Devoir 2

Travail présenté à Audrey Durand dans le cadre du cours *IFT-7201 : Apprentissage par renforcement*

Travail réalisé par :
NAOUSSI SIJOU, Wilfried Armand - NI : 536 773 538
Jack Goodall - NI : 537 010 048
Hugo Meignen - NI : 537 016 570

Automne 2022

1 MountainCar-v0 OpenAI Gym

Etant donné $r_t = -1 \ \forall t$ et $\gamma = 1 \implies Q(s,a) \leq 0 \ \forall s,a$. Ainsi pour un état donné s, si une action a a déjà été jouée, $Q(s,a)0$. L' action sélectionnée est celle qui maximise Q. Soit 0 pour une action qui n'a pas encore été jouée pour un état donné s.

Dans les bandits stochastiques, cela équivaut à jouer chaque action au moins une fois.

2 SARSA(λ)

```
import numpy as np
import gym
from tilecoding import TileCoder


def get_tile_coder(environment):
    return TileCoder(
        environment.observation_space.high,
        environment.observation_space.low,
        num_tilings=8,
        tiling_dim=8,
        max_size=4096,
    )


def set_random_seed(environment, seed):
    environment.seed(seed)
    np.random.seed(seed)

def choose_action(state, action_space, theta, tile_coder):
    Q = []
    for action in range(action_space.n):
        x = tile_coder.phi(state, action)
        Q.append(np.sum(theta[x]))

    return np.argmax(Q)

def run(seed, epochs=100, T=200, alpha=0.1, lambda_=0.9, gamma=1.0):
    # alpha = learning_rate
    # T = Horizon
    # lambda_ = trace_decay
    # gamma = discount_rate

    environment = gym.make("MountainCar-v0")
    set_random_seed(environment, seed)
    tile_coder = get_tile_coder(environment)

    theta = np.zeros(tile_coder.size)

    r_cumuls = np.zeros(epochs)

    action_space = environment.action_space

    for n_epoch in range(epochs):
        s = environment.reset()
        a = choose_action(s, action_space, theta, tile_coder)
```

```
        # Traces
        z = np.zeros(tile_coder.size)

        for _ in range(T):
            s_, r, is_terminal_state, _ = environment.step(a)
            r_cumuls[n_epoch] += r
            delta = r

            for x in tile_coder.phi(s, a):
                delta -= theta[x]
                # Replacing traces
                z[x] = 1

            if is_terminal_state:
                theta += alpha * delta * z
                break

            a_ = choose_action(s_, action_space, theta, tile_coder)

            for x in tile_coder.phi(s_, a_):
                delta += gamma * theta[x]

            theta += alpha * delta * z
            z *= gamma * lambda_

            s, a = s_, a_

        if n_epoch >= 99 and np.mean(r_cumuls[n_epoch-99:n_epoch+1]) >= -110:
            print(f"Environment resolved with r_cumuls mean = {np.mean(r_cumuls[
                                                    n_epoch-99:n_epoch+1])},
                                                    n_epoch = {n_epoch}")
            break

    environment.close()

if __name__ == "__main__":
    seed = 42
    run(seed, epochs=500)
```

Analyse de l'impact du paramètre λ

```
  if __name__ == "__main__":
    from matplotlib import pyplot as plt

    seed = 42

    for l in np.linspace(0,1,5):
        plt.plot(run(seed, epochs=500, lambda_=l), label="{0} = {1}".format(r"$\
                                                lambda$", str(l)))

    plt.title("Cumulative reward mean over the 100 steps")
    plt.xlabel("n_epoch")
    plt.ylabel("Cumul rewards mean")
    plt.yticks(np.linspace(-200, -100, 11))
    plt.legend()
    plt.show()
```

λ est un rapport de diminution des traces. Il permet de contrôler la quantité d'information des traces (actions sélectionnées antérieures) à considérer pour le calcul de l'erreur. Il joue donc un rôle majeur dans la convergence et la stabilité de Sarsa(λ).

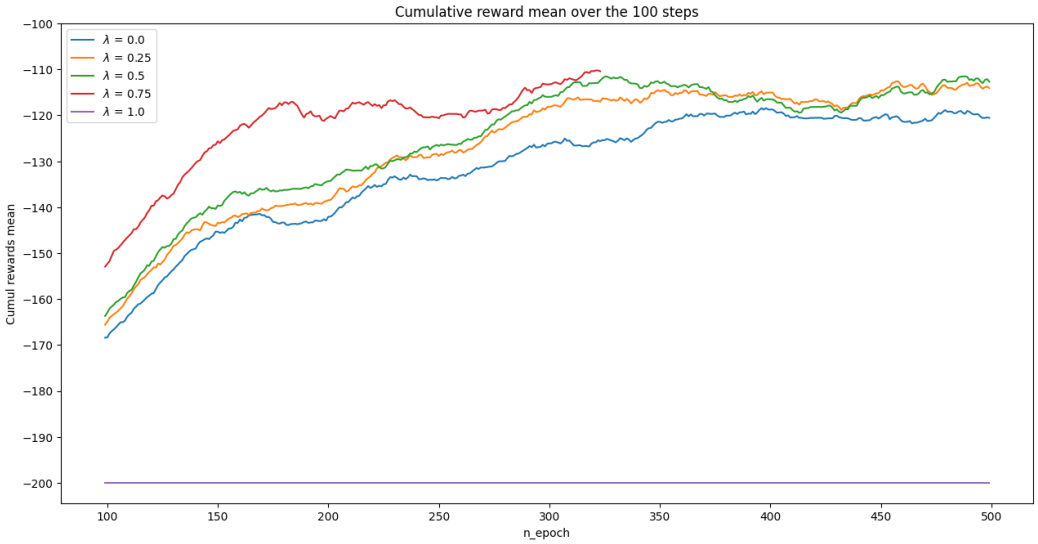

FIGURE 1 – Moyenne cumulatif des 100 derniers épisodes

Pour les valeurs de λ égales à 0, 0.25, 0.5 et 0.75, la vitesse de convergence semble diminuer avec la croissance de λ.

Par contre, pour de $\lambda = 1$, l'agent n'arrive pas à terminer ne serait-ce qu'un épisode dans le temps imparti (ie $R_T = -200$ à tous les épisodes).

Pour mieux comprendre ce qui se passe nous avons testé avec des valeurs de λ plus proches de 1 (0.9, 0.95, 0.96, 0.97, 0.98, 0.99). On remarque alors que l'algorithme arrive à résoudre MontCar-v0 en moins de 150 époques ce qui constitue meilleure performance enregistrée. Cependant, 0.95 à 0.98 cette performance se détériore en fonction de la grandeur de λ. On observve alors pour 0.99 que l'algorithme obtient le même score qu'avec 1.

D'après ces observations, la prise en compte de la totalité ou la quasi-totalité des traces pour le calcul de l'erreur entraine une grande instabilité et par conséquence la divergence de l'algorithme. Ceci pourrait s'expliquer par le fait que l'état actuel et celles antérieurs ont le même poids dans le temps. Ce qui déboussole complètement l'agent car il n'a plus de repère.

3 Motivez les astuces utilisées pour stabiliser l'entraînement d'une stratégie Deep Q-Learning

(a) Principal avantage d'utiliser $q_\theta(s) \in \mathbb{R}^{|\mathcal{A}|}$ dans un modèle d'approximation dans les environnements à actions discrètes

Réduis la latence grâce à un gain de temps de calcul. En effet, l'apprentissage par lot simplifie les opérations d'aller/retour du réseau de neuronnes (foward propagation/ Backward propagation).

(b) Utilité d'un réseau cible

En RL, la cible $r_t + \gamma \max_{a' \in A} q_\theta(s_{t+1}, a')$ change continuellement à chaque itération. Ce qui rend l'apprentissage instable avec les réseaux de neuronnes. Pour remédier à cela, une solution est d'utiliser un réseau cible. Ce réseau à la même architecture que le réseau qui approxime la fonction mais avec un paramètre gelé qui est mis à jour à chaque itération. Cela conduit à un entrainement plus stable car la fonction cible reste fixe pendant un certain temps.

(c) Heuristique pour déterminer θ^-

Soit l'heuristique : $\theta^- = (1-\tau)\theta^- + \tau\theta$. Le choix de τ permet de contrôler la variation du paramètre θ^- en fonction de θ.

Si $\tau = 0,\ \theta^- = (1-0)\theta^- + 0 * \theta \implies \theta^- = \theta^-$
θ^- est constante. l'entrainement du réseau est stable, mais probablement pas adapté car l'écart entre θ et θ^- peut être considérablement grand.

Si $\tau = 1,\ \theta^- = (1-1)\theta^- + 1 * \theta \implies \theta^- = \theta$
le réseau cible utilise la valeur de θ pour calculer la cible. Cela revient pratiquement à ne pas avoir de réseau cible. Par conséquent cause un entrainement instable.

Choisir une bonne valeur de τ dans ce cas revient à déterminer une valeur comprise entre 0 et 1 de telle sorte que l'entrainement du réseau soit le plus stable possible tout en restant le plus proche possible de la paramétrisation estimée ou réelle du problème.

(d) Utilité d'un replay buffer

En RL nous recevons des échantillons séquentiels provenant des interactions avec l'environnement. Le réseau peut être amené à voir trop d'échantillons d'un même type et à oublier les autres. Une solution à cela consiste à utiliser un replay buffer.

- Il permet de briser la corrélation temporelle des échantillons d'entraînement et un meilleur comportement de convergence lors de l'apprentissage d'un approximateur de fonction. Cela s'explique en partie par le fait que les données sont plus proches des données i.i.d. supposées dans la plupart des preuves de convergence de la descente de gradient stochastique.

- De plus, Lorsqu'une expérience est coûteuse(transition à éviter) ou rare(manque d'informations), la connaissance des transitions antérieures permet de tirer pleinement parti de celle-ci.

(e) Différence Replay buffer et Apprentissage supervisé

- La régression logistique sépare $\mathcal{D}$ en train/test/validation tandis que le replay buffer pige de manière i.i.d. dans $\mathcal{D}$ pour entrainer le modèle.

- Dans la regression logistique, la sortie y est connue pour chaque entrée x tandis que le replay buffer se sert d'une estimation de la cible $\hat{y} = r_t + \gamma \max_{a \in A} q_\theta(s_{t+1}, a')$ pour chaque entrée $x = (s, a)$

```
    import random

from poutyne import Model
from copy import deepcopy  # NEW

import numpy as np
import gym
import torch


class ReplayBuffer:
    def __init__(self, buffer_size):
        self.__buffer_size = buffer_size
        self.__buffer = []

    def __len__(self):
      return len(self.__buffer)

    def store(self, element):
        '''
        Stores an element. If the replay buffer is already full, deletes the oldest
        element to make space.
        '''
        self.__buffer.append(element)
        if len(self.__buffer) > self.__buffer_size:
            del self.__buffer[0]

    def get_batch(self, batch_size):
        '''
        Returns a list of batch_size elements from the buffer.
        '''
        return random.choices(self.__buffer, k=batch_size)


class DQN(Model):
    def __init__(self, actions, *args, **kwargs):
        self.actions = actions
        super().__init__(*args, **kwargs)

    def get_action(self, state, epsilon):
        '''
        Returns the selected action according to an epsilon-greedy policy.
        '''
        if(np.random.random() < epsilon):
            return np.random.choice(self.actions)

        return np.argmax(self.predict_on_batch(state))

    def soft_update(self, other, tau):
        '''
        Code for the soft update between a target network (self) and
        a source network (other).

        The weights are updated according to the rule in the assignment.
        '''
        new_weights = {}
```

```
        own_weights = self.get_weight_copies()
        other_weights = other.get_weight_copies()

        for k in own_weights:
            new_weights[k] = (1 - tau) * own_weights[k] + tau * other_weights[k]

        self.set_weights(new_weights)


class NNModel(torch.nn.Module):
    '''
    Neural Network with 3 hidden layers of hidden dimension 64.
    '''
    def __init__(self, in_dim, out_dim, n_hidden_layers=3, hidden_dim=64):
        super().__init__()
        layers = [torch.nn.Linear(in_dim, hidden_dim), torch.nn.ReLU()]
        for _ in range(n_hidden_layers - 1):
            layers.extend([torch.nn.Linear(hidden_dim, hidden_dim), torch.nn.ReLU()
                                                                    ])
        layers.append(torch.nn.Linear(hidden_dim, out_dim))

        self.fa = torch.nn.Sequential(*layers)

    def forward(self, x):
        return self.fa(x)


def format_batch(batch, target_network, gamma):
    '''
    Input :
        - batch, a list of n=batch_size elements from the replay buffer
        - target_network, the target network to compute the one-step lookahead
                                                    target
        - gamma, the discount factor

    Returns :
        - states, a numpy array of size (batch_size, state_dim) containing the
                                                    states in the batch
        - (actions, targets) : where actions and targets both
                    have the shape (batch_size, ). Actions are the
                    selected actions according to the target network
                    and targets are the one-step lookahead targets.
    '''
    states, actions, rewards, states_, are_terminal_states = zip(*batch)
    states = np.vstack(states)
    actions = np.array(actions)
    rewards = np.array(rewards)
    states_ = np.vstack(states_)
    are_terminal_states = np.array(are_terminal_states)

    q_theta_ = target_network.predict_on_batch(states_)
    targets = rewards + gamma * np.max(q_theta_, axis=1) * (1-are_terminal_states)

    return states, (actions, targets.astype(np.float32))

def dqn_loss(y_pred, y_target):
    '''
    Input :
        - y_pred, (batch_size, n_actions) Tensor outputted by the network
```

```
        - y_target = (actions, targets), where actions and targets both
                     have the shape (batch_size, ). Actions are the
                     selected actions according to the target network
                     and targets are the one-step lookahead targets.

    Returns :
        - The DQN loss
    '''
    actions, targets = y_target
    q_thetas = y_pred.gather(1, actions.type(torch.int64).unsqueeze(-1)).squeeze()

    return torch.nn.functional.mse_loss(targets, q_thetas)

def set_random_seed(environment, seed):
    environment.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)  # NEW

def run(batch_size, gamma, buffer_size, seed, tau, training_interval, learning_rate
                                        , epsilon=1.0, epsilon_min=5e-3, epochs=
                                        800, T=1000, start_replay=64):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using {device} device")

    environment = gym.make("LunarLander-v2")
    set_random_seed(environment, seed)

    observation_space = environment.observation_space
    action_space = environment.action_space

    nn_network = NNModel(in_dim=observation_space.shape[0], out_dim=action_space.n)
    optimizer = torch.optim.Adam(nn_network.parameters(), lr=learning_rate)

    prediction_network = DQN(actions=action_space.n, network=nn_network, optimizer=
                                            optimizer, loss_function=dqn_loss).to
                                            (device)
    target_network = (deepcopy(prediction_network)).to(device)

    replay_buffer = ReplayBuffer(buffer_size)

    r_cumuls = np.zeros(epochs)
    loss_cumuls = np.zeros(epochs)

    for n_epoch in range(epochs):
        s = environment.reset()

        for t in range(T):
            a = prediction_network.get_action(s, epsilon)
            s_, r, is_terminal_state, _ = environment.step(a)

            r_cumuls[n_epoch] += r

            replay_buffer.store((s,a,r,s_,is_terminal_state))
            s = s_

            if (len(replay_buffer) < start_replay):
                continue

            if t % training_interval == 0:
```

```
                batch = replay_buffer.get_batch(batch_size)
                states, (chosen_actions, targets) = format_batch(batch,
                                                        target_network, gamma)
                loss_cumuls[n_epoch] += prediction_network.train_on_batch(states, (
                                                        chosen_actions, targets))

                target_network.soft_update(prediction_network, tau)

            if is_terminal_state or r_cumuls[n_epoch] > 200:
                print(f"Is terminal state? {is_terminal_state}\t n_epoch={n_epoch}
                                                        step={t} r_cumul={
                                                        r_cumuls[n_epoch]}
                                                        loss_cumul={loss_cumuls[
                                                        n_epoch]}")
                break

        epsilon = max(0.99 * epsilon, epsilon_min)

    environment.close()

    return r_cumuls, loss_cumuls


if __name__ == "__main__":
    '''
    All hyperparameter values and overall code structure are only given as a
                                            baseline.

    You can use them if they help  you, but feel free to implement from scratch the
    required algorithms if you wish!
    '''
    import time
    from matplotlib import pyplot as plt

    batch_size = 64
    gamma = 0.99
    buffer_size = 1e5
    seed = 42
    tau = 1e-3
    training_interval = 4
    learning_rate = 1e-3

    start = time.time()

    r_cumuls, loss_cumuls = run(batch_size, gamma, buffer_size, seed, tau,
                                            training_interval, learning_rate)

    end = time.time()
    print(f"Run duration = {(end-start)/60} min")

    plt.figure()
    plt.plot(r_cumuls, label="Cumulative return per epoch")
    plt.xlabel("n_epoch")
    plt.legend()

    plt.figure()
    plt.plot(loss_cumuls, label="Cumulative loss per epoch")
    plt.xlabel("n_epoch")
    plt.legend()
```

```
plt.show()
```

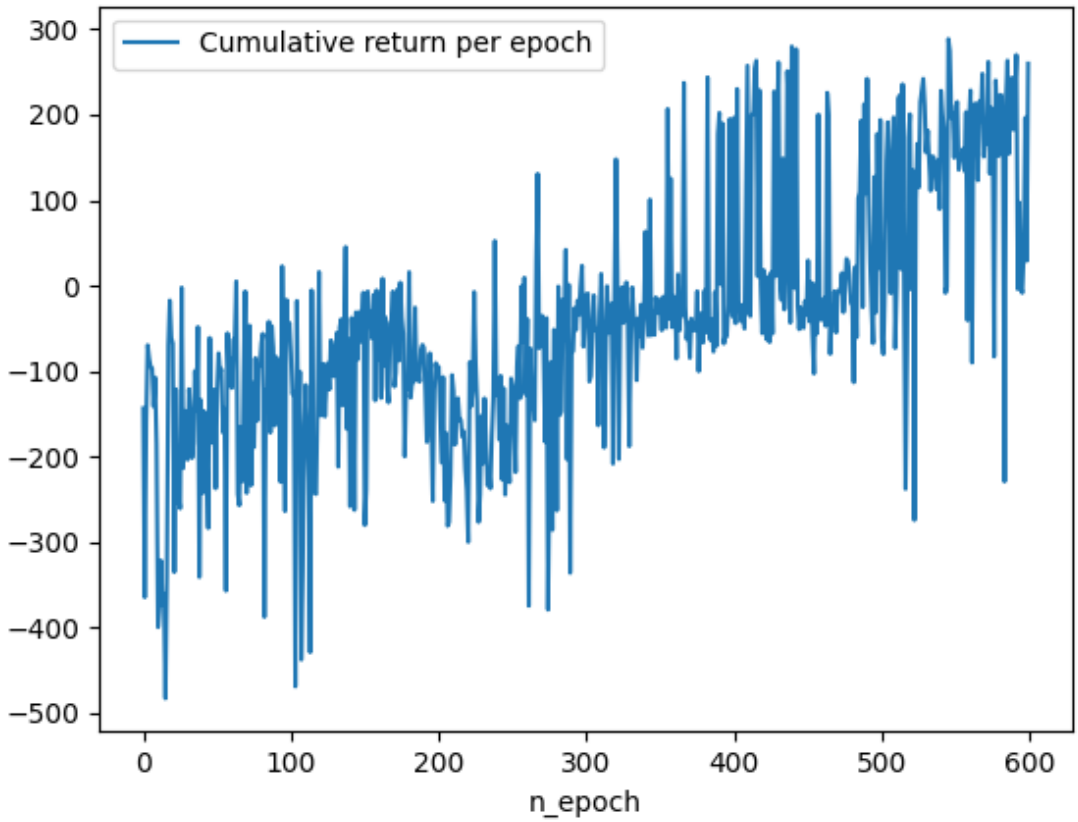

FIGURE 2 – somme des r´ecompenses en function du nombre de pas

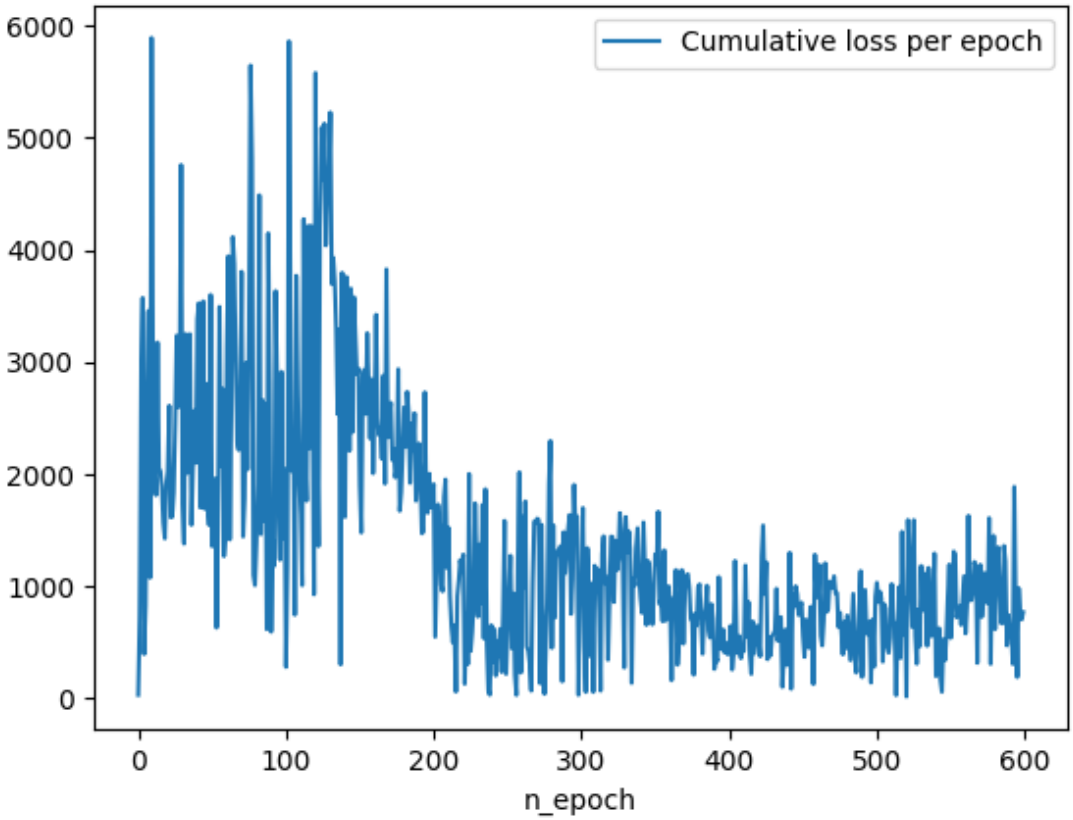

FIGURE 3 – fonction de perte de votre mod'ele en fonction du nombre de pas