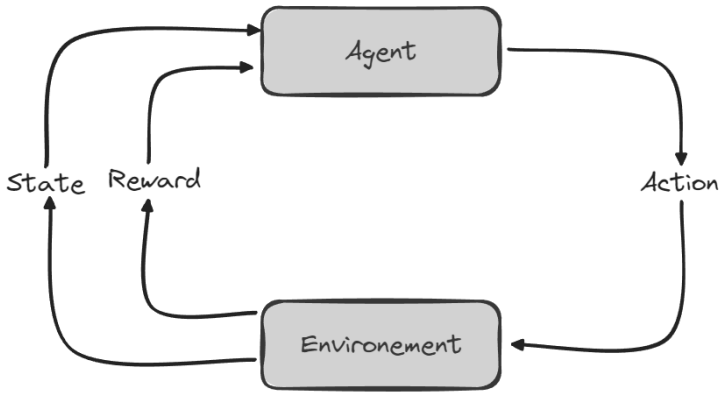
Apprentissage par renforcement

Dans ce chapitre, vous allez apprendre à développer des modèles d'apprentissage par renforcement. En machine learning, l'apprentissage par renforcement est une branche très différente de celles de l'apprentissage supervisé et non supervisé. En effet, dans cette discipline, on ne fournit pas de données (X, y) à notre machine, mais on la programme afin qu'elle puisse générer ses propres expériences et apprendre de ses erreurs.

L'idée centrale est donc de créer un **agent**, libre d'entreprendre des **actions** dans un **environnement**. Ces actions modifient l'**état** (*state*) de l'agent, et ce changement d'état s'accompagne d'une **récompense** (*reward*). Pour l'agent, le but du jeu est de maximiser ses récompenses, ce qui le pousse à apprendre quelles actions effectuer pour obtenir le plus de récompenses.



Cette discipline est très utilisée pour les applications suivantes :

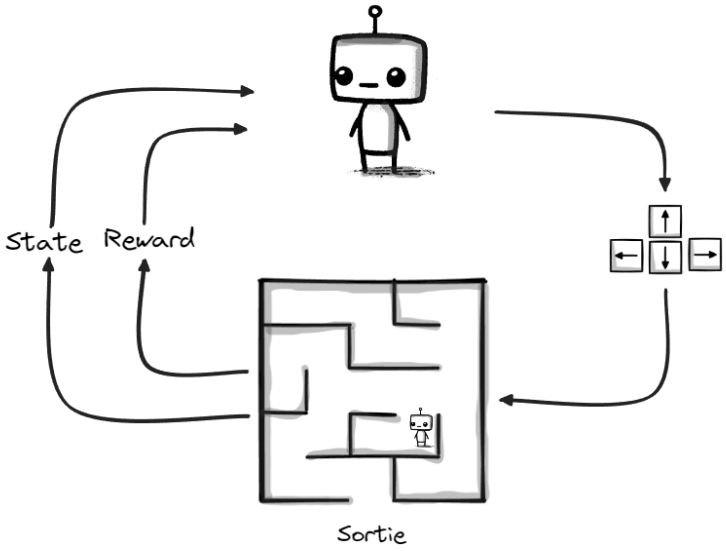
- Robotique
- Véhicules autonomes (voitures, drones)
- Trading
- Algorithmes de prise de décision

Mise en situation

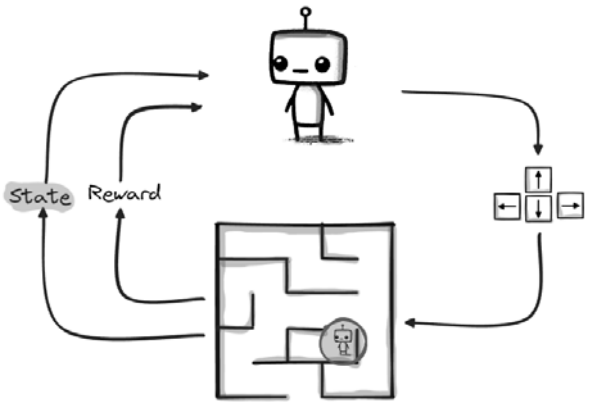
Imaginez : vous programmez un petit robot pour qu'il apprenne à sortir d'un labyrinthe le plus rapidement possible. Pour rendre les choses plus compliquées, le labyrinthe est généré aléatoirement.

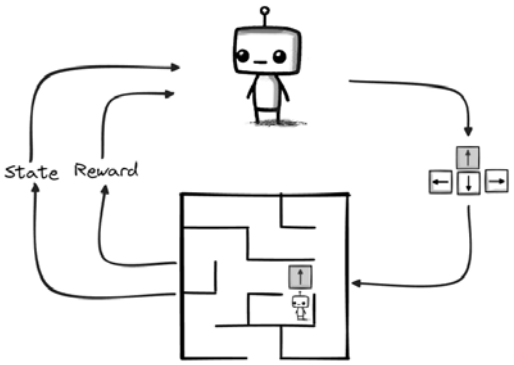
Bien sûr, nous pourrions pour cela construire un algorithme de recherche du chemin le plus court basé sur la théorie des graphes, mais cela reviendrait à coder explicitement le comportement de la machine. Et c'est là tout l'inverse du machine learning. À la place, nous voulons programmer la machine pour qu'elle développe elle-même la stratégie lui permettant de quitter le labyrinthe.

Nous allons donc utiliser une approche d'apprentissage par renforcement. Notre agent sera le petit robot, et le labyrinthe l'environnement dans lequel il évolue.



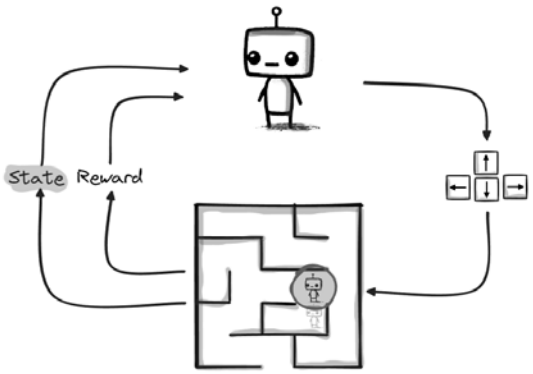
L'état (*State*) correspond à la position de l'agent dans le labyrinthe, et les actions à ses déplacements possibles (haut, bas, droite, gauche)



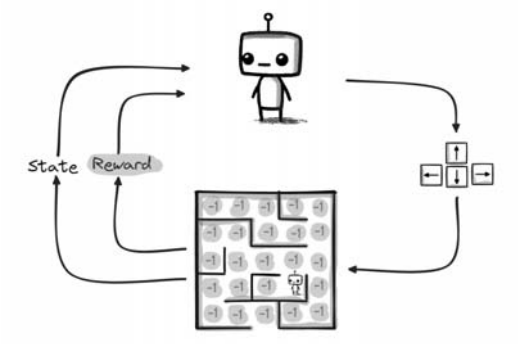


Depuis cet état, l'agent peut choisir une action, par exemple "aller en haut".

En effectuant cette action, l'agent change son état, ce qui s'accompagne d'une récompense. Ici, nous attribuons à la machine un score de -1 point.



Quoi ?! Mais l'agent s'est pourtant rapproché de la sortie, pourquoi le sanctionner avec un score négatif ? C'est vrai, cependant, rappelez-vous de notre objectif : "trouver la sortie le plus vite possible".



Ainsi, nous infligeons un point négatif chaque fois que la machine effectue un déplacement. Son but étant de maximiser son score final, elle cherchera donc le moyen le plus rapide de rejoindre la sortie, préférant marquer -10 points plutôt que -15. Ne vous en faites pas, la machine ne sera pas triste ☺

En résumé, notre objectif est d'apprendre à la machine, *quelle action effectuer lorsqu'elle se situe dans un état donné*. Autrement dit, nous cherchons à développer une fonction $f(s) = a$ qui prend en entrée l'état s et produit une action a. Cette fonction est couramment nommée ***politique d'action***.

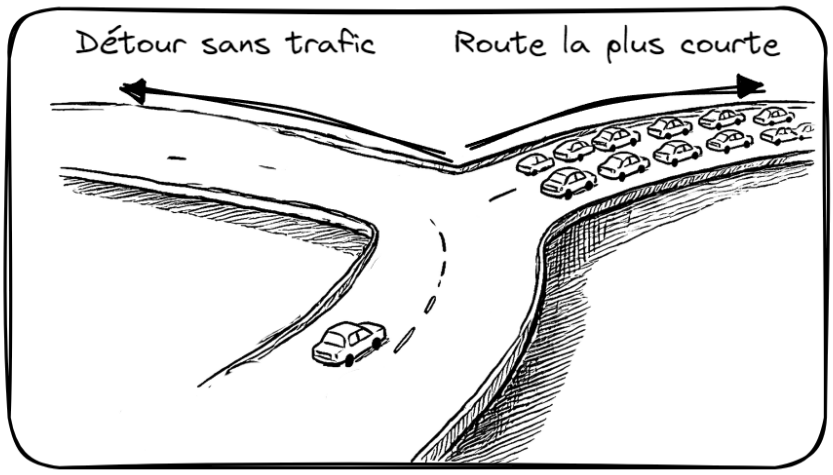
En apprentissage par renforcement, il existe plusieurs méthodes permettant d'apprendre à développer cette politique d'action. Celle que nous allons voir ici est l'approche appelée **Q-Learning**.

Le fonctionnement du Q-Learning

Le Q-Learning consiste à apprendre quelle est la **valeur** d'une action A dans un état S donné.

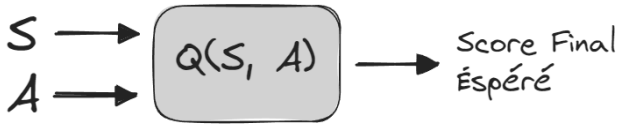
Par exemple, je mets habituellement 30 minutes en voiture pour aller à mon travail. Aujourd'hui, cette route est encombrée par le trafic. J'aperçois cependant qu'une autre route, un peu plus longue, est également possible. Ainsi, deux choix s'offrent à moi : l'action de tourner à gauche ou l'action de tourner à droite.

Pour choisir laquelle emprunter, je vais évaluer la valeur de chacune. À mon avis, quel sera le temps de trajet total si je tourne à gauche ? Et si je tourne à droite ?

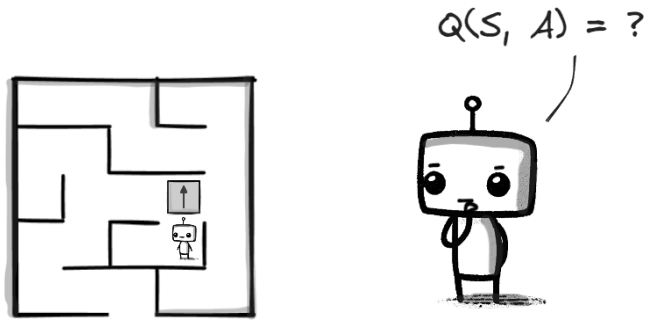


Voilà le but du Q-Learning : apprendre à prédire la valeur de mes actions, en tenant compte de la situation dans laquelle je me trouve.

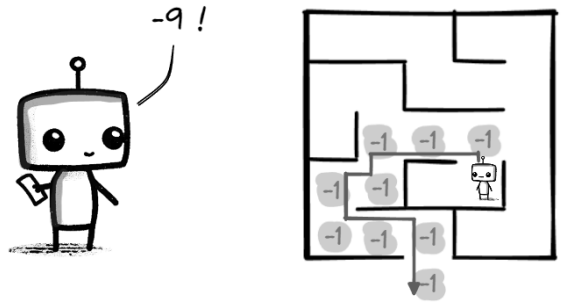
Plus formellement, il s'agit donc d'apprendre une fonction Q qui prend en entrée un état S et une action A, et qui prédit le **score final** que l'on peut espérer obtenir si tout se passe pour le mieux par la suite.



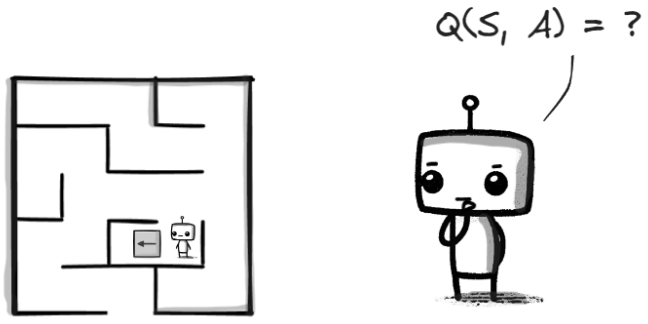
Prenons un exemple : dans la position actuelle, quel est le meilleur score que la machine puisse espérer obtenir si elle choisit l'action "allez en haut" ?



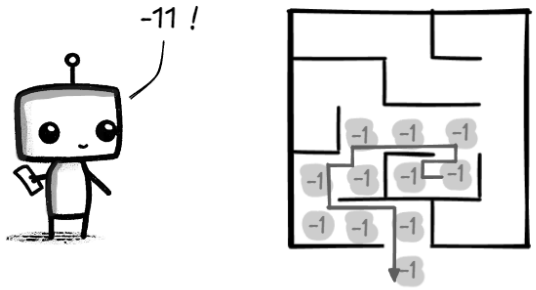
En tenant compte du fait que la machine perd un point pour chaque déplacement, le meilleur score qu'elle puisse espérer obtenir est -9.



Et dans le cas où la machine choisit d'aller à gauche ?



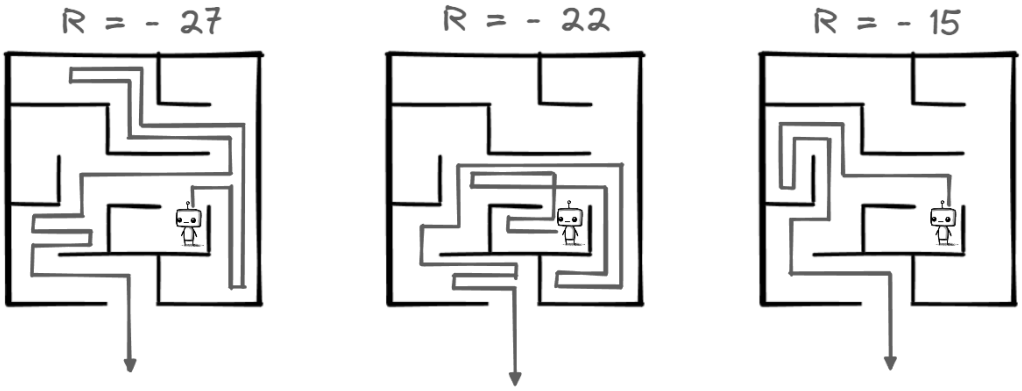
... Cette fois-ci, la réponse est -11, car la machine devra revenir sur son chemin. Ainsi, il est préférable de choisir l'action "aller en haut" car elle donne un score final plus élevé (-9) que l'action "aller à gauche" (-11)



Il reste maintenant une question : Comment apprendre cette fonction Q ?

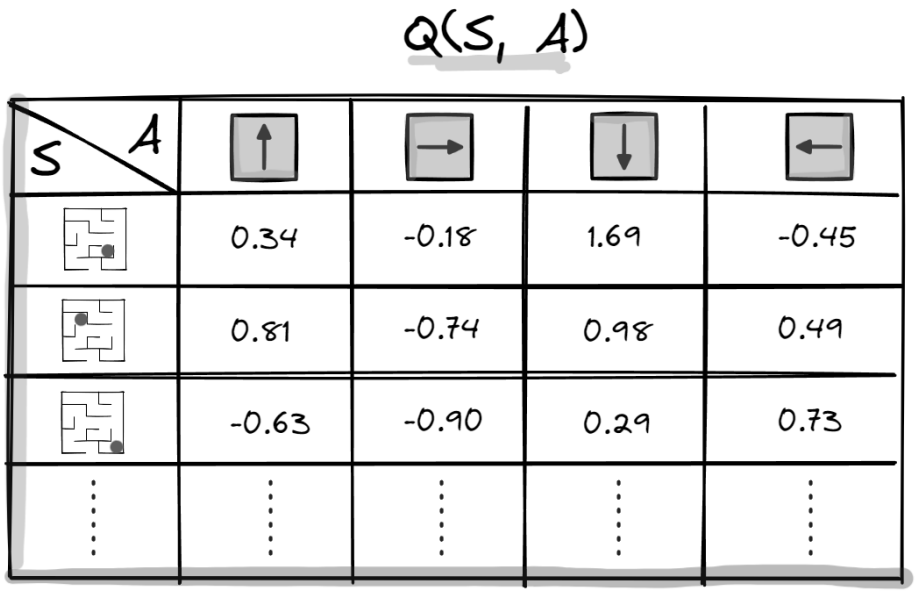
L'apprentissage de la fonction Q

Pour apprendre à prédire la valeur de ses actions, la machine doit explorer son environnement en s'y déplaçant au hasard, afin de générer des données (S, A, R) (état, action, récompense).

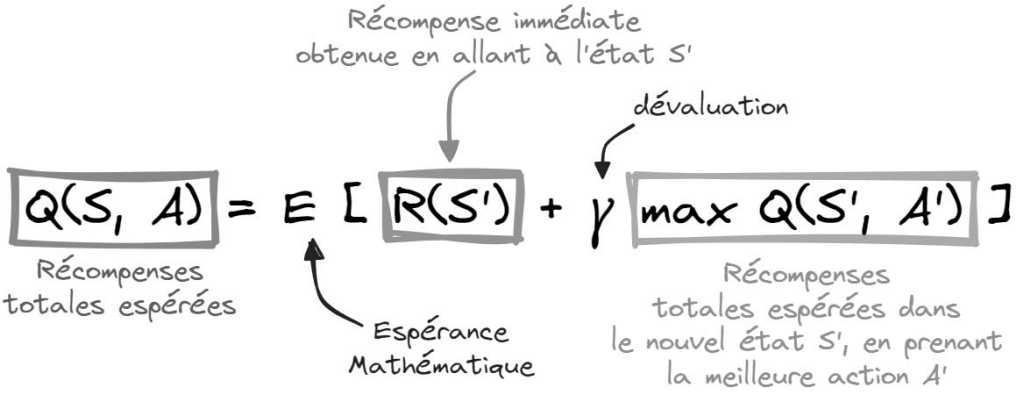


Ces données sont ensuite utilisées pour mettre à jour un tableau Q(S, A) qui informe la machine des récompenses qu'elle est censée obtenir à l'avenir en choisissant telle ou telle action dans l'état présent.

Au début de son apprentissage, ce tableau est rempli de valeurs aléatoires, comme illustré ci-dessous.



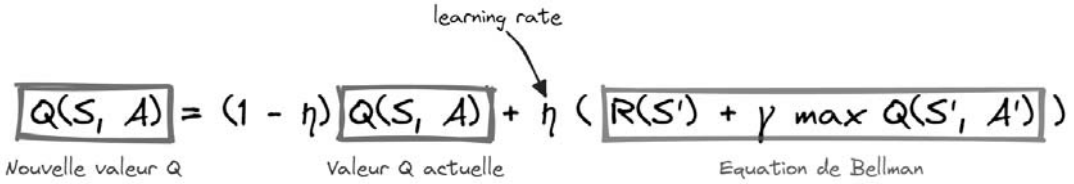
Une formule, connue sous le nom de **l'équation de Bellman**, permet d'exprimer le score total que l'on peut espérer obtenir à l'avenir. Ce score est égal à la récompense obtenue immédiatement dans l'état S, à laquelle on ajoute le même score pour l'état suivant S', si l'on choisit la meilleure action possible dans ce nouvel état.



Pour bien comprendre, reprenons l'analogie avec la voiture.

L'équation de Bellman vous permet d'estimer que la valeur de tourner à gauche vous donne la récompense immédiate d'éviter le trafic, en plus de la valeur que vous allez obtenir dans votre prochain état. Si vous tombez sur des bouchons 1 km plus loin, le fait de tourner à gauche n'était peut-être pas la meilleure option.

À chaque fois que la machine réalise une action, elle peut mettre à jour sa table Q, car elle reçoit une récompense associée à son changement d'état. D'après la formule de Bellman, cette récompense R(S') peut être utilisée pour recalculer la valeur de Q(S,A) en se basant sur sa valeur actuelle et cette nouvelle donnée :



Exploration et exploitation

Cependant, dès que la table-Q commence à donner de bons résultats, la machine peut tomber dans le piège de suivre une politique d'action limitée, ignorant au passage les autres possibilités encore inexplorées.

De la même manière qu'après avoir trouvé un bon restaurant dans votre ville, vous choisissez systématiquement d'aller dîner dans ce restaurant car vous pensez qu'il n'y en a pas de meilleurs.

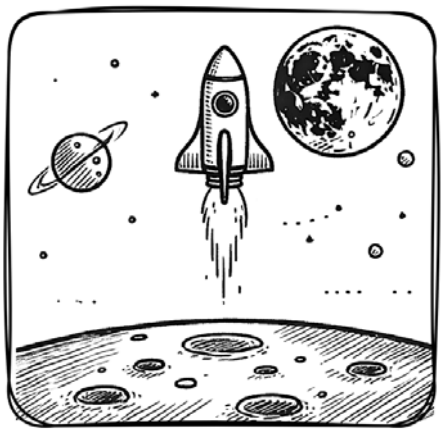
Pour éviter ce phénomène, on définit dans l'algorithme du Q-learning une probabilité **epsilon** qui pousse la machine à continuer **d'explorer** l'environnement des possibles, plutôt que de simplement **exploiter** sa politique d'action déjà apprise.

C'est là le dernier élément qui compose l'algorithme du Q-Learning.

Pour finir ce chapitre, je vous propose donc de passer à l'action avec une implémentation des plus amusantes ☺

Implémentation Python

Dans cette partie, nous allons développer un algorithme de Q-Learning dont le but sera d'apprendre à poser un vaisseau spatial sur la lune sans qu'il ne s'écrase.



Pour cela,nous aurons besoin d'un environnement (la lune) ainsi que d'un agent (le vaisseau spatial). Il existe de nombreuses librairies permettant de générer de tels environnements, l'une des plus populaires étant la librairie **gymnasium** de OpenAI.

Avant de commencer, il vous faudra donc créer un nouvel **environnement virtuel** pour y installer le package *gymnasium* dans sa version 0.29.1 ainsi que son extension *gymnasium[box2d]*.

Si vous rencontrez des difficultés lors de l'installation, les lignes de commandes suivantes devraient permettre de tout installer pour la plupart des utilisateurs :

1. pip install wheel setuptools pip –upgrade
2. pip install swig
3. pip install gymnasium==0.29.1
4. pip install gymnasium[box2d]

Une fois que vos packages sont bien installés, le code ci-dessous devrait fonctionner et vous montrer un joli petit vaisseau spatial.

```
import gymnasium as gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)

for _ in range(1000):
   action = env.action_space.sample()  # this is where you would insert your policy
   observation, reward, terminated, truncated, info = env.step(action)

   if terminated or truncated:
      observation, info = env.reset()

env.close()
```

Dans ce projet, le vaisseau spatial dispose de **quatre actions** possibles :

- Ne rien faire
- Activer le réacteur du bas, afin de ralentir sa descente
- Activer le réacteur de gauche, afin de s'incliner à droite
- Activer le réacteur de droite, afin de s'incliner à gauche

Son **état** lui est décrit pas sa position dans l'espace, ainsi que sa vitesse, son angle, et son contact avec le sol.

Pour finir, le système de **récompenses** est défini par plusieurs règles, parmi lesquelles :

- Le vaisseau gagne 100 points s'il réussit à se poser sans dégâts
- Le vaisseau perd 100 points s'il s'écrase au sol (ne se pose pas sur ses pieds)
- Le vaisseau perd des points chaque fois qu'il active un moteur

Nous allons donc commencer par initialiser notre environnement lunaire, ainsi que les différents hyper-paramètres de notre modèle : le *learning rate*, le facteur de dévalorisation, la valeur epsilon, et le nombre d'épisodes que la machine va jouer durant son apprentissage.

```
env = gym.make("LunarLander-v2", render_mode="human")

# hyper-parametres
alpha = 0.1
gamma = 0.99
epsilon = 1.0
epsilon_decay = 0.995
epsilon_min = 0.01
episodes = 1000
max_steps = 1000
```

Tout comme notre petit robot qui se déplaçait dans le labyrinthe d'une case à l'autre, nous allons considérer que le vaisseau spatial se déplace sur une grille de coordonnées. C'est pourquoi nous allons découper l'espace des états du vaisseau spatial en plusieurs parties, afin de regrouper tous ces états dans la table-Q (initialement remplie de 0).

```
state_bins = [np.linspace(-1.0, 1.0, 10) for _ in range(env.observation_space.shape
    ↪ [0])]
n_bins = tuple(len(bins) + 1 for bins in state_bins)
q_table = np.zeros(n_bins + (env.action_space.n,))

def discretize_state(state):
    return tuple(np.digitize(state[i], state_bins[i]) for i in range(len(state)))
```

Ensuite, nous allons créer une boucle pour entraîner notre vaisseau spatial à se poser sans dégât. Cette boucle comprendra 1000 épisodes, et chaque épisode comprend au maximum 1000 pas.

À chaque pas, le vaisseau spatial "lancer un dé", et en fonction du résultat, il choisira soit **d'explorer** son environnement en choisissant une action au hasard, soit **d'exploiter** sa politique d'action en choisissant l'action qui, selon lui, a la plus grande valeur (cette valeur est définie par la table-Q)

Une fois son action effectuée, le vaisseau recevra une récompense, qu'il utilisera pour mettre à jour la table-Q en appliquant la formule de mise à jour et l'équation de Bellman.

```
for episode in range(episodes):
    state, _ = env.reset(seed=42)
    state = discretize_state(state)
    total_reward = 0

    for step in range(max_steps):
        # Epsilon-greedy policy
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample()  # Exploration
        else:
            action = np.argmax(q_table[state])  # Exploitation

        next_state, reward, terminated, truncated, _ = env.step(action)
        next_state = discretize_state(next_state)


        old_value = q_table[state][action] ] # valeur Q actuelle
        next_max = np.max(q_table[next_state]) # valeur Q max de l'état S'

        # mise a jour de la table q avec la formule de Bellman
        new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
        q_table[state][action] = new_value

        state = next_state
        total_reward += reward

        if terminated or truncated:
            break

    # Reduction de Epsilon
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

    print(f"Episode {episode + 1}, Total Reward: {total_reward}")

env.close()
```

Bravo Commandant ! Mission accomplie ! ☺