APPRENTISSAGE PAR RENFORCEMENT

UNE BRANCHE DE L'I.A

Résumé

Ce document décrit les recherches sur les méthodes d'apprentissage par renforcement, qui ont été menées durant mon stage de fin d'étude au LORIA

Tuteurs : Adrien GUENARD et Cyril REGAN

Theo BOUNACEUR

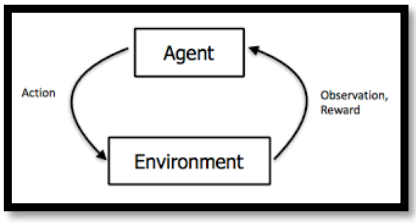
theo.bounaceur@loria.fr

TABLE DES MATIERES

I. INTRODUCTION

1.1 DEFINITION

Le Reinforcement Learning (R.L.), ou apprentissage par renforcement, est une branche de l'intelligence artificielle (I.A.) qui se concentre sur la manière dont un agent peut apprendre à interagir avec son environnement pour atteindre un objectif.



Contrairement à l'apprentissage supervisé, où les données sont étiquetées et l'algorithme apprend à faire des prédictions basées sur ces étiquettes, le R.L. repose sur un cadre d'interaction. Dans ce cadre, l'agent explore l'environnement, prend des décisions, et reçoit des récompenses ou des punitions en fonction des actions qu'il entreprend.

1.2 OBJECTIF

Cet état de l'art a pour but d'explorer les principales approches de l'apprentissage par renforcement, en mettant l'accent sur les algorithmes "model-free" tels que le Q-learning et le Deep Q-Network (DQN). Il couvre également des algorithmes plus avancés comme le Proximal Policy Optimization (PPO) et le Soft Actor-Critic (SAC). Ce rapport s'intéresse aussi à l'utilisation d'environnements d'apprentissage simulés simples, comme FrozenLake et MiniGrid, ainsi qu'à des outils d'expérimentation (Stable Baselines3) et de suivi des performances (MLflow).

2. FONDAMENTAUX DE L'APPRENTISSAGE PAR RENFORCEMENT

2.1 DEFINITION DES CONCEPTS CLES

L'apprentissage par renforcement repose sur plusieurs concepts fondamentaux :

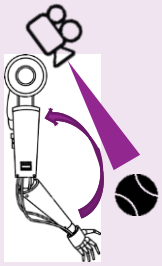
Agent

L'agent est l'entité qui prend des décisions et interagit avec l'environnement. Son objectif est d'apprendre à maximiser les récompenses qu'il reçoit à travers ses actions. Il peut s'agir d'un robot réel ou d'une entité informatique.

Environnement

L'environnement représente le milieu dans lequel l'agent opère. Il peut être réel comme le cas d'un robot, ou simulé par ordinateur.

Ici l'environnement est une grille de 4x4 qui contient des trous et un cadeau

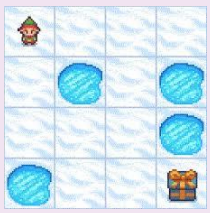


Ici l'environnement est un espace réel qui contient une balle

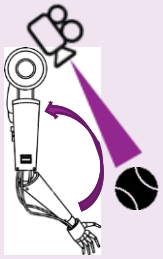
État (State)

L'état est une représentation complète de la situation actuelle de l'environnement dans laquelle se trouve l'agent. En d'autres termes c'est ce que perçoit l'agent à un instant donné.

Cela peut être des informations tels que sa position, sa vélocité… Les informations peuvent aussi être subjectives comme les données d'une caméra placé sur le robot ou encore l'orientation de ses articulations.



Ici l'état est représenté par un entier de 0 à 15 qui représente la case où le lutin est positionné.

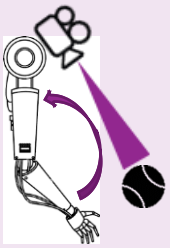


Ici l'état est représenté par l'orientation du bras du robot (fournit par un capteur) et par l'image fournit par la caméra

Action

Les actions sont les choix que l'agent peut faire pour interagir avec l'environnement. Les actions peuvent influencer l'état de l'environnement et déterminer la récompense que l'agent recevra.

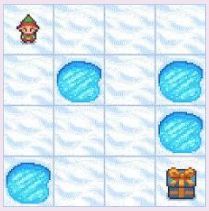
Ici le lutin peut aller à Gauche/Droite/Haut/Bas



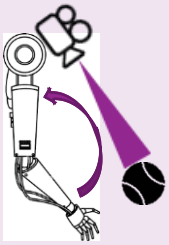
Ici le robot peut changer l'angle de son bras

Récompense (Reward)

La récompense est un signal de rétroaction que l'agent reçoit après avoir effectué une action. Elle peut être positive (**encourageant l'action**) ou négative (**incitant à éviter l'action**). C'est l'environnement qui fournit les récompenses, et l'objectif de l'agent est de maximiser la somme des récompenses reçues au fil du temps.



Ici la récompense peut être négative si le lutin se déplace vers un trou ou positive s'il se déplace vers le cadeau



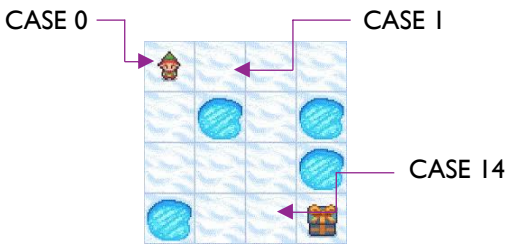
Ici le robot recevra une récompense s'il attrape la balle

Politique (Policy)

La politique est une stratégie utilisée par l'agent pour déterminer quelles actions entreprendre en fonction de l'état actuel de l'environnement. Une fois entraînée, cette politique guidera l'agent pour maximiser les récompenses.

2.2 LA TABLE DES Q-VALEURS (Q-VALUES)

Le principe fondamental de l'apprentissage par renforcement est que l'agent cherche à estimer la qualité de ses actions dans différents états à travers plusieurs tests. Pour garder en mémoire ces informations, l'agent utilise une table de Q-valeurs. Cette table associe à chaque couple (état, action) une valeur, appelée Q-valeur, qui représente une estimation de la récompense totale attendue (**récompenses immédiates + futures**) si l'agent prend cette action dans cet état.



Exemple avec FrozenLake

	GAUCHE	BAS	DROITE	HAUT
CASE 0	+0	+0 ,5	+0,5	+0
CASE 1	+0	-1	+1	+0
…				
CASE 15	+0	+0	+1	+0

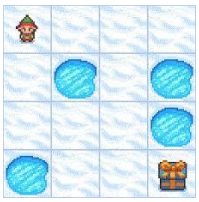
La table des Q-valeurs pour cet exemple de FrozenLake ressemblera à cela lorsque l'agent aura effectué suffisamment de tests. Elle constitue une politique à suivre dans cet environnement, qu'on appelle **politique cible** (Target policy). En mode *évaluation*, l'agent n'aura qu'à suivre les actions qui mènent aux plus grandes valeurs de Q.

2.3 ENTRAINEMENTS ET EPISODES

L'apprentissage par renforcement est généralement organisé en **épisodes**. Un épisode correspond à une séquence d'actions prises par l'agent depuis un état initial jusqu'à un état terminal, où il atteint son objectif ou échoue.

Dans FrozenLake :

- Etat initial = Case 0 (en haut à gauche)
- Etat terminal = Case 10 (en bas à droite)
- Cas d'échec = Les cases où il y a un trou

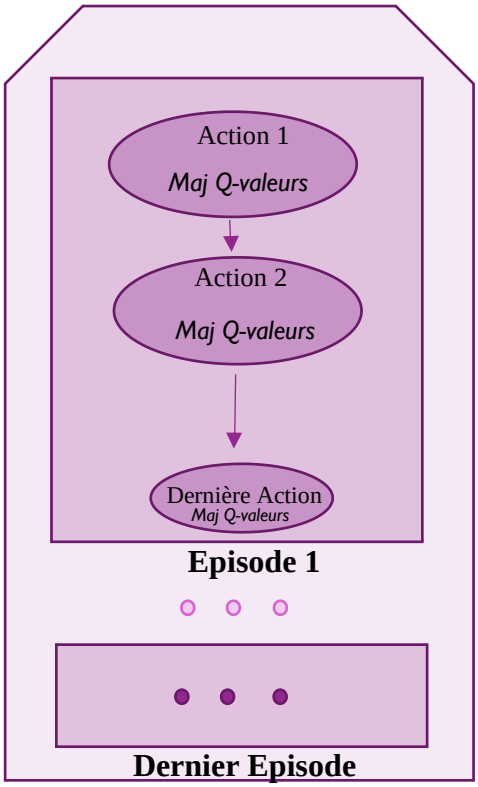


FrozenLake

Au cours de ces épisodes, l'agent explore l'environnement, en effectuant des actions et met donc à jour ses Q-valeurs en fonction des récompenses qu'il reçoit. Tout cela afin d'ajuster sa politique et ainsi apprendre de son environnement.

2.4 MISE A JOUR DES Q-VALEURS

L'agent met à jour ses Q-valeurs au cours des épisodes, à chaque action effectuée et récompense obtenue. La formule de mise à jour courante est l'équation de Bellman (développée en section 4.2), qui prend en compte l'action effectuée et la récompense reçue.



Entraînement

2.5 EXPLORATION VS EXPLOITATION

Lors de la phase d'apprentissage, l'agent doit choisir des actions à opérer en prenant en compte le dilemme entre EXPLORATION et EXPLOITATION.

- **Exploration** : L'exploration consiste à essayer de nouvelles actions pour découvrir leurs conséquences. C'est essentiel pour apprendre des informations sur l'environnement et améliorer la politique de l'agent. Cependant, trop d'exploration peut empêcher l'agent de tirer parti des connaissances qu'il a déjà acquises.
- **Exploitation** : L'exploitation implique d'utiliser les connaissances actuelles de l'agent pour maximiser la récompense. Cela signifie choisir les actions qui ont déjà montré de bons résultats. Cependant, se concentrer uniquement sur l'exploitation peut conduire à manquer de meilleures opportunités que l'agent n'a pas encore découvertes.

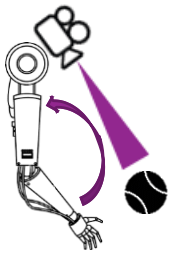
Stratégies courantes pour gérer ce dilemme :

- **ε-greedy :** Choisir la meilleure action avec une probabilité de 1 - ε, ou une action aléatoire avec une probabilité de ε. La valeur de ε diminue au fil des entraînements.
- **Softmax :** Sélectionner les actions selon une distribution de probabilité basée sur les récompenses obtenues. Les actions avec de meilleures récompenses ont une probabilité plus élevée d'être choisies, tout en permettant une certaine exploration.

La politique suivie pendant cette phase est la **politique de comportement (Behavior policy)**, à distinguer de la **politique cible (Target Policy)**, qui est celle qu'on entraîne et qu'on utilisera lors de l'évaluation.

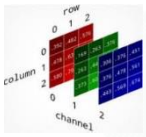
2.6 LES RESEAUX DE NEURONNES EN APPRENTISSAGE PAR RENFORCEMENT

Lorsque **l'espace des états ou des actions devient trop grand ou continu**, la table des Q-valeurs devient alors trop conséquente pour être stockée. C'est le cas de l'exemple du bras de robot.



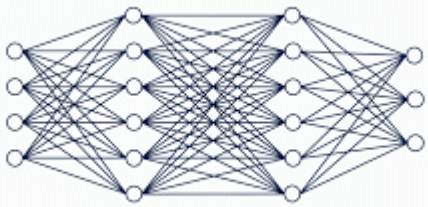
- *Entrées = Angle du bras + Image*
- *Sortie = Rotation du bras*

En effet, l'agent reçoit en entrée un image constituée de plusieurs pixel (*r*,*g*,*b*). Si l'on devait regrouper dans une table toutes les configurations possibles de l'image, cela serait trop gourmand en mémoire. On utilise alors un **réseau de neurones** qui prend en entrée l'état du système et qui renvoi les Q-valeurs pour les différentes actions possibles.



45° Angle

Entrées = Angle du bras + Image



Réseau de neurones

Q-Network

- $Q(s,a_1)$
- $Q(s,a_2)$
- ...
- $Q(s,a_N)$

Sortie = Q-valeurs pour les N action possibles

2.7 RECAPITULATIF DES FONDEMENTS DU R.L

On a vu que pour entrainer un modèle, l'agent doit effectuer **plusieurs actions regroupées en épisodes**. Chaque action prise par l'agent forme une boucle d'apprentissage : à chaque action l'environnement récompense l'agent ce qui va lui permettre de **mettre à jour sa fonction de Q-valeurs**. Cette fonction peut être représentée par une table pour des cas simples et discrets ou par un réseau de neurones pour des cas continus ou possédant des multiples valeurs.

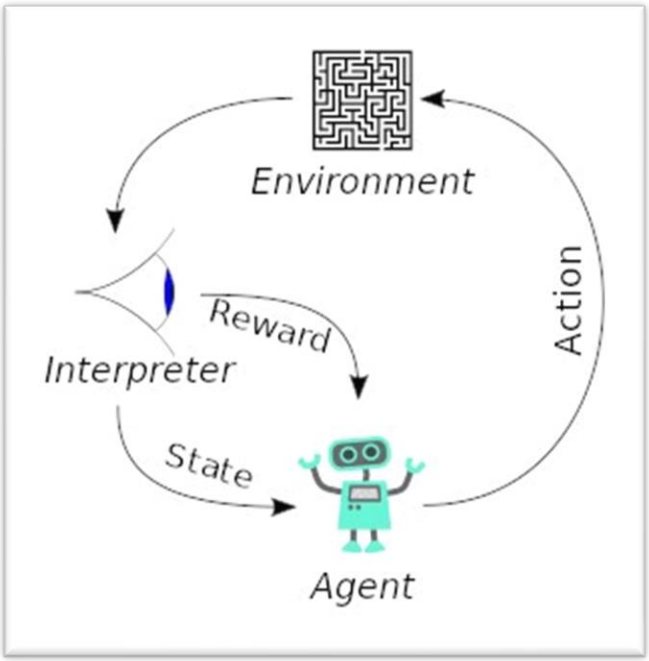


Schéma de la boucle d'apprentissage

Les actions choisies par l'agent au fur et à mesure de l'apprentissage sont choisis suivant une **politique de comportement** qui peut être **ε-greedy**. C'est-à-dire qu'au départ l'agent choisira des actions aléatoires pour explorer l'environnement. Puis au fur et à mesure de l'entrainement, il choisira plutôt des solutions déjà vu qui possèdent de fortes récompenses pour ainsi privilégier l'exploitation.

Au fil des épisodes d'entrainements, la fonction des Q-valeurs s'affine pour ainsi former une **politique cible**. L'agent pourra alors se diriger dans son environnement en mode évaluation grâce à celle-ci.

3. LES TYPES D'ALGORITHMES EN R.L

3.1 MODEL FREE / MODEL BASED

Dans l'apprentissage par renforcement, il existe deux grandes familles d'algorithmes :

1. **Les algorithmes Model Free** : Ces algorithmes, comme le **Q-Learning**, **DQN**, **SARSA**, **PPO** ou encore **SAC** ne cherchent pas à créer une représentation précise de l'environnement dans lequel l'agent évolue. Autrement dit, l'agent ne sait pas vraiment comment l'environnement fonctionne, mais il apprend simplement quelles actions sont bonnes ou mauvaises dans chaque situation (ou "état"). Ils associent seulement quelle action effectuer dans quel état.

2. **Les algorithmes Model Based** : Ces algorithmes comme **Dyna DQN**, quant à eux, tentent de construire une sorte de "modèle" de l'environnement. Cela signifie qu'ils essaient de comprendre comment l'environnement réagit à chaque action prise par l'agent. Grâce à ce modèle, l'algorithme peut ensuite anticiper les résultats des différentes actions et choisir les meilleures en conséquence. Ces méthodes sont plus complexes car elles nécessitent une prédiction des conséquences des actions.

3.2 ALGORITHMES DE POLICY GRADIENT

Les algorithmes de **Policy Gradient** en R.L. apprennent directement une politique, c'est-à-dire une fonction qui détermine quelles actions l'agent doit prendre en fonction de l'état dans lequel il se trouve. Contrairement aux méthodes basées sur la fonction de valeur, comme les DQNs qui apprennent une estimation de la qualité des actions (Q-valeurs) pour chaque état, les algorithmes de policy gradient ajustent la politique elle-même afin d'augmenter la probabilité de sélectionner des actions qui mènent à des récompenses élevées.

Caractéristiques : les actions sont choisies en fonction d'une politique stochastique (Politique de Commportement), et la politique est mise à jour pour maximiser les récompenses obtenues à long terme.

Proximal Policy Optimization (PPO) : C'est une amélioration des Policy Gradients qui contraint les mises à jour de la politique pour éviter des changements trop drastiques. Il stabilise l'apprentissage. Il est souvent utilisé pour des environnements complexes.

Les Différences

- **Policy Gradient** apprend une politique directement, ce qui permet d'apprendre des politiques stochastiques ou continues (utile pour les espaces d'action continus).
- **DQN (Deep Q-Network)** est un algorithme basé sur une fonction de valeur qui apprend une Q-value pour chaque action dans un état donné. Il suit une politique de comportement (greedy ou epsilon-greedy) en choisissant les actions qui maximisent la Q-value.

Les algorithmes de **Policy Gradient** sont souvent plus adaptés aux problèmes d'actions complexes ou continues, tandis que les **DQNs** sont mieux adaptés aux problèmes d'actions simples et discrètes car ils sont plus simples à implémenter et à converger vers une solution optimale.

4. ALGORITHMES DE TYPE Q-LEARNING

4.1 Q-LEARNING

Le **Q-learning** est l'un des algorithmes fondamentaux du Reinforcement Learning. Il est model-free, c'est-à-dire qu'il n'a pas besoin d'un modèle explicite de l'environnement. L'agent construit une table de **Q-valeurs**, où chaque couple (état, action) est associé à une estimation de la récompense attendue.

Le Q-learning repose sur l'**équation de Bellman**, qui permet de mettre à jour les valeurs de Q en fonction des récompenses reçues après chaque action. L'algorithme suit un processus d'itérations où l'agent explore l'environnement, prend des actions, et met à jour la table pour maximiser ses récompenses futures.

Voici un résumé des étapes du Q-learning

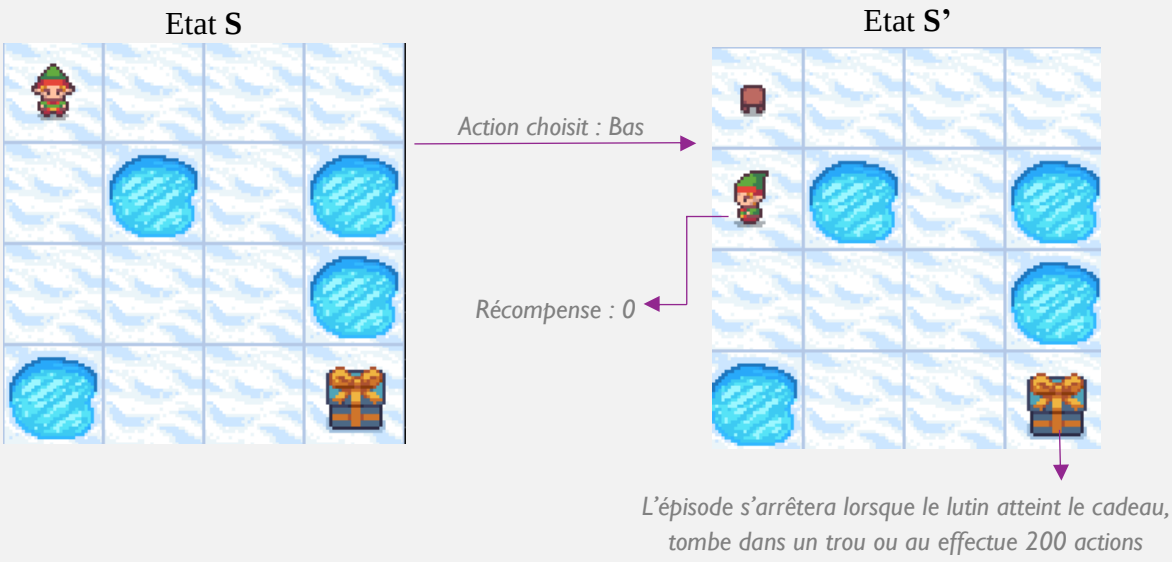
Pour chaque EPISODE :

Pour chaque Action :

1. *L'agent observe l'état s dans lequel il est*
2. *Il choisit une action a via une politique de comportement comme ε-greedy*
3. *Il reçoit une récompense r et observe le nouvel état s'*
4. *Il met à jour la valeur Q(s,a) selon l'équation de Bellman:*

$$Q(s,a) \leftarrow Q(s,a) + \alpha TDerror$$

L'agent recommence et s'arrête lorsqu'il atteint un état final ou dépasse 200 actions



4.2 EQUATION DE BELLMAN

Lors de l'apprentissage, pour une action donnée, l'équation de Bellman permet de calculer la valeur de Q observée, c'est-à-dire celle que l'agent :

$$\boldsymbol{Q(s,a)observée} = \boldsymbol{r} + \boldsymbol{\gamma}.\boldsymbol{max_{a'}Q(s',a')}$$

où :

- ***s*** est l'état courant,
- ***a*** est l'action choisie,
- ***r*** est la récompense obtenue
- ***s'*** est l'état suivant,
- ***a'*** est la meilleure action possible dans cet état.

Le paramètre γ (facteur d'actualisation) influence l'impact des récompenses futures. Cette équation prend en compte à la fois la récompense immédiate (***r***) et la meilleur estimation de la récompense future (***$max_{a'}$Q(s',a')***) qui représente le maximum des récompense pour l'état suivant ***a'***.

L'erreur temporelle ou $\boldsymbol{TD_{error}}$ est définie comme la différence entre $\boldsymbol{Q(s,a)_{observée}}$ et $\boldsymbol{Q(s,a)_{enregistrée}}$:

$$\boldsymbol{TD_{error}} = \boldsymbol{Q(s,a)observée} - \boldsymbol{Q(s,a)enregistrée}$$

On obtient alors l'**équation finale de mise à jour de la Q-table de Bellman** :

$$Q(s,a) \leftarrow Q(s,a) + \alpha TDerror$$

où :

- ***s*** est l'état courant,
- ***a*** est l'action choisie,

Le paramètre ***α*** (taux d'apprentissage) contrôle l'importance de la mise à jour.

Ce processus est répété à chaque épisode pour permettre à l'agent d'affiner ses Q-valeurs et d'améliorer sa politique.

4.3 ON-POLICY ET OFF-POLICY

On dit que Q-learning est **Off-Policy** car dans l'équation de Bellman, le terme $\boldsymbol{max_{a'} Q(s', a')}$ est choisie pour estimer la valeur des récompenses futures à l'étape ***s'***.

SARSA (State-Action-Reward-State-Action) est un autre algorithme model-free, similaire au Q-learning, mais avec une différence clé : SARSA met à jour la fonction Q en fonction de la **politique cible** de l'agent. C'est-à-dire que pour estimer la valeur des récompenses futures en ***s'***, l'agent choisit l'action qui sera réellement choisie dans l'état ***s'*** (action explorée) selon la politique en cours. Ainsi, pour mettre à jour la Q-valeur, contrairement à Q-learning qui prend la meilleure action possible dans cet état, SARSA choisit l'état que la politique actuelle conseil. On dit alors que SARSA est **On-Policy**.

SARSA prend en compte l'action choisie (qui peut ne pas être la meilleure action) pour la mise à jour des Q-valeurs. Cela signifie que si l'agent explore souvent des actions moins optimales, cela se reflétera directement dans les mises à jour des Q-valeurs, favorisant une exploitation plus prudente.

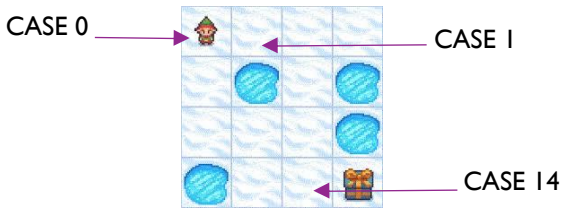
SARSA est donc plus adapté pour des politiques qui favorisent davantage l'exploration dans l'apprentissage.

4.2 DEEP Q-NETWORK (DQN)

Les **Deep Q-Networks (DQN)** sont une extension du Q-learning, où l'approximation des Q-valeurs est faite à l'aide de **réseaux de neurones profonds**. Ces réseaux permettent de gérer des environnements avec de vastes espaces d'états ou continus, là où une table de Q-valeurs serait inefficace.

La Q-fonction n'est donc plus une Q-table mais un Q-Network !

Dans mes travaux sur l'environnement **FrozenLake**, j'ai utilisé un DQN pour permettre à l'agent de naviguer dans un environnement. L'agent connait uniquement l'état dans lequel il est. On rappelle que cet état est représenté par un entier allant de 0 à N-1, où N est le nombre de case.



FrozenLake

Ici on retrouve un cas simple à 16 cases, mais on peut multiplier le nombre de cases.

4.3 POURQUOI UN DEEP Q-NETWORK (DQN) NECESSITE 2 RESEAUX DE NEURONNES

ATTENTION : dans un DQN, on utilise 2 réseaux de neurones !!

En **Q-learning**, l'agent utilise une simple table de Q-valeurs pour stocker la qualité des actions dans chaque état. Cela fonctionne bien quand l'espace d'états est petit, car il est possible de représenter tous les états et actions dans une table.

En **DQN (Deep Q-Network)**, l'agent utilise des réseaux de neurones pour approximer les Q-valeurs lorsque l'espace d'états est trop grand pour être stocké dans une table. Deux réseaux de neurones sont utilisés pour stabiliser l'apprentissage :

1. **Réseau principal (ou d'entraînement)** : Ce réseau **Q-Network** est constamment mis à jour à chaque étape de l'entraînement, en ajustant ses poids pour mieux prédire les Q-valeurs. Il représente la Q-table dans un Q-Learning simple.
2. **Réseau cible** : Ce réseau **Q-Target** est une copie du réseau principal, mais il est mis à jour moins fréquemment. Cela permet de calculer les cibles de Q-valeurs de manière plus stable, en évitant des changements trop brusques à chaque mise à jour.

La formule de Bellman devient alors : $\boldsymbol{Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \cdot max_{a'} Qtarget(s',a') - Q(s,a)]}$

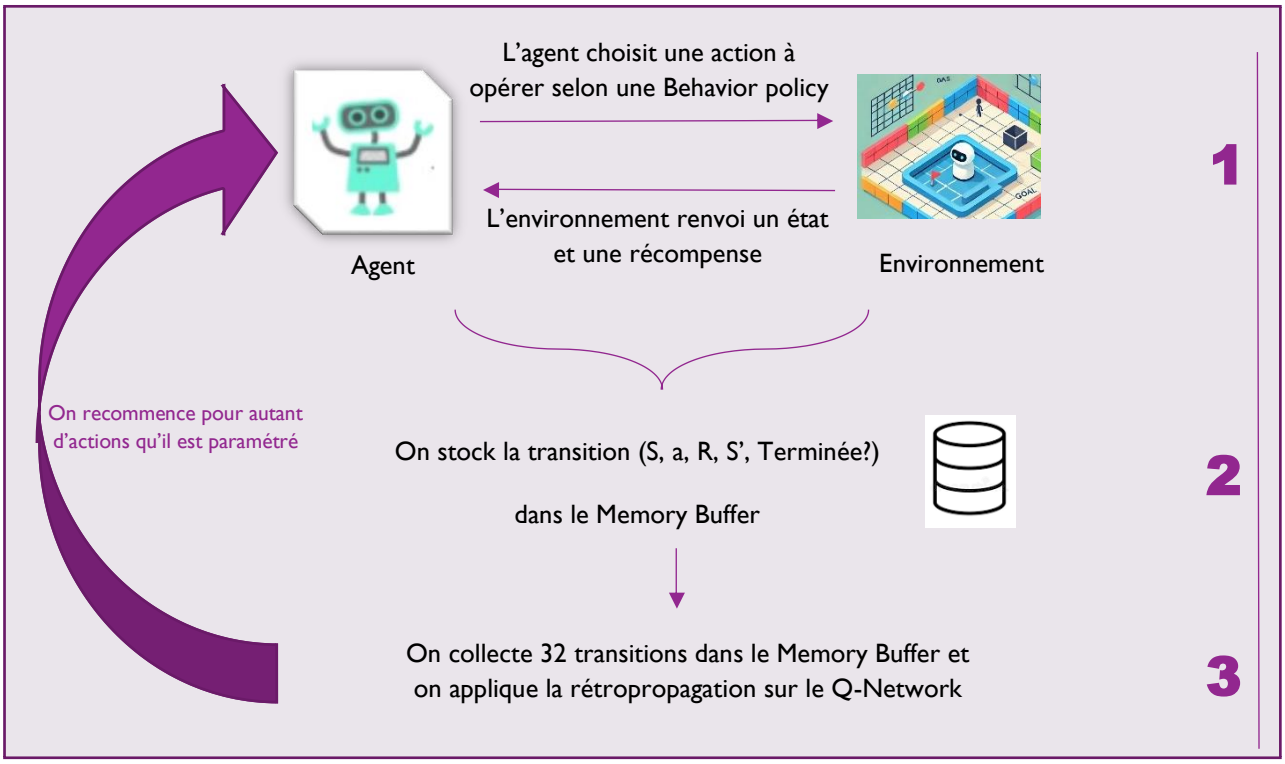
L'utilisation de deux réseaux permet de réduire l'instabilité due à l'apprentissage continu, car si le réseau principal change trop rapidement, les cibles utilisées pour la mise à jour des Q-valeurs peuvent devenir incohérentes, ce qui rendrait l'entraînement chaotique.

4.4 BOUCLE D'APPRENTISSAGE D'UN DQN

La boucle d'apprentissage d'un DQN diffère du Q-learning classique. L'agent continu mettre à jour sa Q-fonction après chaque action, sauf que dans ce cas il utilise un réseau de neurones (Q-Network) et applique l'équation de Bellman via rétropropagation.

Le DQN utilise un **Memory Buffer**, où un échantillon aléatoire de **32 transitions passées** est prélevé. La mise à jour des Q-valeurs est ensuite calculée pour ces 32 transitions, et la moyenne des erreurs est rétro propagée dans le Q-Network. Cette approche stabilise l'apprentissage en réutilisant les expériences passées et en lissant les mises à jour.

Boucle d'apprentissage d'un DQN



La collecte des 32 transitions est effectuée après chaque actions. Les 32 transitions peuvent être choisies aléatoirement ou selon d'autres stratégies comme aléatoire pondérée par la réussite des transitions… La rétropropagation est alors effectuée en appliquant la formule de Bellman sur les 32 transitions. La moyenne des 32 résultats est retro propagé dans le Q-Network.

Synchronisation de Q-Target

Au bout de N-synchronisations actions effectuées, le réseaux cible Q-Target est mis à jour. Dans notre cas la mise à jour se fait de la manière la plus simple possible : Q-Network est copié sur Q-Target (Q-Target←Q-Network).

4.5 ARCHITECTURE DU CODE DQN APPLIQUE A FROZENLAKE

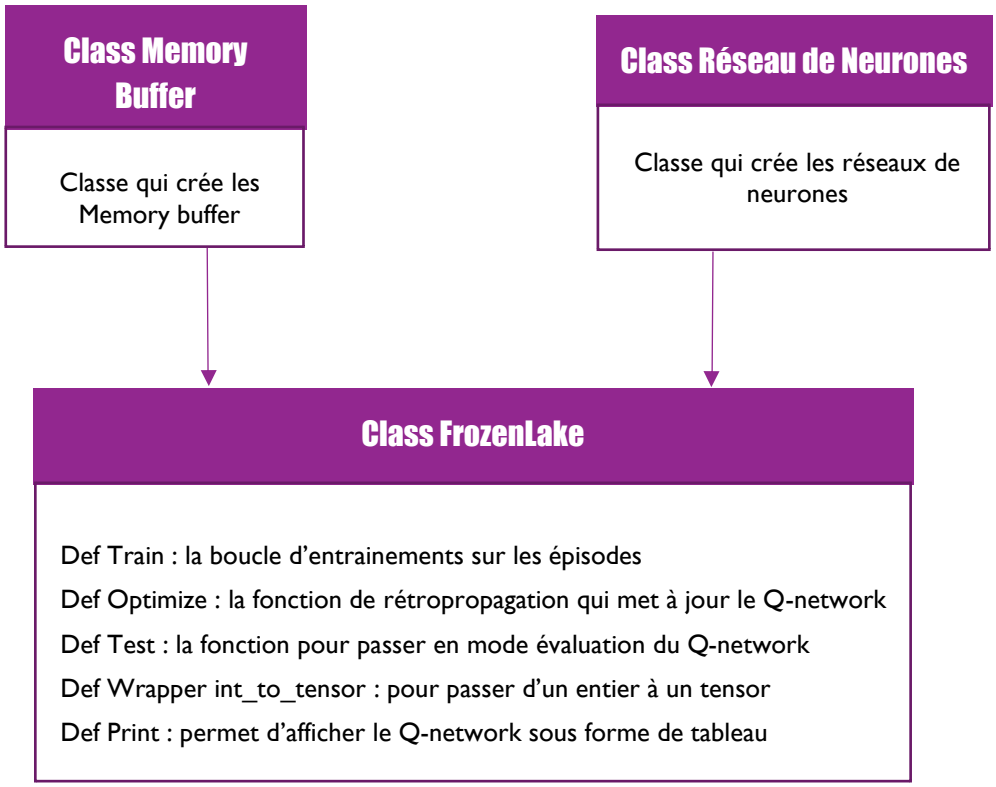


Schéma de l'architecture de mon DQN

4.6 ETUDE DE LA BOUCLE D'ENTRAINEMENT DANS UN CODE

Voici un extrait de mon code pour la résolution de FrozenLake avec un DQN. Le code suivant fait partie de la fonction Train de la classe FrozenLake et représente la boucle d'apprentissage. C'est le cœur de l'algorithme de l'apprentissage par renforcement.

```
######################################## Training ########################################################
for i in tqdm(range(episodes), desc="Entraînement"):
    state = env.reset()[0]  # Initialise l'agent a l'etat 0 qui correspond a la case en haut a gauche
    terminated = False      # True quand l'agent tombe dans un trou ou arrive au bout
    truncated = False       # True quand l'agent fait plus de 200 actions (qu'il se perd)


    ############################## On entre dans la boucle d'un Episode ##############################
    # L'agent effectue des transitions jusqu'a ce qu'il tombe, arrive au bout ou se perde
    while(not terminated and not truncated):

        # 1. L'agent choisit une action selon sa behavior policy qui est ici un epsilon greedy
        if random.random() < epsilon:
            # Action Aleatoire
            action = env.action_space.sample() # actions: 0=left,1=down,2=right,3=up
        else:
            # Action Optimale
            with torch.no_grad():
                action = training_dqn(self.state_to_dqn_input(state, num_states)).argmax().item()

        # 2. Execute action
        new_state,reward,terminated,truncated,_ = env.step(action)

        # 3. Stocker cette transition dans le Replay Buffer
        memory.append((state, action, new_state, reward, terminated))

        # 4. Entraîner le réseau si suffisamment d'expériences sont disponibles
        if (len(memory) > self.mini_batch_size):
            mini_batch = memory.sample(self.mini_batch_size)  # Échantillonnage d'un mini-batch
            self.optimize(mini_batch, training_dqn, target_dqn)  # Optimisation du réseau

        # 5. L'Etat devient le nouvel Etat
        state = new_state

        # 6. Incrementation du step counter
        step_count+=1
    ################################### On sort de l'Episode ########################################

    # Stocker la récompense obtenue dans cet épisode
    rewards = rewards + reward
    rewards_per_episode[i] = rewards

    # Décroître l'epsilon pour favoriser l'exploitation
    epsilon = max(epsilon - 1/episodes, 0)
    epsilon_history.append(epsilon)

    # Synchroniser le Target Network avec le Training Network en fonction du network_sync_rate
    if i % self.network_sync_rate == 0:
        target_dqn.load_state_dict(training_dqn.state_dict())
#########################################################################################################
```

1

2

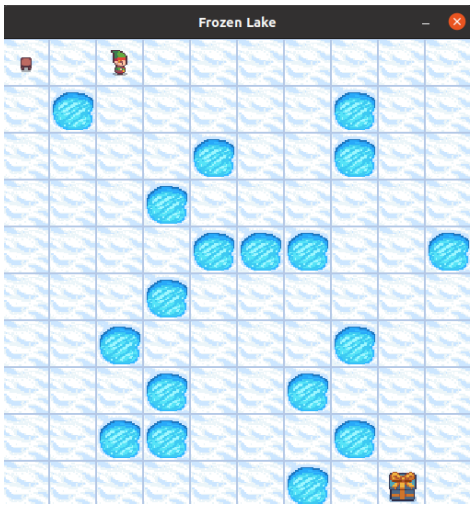
3

Synchronisation

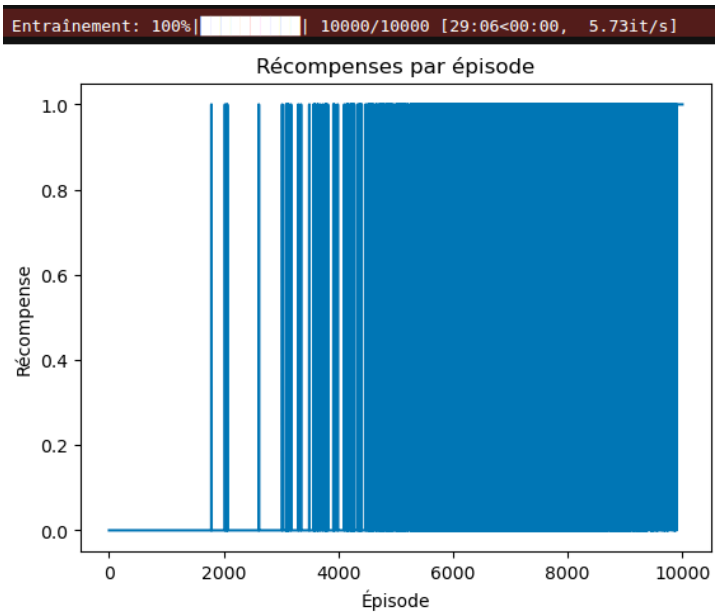
On retrouve les 3 parties de la boucle d'apprentissage décrites précédemment.

4.7 RESULTATS FROZENLAKE

ANALYSE

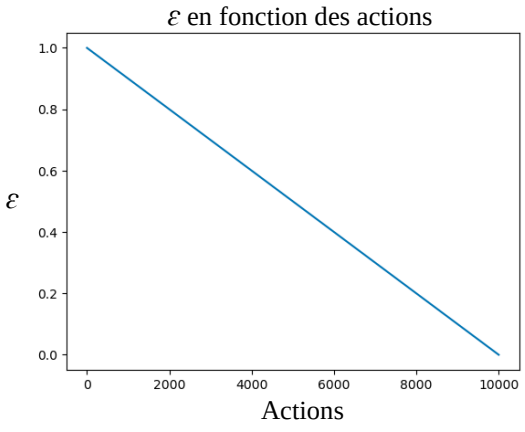


Environnement FrozenLake

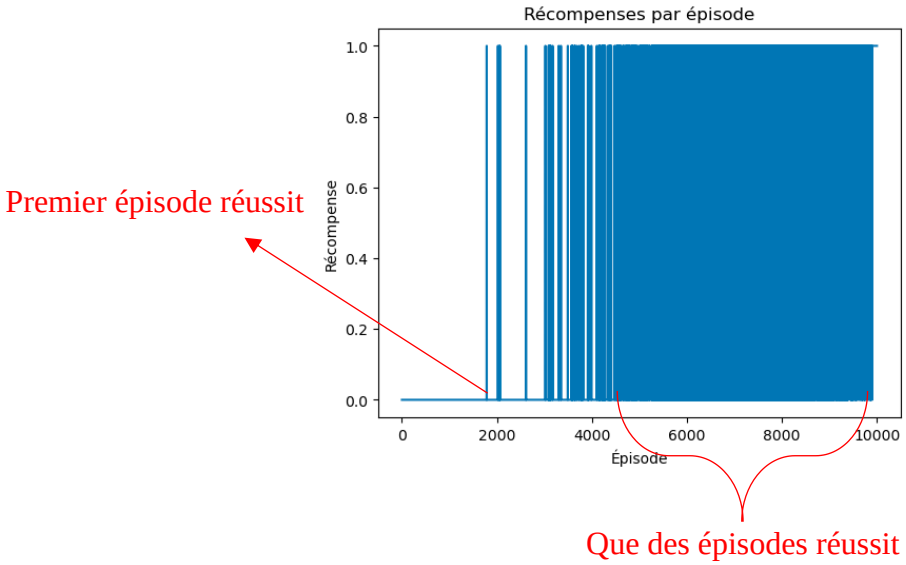


La récompense est égale à 1 si le lutin arrive au cadeau. On remarque que le lutin arrive pour la première fois au cadeau seulement au bout de la 1900ème action effectuée. Avant cela, le lutin ne faisait que des déplacements dans le vide ou tombait dans les trous.

On remarque aussi que la fréquence de réussite du lutin augmente progressivement. A partir de l'action 4300, le lutin arrive même à chaque fois jusqu'au cadeau.



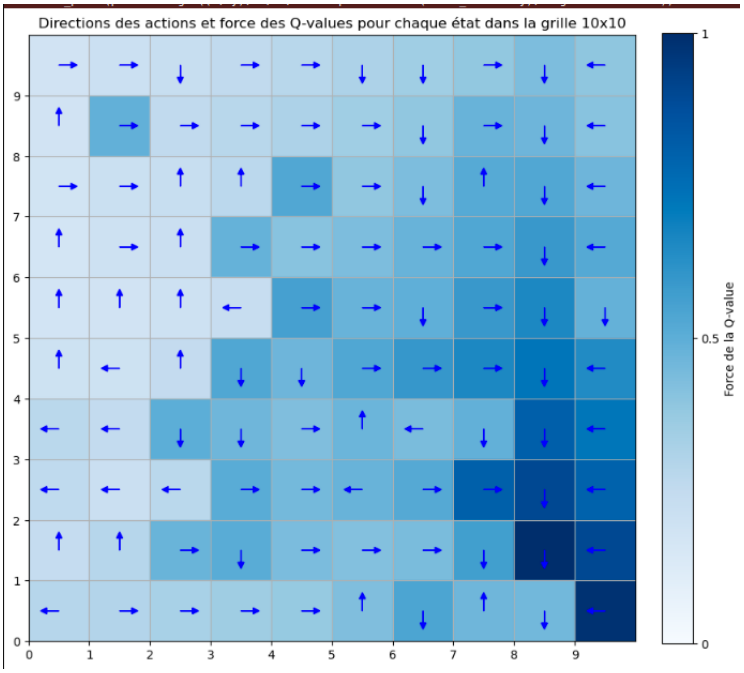
On rappelle que ε correspond à la part d'aléatoire dans le choix des actions prisent par l'agent (lutin). Ici la courbe est simplement linéaire. AU départ les actions sont purement aléatoires puis exploitent de plus en plus la table des Q-valeurs vers la fin de l'entrainement.

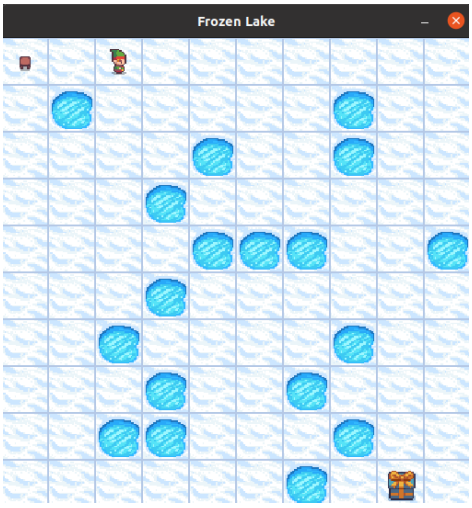


On remarque qu'il y a d'abord une première phase d'exploration où l'agent n'effectue aucune réussite, car ses actions sont principalement guidées par l'aléatoire. Puis arrive une phase où le nombre d'épisode réussit s'accélère. Cette phase met en jeu à la fois l'exploration et l'exploitation des premières réussites. Enfin, vers le 5000ème épisode, on remarque que l'agent effectue uniquement des épisodes réussit. On peut alors en conclure que l'entraînement aurait pu s'arrêter avant puisque l'agent ne semble plus rien apprendre.

Mais attention : Une phase qui contient uniquement des réussites ne veut pas forcément dire que l'agent connait parfaitement son environnement. S'il existe un autre chemin, plus court l'agent peut ne pas le voir. En effet s'il choisit de plus en plus des actions basées sur l'exploitation et qu'il a déjà trouver un chemin menant vers la réussite, il n'explorera plus. C'est pourquoi il faut prendre compte cela pour la suite.

On peut visualiser la HeatMap (après un entraînement de 10000 actions) :





Environnement FrozenLake

Cette HeatMap présente l'action guidée par le Q-Network qui équivaut à la Q-valeur max dans chaque case. Plus la Q-valeur est importante, plus la case apparait foncée. On remarque que les cases près du cadeau apparaissent assez foncées ainsi que les cases près des trous. Cela résulte des récompenses positives et négatives que l'agent à reçu au cours de son entrainement.

COMMENTAIRE

Pour FrozenLake, mon algorithme DQN fournit des résultats très satisfaisant. On remarque qu'au bout de 5000 actions effectuées, l'agent à mis à jour ses Q-valeurs dans son Q-Network pour qu'il puisse atteindre le cadeau à tous les coups. Il ne faudra pas plus de 20 minutes pour entraîner l'agent dans cet environnement. **En ordre de grandeur, le code met environ 10 secondes par cases pour s'entrainer**.

Néanmoins, attention à l'équilibre Exploration/Exploitation et à ne pas contraindre l'agent à rester dans des optimums locaux.

Le code fonctionne aussi si l'on rajoute une notion non déterministe : si on active le paramètre ***Is_Slippery***, l'agent à désormais une probabilité non nulle de se tromper de case lors d'une action. L'agent réussit à apprendre mais plus lentement, et en mode évaluation les cas d'échecs ne sont pas inexistant puisque le robot peut glisser dans un trou.

4.8 RESULTATS MINIGRID

4.7 COMMENTAIRES

Lors de mes expériences, j'ai remarqué que le **Q-learning** et le **SARSA** fonctionnent bien dans des environnements simples comme FrozenLake, où l'espace des états est relativement limité. Cependant, dès que l'environnements devient plus grand ou plus complexe, l'utilisation de réseaux de neurones améliore grandement la rapidité d'exécution. C'est pourquoi pour un environnement FrozenLake de plus de 100 cases, l'utilisation d'un DQN est largement recommandée pour une meilleure rapidité d'exécution.

Les Limites

Une des limites que j'ai observées avec le **DQN** est sa lente convergence, notamment dans des environnements où les récompenses sont retardées ou rares. Dans l'exemple de Minigrid qui sera plus détaillé dans la section **PPO**, la résolution du labyrinthe ne semble pas évidente. En effet, il faudra plusieurs essais avant que l'agent arrive par hasard vers l'objectif final qui est l'unique source de récompense. Même au bout de 3000 épisodes, ce qui représente plus de 10 000 actions, l'agent ne présente pas une ***Q-Target Policy*** fiable. Pourtant on constate quand même une amélioration des résultats au cours des épisodes d'entrainement.

L'agent ne reçoit aucune pénalité s'il fonce dans un mur ou s'il traverse une case vide. De ce fait apparait un problème assez récurent : l'agent se bloque contre les murs. Je ne sais pas exactement à quoi cela est du mais il fat garder à l'esprit que dans le cadre de Minigrid, l'environnement est tellement vaste que le nombre d'épisode pour un entraînement réussit dépasse l'ordre du million. Ce qui n'est pas possible à moins d'entraîner en parallèle les agents simultanément.

5 ALGORITHMES DE POLICY GRADIENT

5.1 INTRODUCTION A PPO ET SAC

Le **PPO** (**Proximal Policy Optimization**) est une méthode d'optimisation de politique qui cherche à maximiser une fonction objectif tout en contrôlant l'ampleur des changements de politique à chaque étape, afin d'éviter des mises à jour trop brutales et instables.

Le **SAC** (**Soft Actor Critic**) est conçu pour des environnements à **espace d'action continu**. Il combine des stratégies d'exploration efficaces avec un réseau qui maximise la récompense tout en maintenant une certaine **entropie**, permettant à l'agent d'explorer l'espace d'action de manière plus variée. Cela le rend particulièrement adapté aux environnements où l'agent doit prendre des décisions dans des espaces d'action continus, comme la robotique.

5.2 PPO DE PYTORCH

Dans mes travaux sur l'environnement MiniGrid, suite à mes échecs à entraîner un modèle, et aux recommendations de mon tuteur, j'ai utilisé **PPO**.

Contrairement aux algorithmes **DQNs** décrits précédemment, je n'ai pas codé directement PPO mais je vais utiliser un code déjà fourni par la bibliothèque **PyTorch**. La bibliothèque ***Stable_baselines3*** met à disposition plusieurs algorithmes de R.L comme PPO et SAC qu'il suffit d'entrainer dans l'environnement de notre choix.

<u>Voici le guide de démarrage pour entrainer un modèle PPO :</u>

Pour importer un environnement virtuel (décrit en 6.1) →

```
import gymnasium as gym
```

Importation de *PPO* et de *make_vec_env* pour entrainer plusieurs agents en parallèle →

```
from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env

# Parallel environments
vec_env = make_vec_env("CartPole-v1", n_envs=4)
```

Entraînement du modèle sur 25000 actions sur l'environnement « ppo_cartpole » →

```
model = PPO("MlpPolicy", vec_env, verbose=1)
model.learn(total_timesteps=25000)
model.save("ppo_cartpole")

del model # remove to demonstrate saving and loading

model = PPO.load("ppo_cartpole")
```

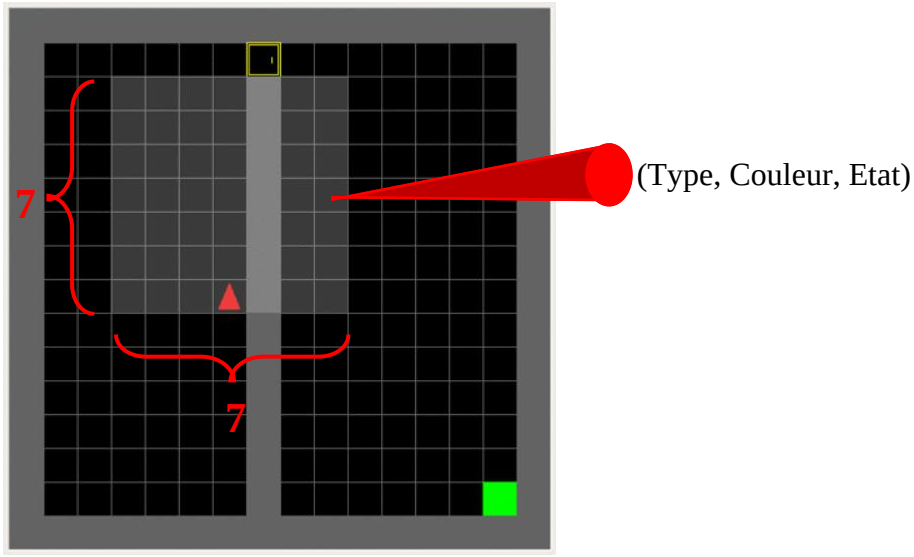
Test du modèle en visualisation « human » →

```
obs = vec_env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

5.3 ADAPTER PPO POUR LES ENVIRONNEMENTS MINIGRID

Un des défis majeurs a été de simplifier les observations de Minigrid pour rendre l'entraînement compatible avec les exigences de **PPO** de la bibliothèque ***Stable_baselines3***. En effet, un état dans le cas de Minigrid est caractérisé par une matrice de 7x7x3. C'est-à-dire que l'agent voit 7x7 cases devant lui (*comme illustré sur l'image suivante*). Chaque case est caractérisée par 3 valeurs : (Type, Couleur, Etat)

Monde de Minigrid



Bien que ***StableBaselines3*** soit compatible avec les environnements Gymnasium, l'architecture CNN par défaut ne supporte pas cette observation Minigrid. Cest pourquoi pour entrainer un agent sur un environnement Minigrid, il faut créer un ***wrapper*** (détaillé en 6.2). Cela peut être fait au moyen d'une classe ***MinigridFeaturesExtractures*** fournit par Minigrid.

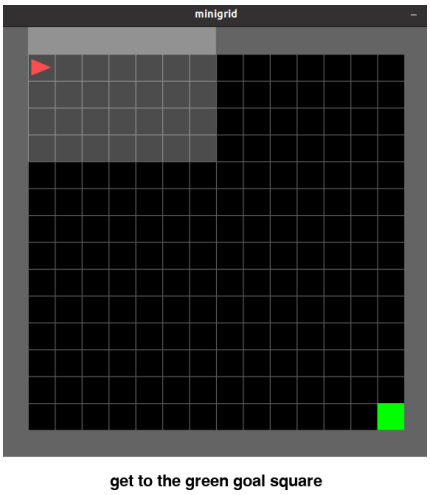
Class MinigridFeaturesExtractor

```python
class MinigridFeaturesExtractor(BaseFeaturesExtractor):
    def __init__(self, observation_space: gym.Space, features_dim: int = 512, normalized_image: bool
        super().__init__(observation_space, features_dim)
        n_input_channels = observation_space.shape[0]
        self.cnn = nn.Sequential(
            nn.Conv2d(n_input_channels, 16, (2, 2)),
            nn.ReLU(),
            nn.Conv2d(16, 32, (2, 2)),
            nn.ReLU(),
            nn.Conv2d(32, 64, (2, 2)),
            nn.ReLU(),
            nn.Flatten(),
        )

        # Compute shape by doing one forward pass
        with torch.no_grad():
            n_flatten = self.cnn(torch.as_tensor(observation_space.sample()[None]).float()).shape[1]

        self.linear = nn.Sequential(nn.Linear(n_flatten, features_dim), nn.ReLU())

    def forward(self, observations: torch.Tensor) -> torch.Tensor:
        return self.linear(self.cnn(observations))
```

5.4 LES RESULTATS DE PPO APPLIQUE A UN ENVIRONNEMENT MINIGRID

L'écriture du code PPO appliqué à ***StableBaselines3*** est relativement facile. J'ai entrainé le modèle dans un environnement vide de 16x16.

Environnement Minigrid



Voici la partie du code qui entraîne le modèle. On peut remarquer la présence de *logs* qui permet de garder en mémoire les modèles entraînés et des informations sur eux pour pouvoir étudier l'amélioration du modèle au cours de plusieurs entraînements. Tout cela se fait grâce à Tensorboard (décrit en 7.)

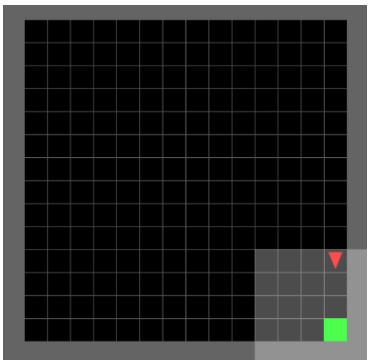
```
4. Train Model + Save training

log_path = os.path.join('Minigrid','Training', 'Logs')

model = PPO("CnnPolicy", env, policy_kwargs=policy_kwargs, verbose=1, tensorboard_log=log_path)
model.learn(total_timesteps=20000,tb_log_name="PPO")
```

Premier entraînement à 20000 actions qu'on nomme « PPO »

Par la suite j'ai refait un entraînement « PPO 3 » à 10 millions d'actions qui a duré 6h.

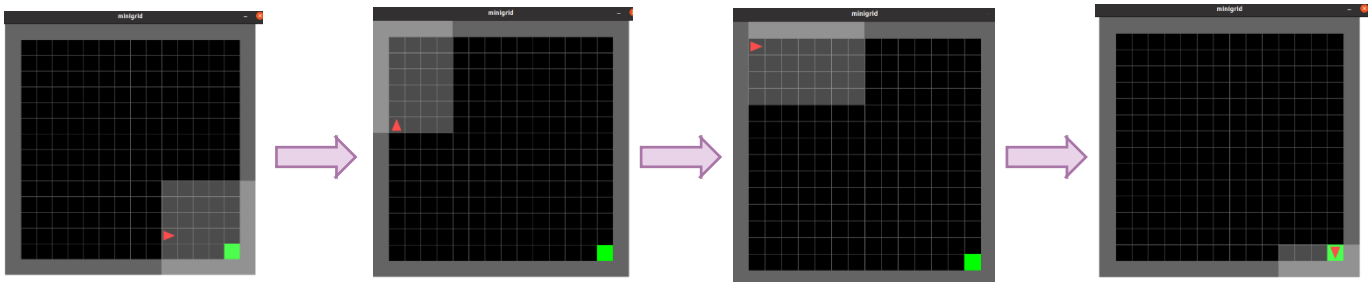
Après cet entraînement, le modèle fonctionnait comme convenu. L'agent parvient bien à la case verte.

Environnement Minigrid



S’il on déplace la position d’origine, l’agent parvient aussi à rejoindre l’objectif. Mais un phénomène étrange apparaît : L’agent veut absolument retourner à sa position d’origine qu’il avait lorsqu’il s’entrainait. Cela suggère qu'au lieu d'apprendre à naviguer en fonction de ce qu'il voit, l'agent a probablement mémorisé une séquence d'actions spécifique associée à son point de départ.

Etapes lorsque la position d’origine est modifiée



Ce comportement peut être dû à un manque de généralisation dans les observations visuelles ou à un entraînement trop centré sur une position initiale fixe. C’est-à-dire qu’à force de démarrer de la même case, la Q-valeurs de cette case à fortement augmenté.

Pour éviter cela, il serait utile de randomiser les positions de départ pendant l'entraînement afin de forcer l'agent à mieux explorer et utiliser ses observations visuelles pour atteindre la récompense.

5.5 COMMENTAIRE SUR PPO

PPO est un algorithme très efficace pour la résolution de Minigrid. Avec la vectorisation des environnements, il est possible d’entraîné le modèle pour qu’il soit encore plus performant. Il paraît dans un premier temps que les algorithmes de Policy Gradient semblent être plus performant que les algorithmes DQN. En revanche il convient de noter que le DQN étudié à été créer par moi-même et qu’il sûrement loin d’être optimal.

En conclusion PPO semble être une bonne architecture d’algorithme d’Apprentissage par Renforcement pour la suite du stage. Je pense notamment pour l’utilisation des drones où comme pour Minigrid, l’environnement n’est pas évident à manipuler.

5.5 SAC POUR LES ESPACES D'ACTION CONTINUS

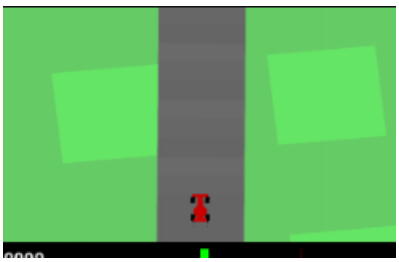
J'ai également étudié l'algorithme **SAC** pour des problèmes avec des espaces d'action continus.

6. ENVIRONNEMENT D'APPRENTISSAGE

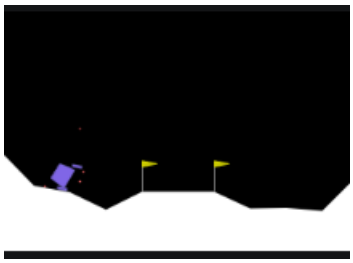
6.1 GYMNASIUM

Gymnasium est une bibliothèque Python utilisée pour développer et tester des environnements de reinforcement learning (RL). C'est une suite standardisée qui permet aux chercheurs et aux développeurs de créer des agents d'apprentissage automatique capables d'interagir avec des environnements variés. Elle fournit une interface commune pour faciliter l'intégration avec des algorithmes de RL, notamment en offrant des outils pour la gestion des actions, des observations et des récompenses.

Il existe de nombreux environnements. Voici 2 que j'ai pu utiliser :



Racing Car



Lunar Landing

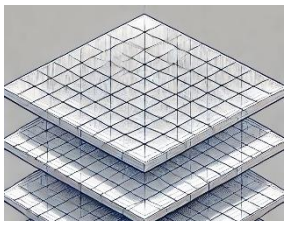
Quelques-unes des commandes importantes sont rappelées ci-dessous :

RECAPITULATIF DES COMMANDES DE GYMNASIUM :

1. **Installation** : `pip install gymnasium`
2. **Importation** : `import gymnasium as gym`
3. **Création de l'environnement** : `env = gym.make('CartPole-v1',render_mode= « human » ou « rgb-array »)`
4. **Réinitialisation** : `state, info = env.reset()`
5. **Exécution d'une action** : `new_state, reward, done, truncated, info = env.step(action)`
6. **Rendu visuel** : `env.render()`
7. **Obtenir des infos sur l'espace d'actions et d'observations** :
   - `env.action_space`
   - `env.observation_space`
8. **Fermer l'environnement** : `env.close()`

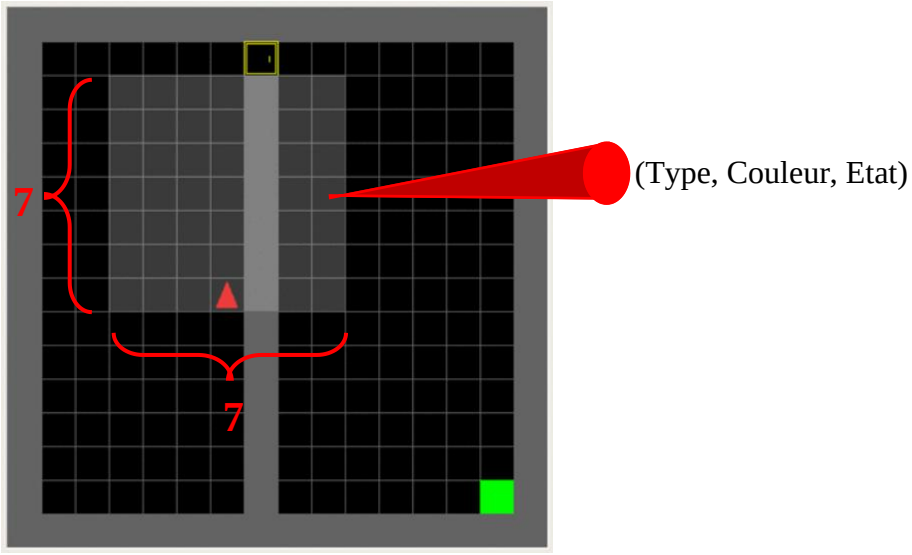
6.2 MINIGRID

Minigrid est un environnement au format Gymnasium mais les données d'états sont plus complexes. L'état est une matrice de 7x7x3. C'est une grille de 7x7 cases où chaque case est représenté par 3 valeurs (Type, Couleur, Etat).



Matrice 7x7x3

Environnement Minigrid



Ici par exemple, il y a des cases (Vide, Noire, Ouverte), des cases (Mur, Gris, Bloqué), et des cases (Porte, jaune, Fermée).

Voici l'action space de minigrid :

Num	Name	Action
0	left	Turn left
1	right	Turn right
2	forward	Move forward
3	pickup	Unused
4	drop	Unused
5	toggle	Unused
6	done	Unused

6.3 LES WRAPPERS

Un wrapper prend un environnement existant et l'entoure de nouvelles fonctionnalités. Gymnasium propose plusieurs types de wrappers, par exemple :

- **Observation Wrappers** : Pour transformer les observations (ex : redimensionner les images, passer l'observation en FullyObservable pour voir tout l'environnement et non que ce qu'il y a devant l'agent).

  Exemple : MinigridFeaturesExtractor pour adapter l'environnement Minigrid au réseau de neuronnes CNN de PPO de StableBaselines3.

- **Action Wrappers** : Pour modifier les actions (ex : passer d'un espace continu à discret).

  Exemple : random_start_wrapper pour passer l'etat d'origine de l'agent en aléatoire.

- **Reward Wrappers** : Pour ajuster les récompenses (ex : ajouter une pénalité).

- **Environment Wrappers** : Pour des fonctionnalités plus globales comme limiter le nombre de pas dans un épisode ou enregistrer une vidéo.

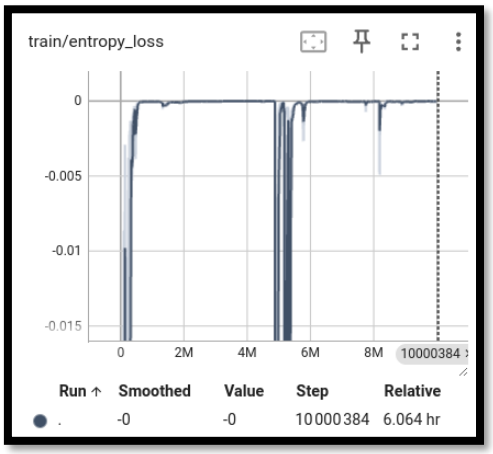
  Exemple : RewardLoggerCallback pour enregistrer le nombre de réussite par épisode.

7. OUTILS POUR L'ANALYSE

7.1 TENSORBOARD

TensorBoard permet de visualiser en temps réel les métriques d'entraînement, comme les pertes et les récompenses cumulées. Cela aide à évaluer la progression de l'agent et ajuster les hyperparamètres si nécessaire.

Courbe de l'entropy_loss fournie par TensorBoard



Cette courbe correspond à l'entropie de modèle PPO sur Minigrid pour 10 millions d'actions effectuées. On voit que cela a mis 6h.

7.2 MLFLOW

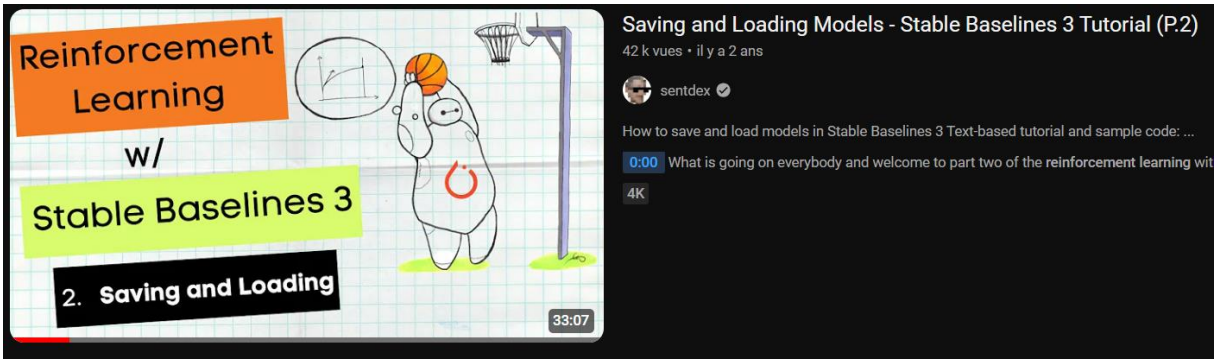
MLflow, de son côté, te permet de suivre et comparer les expériences en sauvegardant les runs, hyperparamètres, et modèles. Cela facilite la gestion, la reproductibilité et l'analyse des différents essais.

8. CONCLUSION ET PERSPECTIVES

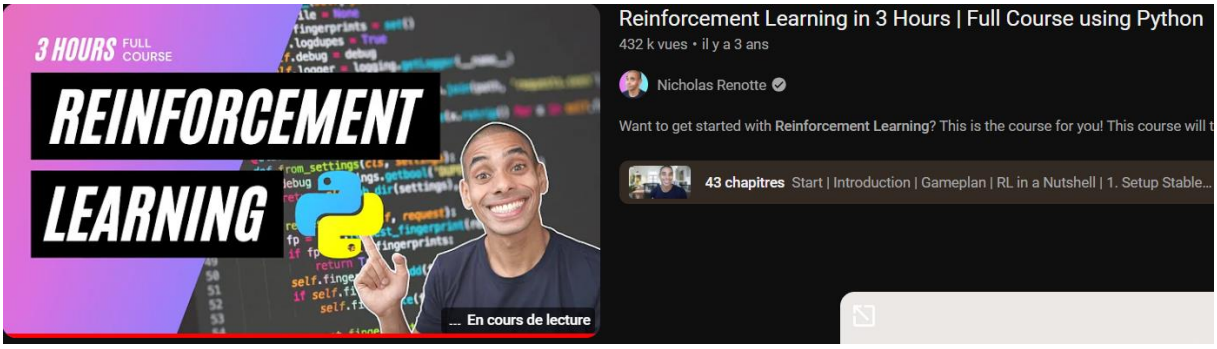
9. LES RESSOURCES

J’ai utilisé de nombreuses ressources pour la réalisation de mes premières recherches :

Pour l’utilisation de la bibliothèque StableBaselines3

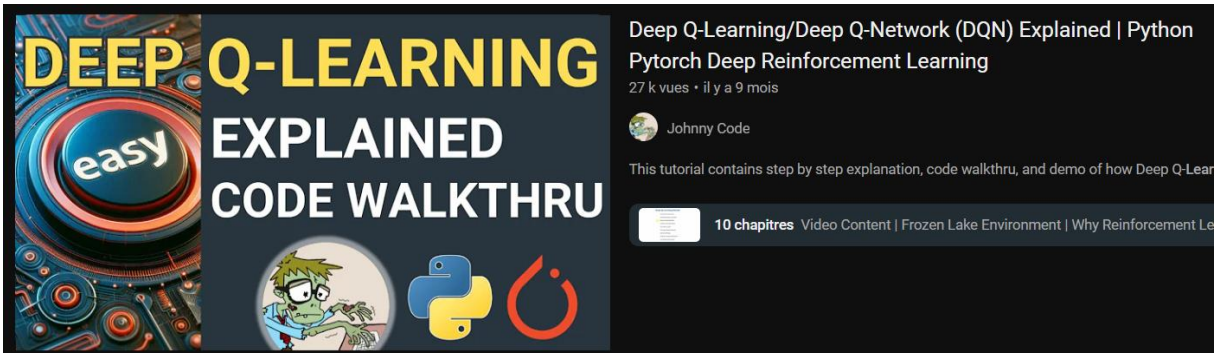


https://github.com/TheAILearner/Snake-Game-using-OpenCV-Python/blob/master/snake_game_using_opencv.ipynb



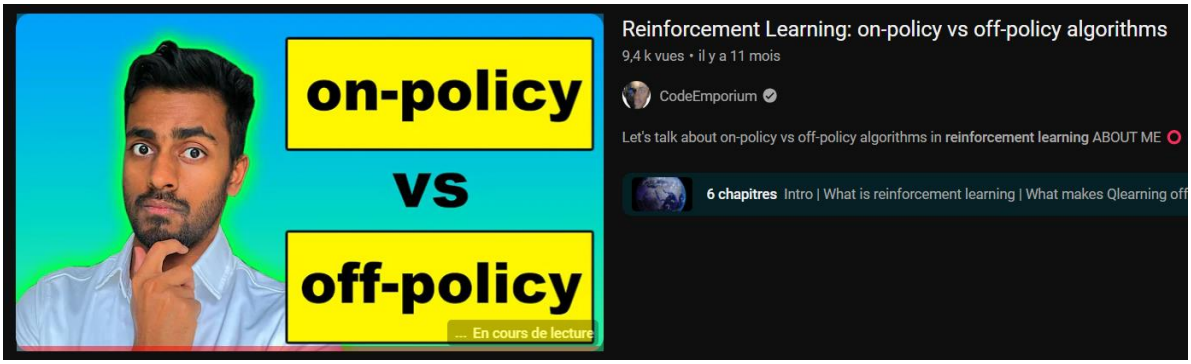
https://github.com/nicknochnack/ReinforcementLearningCourse

Pour la création d’un DQN

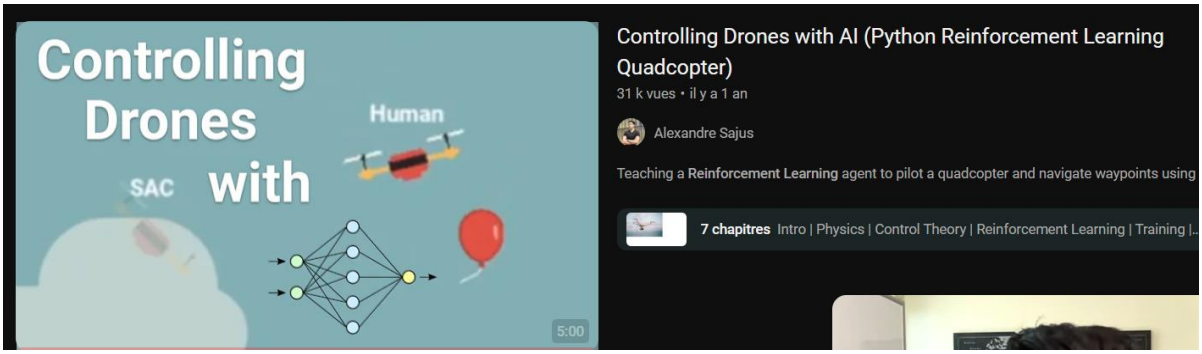


https://github.com/johnnycode8/gym_solutions

Pour la compréhension de l'Apprentissage par Renforcement



Pour le DQN appliqué aux drones



https://github.com/AlexandreSajus/Quadcopter-AI