TP - Deep Reinforcement Learning

February 23, 2018

Le but de ce TP est de vous faire prendre en main un code simple de Deep RL (`dqn.py`), et de l'améliorer en rajoutant des fonctionnalités.

Vous travaillerez dans une machine virtuelle en python, vous ouvrirez avec `Spyder` le code `dqn.py` et commencerez à l'étudier.

1 Rappels

D'abord, rappelons l'algorithme de base du Q-learning:

algorithme de q-learning

1. Répéter durant M épisodes:
   (a) Pour $t = 1 \dots T$:
      i. Faire l'exploration ϵ-greedy de la façon suivante:
         A. avec une probabilité ϵ, on choisit une action a aléatoirement
         B. avec une probabilité $1 - \epsilon$, on choisit $a = \arg\max Q(s, a)$
      ii. Exécuter l'action a, et reçoit la récompense et l'état suivant r, s'
      iii. Mettre à jour de la fonction Q:
      $Q(s,a) := Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a)) \right]$

Pour l'algorithme du **Deep-Q-learning**, on rappelle que la fonction Q est représentée par une fonction (par exemple ici un réseau de neurones) qui en général prend un état s en entrée, et génère en sortie un vecteur des valeurs de chaque action $(Q(s, a_1), \dots, Q(s, a_n))$. Un exemple d'algorithme de deep-Q-learning est donné en annexe.

2 Exploration de Boltzmann

A la place de l'exploration ϵ-greedy, une alternative consiste à faire de l'exploration de Boltzmann.

Ca consiste à choisir une action a avec la probabilité:

$$p_x(a) = \frac{e^{\frac{Q(x,a)}{T}}}{\sum_{b\in A} \; e^{\frac{Q(x,b)}{T}}}$$

pour une valeur de T fixée manuellement.
Implémentez cette procédure, et comparez le résultat avec l'ϵ-greedy

3 Double Q-Learning

Supposons qu'on ait un ensemble de dés $D_1 \ldots D_n$ tous biaisés différement. Pour trouver le biais le plus élevé (le dé qui donne les nombres les plus élevés en moyenne), on fait naïvement un tirage de chaque dé $A_1 \ldots A_n$, et on prend $\max_i A_i$. Evidement, le biais obtenu est surévalué: si le nombre de dés est grand, on aura $\max_i A_i = 6$, ce qui ne reflête pas le biais d'un dé. Une autre solution moins naïve consiste à calculer $i^* = \arg\max_i A_i$, puis de refaire un tirage de chaque dé $B_1 \ldots B_n$, et de renvoyer B_{i^*}. Le résultat est moins biaisé.

En Q-learning classique, le calcul de $\max_{a'} Q(s', a')$ est biaisé de la même façon. Pour lutter contre ce biais, le double-Q-learning utilise deux fonction Q: la fonction Q^A et la fonction Q^B.

A chaque étape de mise à jour, on tire l'une ou l'autre aléatoirement et on ne mettra à jour que celle-ci, de la façon suivante:

$$Q^A_{t+1}(s,a) \leftarrow Q^A_t(s,a) + \alpha\Big(r + \gamma Q^B_t(s', a^*) - Q^A_t(s,a)\Big) \quad \text{or}$$
$$Q^B_{t+1}(s,a) \leftarrow Q^B_t(s,a) + \alpha\Big(r + \gamma Q^A_t(s', b^*) - Q^B_t(s,a)\Big)$$
$$\text{where} \quad a^* = \arg\max_a Q^A(s', a)$$
$$b^* = \arg\max_a Q^B(s', a)$$

Dans notre algorithme, on aura besoin de deux réseaux de neurones, un qui représente Q^A, et un autre qui représenter Q^B. Implémentez cela.

4 Annexes:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for

Information sur le “cart-pole problem”:
l’état est composé de 4 variables: cart position, velocity, angle and derivative
l’action peut être “gauche” ou “droite”