Répondez dans un rapport PDF. Remettez une archive contenant le rapport PDF ainsi que le code Python associé aux différentes questions. Votre rapport doit indiquer clairement quel fichier Python doit être exécuté pour répliquer les résultats présentés dans votre rapport.
Remise : Mercredi 25 novembre à 17h00

Les librairies requises pour ce TP se trouvent dans un fichier `requirements.txt` fourni avec cet énoncé. Vous pouvez les installer avec la commande *pip install -r requirements.txt*. Vous devez aussi suivre les instructions ici pour installer la librairie *box2d*. Enfin, ce TP fait un usage exhaustif des environnements fournis par OpenAI gym. Pour plus d'informations sur ces derniers, référez vous à leur documentation.

1. **Deep Q-learning**

   Pour ce numéro, nous allons nous familiariser avec plusieurs des concepts clés derrière le Deep Q-learning. Cet algorithme d'apprentissage est théoriquement applicable à n'importe quel problème d'apprentissage par renforcement, mais son entraînement n'en demeure pas moins notoirement capricieux. Nous explorerons et justifierons diverses des astuces utilisées pour tenter de stabiliser son entraînement.

   Dans la plupart des problèmes du monde réel, il est irréaliste de penser pouvoir entreposer toutes les paires action-état d'un environnement pour utiliser les approches tabulaires. C'est pour cette raison que l'on approxime les valeurs de Q à partir d'une fonction paramétrée $q_\theta(s, a)$, où θ représente habituellement les poids d'un réseau de neurones. Dans ce contexte, on cherche alors à minimiser la fonction de perte

   $$\mathcal{L}(\theta) = \frac{1}{b}\sum_{t=1}^{b}\left(r_t + \gamma \max_{a' \in \mathcal{A}} q_\theta(s_{t+1}, a') - q_\theta(s_t, a_t)\right)^2,$$

   qui représente la moyenne empirique du *one-step lookahead target*.

   (a) (1 point) **État comme seule entrée.** La plupart du temps dans les environnements à actions discrètes, il est choisi d'avoir $q_\theta(s) \in \mathbb{R}^{|\mathcal{A}|}$. Dans cette optique, on prend donc seulement en entrée un état et on ressort les valeurs de Q pour toutes les actions à la fois. Quel est l'avantage principal à faire de la sorte comparativement à prendre en entrée une paire état-action ?

   (b) (1 point) **Réseau cible.** Une des astuces pour faciliter l'apprentissage utilisée en pratique est l'emploi d'un réseau cible q_{θ^-}. Ce réseau est utilisé pour le calcul du *one-step lookahead target* et donne la nouvelle fonction de perte:

   $$\mathcal{L}(\theta) = \frac{1}{b}\sum_{t=1}^{b}\left(r_t + \gamma \max_{a' \in \mathcal{A}} q_{\theta^-}(s_{t+1}, a') - q_\theta(s_t, a_t)\right)^2.$$

   Justifiez dans vos mots quelle est l'utilité de l'introduction d'un réseau cible par rapport à la stabilité de l'entraînement. Basez votre réponse sur le comportement indésirable qu'est l'*oubli catastrophique*, où un réseau de neurones qui se dirigeait vers de bons paramètres semble perdre tout son progrès de manière subite.

   (c) (1 point) Une des heuristiques utilisées pour déterminer le réseau cible est de faire une moyenne exponentielle mobile sur les paramètres θ selon un certain τ près de 0 et la mise à jour

   $$\theta^- = (1 - \tau)\theta^- + \tau\theta.$$

   Cette mise à jour est effectuée après chacune des mises à jours des poids θ.

   Expliquez quel est le dilemme fondamental sur le choix d'un bon τ. Basez votre analyse sur les cas particuliers $\tau = 0$ et $\tau = 1$ et l'objectif d'un apprentissage aussi stable que possible.

   (d) (1 point) **Replay buffer.** Au fur et à mesure d'un épisode, on entrepose les transitions (s_t, a_t, r_t, s_{t+1}) dans un *buffer* $\mathcal{D}$ avec une taille maximale fixée. La mise à jour des poids θ est faite en pigeant une *minibatch* de manière i.i.d. depuis $\mathcal{D}$.

Justifiez quels sont les avantages d'utiliser un *replay buffer*. Basez votre réponse sur la comparaison avec une *minibatch* constituée de trajectoires obtenues par les poids θ courants ou toute autre alternative que vous considérez envisageable.

(e) (1 point) **Apprentissage supervisé.** Dans un contexte de régression en apprentissage supervisé, on suppose qu'il existe une distribution $\mathcal{D}$ sur des paires entrées-sorties (x, y). On souhaite apprendre une approximation de fonction f_θ minimisant:

$$\mathcal{L}(\theta) = \mathbb{E}_{x,y\sim\mathcal{D}}\left[(f_\theta(x) - y)^2\right].$$

Cette fonction objectif semble très similaire à celle utilisée dans notre mise à jour de Deep Q-learning. En quoi les deux objectifs sont-ils différents ? *Indice:* Quelles sont les différences entre la distribution $\mathcal{D}$ ici et le *replay buffer* $\mathcal{D}$ présenté plus haut ?

(f) (3 points) Implémentez l'algorithme Deep Q-learning à partir du fichier `q1.py` et testez le sur l'environnement *LunarLander-v2*. Pour votre implémentation, nous vous demandons de:

- Ajouter un réseau cible. Le code de mise à jour des poids cibles pour un certain τ vous est founi. Fixez $\tau = 0.001$ pour commencer.
- Utiliser un algorithme ϵ-greedy pour la génération de trajectoires. Faites passer ϵ de 1.0 à 0.01 en le multipliant par 0.99 à la fin de chaque trajectoire.
- Employer un *replay buffer* pour entreposer les trajectoires. Fixez sa taille à 10^5 pour commencer.
- Utiliser une taille de *minibatch* de 64 avec un *learning rate* de 10^{-4}. Ces paramètres sont déjà présents par défaut dans le code fourni.
- Faire un entraînement du réseau à chaque N pas dans l'environnement. Pour commencer, utilisez $N = 4$.
- Vous assurer de bien gérer les états terminaux pour les *one step lookahead targets*. Si l'état s_{t+1} d'une transition est terminal, la cible pour la mise à jour devrait seulement être r_t.
- Utiliser $\gamma = 0.99$ pour vos expérimentations.

Rapportez dans un graphique l'évolution de la somme des récompenses et de la fonction de perte de votre modèle en fonction du nombre de pas effectués dans l'environnement. Discutez des différents défis d'implémentation auxquels vous avez fait face.

Note: Pour cet environnement, une trajectoire est considérée comme un succès si la somme des récompenses est de 200 ou plus. Avec les hyperparamètres suggérés, vous devriez atteindre au moins 200 la plupart du temps après vous être entraînés sur environ 600 trajectoires. Ce processus d'entraînement devrait prendre entre 5 et 10 minutes.

N.B. Pour cette question, une erreur s'était glissée dans l'implémentation du réseau cible fournie. Une nouvelle version mise à jour du fichier `q1.py` corrigeant l'erreur vous est fournie. Dans celle-ci, certains hyperparamètres ont été modifiés pour accélérer l'apprentissage avec la bonne version du code. Ces paramètres sont les suivants:

- On prend $\tau = 0.01$ (*au lieu de 0.001*).
- On multiplie ϵ par 0.9 à la fin de chaque trajectoire (*au lieu de 0.99*).
- On utilise une taille de *minibatch* de 32 (*au lieu de 64*).
- On utilise un *learning rate* de 0.0005 (*au lieu de* $10^{-4} = 0.0001$).
- On peut maintenant converger en près de 100 trajectoires et un peu plus d'une minute (*au lieu de 600 trajectoires et 5 minutes*).

2. **SARSA**(λ)

Soit l'environnement *MountainCar-v0* de OpenAI gym. Dans cet environnement, l'espace d'état est continu, les actions représentent l'accélération vers la gauche et la droite ainsi que l'absence d'accélération. La récompense est de -1 pour chaque pas de temps effectué, avec une fin d'épisode survenant soit quand 200 pas de temps se sont écoulés ou que la voiture a réussi à gravir la pente. On utilise alors généralement $\gamma = 1$ et on considère que l'environnement est résolu lorsqu'un agent obtient un retour cumulatif supérieur à -110 en moyenne sur un intervalle de 100 épisodes consécutifs.

(a) (1 point) **Exploration.** Lorsque l'on utilise des approches approximant les valeurs de Q sur cet environnement, il n'est pas nécessaire d'ajouter une exploration forcée (i.e. ϵ-greedy) à notre algorithme si on initialise $Q(s, a) = 0$ pour tous les états et les actions. Pourquoi ? À quel principe tiré des approches par bandit fait-on alors appel ?

(b) (5 points) Implémentez l'algorithme SARSA(λ) avec caractéristiques binaires et approximation de fonction linéaire tel qu'il est présenté au chapitre 12.7 du livre de Sutton. Utilisez le fichier `q2.py` comme point de départ. Votre implémentation doit inclure :

- Une représentation des caractéristiques tirée d'un carrelage sur l'espace joint des états et des actions. Cette portion est fournie avec des valeurs par défaut adéquates.
- Des traces obtenues par **remplacement**.
- Aucune exploration forcée (pas de ϵ-greedy). L'exploration sera faite naturellement par votre algorithme.

De plus, nous vous suggérons les hyperparamètres suivants par défaut : $\alpha = 0.1$ et $\lambda = 0.9$. Avec ces hyperparamètres, votre algorithme devrait résoudre en quelques secondes et moins de 500 épisodes l'environnement.

Rapportez la courbe de performance de votre implémentation pour différentes valeurs de λ. Testez minimalement $\lambda = 0, 0.9, 1$ et une autre valeur. Analysez l'impact qu'a le paramètre λ sur la stabilité de l'apprentissage et la qualité de la solution trouvée.