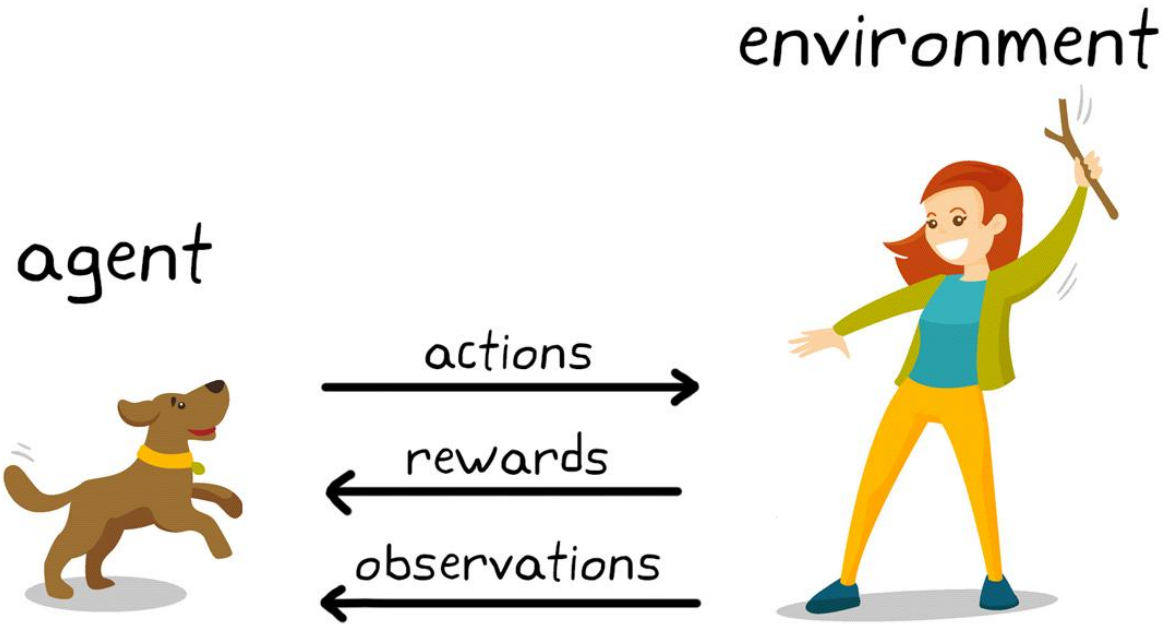
Machine-Learning : Projet-3

Apprentissage Par Renforcement



AL NATOUR Mazen et HERVOUET Léo

ML 2024-2025

Table des matières

Introduction : 3
I. Algorithme Q-Learning 4
A. Mise en place de l'environnement de jeu 4
1. Description 4
2. Implémentation 4
2. Structure 5
B. Développement de l'algorithme de Q-Learning 6
1. Explication de la stratégie utilisée 6
2. Pseudo-code du processus d'apprentissage 7
3. Difficultés rencontrées et optimisations effectuées 7
4. Annexes 8
C. Analyse de la table Q obtenue 9
D. Impact des paramètres 11
II. Algorithme Deep Q-Learning 12
A. Développement de l'algorithme 12
1. Description de la méthode 12
2. Décomposition de l'apprentissage 13
B. Validation de la politique 14
1. Simulation de la politique obtenue 14
2. Comparaison entre les deux modèles 16
C. Impact des paramètres 17
III. Comparaison Q-Learning vs Deep Q-Learning 18
Conclusion : 20

Introduction :

L’apprentissage par renforcement (Reinforcement Learning) constitue une branche clé de l’intelligence artificielle, axée sur la manière dont un agent peut apprendre à interagir de manière optimale avec son environnement à travers un système de récompenses et de pénalités. Ce rapport explore deux méthodes fondamentales de cette discipline : le **Q-Learning** dans un premier temps et le **Deep Q-Learning** dans un second temps. Nous nous concentrerons sur la mise en œuvre de ces algorithmes dans un environnement simulé, où un agent doit naviguer dans un labyrinthe en évitant les ennemis tout en atteignant un objectif final. Ce rapport synthétise les étapes de conception, les défis rencontrés, les solutions apportées, ainsi que les performances des algorithmes.

I. Algorithme Q-Learning

A. Mise en place de l'environnement de jeu

1. Description

L'environnement choisi est une grille en deux dimensions représentant un labyrinthe. L'objectif principal est de permettre au joueur, représenté par un symbole, de naviguer de la case de départ à la case d'arrivée tout en évitant les dragons. La grille contient plusieurs types de cases, les voici :

- **Case de départ (S)** : position initiale du joueur.
- **Case d'arrivée (E)** : position finale où le joueur remporte la partie.
- **Cases dragons (D)** : obstacles dangereux, dont l'atteinte entraîne une fin de partie.
- **Cases vides (.) :** cases libres que le joueur peut parcourir.
- **Position du joueur (P)** : position actuelle du joueur sur la grille.

2. Implémentation

L'environnement est implémenté dans la classe RLGame, qui fournit les fonctionnalités suivantes :

- **Création de la grille** : La méthode create_board initialise une grille avec des cases spécifiques.
- **Application des actions** : La méthode apply_action gère les transitions d'état et retourne la nouvelle position, la récompense associée et l'état de la partie (en cours ou terminée).
- **Réinitialisation** : La méthode reset_player_position permet de réinitialiser la position du joueur à son point de départ.
- **Affichage** : La méthode display_board visualise l'état actuel de la grille, en indiquant la position du joueur et des obstacles.

2. Structure

- **États** : Chaque case de la grille représente un état. Par exemple, dans une grille 4x4, il y a 16 états au total.
    - **Actions possibles** : Le joueur peut se déplacer dans quatre directions :

      - Haut (UP) **- Bas** (DOWN) **- Gauche** (LEFT) - **Droite** (RIGHT)
- **Transitions** : Les transitions sont déterminées par les mouvements du joueur, en tenant compte des limites de la grille. Si un mouvement dépasse les frontières de la grille, la position reste inchangée.
- **Récompenses** : Un système de récompenses est utilisé pour guider l'apprentissage :
    - **Case normal :** R=0
    - **Case avec dragon** : R=−2, la partie se termine.
    - **Case d'arrivée** : R=2, la partie se termine.
    - **Collision avec un mur (sorti du plateau de jeu)** : R=−1

B. Développement de l'algorithme de Q-Learning

1. Explication de la stratégie utilisée

a. Formulation mathématique de l'algorithme

L'algorithme de Q-Learning repose sur la mise à jour itérative des valeurs d'une fonction d'action-valeur ***Q(s,a)***, qui représente la qualité d'une action α prise dans un état s. La mise à jour de ***Q(s,a)*** suit l'équation suivante :

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

- α : le taux d'apprentissage, contrôle la vitesse à laquelle Q(s,a) converge.
- γ : le facteur de discount, pondère l'importance des récompenses futures.
- r : la récompense immédiate obtenue après avoir effectué l'action α.
- $\underbrace{max}_{a'} Q(s',a')$: la meilleure valeur d'action possible dans l'état suivant s′.

b. Exploration vs exploitation : choix de la politique

Une politique **ϵ-greedy** est utilisée pour équilibrer exploration et exploitation :

- Avec une probabilité **ϵ**, l'algorithme explore l'environnement en choisissant une action aléatoire.
- Sinon, l'algorithme exploite la connaissance actuelle en choisissant l'action qui maximise **Q(s,a)**. Le paramètre **ϵ** décroît progressivement au fil des épisodes pour favoriser l'exploitation à mesure que ***Q*** converge.

c. Paramètres du modèle

- **Taux d'apprentissage (α)** : α=0.9, ce qui favorise des mises à jour rapides des valeurs **Q(s,a)**.
- **Facteur de discount (γ)** : γ=0.5, mettant un poids modéré sur les récompenses futures.
- **Nombre d'épisodes** : 1000, permettant à l'algorithme de converger dans un environnement complexe.

2. Pseudo-code du processus d'apprentissage

Voici une version simplifiée du processus de Q-Learning :

1. Initialiser la table **Q(s,a)** avec des valeurs nulles pour tous les états et actions.
2. Pour chaque épisode :
   a. Réinitialiser la position du joueur dans l'environnement.
   b. Définir **ϵ** = 1 − (épisode / nombre d'épisodes).
   c. Tant que l'épisode n'est pas terminé :
      i. Choisir une action **α** en suivant la stratégie **ϵ-greedy**.
      ii. Exécuter l'action **α** et observer la nouvelle position **s′**, la récompense **r**, et l'état de fin.
      iii. Mettre à jour **Q(s,a)** avec l'équation de mise à jour.
      iv. Passer à l'état suivant **s′**
3. Retourner la table **Q** optimisée.

3. Difficultés rencontrées et optimisations effectuées

a. Difficultés rencontrées

- **Exploration inefficace au début de l'apprentissage** : Les premières explorations aléatoires menaient souvent le joueur à des états invalides ou à des dragons.

  Solution : Ajuster **ϵ** pour maximiser l'exploration initiale tout en réduisant de manière progressive cette probabilité à mesure que l'algorithme converge.

- **Convergence lente dans des environnements complexes** : Les récompenses mal équilibrées ralentissaient l'apprentissage.

  Solution : Modifier les valeurs des récompenses pour pénaliser plus fortement les erreurs (dragons, sorti du plateau) et récompenser généreusement les succès (case d'arrivée).

- **Visualisation des résultats** : Observer directement les mises à jour des **Q(s,a)** était compliqué.

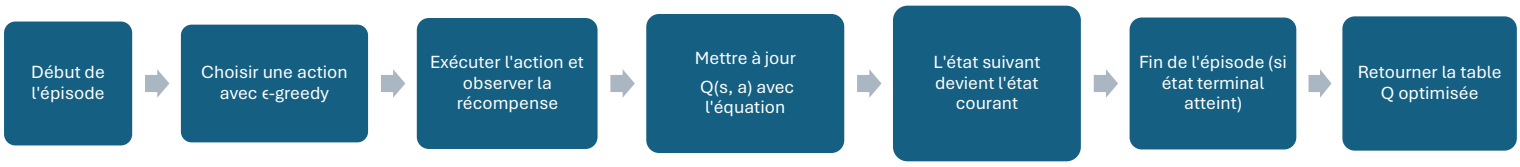
  Solution : Une méthode print_q_table a été ajoutée pour afficher la progression des valeurs **Q** à chaque étape.

b. Optimisations effectuées

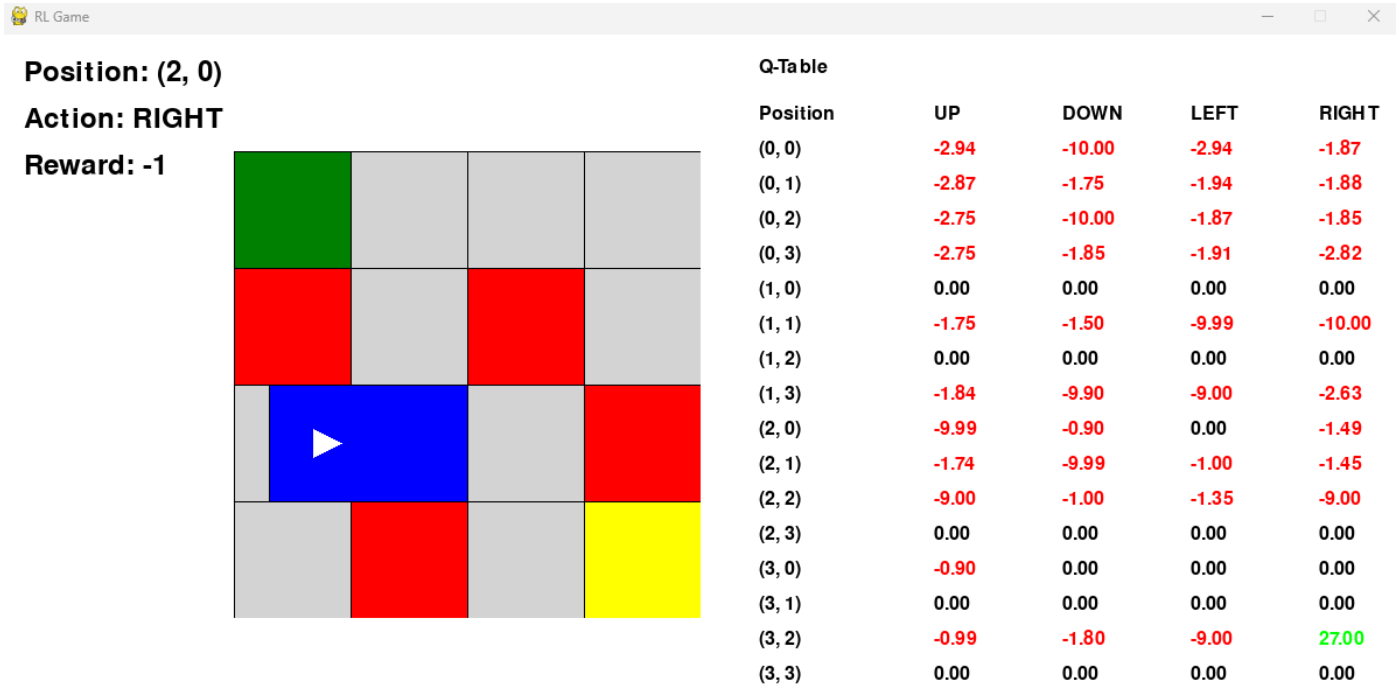
- **Initialisation du tableau Q** : Une initialisation explicite pour chaque état-action a permis de garantir une structure uniforme et facile à déboguer.
- **Gestion dynamique d'ϵ** : La réduction progressive d'ϵ jusqu'à 0 dans les derniers épisodes a assuré une convergence efficace.
- **Amélioration des logs** : Enregistrement des actions, récompenses et états à chaque étape pour faciliter l'analyse et ajuster les paramètres en fonction.

4. Annexes

Voici un schéma explicatif du processus d'apprentissage :



Voici une capture d'écran durant une partie du jeu :



C. Analyse de la table Q obtenue

1. Présentation de la table Q finale

La table Q obtenue à la fin de l'entraînement est une matrice où chaque état (x,y) est associé à quatre actions possibles : UP, DOWN, LEFT, et RIGHT. Les valeurs représentent la "qualité" de chaque action dans un état donné.

Extrait de la table Q finale (dernier épisode) :

Position	UP	DOWN	LEFT	RIGHT
(0, 0)	-0.97	-2.00	-0.97	0.06
(2, 2)	-2.00	1.00	0.25	-2.00
(3, 2)	0.50	0.00	-2.00	2.00
(3, 3)	0.00	0.00	0.00	0.00

Dans cet extrait, on peut remarquer :

- Les actions associées à des valeurs négatives indiquent des mouvements peu favorables (risque de sortir du plateau de jeu ou tomber sur des dragons).
- Les valeurs positives reflètent des actions qui mènent vers des récompenses élevées (par exemple, atteindre la case d'arrivée).
- Les positions terminales (comme (3, 3)) n'ont pas de valeurs significatives, car elles ne nécessitent plus de décision.

2. Évaluation de la politique optimale

a. Politique dérivée de la table Q

La politique optimale est définie comme suit : pour chaque état (x,y), l'action optimale est celle ayant la valeur Q(s,a) maximale.

Exemple :

- Pour l'état (3,2), la meilleure action est **RIGHT**, car Q ((3,2), RIGHT) =2.00.
- Pour l'état (2,2), la meilleure action est **DOWN**, car Q ((2,2), DOWN) =1.00.

b. Simulation des actions afin de démontrer le bon fonctionnement (Chemin simulé pour atteindre la case finale (3,3) depuis (0,0)) :

1. (0,0) → **RIGHT** → (0,1)
2. (0,1) → **DOWN** → (1,1)
3. (1,1) → **RIGHT** → (1,2)
4. (1,2) → **DOWN** → (2,2)
5. (2,2) → **DOWN** → (3,2)
6. (3,2) → **RIGHT** → (3,3)

D. Impact des paramètres

1. Étude des variations des récompenses

Deux configurations de récompenses ont été testées :

1. **Récompenses simples** :
    - NORMAL = 0, ENEMIES = -2, END = 2, WALL = -1
    - L'apprentissage converge lentement, car les récompenses sont faibles et les pénalités modérées.
2. **Récompenses renforcées** :
    - NORMAL = -1, ENEMIES = -10, END = 30, WALL = -2
    - L'apprentissage converge rapidement, car les récompenses et les pénalités sont plus marquées, poussant l'agent à adopter des actions stratégiques dès les premiers épisodes.

2. Impact des paramètres d'apprentissage

Paramètres	**Description**	**Comportement observé**
α = 0.9	Taux d'apprentissage élevé	Convergence rapide mais risque de surapprentissage si les récompenses sont mal définies.
α = 0.1	Taux d'apprentissage faible	Convergence très lente.
γ = 0.9	Importance des récompenses futures	L'agent planifie mieux ses actions, mais peut négliger les récompenses immédiates.
γ = 0.1	Importance des récompenses immédiates	L'agent privilégie des solutions rapides mais sous-optimales.
ϵ	Exploration (ϵ-greedy)	Décroissance progressive favorise la stabilité à la fin de l'apprentissage.

3. Résultats comparatifs

Configuration	**Convergence**	**Qualité de la politique**
Récompenses simples	Lent	Bonne, mais nécessite plus d'épisodes.
Récompenses renforcées	Rapide	Excellente, l'agent évite les ennemis et trouve la sortie efficacement.

II. Algorithme Deep Q-Learning

A. Développement de l’algorithme

1. Description de la méthode

La méthode Deep Q-Learning consiste à utiliser un réseau de neurones pour résoudre le jeu. Cette méthode permet d'entraîner le réseau en le faisant jouer : lorsqu'il effectue une bonne action, il est récompensé et continue à apprendre dans cette direction. À l'inverse, lorsqu'il commet une erreur, il fait des corrections pour maximiser la meilleure récompense possible. Grâce à cette approche, le réseau explore le contexte et découvre la meilleure stratégie pour jouer en fonction des récompenses reçues.

Comme pour le Q-Learning, le jeu consiste à se déplacer jusqu'à l'arrivée sans entrer en contact avec les ennemis.

Pour ce faire, nous utilisons un réseau de neurones avec en entrée 16 valeurs, une pour chaque case du plateau de jeu et en sortie c’est un vecteur de 4 valeurs qui est retourné (pour les déplacements : haut, droite, bas et gauche).

Deux architectures de réseaux sont testées :

- Le premier réseau ne comporte qu'une seule couche cachée de 8 neurones.
- Le second réseau possède deux couches cachées de 8 neurones chacune.

Ces réseaux sont suffisants, car le jeu n'est pas particulièrement complexe à résoudre.

La stratégie d'apprentissage adoptée repose sur une phase d'exploration initiale avec des mouvements aléatoires (epsilon fixé à 0). Ensuite, la valeur d'epsilon augmente progressivement pour atteindre 1. Ce qui signifie que, plus le modèle apprend, plus il est exploité pour trouver la solution optimale en mettant de côté les mouvements aléatoires. Cette technique permet une exploration dégressive, réduisant ainsi les risques de fins de jeu brutales.

De plus, pour rendre l'apprentissage plus efficace et moins dépendant d'une configuration fixe, la position de départ du joueur est aléatoire choisie à chaque début de partie pendant la phase d'exploration. Cette approche améliore l'exploration du plateau de jeu en couvrant un plus grand nombre de situations possibles.

2. Décomposition de l’apprentissage

Tout d'abord, dans le code, nous commençons par initialiser notre modèle selon celui que nous avons sélectionné (une couche ou deux couche). Ensuite, nous configurons l'optimiseur "Adam" ainsi que la fonction d'erreur "MSE" (Mean Squared Error).

Juste après, nous séparons notre modèle en deux en :

- Un modèle pour l'apprentissage, appelé "model".
- Un modèle plus stable, appelé "target", qui permet de réaliser des prédictions pour améliorer "model".

Ensuite, nous entrons dans la boucle principale qui gère les épisodes. Ici, nous calculons l'epsilon pour chaque épisode en fonction du nombre d'épisodes aléatoires, défini par la variable "episodes_random" (c'est le nombre d'épisodes où des actions peuvent être choisies aléatoirement).

Nous lançons ensuite la boucle de jeu, où "model" effectue son apprentissage. À chaque itération, "model" calcule le prochain coup à effectuer sur le plateau. Puis, nous utilisons "target" pour déterminer le meilleur coup suivant. Avec l'action réalisée et la prédiction du prochain coup, nous calculons la valeur cible que notre modèle doit prédire pour l'étape suivante. Cette valeur cible est la somme de la récompense du coup joué et de la récompense prédite pour le prochain coup par "target", multipliée par gamma, un hyperparamètre.

Pour limiter l'impact de la prédiction, si le joueur rencontre un mur ou un ennemi, la valeur de prédiction pour le prochain coup est fixée à 0. Dans ce cas, seule la récompense immédiate est prise en compte. C'est une partie essentielle de l'algorithme d'apprentissage du modèle. La récompense d'une direction est définie par la récompense immédiate et la valeur du meilleur coup possible de la case d'arrivée.

Nous effectuons ensuite une prédiction avec "model" et ajustons ses poids en fonction de la différence entre la prédiction et la valeur cible. Cette étape correspond à la descente de gradient, réalisée par l'optimiseur Adam.

Une fois cette étape terminée, nous répétons le processus jusqu'à la fin du jeu. À la fin du jeu, une nouvelle époque commence avec un nouvel epsilon. Quand l'epsilon est égal à 0, c’est-à-dire lorsque nous utilisons le modèle à 100 % pour prendre les décisions, nous réinitialisons la position de départ à l’emplacement prévu, situé en (0, 0) sur le plateau de jeu.

Pour terminer l’entraînement, nous attendons que notre modèle termine le jeu 20 fois de suite avec succès. Une fois cet objectif atteint, nous pouvons retourner le modèle.

B. Validation de la politique

1. Simulation de la politique obtenue

Une fois que nous avons terminé d'entraîner notre modèle, nous pouvons afficher les résultats obtenus pour chaque position ainsi que les poids associés à chaque action.

Pour l'apprentissage nous avons utilisé les récompenses suivantes :

Case **normal** : **-1**, case **dragon** : **-20**, case **d'arrivée** : **100** et collision **mur** : **-5**

Pour éviter les boucles de jeu, lorsque nous faisons plus 100 actions, nous avons une récompense à **-25** et nous terminons le jeu.

Préambule : Avant d'expliquer la simulation de la politique obtenue, voici comment nous allons présenter les résultats :

	5			20	
5	Position1	50	40	Position2	5
	20			100	

Les cases en gris et en vert montrent les récompenses attribuées par le modèle pour une position sur le plateau de jeu. Si un modèle prend en entrée deux positions et retourne quatre actions possibles, alors pour chaque position, les actions (haut, droite, bas, gauche) sont affichées dans le tableau, avec en vert l'action ayant le poids le plus élevé.

Par exemple, nous pouvons passer de Position1 à Position2 en utilisant le modèle, car la case avec le poids le plus élevé pour Position1 est celle située à droite. Pour Position2, la case ayant la valeur la plus haute est celle du bas, donc le modèle choisira cette action. Cette action mène à un mur, ce qui ramène le joueur à la même case. Toutes les cases autour du tableau représentent des murs.

Cette représentation du modèle permet de visualiser simplement l'itinéraire choisi pour aller du point de départ à la fin du jeu. Les poids associés aux dragons ne sont pas pertinents, car ils n'ont pas été entraînés étant donné que nous ne passons jamais par ces cases pendant l'entraînement.

Pour notre premier réseau simple (avec une couche de 8 neurones), voici les récompenses prédites:

	26.24			27.95			23.80			-0.62	
8.49	**Départ**	87.46	30.58	Chemin	14.46	76.75	Chemin	23.17	40.48	Chemin	18.66
	-12.51			95.73			-8.36			-6.03	
	54.30			44.34			48.13			3.82	
24.84	Dragon	72.80	9.90	Chemin	5.89	26.18	Dragon	65.74	-2.04	Chemin	30.47
	25.44			96.60			29.51			-11.65	
	49.12			59.56			8.02			49.00	
44.59	Chemin	112.68	57.82	Chemin	102.28	81.27	Chemin	2.00	25.57	Dragon	68.59
	-5.27			-6.09			98.87			22.69	
	81.61			47.68			90.13			47.91	
16.42	Chemin	59.02	24.95	Dragon	59.36	2.12	Chemin	102.83	27.17	**Fin**	66.28
	-10.42			28.92			11.54			27.00	

Tableau des récompenses du premier modèle

En suivant les récompenses que nous donne le modèle, nous obtenons un chemin allant du départ jusqu'à la fin du jeu. Ce chemin est le plus court que nous pouvons obtenir. Le modèle a réussi à propager jusqu'au départ la récompense attribuée lorsque nous terminons le jeu, ce qui lui permet de trouver un chemin optimal du début à la fin (cases vertes).

Pour notre deuxième réseau (avec deux couches de 8 neurones), voici les récompenses prédites :

	-12.50			-12.07			-11.30			-15.86	
-11.71	**Départ**	-0.05	-4.19	Chemin	-11.69	-7.45	Chemin	-2.64	-7.55	Chemin	-19.53
	-20.74			1.85			-21.36			-7.39	
	-10.04			-13.62			-8.39			-14.34	
-3.63	Dragon	-3.44	-9.47	Chemin	-34.14	-8.10	Dragon	-4.29	-22.94	Chemin	-18.71
	-8.61			5.81			-11.19			-28.01	
	-11.07			-6.72			-20.02			-6.12	
-5.20	Chemin	4.07	-6.10	Chemin	9.04	6.02	Chemin	-24.73	-6.68	Dragon	7.40
	-8.61			-10.92			13.37			-8.65	
	-9.77			-9.64			-1.49			-12.86	
-6.50	Chemin	-23.36	-5.64	Dragon	2.32	-1.55	Chemin	15.11	-5.31	**Fin**	-7.60
	-8.22			-7.98			-2.89			-6.17	

Tableau des récompenses du deuxième modèle

Ici aussi, en utilisant le modèle, nous obtenons bien un chemin allant du départ jusqu'à la fin du jeu, en empruntant le plus court chemin. Comme pour le modèle précédent, celui-ci a réussi à propager jusqu'au départ la récompense attribuée lorsque nous terminons le jeu.

2. Comparaison entre les deux modèles

Tout d'abord, pour les deux modèles, nous obtenons des résultats assez différents : le premier modèle a appris beaucoup plus vite que le second, avec des récompenses se rapprochant des valeurs optimales qui sont les suivantes :

	-5			-5			-5			-5	
-5	**Départ**	95	94	Chemin	94	95	Chemin	93	94	Chemin	-5
	-20			96			-20			92	
	nombre			95			nombre			93	
nombre	Dragon	nombre	-20	Chemin	-20	nombre	Dragon	nombre	-20	Chemin	-5
	nombre			97			nombre			-20	
	-20			96			-20			nombre	
-5	Chemin	97	96	Chemin	98	97	Chemin	-20	nombre	Dragon	nombre
	95			-20			99			nombre	
	96			nombre			98			nombre	
-5	Chemin	-20	nombre	Dragon	nombre	-20	Chemin	100	nombre	**Fin**	nombre
	-5			nombre			-5			nombre	

Dans le tableau des récompenses optimales, on peut voir clairement la propagation de la récompense de 100 points obtenue en terminant le jeu. On remarque aussi que toutes les récompenses vers les dragons sont de -20 et celles vers les murs sont de -5.

Parmi les deux modèles, le premier est celui qui se rapproche le plus des récompenses optimales. Le second en est très éloigné.

Pour les deux modèles, les récompenses obtenues varient beaucoup, avec des valeurs parfois bien au-dessus ou en dessous de la normale. Cela est dû à un apprentissage insuffisant, mais il permet quand même d'obtenir un résultat concluant.

C. Impact des paramètres

Nous allons examiner l'impact des paramètres et hyperparamètres sur l'évolution du modèle :

• **La taille du réseau :** Comme nous l'avons observé, plus le modèle est grand, plus le temps d'apprentissage devient important. Cependant, un réseau de grande taille est capable de capturer des cas spécifiques à notre environnement, ce qui améliore la performance globale du modèle. Trouver un juste équilibre entre la taille et la complexité reste essentiel pour optimiser le temps d'entraînement sans sacrifier la précision.

• **Epsilon :** Adopter une stratégie où epsilon diminue progressivement au fil des époques est l'approche la plus efficace que nous ayons testée. Cette méthode commence par une phase d'exploration où le modèle teste diverses actions. Progressivement, il entre dans une phase d'exploitation où il utilise principalement les décisions basées sur son apprentissage. Cela permet au modèle d'optimiser sa performance tout en explorant l'environnement de manière suffisante au départ.

• **Taux d'apprentissage :** Pour les deux modèles, nous avons opté pour un taux d'apprentissage de 0.01, particulièrement adapté à l'optimiseur "Adam". Ce taux favorise un apprentissage rapide et génère de bons résultats. Une valeur inférieure (par exemple, 0.001) pourrait ralentir l'apprentissage au point de ne pas atteindre une solution optimale, tandis qu'une valeur supérieure (au-delà de 0.01) pourrait provoquer des oscillations ou empêcher la convergence.

• **Gamma :** Le choix d'un gamma élevé, ici 0.999, s'est révélé important pour capturer l'impact des décisions à long terme. Un gamma plus faible réduirait cet effet, ralentissant ainsi l'apprentissage. Cependant, dans des situations spécifiques (comme lorsqu'un mur ou un dragon est rencontré), nous ajustons temporairement le gamma à 0 pour prioriser la récompense immédiate et réagir rapidement.

• **Nombre d'épisodes :** Nous avons fixé à 1000 le nombre d'épisodes d'entraînement pour les deux modèles. Ce choix permet un apprentissage à la fois rapide et stable, avec une diminution progressive de l'epsilon de 1 à 0. Cette durée d'entraînement assure une exploration suffisante tout en laissant au modèle le temps de converger vers une solution optimale.

• **Récompenses :** Nous avons expérimenté diverses configurations de récompenses avant de retenir celles proposées dans le sujet. Ces récompenses ont offert les meilleurs résultats, en orientant efficacement l'apprentissage du modèle et en encourageant les comportements souhaités dans l'environnement. Nous avons également noté que des ajustements trop fréquents des récompenses pouvaient perturber la stabilité du processus d'entraînement.

III. Comparaison Q-Learning vs Deep Q-Learning

Comparons maintenant l'approche du Q-Learning et celle du Deep Q-Learning.

- **Pour le Q-Learning :**
  - **Avantages :**
    - **Apprentissage rapide :** l'utilisation d'une table de poids pour mettre à jour directement les valeurs associées à chaque état-action permet d'apprendre rapidement les politiques optimales dans les environnements simples. En pratique, une solution peut être trouvée en quelques secondes, comme cela a été observé dans nos tests où l'entraînement n'a pris que 2 secondes.
    - **Simplicité :** C'est une approche intuitive, facile à comprendre et à implémenter.
    - **Convergence garantie :** Le Q-Learning, sous certaines conditions (comme une exploration suffisante et un taux d'apprentissage décroissant), est mathématiquement prouvé pour converger vers la politique optimale.
  - **Inconvénients :**
    - **Problèmes d'échelle :** Le principal inconvénient réside dans l'évolution exponentielle de la taille de la table de poids lorsque le nombre d'états augmente. Cette limitation le rend inefficace pour les environnements de grande taille où la table devient rapidement ingérable.
    - **Adapté aux petits environnements :** En raison de cette limitation, le Q-Learning est principalement utilisé pour des problèmes simples ou des environnements à faible complexité.

- **Pour le Deep Q-Learning :**
    - **Avantages :**
        - **Adapté aux grands environnements :** Contrairement au Q-Learning classique, le Deep Q-Learning s'appuie sur des réseaux de neurones profonds pour approximer la fonction Q. Cela ne nécessite pas une augmentation linéaire de la taille du réseau en fonction du nombre d'états, ce qui le rend bien plus évolutif et efficace dans des environnements complexes avec un grand espace d'états.
        - **Stabilité des poids :** Les méthodes d'optimisation comme le replay buffer évitent les changements brusques des poids, améliorant ainsi la stabilité et la précision de l'apprentissage.
    - **Inconvénients :**
        - **Temps d'apprentissage :** Le Deep Q-Learning nécessite un temps d'entraînement significatif pour atteindre une performance optimale, en particulier lorsque l'environnement est complexe.
        - **Complexité :** Comparé au Q-Learning, le Deep Q-Learning est plus difficile à comprendre et à implémenter.

Donc il est clair que le choix de l'approche dépend principalement de la taille et de la complexité de l'environnement. Pour notre jeu, qui se déroule dans un environnement relativement simple, le Q-Learning s'avère être l'option la plus adaptée. Il offre une solution rapide, efficace et facile à implémenter, sans les contraintes du Deep Q-Learning. Cela dit, si l'environnement venait à évoluer vers une configuration plus complexe, le passage au Deep Q-Learning pourrait alors devenir une stratégie pertinente pour maintenir des performances optimales.

Conclusion :

Ce projet a permis d’explorer l’efficacité et les limitations des algorithmes de Q-Learning et de Deep Q-Learning dans un environnement simulé. Le Q-Learning a démontré sa robustesse dans des environnements simples, mais son efficacité décroît face à des espaces d'états complexes. En revanche, le Deep Q-Learning, grâce à l'utilisation de réseaux neuronaux, s’est montré plus adapté à ces environnements, offrant une meilleure généralisation et des politiques optimales.

Les résultats obtenus mettent en évidence l’importance des paramètres d’apprentissage et des récompenses dans la convergence des modèles. Ce projet montre le potentiel des algorithmes d’apprentissage par renforcement dans des applications concrètes, tout en ouvrant des perspectives pour des améliorations et des adaptations à des scénarios plus complexes.

Ce projet a également renforcé notre compréhension de l’utilisation et de l’entraînement de ces algorithmes. Il nous a également permis d’améliorer nos compétences en Machine Learning, tout en nous préparant à relever des défis plus complexes.