de

manière aléatoire sur la carte. Pour améliorer la lisibilité, les ennemis ne peuvent plus se superposer complètement, ce qui renforce l'impression de "troupeau".

4.3 Les Collisions

Le troisième point concernait la gestion des collisions, en particulier celles avec les murs.

Initialement, les murs étaient représentés simplement par un ensemble de lignes, ce qui rendait leur ajout fastidieux lors de la création de nouvelles cartes, en plus d'être peu performant et sujet à des bugs.

Il a donc été choisi de réimplémenter complètement ce système.

La nouvelle approche consiste à diviser la carte en carreaux, chacun pouvant contenir un mur (avec plusieurs formes possibles) ou rester vide.

Cette structure permet non seulement de faciliter la conception des cartes, mais aussi d'optimiser les calculs de collision : seules les cases adjacentes à une entité (joueur ou balle) sont désormais prises en compte, au lieu de vérifier l'ensemble des murs de la carte.

Enfin, cette refonte avait également pour but de garantir que joueurs et ennemis ne puissent plus traverser les murs.

Il a ensuite fallu implémenter les conditions de victoire et de défaite pour que le jeu soit enfin prêt à être joué.

5. Choix de l'IA

5.1 Options Disponibles

Les deux options les plus populaires lors de la création d'un agent évoluant dans un environnement dans le domaine du Deep Learning sont l'apprentissage par imitation et l'apprentissage par renforcement.

L'apprentissage par imitation, comme son nom l'indique, se base sur le principe d'entraîner un agent à imiter le comportement d'un expert, ici un joueur. Cette méthode requiert un grand nombre d'expériences (ici des parties), cette méthode est donc impossible car le jeu étant encore en développement. À noter, ce problème serait également présent dans des plus grands studios de développement qui n'auraient pas non plus cette ressource avant d'avoir sorti le jeu.

La seconde option, l'apprentissage par renforcement, est une méthode dans laquelle un agent apprend par essais et erreurs en interagissant avec l'environnement du jeu. L'idée est de laisser faire l'agent des actions aléatoires et de le récompenser positivement ou négativement lors de son exploration de l'environnement lorsqu'on juge que cette action pousse à la victoire ou non, ce qui va le pousser à reproduire ou éviter certaines situations en fonction de ses expériences passées. Les récompenses doivent donc être choisies méticuleusement par le développeur afin de donner l'impression que l'agent essaye de gagner la partie et non de récolter le plus de récompenses possible. Une implémentation possible pourrait être de simplement donner une récompense à l'agent lorsqu'il gagne la partie. Cette approche fonctionne dans des environnements simples avec un nombre de possibilités relativement restreint tels que des jeux de plateau (comme Tic-Tac-Toe ou même des versions simplifiées d'échecs). Toutefois, dans un jeu vidéo complexe où les actions possibles sont nombreuses, les états du jeu très variés, et les parties longues, ce type de récompense pose un problème majeur : l'agent risque de ne jamais réussir à finir le jeu par hasard et ne saura pas forcément quelles actions précises ont conduit à la victoire, ce qui rend l'apprentissage extrêmement lent et instable.
Une meilleure approche est celle de récompenser l'agent tout au long de son exploration pour qu'il soit guidé vers la victoire. Un exemple ici serait de le récompenser lorsqu'il tue un ennemi, fait une passe ou récupère la balle, et évidemment lorsqu'il remporte la partie. Les récompenses négatives étant également possibles, on peut en profiter pour définir des situations que l'agent devrait éviter, comme tirer la balle à l'opposé de son allié ou bien perdre des points de vie.

5.2 Apprentissage par Renforcement

L'apprentissage par renforcement (Reinforcement Learning, ou RL) est un cadre d'apprentissage dans lequel un agent apprend à prendre des décisions par interaction avec un environnement. À chaque étape, l'agent observe un état s_t, choisit une action a_t, reçoit une récompense r_t, et se retrouve dans un nouvel état s_{t+1}. L'objectif est de maximiser la somme des récompenses futures attendues, appelée **récompense cumulée** ou **return**, souvent notée G_t qu'on peut calculer de cette manière :

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

où $\gamma \in [0, 1]$ est un facteur d'actualisation (souvent égale à 0.99) qui permet de donner plus de poids aux récompenses proches dans le temps. Cela reflète l'idée que les récompenses immédiates sont souvent préférées à celles qui arrivent plus tard.

5.3 Descente de Gradient

Pour que l'agent apprenne une bonne politique, il faut ajuster les paramètres du modèle (ici, les poids du réseau de neurones) de manière à améliorer ses décisions. C'est ici qu'intervient la **descente de gradient**, une méthode d'optimisation très utilisée en machine learning.

L'idée est de mesurer **l'erreur** entre ce que l'agent prévoit et ce qu'il aurait dû prévoir, puis on modifie légèrement les paramètres dans le sens qui réduit cette erreur.

Prenons un cas simple, comme l'algorithme **Q-learning**, où l'agent apprend la **valeur d'une action dans un état donné**, notée $Q(s, a)$. Si on observe une récompense r après une action, on peut mettre à jour notre estimation avec la règle :

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

Ici, α est le **taux d'apprentissage**, et la partie entre crochets représente l'erreur qu'on cherche à minimiser. Cette mise à jour est un exemple de descente de gradient sur la fonction de perte liée à la valeur Q.

5.4 Choix DDPG

Dans le cas de ce jeu, l'espace d'action n'est pas représenté par la probabilité d'un choix à effectuer, mais plutôt par des valeurs réelles telles que la direction en x, y, ou encore l'angle de tir — contrairement à un réseau de neurones qui essaierait de choisir si la meilleure action serait d'aller en haut, en bas, ou de tirer, etc.

C'est pourquoi j'ai choisi DDPG (Deep Deterministic Policy Gradient), un algorithme spécialement conçu pour gérer des environnements à actions continues. Il combine les idées du policy gradient (où l'on apprend directement la politique) et du Q-learning (où l'on apprend une fonction de valeur), en utilisant des réseaux de neurones pour estimer cette fonction.

5.5 Principe du DDPG

DDPG (Deep Deterministic Policy Gradient) est un algorithme de reinforcement learning particulièrement adapté aux environnements où les actions sont **continues** et non discrètes. Son fonctionnement repose sur deux concepts clés : l'apprentissage d'une politique déterministe et l'utilisation conjointe de deux réseaux de neurones, que l'on appelle **l'acteur** et le **critique**.

L'acteur est un réseau de neurones qui reçoit en entrée l'état courant du jeu (par exemple la position de l'agent, celle des adversaires, la balle, etc.) et qui produit en sortie directement les actions sous forme de valeurs réelles (comme l'angle de tir ou la direction). Contrairement à d'autres méthodes qui donnent une probabilité pour chaque action possible, l'acteur ici prédit une action précise et continue.

Le critique, quant à lui, est un autre réseau qui évalue la qualité de cette action dans cet état, c'est-à-dire qu'il estime la valeur $Q(s, a)$, c'est-à-dire la récompense cumulée que l'agent peut espérer obtenir en suivant cette action puis la politique optimale ensuite.

Le but de l'apprentissage est double :

- Le **critique** s'améliore en ajustant ses paramètres pour mieux prédire la valeur $Q(s, a)$ à partir des expériences passées, en minimisant l'erreur entre sa prédiction et la récompense réellement obtenue plus la valeur future estimée (comme dans le Q-learning classique).

- L'**acteur** est entraîné à ajuster ses paramètres pour proposer des actions qui maximisent la valeur estimée par le critique. En d'autres termes, il apprend à choisir les meilleures actions possibles selon le retour attendu.

L'apprentissage se fait par **descente de gradient** :

- Le critique minimise une fonction de perte qui représente la différence entre la valeur prédite et la valeur réelle (récompense observée + valeur estimée de l'état suivant).

- L'acteur maximise la fonction de valeur estimée en ajustant ses paramètres dans la direction du gradient qui augmente la récompense.

De plus, DDPG utilise une technique appelée **replay buffer**, qui consiste à stocker les expériences passées (état, action, récompense, état suivant) et à les réutiliser pour entraîner les réseaux. Le replay buffer agit comme une mémoire où sont conservés ces tuples (s, a, r, s'). Lors de l'entraînement, l'algorithme prélève au hasard un échantillon d'expériences dans cette mémoire plutôt que d'utiliser les données dans l'ordre chronologique. Cela permet de casser la dépendance temporelle entre les expériences successives, améliorant ainsi la stabilité et l'efficacité de l'apprentissage.

Enfin, en DDPG il est possible d'ajouter un mécanisme appelé **target networks** (réseaux cibles), qui sont des copies légèrement décalées dans le temps des réseaux acteur et critique. Plutôt que de mettre à jour directement les réseaux principaux à chaque étape, ces réseaux cibles sont mis à jour de façon progressive en faisant une moyenne pondérée des anciens et nouveaux poids. Lors du calcul des cibles pour la fonction de perte du critique, on utilise ces réseaux cibles, ce qui évite que les estimations évoluent trop rapidement, stabilisant ainsi l'entraînement et prévenant les oscillations. Cependant, je n'ai pas ajouté de target networks dans mon cas.

6. Ajout des éléments utiles à l'IA

6.1 Paramètres d’entrée

Avant d'implémenter l’IA il a fallu ajouter quelques éléments au jeu pour que l’agent puisse comprendre son environnement, ainsi que interagir avec.

La première étape a été d’implémenter la possibilité de faire jouer deux joueurs sur une même instance du jeu. Jusqu’à présent, le jeu ne permettait les affrontements qu’à travers une communication réseau, ce qui s’avérait contraignant. En effet, les communications réseau impliquent une gestion complexe des connexions, une latence non négligeable et une consommation accrue de ressources système, même en local. À l’inverse, regrouper les deux joueurs dans une même instance permet de centraliser les actions, d’éviter la surcharge liée au protocole réseau, et de simplifier grandement la simulation des parties, notamment dans le cadre d’un entraînement IA où des centaines voire milliers de parties doivent être lancées rapidement et efficacement.

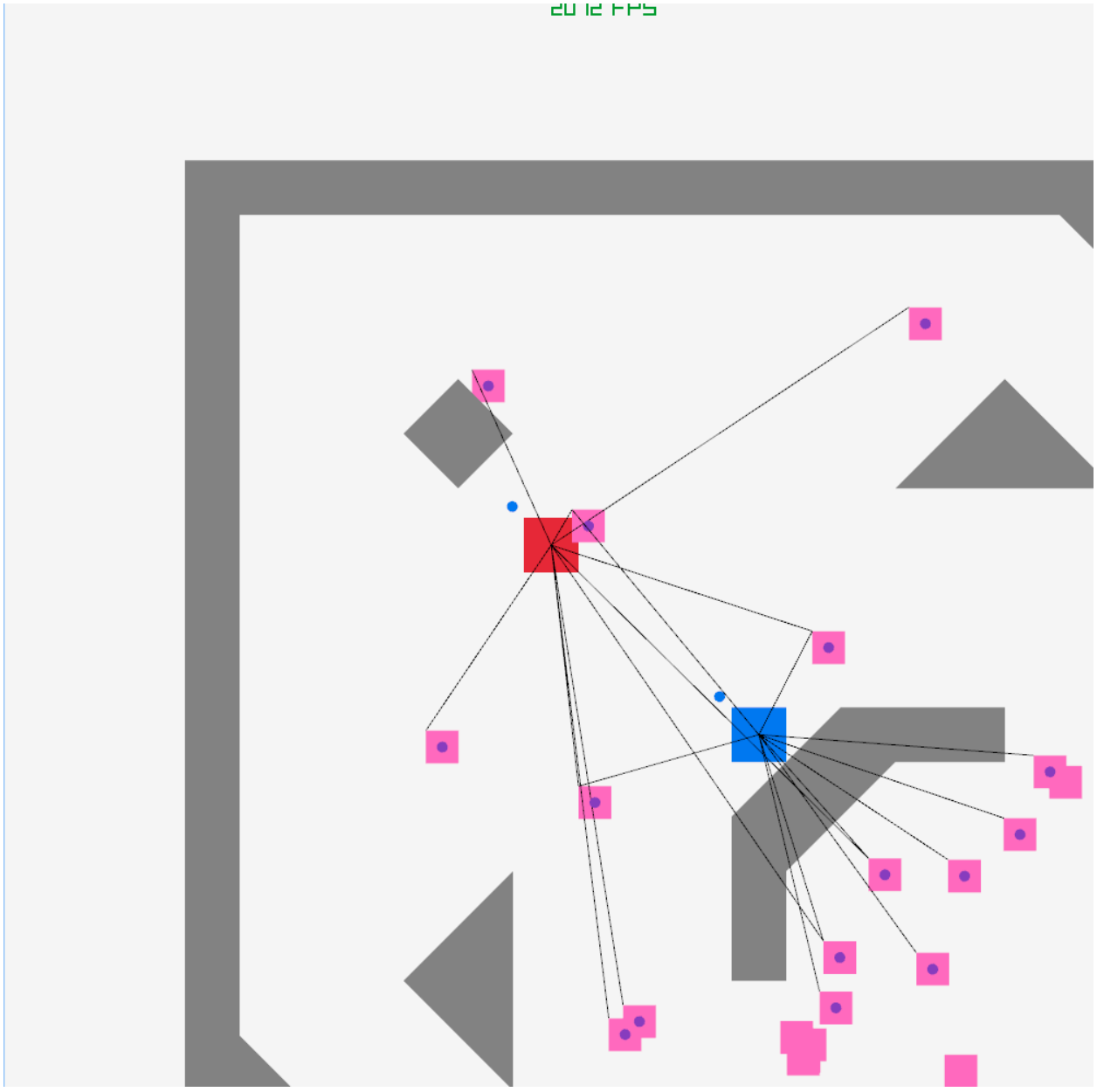
Ensuite il a fallu créer un moyen d’obtenir les features d'entrée qui allaient être passées au réseau de neurones (les informations que l’agent a sur son environnement). Le choix effectué a été celui de passer une représentation relative du jeu en fonction de l’agent et non une vision globale du jeu. Les informations concernant la position des entités du jeu sont alors passer par 3 paramètres : l’angle cosinus, sinus et la distance euclidienne entre l’agent et l'entité.

Un agent peut donc obtenir la position relative de :

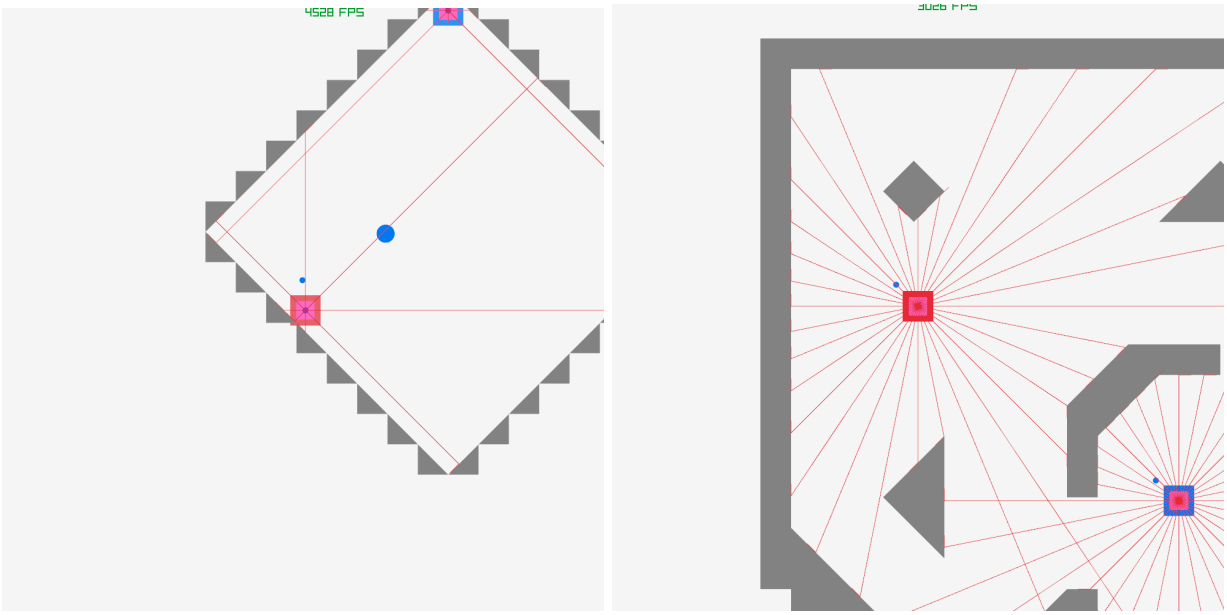
-la balle

-son allie

-les n ennemies les plus proches (n modifiable), ci dessous une représentation imagée de comment l’agent pourrait “percevoir” les ennemis.



-n murs autour du joueur obtenu par raycasting (n modifiable), a noté que en plus de la position relative d'un mur, l'agent possède aussi la normale du mur (son angle cosinus et sinus). Voir images ci dessous de calcul avec $n = 8 \, et \, n = 32$:



Et voici la liste des autres features qui sont passés en paramètres :

-vitesse de la balle $[0, 1]$

-direction (cosinus) de la balle $[-1, 1]$

-direction (sinus) de la balle $[-1, 1]$

-dégâts de la balle $[0, 1]$

-vitesse de l'agent $[0, 1]$

-points de vie l'agent $[0, 1]$

-boolean : si l'agent possede la balle $[-1, 1]$

-boolean : si l'agent a tiré la balle en dernier $[-1, 1]$

-vitesse du joueur allié $[0, 1]$

-points de vie du joueur allié $[0, 1]$

-boolean : si le joueur allié possède la balle $[-1, 1]$

-boolean : si le joueur allié a tiré la balle en dernier $[-1, 1]$

-pourcentage d'ennemis en vie $[0, 1]$

-boolean : si la partie est perdue $[-1, 1]$

-boolean : si la partie est gagnée $[-1, 1]$

À droite de chaque feature est présent l'intervalle de normalisation qui a été appliqué avant de le passer dans le futur réseau de neurones. En effet, il est important de normaliser toutes les valeurs pour éviter que certaines valeurs aient trop d'impact dans les calculs effectués au sein du réseau de neurones, notamment si elles ont naturellement tendance à être élevées (par exemple, la distance euclidienne). Les distances utilisées dans l'implémentation de la position relative mentionnée précédemment sont donc également normalisées (entre [0,1], 1 correspondant à la taille maximale possible dans la carte actuelle).

6.2 Valeurs de sortie

Comme explique precedemment, la sortie du réseau de neurones "actor" sera un ensemble de valeurs représentant chacune une action, voici les 5 valeurs de sorties choisi :

-vitesse x $[-1, 1]$

-vitesse y $[-1, 1]$

-angle de visée (cosinus) $[-1, 1]$

-angle de visée (sinus) $[-1, 1]$

-boolean : si l'agent veut tire la balle $[-1, 1]$

Une fois cela effectué, la dernière étape a été d'ajouter ces valeurs de sortie directement dans le programme du jeu pour permettre à l'agent de contrôler son personnage, le jeu possède désormais la possibilité de choisir de jouer en mode IA, ou en mode HUMAIN.

7. Implementation DDPG

7.1 Réseau de neurones

L'idée d'un réseau de neurones est la suivante : il est constitué de couches, chaque couche contient un ou plusieurs neurones, et chaque neurone d'une couche est lié à tous les neurones de la couche précédente. On distingue trois types de couches : la couche d'entrée, qui correspond aux caractéristiques (features) ; la couche de sortie, qui donne le résultat final ; et les couches intermédiaires (aussi appelées couches cachées) dont le rôle est de manipuler les données pour que le résultat présenté dans la couche de sortie soit le plus précis possible. Voici l'implémentation d'un neurone dans mon programme.

```
typedef struct neuron {
    float* weights;
    float bias;
    float value;
    float delta;
    ActivationType activation;
} neuron;
```

Explication des paramètres :

`float* weights;`

Les poids représentent l'importance des connexions entre les neurones. Chaque neurone a un poids par entrée. Ils sont ajustés pendant l'entraînement pour minimiser l'erreur du réseau.

$$z = \sum_{i=1}^{n} (w_i . x_i) + b$$

où :

- z est la somme pondérée + biais (avant activation)
- w_i est le poids de la connexion i
- x_i est l'entrée i
- b est le biais du neurone

`float bias;`

Le biais est une constante ajoutée à la somme pondérée des entrées. Il permet de décaler la sortie du neurone, ce qui améliore la capacité d'apprentissage du réseau, en particulier quand toutes les entrées sont nulles ou égales.

`float value;`

La **valeur** correspond à la sortie du neurone après application de la fonction d'activation. C'est cette valeur qui sera transmise aux neurones suivants dans le réseau.

$$a = f(z)$$

où :

- a est la valeur (le champ value),
- f est la fonction d'activation ,
- z est la somme pondérée avec biais.

`float delta;`

Le delta représente l'erreur du neurone utilisée pour mettre à jour les poids lors de la rétropropagation (backpropagation). Il dépend de la dérivée de l'erreur par rapport à l'entrée du neurone.

Formule (sortie du réseau) :

$$\delta = (a - y) \,.\, f'(z)$$

où :

- y : la vraie valeur cible,
- $f'(z)$: la dérivée de la fonction d'activation.

Formule (neurone présent dans les couches cachées) :

$$\delta = (\sum_{j} w_j . \delta_j) . f'(z)$$

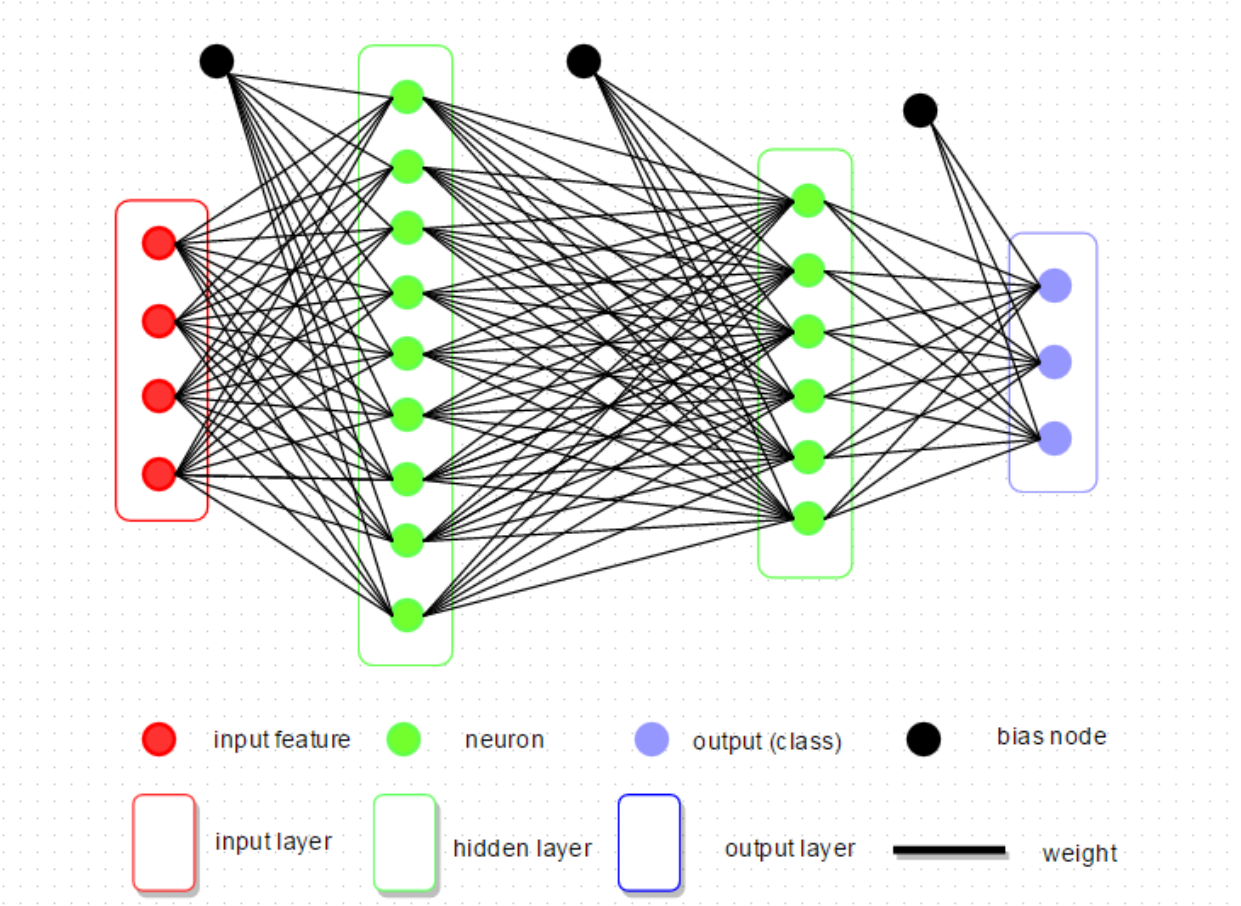
où la somme porte sur tous les neurones j connectés à ce neurone en sortie.

`ActivationType activation;`

Spécifie la fonction d'activation à appliquer au neurone. Cette fonction introduit une non-linéarité, ce qui permet au réseau d'apprendre des relations complexes.

Utilité de la fonction d'activation :

- Sans elle, le réseau serait simplement une combinaison linéaire des entrées, donc incapable de modéliser des fonctions complexes.



```
typedef struct layer{
    neuron* neurons;
    size_t size;
}layer;


typedef struct neural_network{
    layer* layers;
    size_t total_layers;
}neural_network;
```

Voici respectivement l'implémentation d'une couche et d'un réseau de neurones, qui sont simplement des listes de neurones (pour les couches) et de couches (pour les réseaux de neurones).

7.2 Fonctions

Maintenant que la structure d'un réseau de neurones est prête, il faut désormais définir les fonctions qui vont nous permettre d'utiliser et de faire apprendre un réseau de neurones. Voici un aperçu des fonctions les plus importantes au bon fonctionnement du DDPG.

7.2.1 Actor

```
float* forward_pass_actor(neural_network* net, float* state);
```

Cette fonction place l'état actuel du jeu dans la première couche du réseau de neurones, active les neurones comme vu précédemment, et renvoie un vecteur de valeurs correspondant aux 5 actions possibles (output).

```
void backward_pass_actor(neural_network* net, float* action_taken, float* grad_q_action);
```

Cette fonction calcule la rétropropagation pour mettre à jour le réseau de l'acteur en fonction du gradient de la Q-value reçu. Ici, grad_q_action représente le gradient du critique par rapport à l'action, utilisé pour améliorer la politique de l'acteur. Chaque neurone stocke le gradient obtenu dans le champ delta.

7.2.2 Critic

```
float forward_pass_critic(neural_network* critic, float* state, float* action);
```

Cette fonction fait passer l'état (state) et l'action (action) dans le réseau de neurones du Critic. Elle concatène ces deux vecteurs en entrée, active les neurones couche par couche, puis renvoie une seule valeur en sortie : la Q-value estimée (valeur attendue de la récompense).

```
void backward_pass_critic(neural_network* critic, float* state, float* action, float target, float learning_rate);
```

Cette fonction entraîne le réseau du Critic en comparant sa prédiction actuelle avec la valeur cible ($target = reward + \gamma . Q_{next}$). Elle calcule l'erreur, la propage en arrière dans le réseau pour remplir les deltas (gradients).

```
void backward_pass_critic_action_gradient(neural_network* critic, float* state, float* action, float* grad_action_out);
```

Cette fonction calcule le gradient de la Q-value par rapport aux actions en effectuant une rétropropagation partielle dans le réseau du Critic, avec une dérivée de sortie fixée à 1. Le résultat, stocké dans grad_action_out, indique à l'acteur comment ajuster ses actions pour maximiser la valeur estimée par le Critic.

7.2.3 Autres fonctions

```
void update_weights(neural_network* net, float learning_rate);
```

Cette fonction met à jour les poids du réseau en utilisant les gradients stockés dans delta, le learning rate permettant de contrôler la taille des ajustements afin d'éviter des changements trop brusques et d'assurer un apprentissage stable.

```
void add_exploration_noise(AI_output_action* action);
```

Ajoute du bruit à l'action générée pour favoriser l'exploration, ce qui permet à l'agent d'effectuer des actions différentes de celles initialement prévues, puis de comparer si ces actions sont meilleures ou non. La probabilité que ce bruit soit ajouté dépend du paramètre epsilon. Le bruit utilisé ici est un bruit gaussien de moyenne zéro.

7.3 Implémentation dans le jeu

Il suffit maintenant d'ajouter les fonctions dans le jeu. Deux parties majeures composent cette implémentation : la première est l'appel aux fonctions "forward_pass_actor" et "add_exploration_noise" pour chaque agent, qui se produit avant l'actualisation de l'environnement de jeu. On peut assimiler cette partie à l'agent qui choisit son action avant que le jeu ne se déroule.

La deuxième partie est l'apprentissage des réseaux Critic et Actor (dans cet ordre).

L'entraînement du Critic consiste à utiliser le buffer d'expérience vu précédemment. On sélectionne un échantillon aléatoire dans la mémoire, puis on estime la récompense future attendue (target) pour chaque expérience à l'aide de l'Actor et du Critic. Enfin, on ajuste les poids du Critic en comparant ses prédictions actuelles à ces valeurs cibles, afin d'améliorer sa précision à long terme.

Pour l'Actor, l'entraînement utilise également des expériences stockées dans le buffer. Pour chaque état contenu dans l'échantillon, on génère une action grâce au réseau de l'Actor. Ensuite, on demande au Critic d'évaluer cette action en calculant le gradient de la Q-value par rapport à celle-ci. Ce gradient est ensuite utilisé pour ajuster les poids du réseau de l'Actor, afin qu'il apprenne à produire des actions qui maximisent la valeur estimée par le Critic.

7.4 Note sur la modularité

Durant l'implémentation du jeu et du système d'apprentissage, la modularité a été un point d'attention important. En effet, la quasi-totalité des paramètres concernant le jeu sont facilement modifiables, tels que les paramètres de base des joueurs (vitesse de base, vitesse maximale, gain de vitesse, points de vie, taille), le comportement de la balle (vitesse de base, vitesse maximale, vitesse ajoutée à chaque passe ou rebond, etc.), les paramètres des ennemis, ainsi que ceux de la carte, qui sont sauvegardables et aisément modifiables.

Concernant les réseaux de neurones, il est également possible de changer le nombre de couches cachées et de neurones par couche, tout comme le nombre de murs et de monstres détectés (comme mentionné précédemment).

Un paramètre nommé `skipping_loops` a aussi été ajouté. Il permet de sauter un certain nombre de cycles d'actualisation de l'état du jeu. Étant donné que le jeu s'actualise 500 fois par seconde, calculer l'apprentissage des réseaux de neurones à cette fréquence devient extrêmement coûteux en ressources.

Ce paramètre permet donc de ne prendre en compte qu'un cycle sur N, ce qui améliore considérablement les performances.

De plus, cela s'avère bénéfique pour la qualité de l'apprentissage : en effet, si les états sont trop proches les uns des autres (car mis à jour trop fréquemment), le réseau de neurones peut avoir du mal à distinguer ce qui différencie un bon d'un mauvais état. En espaçant davantage les états observés, on augmente leur diversité, ce qui favorise un apprentissage plus pertinent.

7.5 Identifier les bons paramètres

Une fois l'implémentation terminée, il est désormais nécessaire d'identifier quels paramètres, parmi tous ceux modifiables, permettront d'obtenir les meilleurs résultats.

Voici un rappel des éléments ajustables : le taux d'apprentissage (learning rate), la fonction de récompense (reward), le nombre de couches cachées, le nombre de neurones par couche, ainsi que les paramètres utilisés dans la fonction de calcul du return() et ceux liés à l'ajout du bruit d'exploration(ϵ).

8. Resultats

Pour rappel, voici ce que l'on cherche à optimiser :

Pour l'Actor, on cherche à apprendre à choisir les meilleures actions possibles dans chaque situation.

On souhaite donc maximiser la Q-value qui sera estimée par le Critic.

$$max\ Q(s,\ \mu(s))$$

s : l'état du jeu (ce que l'agent voit)

$\mu(s)$: l'action proposée par l'Actor

$Q(s,\ \mu(s))$: ce que pense le Critic de cette action

Pour le Critic, on souhaite estimer les Q-value les plus proches de la réalité. On cherche donc à minimiser l'écart entre la Q-value prédite et la target.

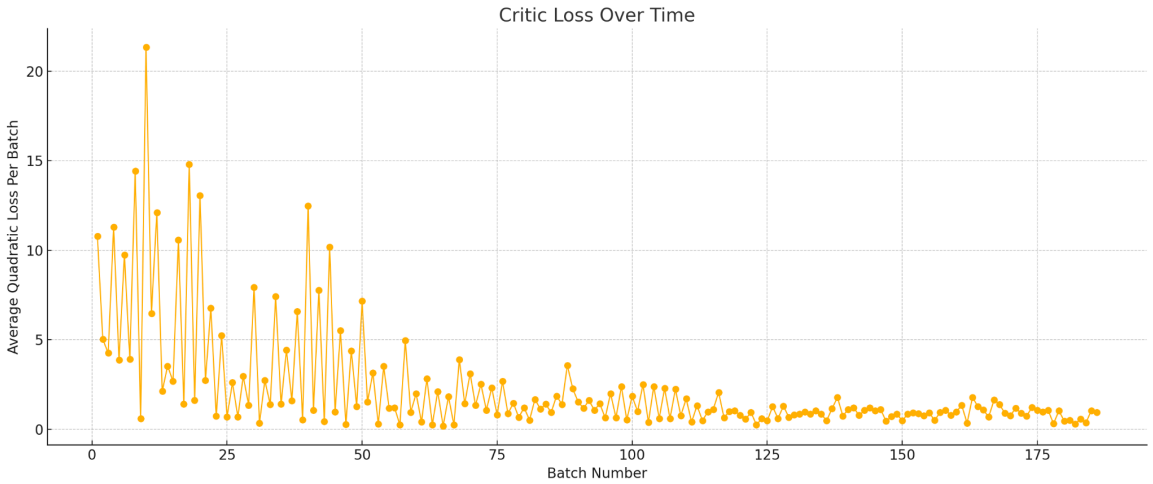
$$min\ (Q(s,a)\ -\ [r\ +\ \gamma.\ Q(s',\mu(s'))])^2$$

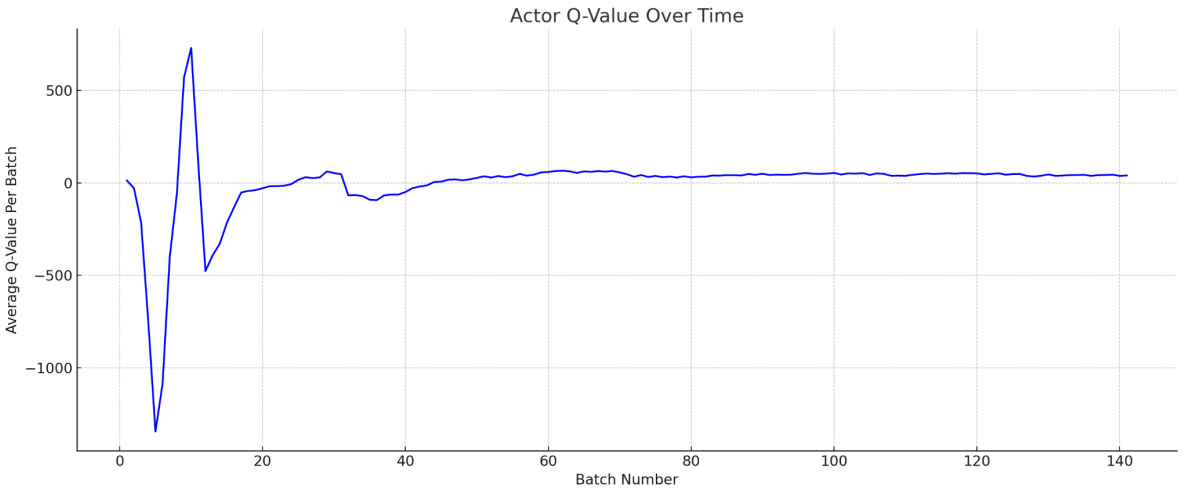
r: la récompense obtenue

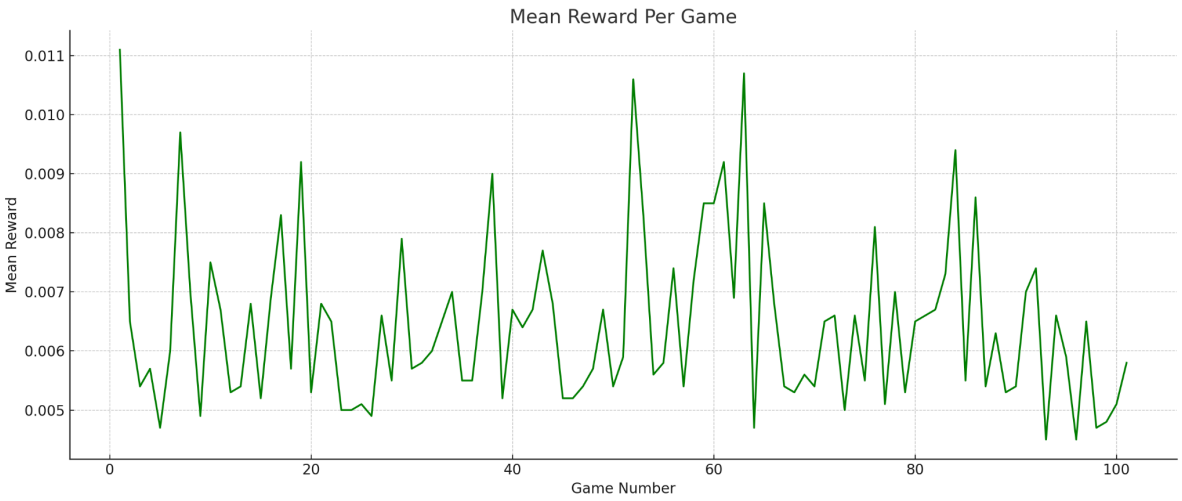
γ: un coefficient entre 0 et 1 qui réduit de l'importance des récompenses futur

s' : l'état suivant

$\mu(s')$: l'action suivante choisie par l'Actor


Critic Loss Over Time
Average Quadratic Loss Per Batch
Batch Number
0
5
10
15
20
0
25
50
75
100
125
150
175


Actor Q-Value Over Time
Average Q-Value Per Batch
Batch Number
500
0
−500
−1000
0
20
40
60
80
100
120
140


Mean Reward Per Game
Mean Reward
Game Number
0.011
0.010
0.009
0.008
0.007
0.006
0.005
0
20
40
60
80
100

La tendance des graphes reste identique, même après un entraînement supplémentaire important.

Voici ci-dessus les résultats obtenus avec les paramètres suivants :

Learning rate = 0.0005

Nombre couches caches = 3

Nombres de neurones par couches caches = 196

$\gamma = 0.99$

$\epsilon = 0.1$

Nombre de mobs passés en feature = 5

Nombre de murs passés en feature = 8

Skipping_loops = 100

Fonction d'activation = sigmoid

Taille d'un batch = 30

Fonction reward: L'agent reçoit des récompenses lorsque la distance avec les ennemis augmente, quand il tire la balle, quand il inflige des dégâts aux ennemis, quand il récupère la balle provenant de son allié, et quand il la tire.

Le premier graphe représente le MSE (Mean Squared Error) pour le Critic. On observe qu'il diminue rapidement, ce qui semble indiquer que le Critic parvient à apprendre efficacement à estimer la valeur des états en réduisant progressivement l'erreur entre ses prédictions et les valeurs cibles. Cela montre que l'entraînement du Critic converge correctement.

Le deuxième graphe montre la Q-value moyenne de l'Actor (Actor Q-Value) au cours de l'entraînement. On note d'abord de fortes oscillations, probablement dues à une exploration instable ou à des valeurs initiales mal calibrées. Cependant, après une vingtaine de batchs (soit 30*20 itérations d'apprentissage), la courbe se stabilise progressivement autour de zéro, ce qui

suggère que l'Actor apprend à proposer des actions raisonnables et que l'entraînement devient plus stable. Néanmoins, une Q-value autour de zéro reste très faible, ce qui est possiblement dû à la difficulté d'apprentissage dans un environnement peut-être trop complexe.

Enfin, le dernier graphe représente la récompense moyenne sur un jeu de 100 parties. On constate qu'elle n'augmente pas vraiment, ce qui est cohérent avec les résultats de la Q-value obtenus précédemment.

Il est difficile de déterminer la raison de ces résultats peu concluants. Mon hypothèse est que l'environnement est complexe, ce qui pousse l'agent à rester dans un optimum local relativement passif, causé par la difficulté d'exploration (le constat reste le même même en augmentant les paramètres). À noter que l'environnement a déjà été simplifié par rapport au jeu original (notamment une carte beaucoup plus simple, moins de monstres, un comportement de la balle moins punitif), mais les résultats restent très similaires.

Malgré de nombreuses tentatives de modifier les paramètres de toutes les manières possibles, je n'arrive pas à obtenir de meilleurs résultats. L'agent reste à côté d'un mur, bougeant et tirant la balle de temps à autre.

Je n'ai donc pas essayé de l'implémenter dans l'environnement du jeu original ni de le faire jouer avec un humain, les résultats étant trop peu satisfaisants, cela n'ayant que peu d'intérêt.

9. Conclusion

Bien que les résultats finaux n'aient pas atteint le niveau de performance espéré en termes de création d'une intelligence artificielle réellement efficace, je ne considère pas ce stage comme un échec.

Au contraire, cette expérience m'a permis d'approfondir mes connaissances sur l'apprentissage par renforcement, de mieux comprendre les difficultés liés à sa mise en œuvre concrète, et de développer des compétences précieuses en matière de recherche, de rigueur expérimentale et d'analyse critique.

Je suis convaincu que cette expérience me sera très utile dans mes projets futurs. Elle m'a permis de me confronter à des problématiques complexes et concrètes, et j'espère avoir l'opportunité de relever à nouveau ce type de défis, avec, je l'espère, de meilleurs résultats.

Je pense également que l'apprentissage a un avenir très prometteur dans le domaine du jeu vidéo, et j'ai hâte d'en voir les avancées. J'espère pouvoir, un jour, avoir la chance de participer à ce progrès.

Bibliographie / Sources

Intro to Goal Oriented Action Planning (GOAP)

Building the AI of F.E.A.R. with Goal Oriented Action Planning | AI 101

Goal-Oriented Action Planning: Ten Years of AI Programming

Behaviour Trees: The Cornerstone of Modern Game AI | AI 101

Behavior Trees - Godot Game Dev (BETTER AUDIO)

The AI of Half-Life: Finite State Machines | AI 101

Reinforcement Learning - "DDPG" explained

Deep Deterministic Policy Gradients

DDPG | Deep Deterministic Policy Gradient (DDPG) architecture | DDPG Explained

The FASTEST introduction to Reinforcement Learning on the internet

https://spinningup.openai.com/en/latest/algorithms/ddpg.html DDPG - OpenAI

https://en.wikipedia.org/wiki/Reinforcement_learning Reinforcement Learning - Wikipedia

https://medium.com/intro-to-artificial-intelligence/deep-deterministic-policy-gradient-ddpg-an-off-policy-reinforcement-learning-algorithm-38ca8698131b Deep Deterministic Policy Gradient (DDPG) — an off-policy Reinforcement Learning algorithm - Medium

https://en.wikiversity.org/wiki/Reinforcement_Learning Reinforcement Learning - Wikiversity

https://eprints.qut.edu.au/45741/ Current AI in games: A review - P Sweetser, J Wiles