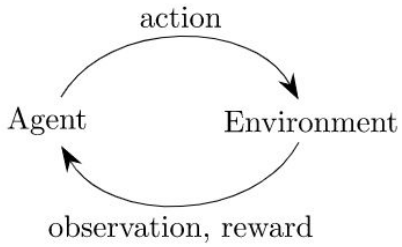
Deep Learning LunarLander-v2 - DQN

Carl HATOUM

Introduction

LunarLander est un “environnement” développé par *OpenAI*. Concrètement, nous avons un **agent**, le module lunaire, qui évolue dans un **environnement**, et en fonction de ses **actions**, reçoit une **récompense**. C’est avec ce paradigme que l’on conçoit le reinforcement learning, ou apprentissage par renforcement.



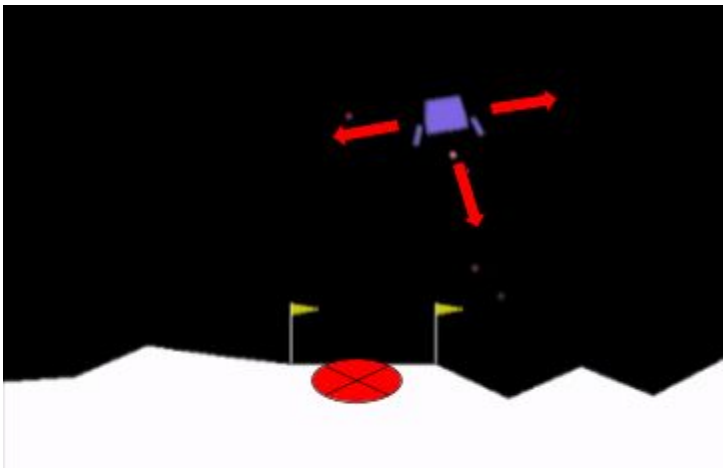
Boucle agent-environnement

Le module lunaire interagit avec un environnement. Pendant une séquence d’états et actions associées, se déroule ce qu’on appelle un épisode.
Ici, l’objectif à atteindre est d'atterrir sur la cible. Pour cela, il est doté des quatres actions, discrètes dans ce cas, possibles :

- 1 Ne rien faire
- 2 Moteur gauche
- 3 Moteur du bas
- 4 Moteur droit

En fonction de ses actions, l’agent reçoit une **récompense**, qui est un réel, de son environnement. Par exemple, déclencher le moteur du bas reçoit -0.3, l’épisode se termine lorsque le module se crashe, ou atterrit sur la cible, recevant respectivement -100 ou 100. Ces récompenses, ont pour objectif d'être maximisées, et nous aideront à mesurer la pertinence d’une **politique**, qui est un certain enchaînement d’actions en fonction des états, pour en avoir une optimale.

Ici, notre but est d'alunir de la meilleure manière, donc avec la politique la plus optimale. Pour le savoir, il va falloir explorer différentes politiques, regarder leur récompense associée, et en apprendre.



Le module lunaire, la cible, et les actions possibles

Apprentissage

En Q-Learning, l'objectif est d'apprendre la fonction *Q*, qui pour deux paramètres action *a* et état *s*, donne la meilleure qualité : *Q(s, a).*
Ainsi, la politique optimale suit les actions qui maximise la fonction *Q*. Autrement dit, pour chaque étape *s*, choisir l'action *a* qui va maximiser la récompense à la fin du jeu. Pour cela, nous définissons une *target*, qui découle de l'équation de *Bellman* : la qualité d'une action *a* en l'état *s* est définie par la somme de sa récompense, et celle de la récompense maximale des futur états, pondérée par *gamma*.

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

Nous allons approximer la fonction Q de manière itérative, en se basant sur cette équation.
Une première manière d'apprendre cette fonction, est de construire une *Q-table*, qui contient toutes les combinaisons d'actions et d'état, avec la qualité associée.

Le nombre total de combinaisons d'actions est très grands, par conséquent difficile à explorer avec une *Q-table*. Au lieu de cela, nous utiliserons un algorithme *Deep Q Network*, et donc un réseaux de neurones pour approximer la fonction *Q*.

Nous commençons par instancier un agent Q-Learning, celui-ci possède un réseaux de neurones, ainsi qu'une mémoire.

Cet agent va passer sur plusieurs épisodes, qui est un enchaînement d'actions, jusqu'à la fin (alunissage, ou crash). À chaque épisode, l'environnement est réinitialisé.

Pour chaque étape dans un épisode, l'agent va choisir une action dite *epsilon-greedy* :

On tire un nombre n, aléatoirement entre 0 et 1.

- Si n est inférieur à *epsilon*, une choisit une action aléatoire
- Sinon, on choisit celle avec la plus grande récompense

Au départ, *epsilon* est égal à 1, puis est décrémenté. Au début, nous allons donc choisir uniformément toutes les actions pour toutes les explorer et favoriser leur diversité, au fur à mesure du temps epsilon est plus petit, et favorise les actions qui ont une grande récompense. Cependant, *epsilon* ne sera jamais nul, nous aurons ainsi toujours une petite probabilité de choisir une action aléatoire, évitant ainsi de tomber dans un optimum local. L'exploration *epsilon-greedy* permet de faire un compromis entre exploration, et exploitation.

```
def choose_action(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_space)
    act_values = self.model.predict(state)
    return np.argmax(act_values[0])
```

Pour chaque étape dans un épisode, nous enregistrons *state, action, reward, next_state, done* (booléen qui indique si l'épisode est fini) dans notre mémoire.

Dans le reinforcement learning, se pose le problème de l'instabilité de la *target* : au fur et à mesure de son apprentissage, celle-ci change constamment. Un autre problème vient dans l'entraînement du réseaux de neurones : en utilisant les derniers vecteurs de transition comme base d'apprentissage, ils s'avèrent non indépendants et fortements corrélés. Des transitions non i.i.d ne sont pas adaptées pour un apprentissage supervisé.

L'*Experience replay* va permettre de stabiliser l'apprentissage. Comme expliqué précédemment, nous allons, pour chaque étape d'un épisode, enregistrer le vecteur *[[state], action, reward, [next_state], done]* dans notre mémoire.

Lorsque notre mémoire contient suffisamment d'éléments, nous allons aléatoirement en sélectionner dans un *Mini-Batch*,qui servira à entraîner le réseau de neurones. En les choisissant au hasard et non à la suite, nous réduisons leur corrélation.

Pour chaque transition dans le *Mini-Batch*, nous calculons sa target égale à la somme de la reward, et à la meilleure action future donnée par la fonction *Q*.

Q, estimée par un réseau de neurones est initialisé avec des poids aléatoires, donc avec une mauvaise estimation. Nous fixons le réseau pendant tout ce temps, pour stabiliser la *target*, puis nous entraînons le réseau à apprendre les bon poids en réduisant la fonction de perte entre la *target*, et la prédiction.

Le réseau profond est implémenté à partir du modèle *sequential* de *Keras*. Il a en entrée l'espace d'états possible dans l'environnement. Puis deux couches cachées de 128 poids *fully connected*, avec des fonction d'activation *ReLu* pour éliminer les valeurs négatives. En sortie, nous avons les actions possibles. La fonction de perte est l'erreur quadratique moyenne (*mean square error*) et la descente du gradient se fait avec *ADAM*, et un *learning rate* de 1e-3.

En résumé, nous avons conçu un réseau de neurones qui prends en entrée, l'espace d'états, et donnera, une fois le réseau entraîné, la meilleure action.

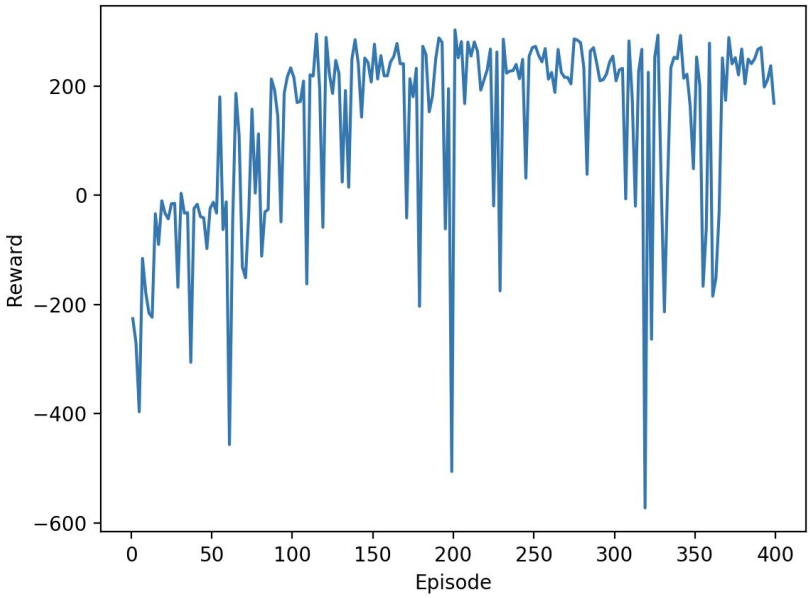
```
def build_model(self):

    model = Sequential()
    model.add(Dense(128, input_dim=self.state_space, activation=relu)) #entrée : state_space
    model.add(Dense(128, activation=relu))
    model.add(Dense(self.action_space, activation=linear)) #sortie action_space
    model.compile(loss='mse', optimizer=adam())
    return model
```

Entraînement et résultats

Pour un épisode nous :

- générons un nouvel environnement
- exécutons des actions *epsilon-greedy*
- enregistrons le vecteur [s, a, r, s'] dans la mémoire
- Si le nombre d'éléments dans la mémoire dépasse la taille du Mini-Batch, procédons à l'entraînement

Nous entraînons l'agent Q-Learning sur 400 épisodes. Chaque épisode est un nouvel environnement, généré aléatoirement, avec des conditions différentes, une politique différente, avec une récompense différente, guidée par l'apprentissage.



Récompense totale en fonction de l'avancement des épisodes

Sur les 100 derniers épisodes, nous avons pour les récompenses :

-Une moyenne de **165.4**
-Une médiane de **229.3**

Plusieurs pistes sont possibles pour améliorer la récompense.
Modifier le réseau de neurones :

- Ajouter des couches cachées supplémentaires
- Modifier le nombre de poids
- Tester différentes fonctions d'activation

Modifier les différents paramètres :

- La taille du *Mini-Batch*
- La taille de la mémoire
- Gamma: le *discount factor* de l'équation de *Bellman*. S'il est égal à 0, l'agent ne s'intéressera qu'au gain immédiat (myopie), s'il est égal à 1 il accorde autant d'importance aux récompenses actuelles qu'à celles futures.

Références :

[1] Playing Atari with Deep Reinforcement Learning
https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[2] DEMYSTIFYING DEEP REINFORCEMENT LEARNING
https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

[3] Multiple Access Channel with Reinforcement Learning
https://colab.research.google.com/github/mgoutay/ml_course/blob/master/RL_exercise_solution.ipynb