Akilhoussen ONALY

S9 MOD Apprentissage profond & Intelligence Artificielle : une introduction

2021-2022

TD3 – Apprentissage par renforcement

Compte rendu

Table des matières

I- Introduction

L'objectif de ce TD est de mettre en œuvre l'apprentissage par renforcement sur des cas d'applications provenant du toolkit Gym. Nous utiliserons principalement des réseaux de neurones en tant qu'algorithmes d'apprentissage. Dans une première partie, il s'agira d'entraîner un chariot à maintenir un pendule vertical en équilibre à l'aide d'un réseau de neurones fully-connected classique. Dans une seconde partie, nous entraînerons un agent à jouer le jeu Breakout d'Atari à l'aide d'un réseau de neurones convolutif.

II- Rappels théoriques

L'apprentissage par renforcement est un domaine de l'apprentissage automatique où le but est d'apprendre à un agent (virtuel) à prendre des actions qui maximisent la récompense dans

un environnement donné. Les cas d'utilisations les plus courants sont les jeux vidéo où des algorithmes sont par exemple capables d'apprendre à jouer aux échecs, au jeu de Go, etc.

Dans l'apprentissage par renforcement, nous avons un agent, qui interagit avec l'environnement dans lequel il se situe. Ces interactions se produisent séquentiellement dans le temps. À chaque étape, l'agent obtient une représentation de l'état de l'environnement. Compte tenu de cette représentation, l'agent sélectionne une action à entreprendre. L'environnement passe alors à un nouvel état, et l'agent reçoit une récompense en conséquence de l'action précédente.

Tout au long de ce processus, l'objectif de l'agent est de maximiser le total des récompenses qu'il reçoit en prenant des actions dans des états donnés. Cela signifie que l'agent veut maximiser non seulement la récompense immédiate, mais aussi les récompenses cumulées qu'il reçoit au fil du temps.

La trajectoire représentant le processus séquentiel de sélection d'une action à partir d'un état, de transition vers un nouvel état et de réception d'une récompense peut être représentée comme suit :

$$S_0, A_0, R_1, S_1, A_1, R_2, \dots$$

où S_t désigne la représentation de l'état à l'instant t, A_t l'action prise à l'instant t depuis l'état S_t et R_{t+1} la récompense obtenue suite à l'action prise à l'instant t.

Il est important de préciser que l'agent n'est pas déterministe (car sinon il n'y aurait pas apprentissage), c'est-à-dire que pour un état donné, l'agent ne choisit pas toujours la même action parmi celles qui sont possibles mais il y a une probabilité qu'il en choisisse une en particulier. Il est donc naturel que S,R et A soient des variables aléatoires.

On définit ensuite les récompenses cumulées (ou le gain) à l'instant t par la somme des récompenses futures à partir de l'instant t :

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T$$

où T désigne l'instant final.

Cependant T pouvant valoir $+\infty$, en pratique on définit G_t comme suit afin d'assurer que G_t ne diverge pas :

$$G_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \cdots$$

avec $\gamma \in [0,1[$

L'ajout du coefficient permet aussi de faire en sorte que les récompenses plus immédiates ont plus ou moins (selon la valeur de γ) d'influence quand il s'agit de prendre une décision concernant une action particulière.

Maintenant que nous avons défini le calcul du gain, il nous reste à répondre à 2 questions : quelle est la probabilité qu'un agent choisisse une action à partir d'un état donné ? En plus de

connaître la probabilité de sélectionner une action, comment savoir à quel point une action ou un état donné est bon pour l'agent en termes de récompenses ?

Pour répondre à ces questions, il faut introduire la notion de *politique* et la notion de *valeur*.

La *politique* est une fonction qui associe un état donné à des probabilités de sélection de chaque action possible à partir de cet état. On la dénote π et $\pi(a \mid s)$ est la probabilité que $A_t = a$ sachant $S_t = s$. On dit qu'un agent « suit » la politique π.

En ce qui concerne la *valeur*, il y a deux quantités qu'on peut définir : la fonction de valeur d'état et la fonction de valeur action – état. La première permet d'évaluer à quel point un état donné est prometteur en termes de gain et la deuxième permet d'évaluer à quel point un couple (état, action) est prometteur en termes de gain.

La fonction de valeur d'état est définie comme le gain moyen (donc l'espérance) sachant l'état :

$$v_\pi(s) = E_\pi[G_t \mid S_t = s]$$

Remarque : L'indice π permet d'indiquer que le calcul est fait sous l'hypothèse que l'agent suit la politique π (la distribution des probabilités des actions est fixée).

De manière similaire, la fonction de valeur action – état est définie comme le gain moyen sachant l'état et l'action :

$$q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a]$$

Ainsi à travers ces deux quantités, on voit qu'une politique π sera meilleure qu'une autre politique π' si $v_\pi(s) \geq v_{\pi'}(s)$ pour tout état s (et aussi $q_\pi(s, a) \geq q_{\pi'}(s, a) \quad \forall (s, a)$) et donc le rôle d'un algorithme d'apprentissage sera d'apprendre une politique qui rend les actions à gain élevé plus probables quel que soit l'état, c'est-à-dire de trouver $\pi = argmax_\pi\, q$, et surtout trouver la fonction q optimale.

Un algorithme classique pour apprendre la meilleure politique est le Q-Learning qui exploite l'équation de Bellman afin de trouver la fonction q optimale (qu'on note q_*). L'équation de Bellman est définie comme suit :

$$q_*(s, a) = E[R_{t+1} + \gamma\, max_{a'}\, q_*(s', a')]$$

Intuitivement, l'équation de **Bellman** signifie que le meilleur gain qu'on peut espérer à un instant t après avoir pris une action a depuis un état s est l'espérance de la récompense immédiate R_{t+1} à laquelle on ajoute l'espérance du meilleur gain suivant (qu'on peut obtenir parmi toutes les actions possibles depuis le nouvel état s' dans lequel on se trouve après l'action a) en prenant en compte le coefficient γ car il s'agit du gain espéré du pas de temps suivant.

Nous ne détaillerons pas l'algorithme de Q-Learning ici mais il est connu aujourd'hui qu'il a ses limites et ne marche que dans des environnements à faible dimension et faible nombre d'états.

L'autre alternative est d'utiliser un réseau de neurones pour apprendre la fonction q. Pour cela, on donne en entrée au réseau l'état et les sorties sont les différentes valeurs de $q(s, a)$ pour

chaque action a. L'état peut prendre plusieurs formes : cela peut être directement des quantités mesurées telles que la position, la vitesse, etc et c'est ce que nous ferons dans la première partie ou alors cela peut être des images de l'agent dans son environnement et cette approche sera utilisée dans la seconde partie.

Dans la pratique, l'utilisation d'un réseau de neurones est couplée à d'autres techniques qui permettent de rendre l'apprentissage plus efficace.

La **première** est ce qu'on appelle l' « **Epsilon-greedy Strategy** ». En effet, au départ, le réseau commence à apprendre et exploiter la sortie du réseau directement n'est pas très fiable donc il faut laisser à l'agent la possibilité de choisir une action aléatoire. Cette stratégie permet un apprentissage qui respecte l'équilibre exploitation – exploration et au fur et à mesure que le réseau apprend, on commence à exploiter de plus en plus l'information qu'il nous donne et on diminue l'exploration. On contrôle cela à l'aide d'un coefficient ε qu'on diminue au fil du temps (exponentiellement à l'aide d'un taux défini). Pour déterminer si l'agent choisira l'exploration ou l'exploitation à chaque pas de temps, nous générons un nombre aléatoire entre 0 et 1. Si ce nombre est supérieur à ε, alors l'agent choisira sa prochaine action via l'exploitation, c'est-à-dire qu'il choisira l'action avec la valeur Q la plus élevée parmi les sorties du réseau. Sinon, sa prochaine action sera choisie par exploration, c'est-à-dire qu'il choisira au hasard son action et explorera ce qui se passe dans l'environnement.

La **deuxième** technique est ce qu'on appelle le **Replay Memory**. En effet, l'apprentissage se fait de manière séquentielle avec l'algorithme suivant (Simple Deep Q-Learning) :

À chaque pas de temps :

- L'agent se trouve dans un état et on choisit une action qui va être simulée par l'agent (avec exploration ou exploitation)
- On observe la récompense obtenue et le nouvel état
- On passe le nouvel état au réseau de neurones et on calcule la perte en comparant les sorties prédites aux sorties optimales. Les sorties optimales sont calculées avec l'équation de Bellman en réutilisant le même réseau avec le nouvel état pour calculer le deuxième terme dans la somme $\gamma\, max_{a'}\, q_*(s', a')$. Une meilleure variante sera introduite dans la suite pour ce calcul.
- On effectue une rétropropagation pour optimiser les poids et on réitère le processus pour l'état suivant

On voit à travers l'algorithme précédent que les échantillons avec lesquels le réseau apprend sont très corrélés car ils sont consécutifs et il y a un risque que le réseau « apprenne » cette corrélation. Pour empêcher cela, on crée un tableau dans lequel on stocke les échantillons au fur et à mesure de l'interaction de l'agent avec l'environnement et au lieu d'entraîner le réseau avec l'échantillon qui succède (temporellement) à l'échantillon précédent, on l'entraîne avec un échantillon qu'on prend aléatoirement dans le tableau. Il a été montré que cette procédure améliore l'apprentissage du réseau.

La **troisième** technique est ce qu'on appelle le « **Double Deep Q-Learning** ». En effet, dans l'algorithme décrit précédemment, on a vu (phrase en rouge) qu'on réutilisait le même réseau pour calculer la perte or cela pose problème car quand on met à jour les poids du réseau, on

le fait à l'aide de la perte qui est calculée avec des cibles qui utilisent les poids actuels et ces cibles auraient changé si on les calcule de nouveau avec les nouveaux poids. Ainsi on a l'impression que l'algorithme « court après sa propre queue ». Pour éviter ce problème, on met en place un deuxième réseau qui est une copie du premier réseau mais qu'on laisse fixe et on le met à jour de temps en temps, par exemple à chaque dizaine d'itérations. C'est ce deuxième réseau qui est utilisée pour calculer les sorties optimales, c'est-à-dire les cibles (et plus précisément le deuxième terme $\gamma\ max_{a'}\ q_*(s', a')$).

Ces trois techniques seront utilisées dans la suite.

III- Premier cas d'application : le CartPole

1) Description du problème

Comme décrit dans l'introduction, dans cette partie, l'objectif est d'apprendre à un agent constitué d'un chariot à se déplacer de façon à maintenir un pendule vertical en équilibre.

Chaque épisode démarre avec un pendule à la verticale, et le but est de l'empêcher de tomber. Une récompense de +1 est fournie pour chaque pas de temps où le poteau reste à la verticale. L'épisode se termine lorsque le poteau s'écarte de plus de 15 degrés de la verticale, ou que le chariot se déplace de plus de 2,4 unités du centre.

Par ailleurs, nous avons 4 quantités qui décrivent l'état à chaque instant : il s'agit de la position du chariot, la vitesse du chariot, l'angle du pendule et la vitesse du pendule au sommet. Comme expliqué dans la partie théorique, ces 4 quantités vont être servies à un réseau de neurone qui va apprendre à prédire les q-valeurs. À chaque instant, l'agent a 2 possibilités : aller à gauche ou à droite. Ainsi, le réseau aura 2 sorties qui correspondent aux q-valeurs pour les 2 actions possibles.

Pour résoudre ce problème, nous utiliserons le Double Deep Q-Learning avec Replay Memory et Epsilon-Greedy Strategy (voir les rappels théoriques pour la signification de ces termes).

2) Description et complétion du code

Le code est constitué de plusieurs classes et nous l'avons adapté à partir de https://github.com/Lazydok/RL-Pytorch-cartpole/blob/master/2_double_dqn.py et de https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Dans cette partie, nous allons décrire chaque classe, les fonctions qu'elle contient et le principe de leur fonctionnement.

Classe ReplayMemory :

Cette classe permet d'implémenter le Replay Memory décrit dans la partie théorique. Elle contient une fonction d'initialisation qui permet de définir la longueur du tableau de stockage (capacité) et créer un tableau vide. Elle contient également une fonction *push* qui permet d'ajouter un échantillon au tableau (et si la taille du tableau dépasse la capacité), on enlève

le premier élément. Enfin, elle contient une fonction *sample* qui permet de récupérer des échantillons au hasard depuis le tableau pour une taille de batch définie.

Classe QNet :

Cette classe permet d'implémenter le réseau de neurones de type perceptron qui sera utilisé pour l'apprentissage.

La fonction d'initialisation est définie comme suit :

```
1.  def __init__(self):
2.      super(QNet, self).__init__()
3.      # VOTRE CODE
4.      #############
5.      # Définition d'un réseau avec une couche cachée (à 256 neurones par exemple)
6.      self.l1 = nn.Linear(4, 164) # 4 entrées d'état
7.      self.l2 = nn.Linear(164, 2) # 2 sorties : 1 pour chaque action (gauche ou droite)
```

La fonction de propagation en avant est définie comme suit :

```
1.  def forward(self, x):
2.      # VOTRE CODE
3.      #############
4.      # Calcul de la passe avant :
5.      # Fonction d'activation relu pour la couche cachée
6.      # Fonction d'activation linéaire sur la couche de sortie
7.      x = F.relu(self.l1(x))
8.      x = self.l2(x)
9.      return x
```

Classe Agent :

Cette classe permet d'implémenter l'agent avec ses différentes méthodes (choisir une action, apprendre, observer l'environnement, etc…).

À l'initialisation, on définit les paramètres et attributs utiles (voir leurs significations dans les commentaires du code) :

```
1.   def __init__(self,env):
2.
3.          self.env = env #l'environnement gym
4.          self.batch_size = 64 #taille de batch
5.          self.gamma = 0.99 #coefficient gamma utilisé dans le calcul du gain
6.          self.eps_start = 0.9 #epsilon au début
7.          self.eps_end = 0.05 #epsilon à la fin
8.          self.eps_decay = 200 #taux de décroissance d'epsilon
9.          self.target_update = 10 #pas de mise à jour du réseau cible
10.         self.num_episodes = 200 #nombre d'épisodes d'entraînement
11.
12.         self.n_actions = env.action_space.n #nombre d'actions possibles
13.         self.episode_durations = [] #liste pour stocker les durées des épisodes
14.
15.         self.policy_net = QNet() #initialisation du réseau apprenant la fonction q
16.         self.target_net = QNet() #initialisation du réseau cible (pour le calcul des sorties cibles)
17.
18.         if use_cuda:
19.             self.policy_net.cuda()
20.             self.target_net.cuda()
21.
22.         # self.target_net.load_state_dict(self.policy_net.state_dict())
23.         # self.target_net.eval()
24.
25.         self.optimizer = optim.Adam(self.policy_net.parameters()) #initialisation de l'optimiseur
26.         self.memory = ReplayMemory(10000) #initialisation du Replay Memory
27.
28.         self.steps_done = 0 #nombre de pas réalisé au total
```

On définit ensuite une fonction select_action qui permettra de choisir une action en suivant l' « Epsilon-greedy Strategy » évoqué dans les rappels théoriques (des commentaires valant explication sont rajoutés dans le code) :

```
def select_action(self,state,train=True):
        sample = random.random() #nombre aléatoire entre 0 et 1
        # on diminue exponentiellement le seuil epsilon
        # selon le nombre de pas réalisé depuis le 1er épisode :
        eps_threshold = self.eps_end + (self.eps_start - self.eps_end) * \
            math.exp(-1. * self.steps_done / self.eps_decay)
        self.steps_done += 1
        if train: #si en mode apprentissage
            if sample > eps_threshold:
                # VOTRE CODE
                ############
                # Calcul et renvoi de l'action fournie par le réseau
                # On calcule l'action qui donne un gain moyen plus élevé d'après le modèle :
                with torch.no_grad():
                    return self.policy_net(Variable(state).type(FloatTensor)).data.max(1)[1].view(1, 1)
            else:
                # VOTRE CODE
                ############
                # Calcul et renvoi d'une action choisie aléatoirement
                return LongTensor([[random.randrange(2)]])
        else: #si en mode test :
            # On calcule l'action qui donne un gain moyen plus élevé d'après le modèle :
            with torch.no_grad():
                return self.policy_net(Variable(state).type(FloatTensor)).data.max(1)[1].view(1, 1)
```

Notons qu'en mode test, nous utilisons directement le modèle car même si ε sera très faible, il se peut que l'action soit choisie aléatoirement dans un cas extrême où le nombre aléatoire est très faible or l'exploration n'a pas lieu d'être en mode test. Pour éviter cela, on empêche ce comportement en ajoutant un argument booléen *train* à la fonction.

La fonction *plot_durations* permet d'afficher les durées de chaque épisode au fur et à mesure de l'apprentissage ainsi que la moyenne mobile sur 100 épisodes.

On définit ensuite une fonction *optimize_model* qui permettra d'optimiser le réseau de neurones (des commentaires sont ajoutés dans le code pour apporter des éléments d'explication) :

```
def optimize_model(self):

        # si la longueur de la mémoire est plus petite que la taille de batch
        # on ne fait rien :
        if len(self.memory) < self.batch_size:
            return

        # on échantillonne des observations depuis la mémoire (replay memory ):
        transitions = self.memory.sample(self.batch_size)
        # on les sépare selon ce qu'elles représentent
        batch_state, batch_action, batch_next_state, batch_reward = zip(*transitions)
        # et on les met au bon format :
        batch_state = Variable(torch.cat(batch_state))
        batch_action = Variable(torch.cat(batch_action))
        batch_reward = Variable(torch.cat(batch_reward)).unsqueeze(-1)
        batch_next_state = Variable(torch.cat(batch_next_state))

        # VOTRE CODE
        ############
        # Calcul de Q(s_t,a) : Q pour l'état courant
        current_q_values = self.policy_net(batch_state).gather(1, batch_action)

        # VOTRE CODE
        ############
        # Calcul de Q pour l'état suivant
        # on utilise le réseau cible comme expliqué dans les rappels théoriques :
        max_next_q_values = self.target_net(batch_next_state).detach().max(1)[0]
        # VOTRE CODE
        ############
        # Calcul de Q future attendue cumulée
```

```
        # équation de Bellman (rappels théoriques)
        expected_q_values = batch_reward + (self.gamma * max_next_q_values).unsqueeze(-1)

        # VOTRE CODE
        ############
        # Calcul de la fonction de perte de Huber
        loss = F.smooth_l1_loss(current_q_values, expected_q_values)
        # VOTRE CODE
        ############
        # Optimisation du modèle
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

On définit ensuite une fonction d'apprentissage *train_policy_model* qui permet de mettre à jour séquentiellement les poids du réseau en fonction de l'interaction de l'agent avec son environnement (voir explications dans les commentaires du code) :

```
def train_policy_model(self):

        for i_episode in range(self.num_episodes):

            state = self.env.reset()
            steps = 0

            while True:

                # Choix de l'action (via exploitation ou exploration)
                action = self.select_action(FloatTensor([state]))
                # observation du nouvel état et récompense obtenue
                # et si l'épisode se termine ou pas :
                next_state, reward, done, _ = self.env.step(action[0,0].item())

                # Définition du reward en fonction de la durée de l'épisode :
                # (on la définit différemment pour créer une "incentive" à l'agent
                # à faire durer l'épisode le plus longtemps possible)
                if done:
                    if steps < 30:
                        reward -= 10
                    else:
                        reward = -1
                if steps > 100:
                    reward += 1
                if steps > 200:
                    reward += 1
                if steps > 300:
                    reward += 1

                # Ajout du nouvel échantillon observé à la mémoire :
                self.memory.push((FloatTensor([state]),
                     action,
                     FloatTensor([next_state]),
                     FloatTensor([reward])))

                # Optimisation du modèle
                self.optimize_model()

                state = next_state
                steps += 1

                # on arrête la boucle si l'épisode se termine
                # ou si une limite de 1000 est atteinte pour la durée
                # cette limite est définie pour empêcher l'épisode
                # de durer indéfiniment
                # et on la définit assez haute
                # pour qu'elle soit satisfaisante comme performance atteinte
                if done or steps >= 1000:
                    self.episode_durations.append(steps)
                    self.plot_durations()
                    break
            # Mise à jour du réseau cible à chaque période (définie au début)
            if i_episode % self.target_update == 0:
                self.target_net.load_state_dict(self.policy_net.state_dict())
                self.save_model()

        # Enregistrement du modèle
        self.save_model()
        print('Training completed')
        plt.show()
```

Ensuite, on définit des fonctions *load_model* et *save_model* qui permettent de charger ou sauvegarder le réseau de neurones entraîné.

On définit ensuite une fonction *test* qui sera lancée au moment de la phase de test de l'agent (voir explications dans les commentaires) :

```
def test(self):
        print('Testing model:')
        for i_episode in range(self.num_episodes):
            print('episode: {}'.format(i_episode))

            state = self.env.reset()

            while True:
                self.env.render()
                # VOTRE CODE
                ############
                # Sélection d'une action appliquée à l'environnement
                # et mise à jour de l'état
                # Sélection d'une action (exploitation seulement) :
                action = self.select_action(FloatTensor([state]),train=False)
                # Mise à jour de l'état :
                next_state, _, done, _ = self.env.step(action[0,0].item())

                state = next_state

                if done:
                    break

        print('Testing completed')
```

Le script principal est alors défini comme suit :

```
if __name__ == '__main__':

    # set up matplotlib
    is_ipython = 'inline' in matplotlib.get_backend()
    if is_ipython:
        from IPython import display

    plt.ion()

    env = gym.make('CartPole-v0').unwrapped
    env.reset()

    agent = Agent(env)

    #Training phase
    agent.train_policy_model()

    #Testing phase
    agent.load_model()
    agent.test()

    env.close()
```

3) Expérimentations et résultats

Nos expérimentations ont consisté à faire varier par exemple le paramètre γ (0.75 et 0.99) ainsi que la taille de batch (64 et 128). Nos meilleurs résultats ont été obtenus pour une taille de batch de 64 ainsi que $\gamma = 0.99$.

Par ailleurs, nous avons le traîné le modèle sur 500 épisodes car nos expérimentations ont montré que ce nombre d'épisodes suffisait pour atteindre une bonne performance.

Après entraînement dans un notebook ayant accès à un GPU, nous obtenons le graphique suivant pour la durée en fonction de l'épisode :

```
[1]: %run qlearning_cartpole.py
```

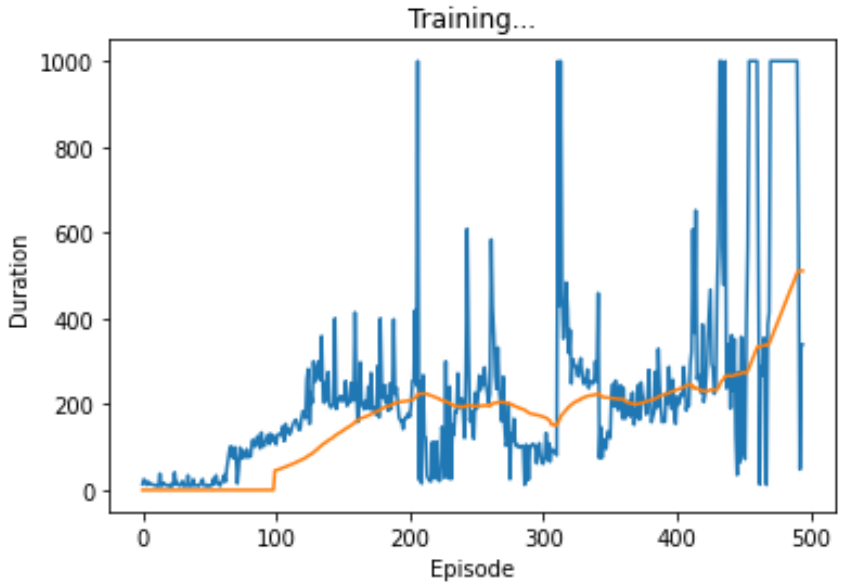


Figure 1 – Courbe des durées en fonction de l'épisode d'entraînement pour le CartPole

On remarque à partir de 450 épisodes environ, l'agent est capable de faire durer l'épisode avec une durée supérieure à 1000 de manière répétée, ce qui signifie qu'il a atteint le régime de bonne performance. Par ailleurs, on observe également que la moyenne est à tendance croissante, ce qui signifie qu'il y a bien eu apprentissage à l'aide du réseau de neurones.

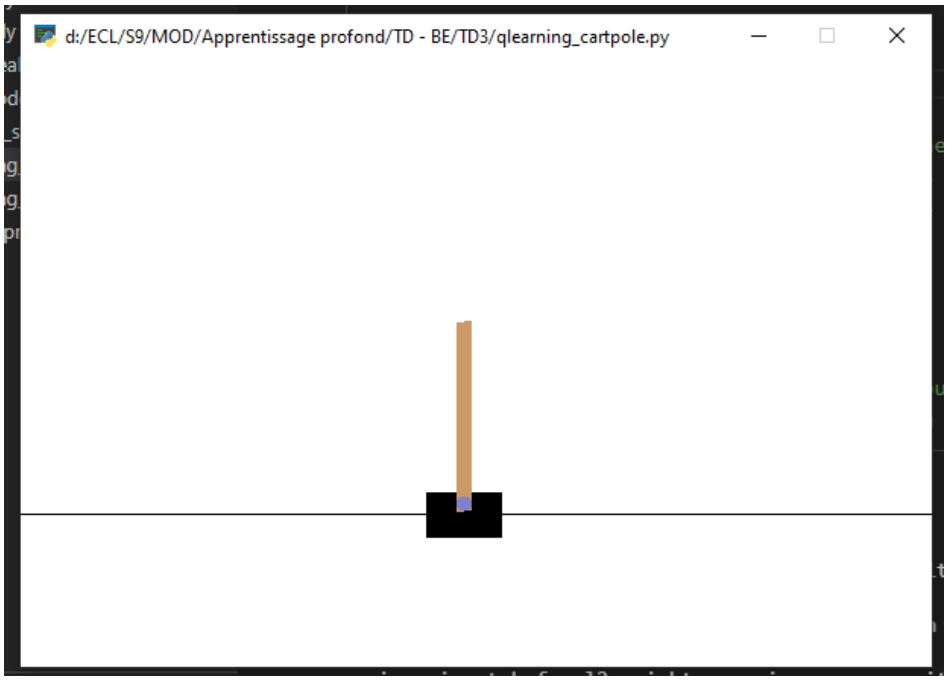


Figure 2 – Capture d'écran de la simulation du chariot et du pendule après apprentissage

Après apprentissage, nous avons testé l'agent. Une vidéo de l'expérimentation est disponible au lien https://www.youtube.com/watch?v=iPucQj1s_7E . Dans cette vidéo, on remarque le chariot effectue des **micro-mouvements** à gauche et à droite afin de garder le pendule en

équilibre et on remarque aussi qu'on est à l'épisode 0 (voir la console) durant toute l'expérimentation. Pour des raisons pratiques, nous avons arrêté après 1 minute mais l'épisode pourrait continuer sans fin. L'agent a donc appris la bonne stratégie compte tenu du fait qu'il ne fallait pas s'éloigner du centre et aussi ne pas faire trop incliner le pendule.

IV- Deuxième cas d'application : le jeu Breakout

1) Description du problème

Dans cette partie, l'objectif est d'apprendre à un agent à jouer le jeu Breakout d'Atari. Cependant, contrairement à la partie précédente, l'observation consistera en une séquence de 4 images consécutives afin d'avoir l'information de mouvement plutôt que l'état directement. On utilisera donc un réseau convolutif qui prendra en entrée la séquence de 4 images et essaiera de prédire les q-valeurs pour les 4 actions possibles qui sont :

- NOOP (ne rien faire)
- FIRE (action permet de commencer le jeu)
- LEFT (aller à gauche)
- RIGHT (aller à droite)

Pour résoudre ce problème, nous utiliserons le Double Deep Q-Learning avec Replay Memory et Epsilon-Greedy Strategy (voir les rappels théoriques pour la signification de ces termes).

2) Description et complétion du code

Nous avons adapté le code de cette partie à partir de
https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Nous allons décrire ici les classes et les fonctions qu'il contient. Cependant, certaines fonctions étant similaires à celles de la partie précédente avec Cartpole, elles ne seront décrites que brièvement.

Classe Replay Memory :

À quelques détails près, cette classe est similaire à celle de la partie précédente. Voir la partie précédente pour plus de détails.

Classe DQN :

Cette classe permet de définir le réseau convolutif qui servira à apprendre la q-fonction.

Elle est définie comme suit (voir les commentaires pour plus de détails) :

```
1.  class DQN(nn.Module):
2.      def __init__(self, num_frames, h, w, num_outputs):
3.          super(DQN, self).__init__()
4.          # VOTRE CODE
5.          ############
6.          # Définition du réseau. Exemple :
7.          # 3 couches de convolution chacune suivie d'une batch normalization
8.          # filtres de taille 5 pixels, pas de 2
9.          # 16 filtres pour la première couche
10.         # 32 filtres pour la deuxième
```

```
        # 64 pour la troisième
        # Finir par une couche fully connected
        self.conv1 = nn.Conv2d(num_frames, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            """Fonction pour calculer la taille de sortie d'une couche convolutive"""
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, num_outputs)

    def forward(self, x):
        # VOTRE CODE
        ############
        # Calcul de la passe avant :
        # Fonction d'activation relu pour les couches cachées
        # Fonction d'activation linéaire sur la couche de sortie
        x = x.to(device)
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

Classe Agent :

Cette classe permet d'implémenter l'agent avec ses différentes méthodes (choisir une action, apprendre, observer l'environnement, etc…).

À l'initialisation, on définit les paramètres et attributs utiles (voir leurs significations dans les commentaires du code) :

```
class Agent:
    def __init__(self,env):

        self.env = env #initialisation de l'environnement
        self.batch_size = 32 #taille de batch
        self.gamma = 0.99 #coefficient gamma utilisé dans le calcul du gain
        self.eps_start = 0.9 #epsilon au début
        self.eps_end = 0.05 #epsilon à la fin
        self.eps_decay = 200 #taux de décroissance d'epsilon
        self.target_update = 10 #pas de mise à jour du réseau cible
        self.num_episodes = 1000 #nombre d'épisodes d'entraînement
        self.num_frames = 4 #nombre d'images dans une séquence

        self.im_height = 84 #hauteur d'une image
        self.im_width = 84 #largeur d'une image

        self.n_actions = env.action_space.n #nombre d'actions possibles

        self.episode_durations = [] #liste pour stocker les durées des épisodes

        #initialisation du réseau apprenant la fonction q :
        self.policy_net = DQN(self.num_frames, self.im_height, self.im_width, self.n_actions).to(device)
        #initialisation du réseau cible (pour le calcul des sorties cibles) :
        self.target_net = DQN(self.num_frames, self.im_height, self.im_width, self.n_actions).to(device)
        # self.target_net.load_state_dict(self.policy_net.state_dict())
        # self.target_net.eval()

        #initialisation de l'optimiseur :
        self.optimizer = optim.Adam(self.policy_net.parameters(),1e-4)
        self.memory = ReplayMemory(10000) #initialisation du Replay Memory

        self.steps_done = 0 #nombre de pas réalisé au total
```

Les fonctions *select_action* et *plot_durations* sont similaires à celles de la partie précédente (se référer à cette partie et au code pour plus de détails).

On définit ensuite une fonction *process* qui permet d'appliquer une transformation à l'image de l'état (voir les détails dans les commentaires) :

```
def process(self, state):
        """Fonction qui permet de transformer l'image d'un état en niveaux de gris
        puis on la redimensionne selon les tailles définies
        et on crée un tenseur Pytorch"""
        state = resize(rgb2gray(state), (self.im_height, self.im_width), mode='reflect') * 255
        state = state[np.newaxis, np.newaxis, :, :]
        return torch.tensor(state, device=device, dtype=torch.float)
```

Ensuite, comme précédemment, on définit une fonction *optimize_model* qui permettra d'optimiser le réseau de neurones (voir les détails dans les commentaires) :

```
def optimize_model(self):
        if len(self.memory) < self.batch_size:
            return
        transitions = self.memory.sample(self.batch_size)

        batch = Transition(*zip(*transitions))

        # Masque des états non finaux :
        # Les états finaux sont ceux pour qui l'épisode se termine
        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                                batch.next_state)), device=device, dtype=torch.uint8)
        # Calcul des états non finaux :
        non_final_next_states = torch.cat([s for s in batch.next_state
                                                    if s is not None])
        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

        # VOTRE CODE
        #############
        # Calcul de Q(s_t,a) : Q pour l'état courant
        state_action_values = self.policy_net(state_batch).gather(1, action_batch)

        # VOTRE CODE
        #############
        # Calcul de Q pour l'état suivant
        next_state_values = torch.zeros(self.batch_size, device=device)
        next_state_values[non_final_mask] = self.target_net(non_final_next_states).max(1)[0].detach()

        # VOTRE CODE
        #############
        # Calcul de Q future attendue cumulée
        expected_state_action_values = (next_state_values * self.gamma) + reward_batch

        # VOTRE CODE
        #############
        # Calcul de la fonction de perte de Huber
        criterion = nn.SmoothL1Loss()
        loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

        # VOTRE CODE
        #############
        # Optimisation du modèle
        self.optimizer.zero_grad()
        loss.backward()
        for param in self.policy_net.parameters():
            param.grad.data.clamp_(-1, 1)
        self.optimizer.step()
```

Ensuite, on définit une fonction *train_policy_model* qui permet d'entraîner le modèle séquentiellement au fur et à mesure qu'il interagit avec l'environnement. Cette fonction est globalement similaire à celle de la partie précédente. Les quelques différences sont évoquées dans les commentaires de la fonction *test* qui est définie comme suit :

```
    def test(self):
        print('Testing model:')
        for i_episode in range(self.num_episodes):
            print('episode: {}'.format(i_episode))

```

```
            state = self.env.reset()
            state = self.process(state)

            while True:
                # ----------------------------------------
                # Cette partie ne fonctionne que lorsque le code
                # est lancé depuis un notebook :
                plt.imshow(self.env.render(mode='rgb_array'),
                           interpolation='none')
                display.clear_output(wait=True)
                display.display(plt.gcf())
                # ----------------------------------------------

                # VOTRE CODE
                ############
                # Sélection d'une action appliquée à l'environnement
                # et mise à jour de l'état

                # On attend qu'on ait assez d'images pour le servir au réseau :
                while state.size()[1] < self.num_frames:
                    action = 1 # Fire

                    new_frame, _, done, _ = self.env.step(action)
                    new_frame = self.process(new_frame)

                    state = torch.cat([state, new_frame], 1)

                # Dès qu'on a assez d'images, on choisit l'action
                # (via exploitation seulement) :
                action = self.select_action(state,train=False)
                # On observe le nouvel état (nouvelle image) :
                new_frame, _, done, _ = self.env.step(action.item())
                # On la transforme :
                new_frame = self.process(new_frame)

                if done: # si l'épisode est terminé
                    new_state = None
                else :
                    # sinon on ajoute la nouvelle image
                    # à l'état
                    new_state = torch.cat([state, new_frame], 1)
                    # et on enlève la première image de l'état
                    # pour qu'on garde toujours le même nombre d'images
                    new_state = new_state[:, 1:, :, :]

                state = new_state

                if done:
                    break

        print('Testing completed')
```

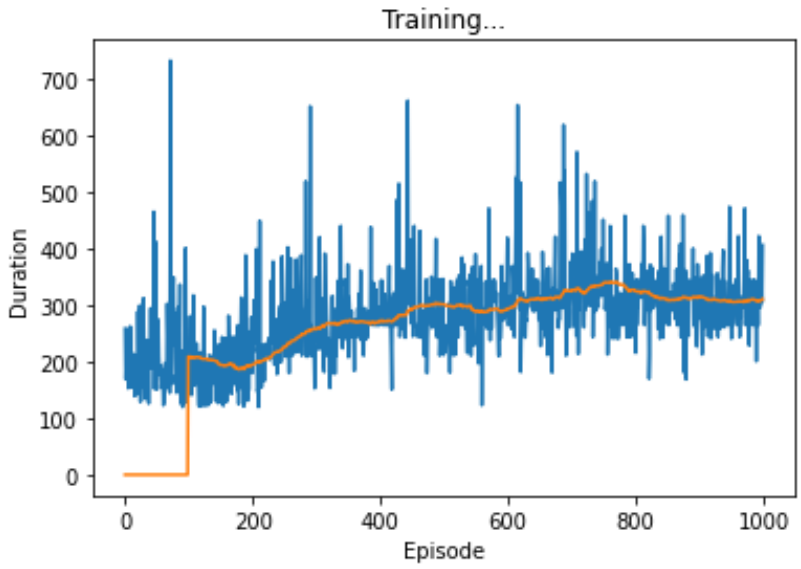
Le script principal de lancement est le même que précédemment, excepté le fait que l'environnement qu'on charge est 'BreakoutDeterministic-v4'.

3) Expérimentations et résultats

Nos expérimentations ont consisté principalement à faire varier le taux d'apprentissage de l'optimiseur ainsi que la taille de batch. Nous avons commencé à obtenir de bons résultats avec le taux $\alpha = 1e - 4$ et une taille de batch de 32. Par ailleurs, nous avons dû entraîner le réseau avec un nombre d'épisodes de 1000 pour commencer à avoir des résultats probants.

Voici le graphe des durées en fonction des épisodes :

```
[2]: %run dqn_breakout.py
```




```
Training completed
```

Figure 3 – Courbe des durées en fonction de l'épisode d'entraînement pour le jeu Breakout

Ce graphique permet de voir que la durée de l'épisode augmente dans sa tendance moyenne au fur et à mesure que le réseau est entraîné. Cela signifie qu'il y a bien eu apprentissage et c'est bon signe.

```
[1]: %load_ext autoreload
     %autoreload 2

[*]: %run dqn_breakout.py
```

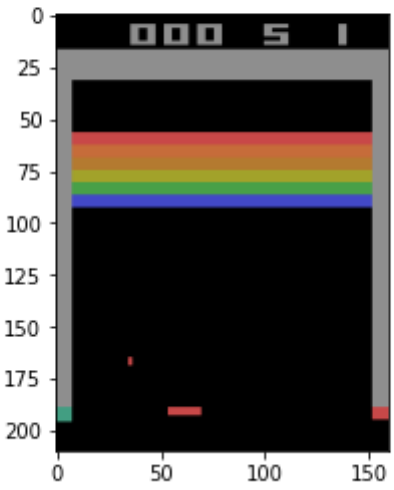


Figure 4 – Capture d'écran de la simulation du jeu Breakout après apprentissage

Après apprentissage, nous avons testé l'agent et une vidéo de l'expérimentation est disponible au lien suivant : https://www.youtube.com/watch?v=HFhRwPIJWYw

Comme on peut le voir dans cette vidéo, l'agent semble avoir appris à se déplacer de manière à ce que la balle atterrisse sur la batte même si ce n'est pas encore parfait. Afin d'améliorer la performance, il faudrait entraîner le réseau davantage ainsi qu'optimiser les hyperparamètres.

V- Conclusion

Ce TD nous a permis de mettre en œuvre l'apprentissage profond par renforcement à l'aide d'environnements de simulation fournis par le toolkit Gym. Comme nous l'avons vu avec le problème du chariot avec le pendule et le jeu Breakout d'Atari, les réseaux de neurones profonds permettent d'avoir d'assez bons résultats et cela illustre tout l'intérêt de leur usage dans ce domaine. Si ces réseaux de neurones peuvent être performants, on peut noter toutefois quelques inconvénients : d'une part, le besoin en ressources de calcul (GPU) pour les entraîner et aussi d'autre part une recherche des bons hyperparamètres qui peut être assez complexe car il peut y en avoir beaucoup. Il n'empêche qu'aujourd'hui, il semble que les réseaux de neurones ont pris le pas sur les techniques classiques d'apprentissage par renforcement et on ne peut que s'en réjouir compte tenu du gain de performance qu'ils peuvent apporter dans des domaines comme la robotique, l'industrie, etc.