Reinforcement Learning & Advanced Deep

M2 DAC

TME 11. Multi-Agents RL

Ce TME a pour objectif d'expérimenter le RL pour les environnements multi-agents, et notamment l'approche MADDPG pour environnements multi-agents à actions continues vue en cours.

Environnements MultiAgents

La librairie des environnements multi-agents que nous considérons au cours de ce TME est donnée sur le site de l'UE sous la forme d'une archive zip. Il suffit de la décompresser dans votre répertoire de travail RL (vous devez avoir un repertoire multiagent au même niveau que le repertoire gridworld).

Dans ce TME, nous proposons de considérer les environnements suivants (les scripts sont dans multiagent/scenarios):

- Cooperative navigation (simple_spread.py): 3 agents (en bleu), 3 cibles (en noir). Les agents sont récompensés selon la distance de chaque cible à l'agent le plus proche. Les agents sont pénalisés si ils entrent en collision. Les agents doivent alors apprendre à collaborer pour décider d'une cible pour chacun d'entre eux afin de se rapprocher de chacune en évitant les collisions. Les rewards sont globaux (identiques pour tous les agents).
- Physical deception (simple_adversary.py): 3 agents, dont 2 bons en bleu et 1 adversaire en rouge, 2 bases dont 1 cible en vert. Tous les agents observent les positions des bases et des autres agents (mais ne savent pas laquelle des bases est la cible). Les agents bleus sont récompensés positivement selon la proximité de l'agent bleu le plus proche de la cible et négativement si l'adversaire est proche de la cible. L'adversaire est récompensé selon sa proximité à la cible. Les agents bleus doivent apprendre à collaborer pour couvrir l'ensemble des bases et empêcher l'adversaire d'atteindre la cible. Les agents bleus partagent le même reward, l'adversaire reçoit une récompense individuelle.

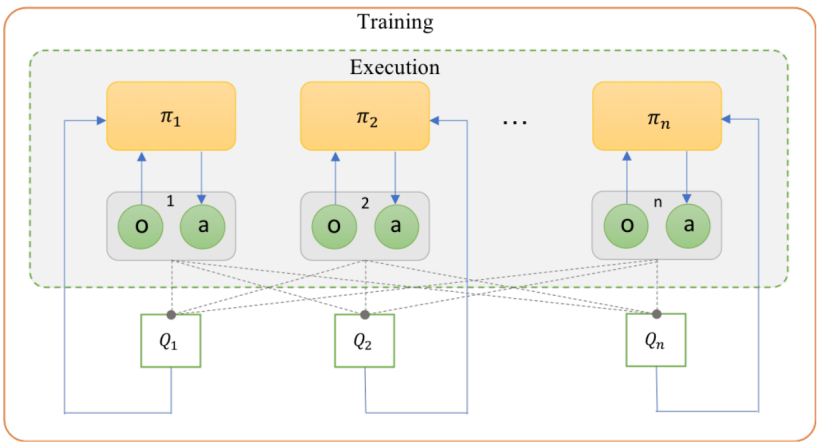
- Predator-prey (simple_tag.py): 3 prédateurs (en rouge), 1 proie (en vert). Les proies sont plus rapides et cherchent à éviter d'être attrapées par les prédateurs. Les prédateurs doivent apprendre à collaborer pour encercler la proie et avoir une chance de l'attraper.

Vous trouverez sur le site de l'UE le code d'une politique aléatoire pour la tâche de Cooperative Navigation. Cela vous montre la manière de se servir des environnements (particulièrement pour le chargement qui est différent de ce qu'on fait habituellement avec les environnements classiques). On note également le fait que maintenant les actions sont des listes d'actions, ainsi que les rewards et les observations retournées par l'environnement. Pour faire la même chose pour les deux autres tâches, il suffit de remplacer simple_spread par simple_adversary ou simple_tag à la ligne $env, scenario, world = make\_env('simple\_tag')$.

Pour vous aider, vous trouverez également une version plus avancée de MADDPG dans le fichier *MADDPG_Start.py*, avec son fichier de configuration correspondant, dans lequel il ne reste plus qu'à spécifier les réseaux à utiliser et écrire la fonction learn de MADDPG. Cette version est dores et déjà fonctionnelle pour les paramètres du fichiers de configuration, qui spécifie des agents aléatoires, mais vous devez l'étendre à des agents du type de ceux considérés dans MADDPG.

MADDPG

L'algorithme MADDPG apprend des politiques individuelles de manière centralisée, de manière à guider l'apprentissage vers des agents qui, selon la tâche collaborent ou s'affrontent de manière collective:



L'idée derrière MADDPG (et l'évaluation de Q selon l'ensemble des actions des agents) est de se dire que si l'on connaît les actions prises par l'ensemble des agents, le problème devient stationnaire même si les politiques des agents évoluent:

$$P(s'|s, a_1, ..., a_N, \pi_1, ...\pi_N) = P(s'|s, a_1, ..., a_N) = P(s'|s, a_1, ..., a_N, \pi'_1, ...\pi'_N)$$

On cherche alors à apprendre des fonctions Q individuelles mais qui prennent en entrée l'ensemble des observations et des actions au temps t. La politique de chaque agent n'est en revanche conditionnée que par sa propre observation, afin de pouvoir être déployé de manière indépendante dans l'environnement cible dans lequel on suppose que les agents ne peuvent pas s'échanger d'infos.
L'algorithme MADDPG est le suivant:

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o'^j_k)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

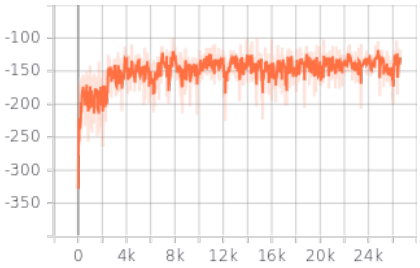
$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

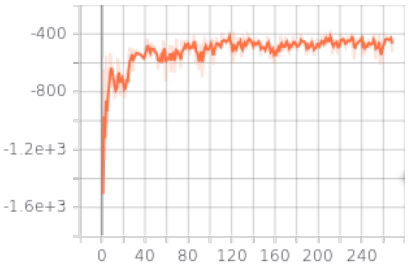
À noter que chaque agent utilise son propre batch d'observations. Selon les configurations x contient l'information de l'état courant ou seulement un ensemble d'observations pour chaque agent. Dans les environnements que l'on considère au cours du TME, on n'obtient qu'une liste des observations $(o_1, \ldots, o_N)$ des N agents. L'apprentissage de Q considère pour chaque transition du batch une cible en fonction de $\mu'_1(o'_1), ..., \mu'_N(o'_N)$ plutôt que l'action destination de la transition pour prendre en compte l'évolution des politiques des différents agents. On considère cependant cela selon des politiques différées $(\mu'_1(o'_1), ..., \mu'_N(o'_N)$ plutôt que $\mu_1(o'_1), ..., \mu_N(o'_N))$ afin d'éviter des changements trop brutaux et stabiliser l'apprentissage de Q. L'apprentissage des politiques des agents ressemble fortement à celui de l'algorithme DDPG, où l'on considère le théorème du PG déterministe.

Appliquer l'algorithme aux 3 problèmes définis à la section précédente.

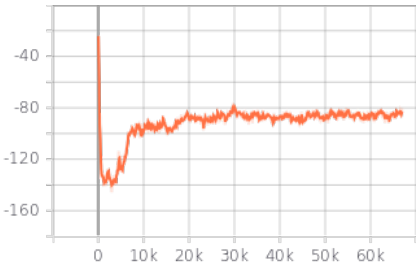
À titre indicatif, voici ce qu'on peut obtenir sur la tâche de Cooperative Navigation avec MADDPG utilisant un processus d'exploration de Ornstein-Uhlenbeck avec $\sigma = 0.2$ (voir core.py pour la méthode d'échantillonnage du bruit, à réinitialiser à la fin de chaque trajectoire), effectuant une optimisation de 1 étape tous les 10 évènements, utilisant un target network Q par agent (mis à jour de manière "soft" à chaque optimisation selon des paramètres $\rho = 0.9$), un discount de 0.95, un pas d'apprentissage de 0.001 pour l'acteur et de 0.01 pour la critique, un batch de 128 transitions échantillonnées dans un buffer de capacité 1000 à chaque pas d'optimisation, un reward instantané clippé entre -100 et 100, une taille d'épisode maximale de 25 évènements en apprentissage (100 en test) et selon des réseaux de neurones acteur et critique à une seule couche cachées de 128 neurones (avec activation leakyRelu sur les couches cachées, pas de batch normalisation, activation finale tanh):



(a) Reward en apprentissage (abscisse: nombre d'episodes)



(b) Reward en test (un test tous les 100 episodes)



(c) Valeur Q moyenne (abscisse: nombre d'optimisations)

Bonus

Pour obtenir des politiques robustes aux changements des autres agents (e.g., éviter de sur-apprendre sur les politiques des agents concurrents), MADDPG propose également de considérer des ensemble de politiques par agent: chaque agent possède K politiques $\mu_i^{(1)}, ..., \mu_i^{(K)}$.

On vise alors à maximiser pour chaque agent i: $\mathbb{E}_{k\sim unif(1,K), s\sim d^\mu, a_i\sim\mu_i^{(k)}}[R_i(s, a_i)]$

Gradient correspondant (avec k replay buffers par agent):

$$\nabla_{\theta_i^{(k)}} J(\mu_i) = \mathbb{E}_{x,a\sim D_i^{(k)}}[\nabla_{\theta_i^{(k)}} \mu_i^{(k)}(o_i) \nabla_{a_i} Q_i^\mu(x, a_1, ..., a_N)|_{a_i=\mu_i^{(k)}(o_i)}]$$

Implémenter cette version de MADDPG et comparer les résultats avec ceux de la version classique.