APRNS-GIA (FIB) Laboratory Assignment

Lab 7: Off-policy Actor-Critic methods:
DDPG and TD3

Gerard Gómez Izquierdo

November 18, 2023

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

1 Introducció

En aquest informe es mostraran els resultats obtinguts a partir de la implementació feta dels algorismes DDPG i TD3. Per mostrar el funcionament correcte dels algorismes i fer una comparació fidel a aquests, s'ha inclòs per cada algorisme el resultat d'executar-lo 3 cops. Per tant, farem una comparació basant-nos en les gràfiques de reforç per episodi resultants de les 3 execucions fetes per algorisme. Després de comparar els resultats es farà una breu explicació on es comentaran els diferents aspectes de la implementació feta per cada algorisme.

Hyper-parameter	**Ours**	**DDPG**
Critic Learning Rate	10^{-3}	10^{-3}
Critic Regularization	None	$10^{-2} \cdot \|\|\theta\|\|^2$
Actor Learning Rate	10^{-3}	10^{-4}
Actor Regularization	None	None
Optimizer	Adam	Adam
Target Update Rate (τ)	$5 \cdot 10^{-3}$	10^{-3}
Batch Size	100	64
Iterations per time step	1	1
Discount Factor	0.99	0.99
Reward Scaling	1.0	1.0
Normalized Observations	False	True
Gradient Clipping	False	False
Exploration Policy	$\mathcal{N}(0, 0.1)$	OU, $\theta = 0.15, \mu = 0, \sigma = 0.2$

Figure 1: Paràmetres utilitzats per execucions de TD3 i DDPG

2 Resultats DDPG

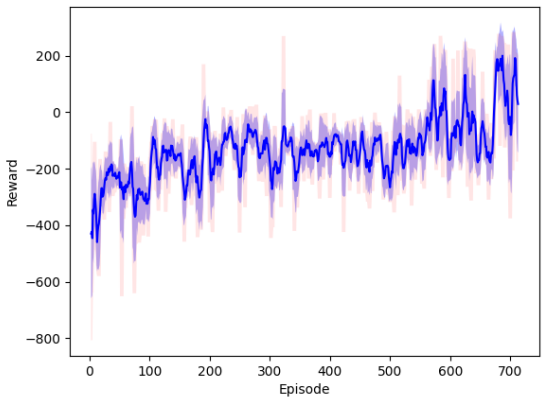

Figure 2: Rewards 1 DDPG

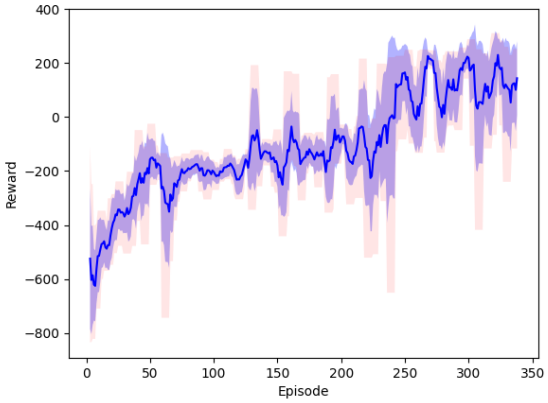

Figure 3: Rewards 2 DDPG

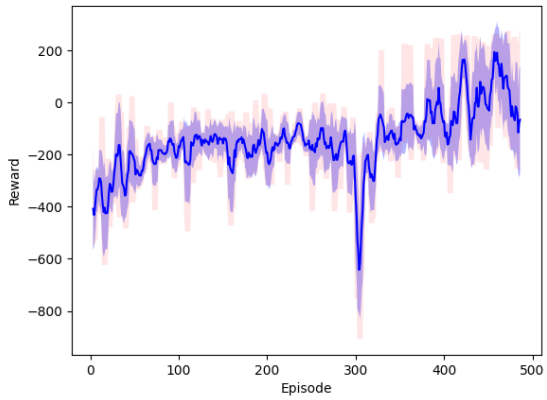

Figure 4: Rewards 3 DDPG

3 Resultats TD3

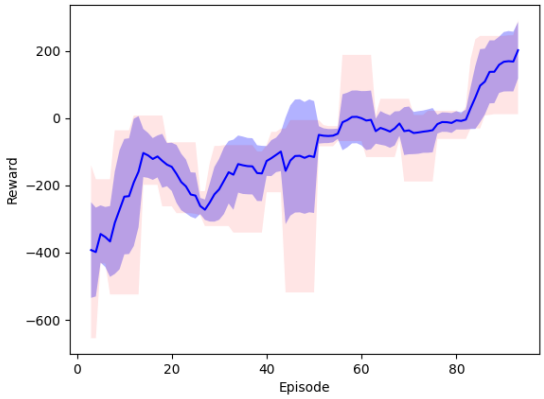

Figure 5: Rewards 1 TD3

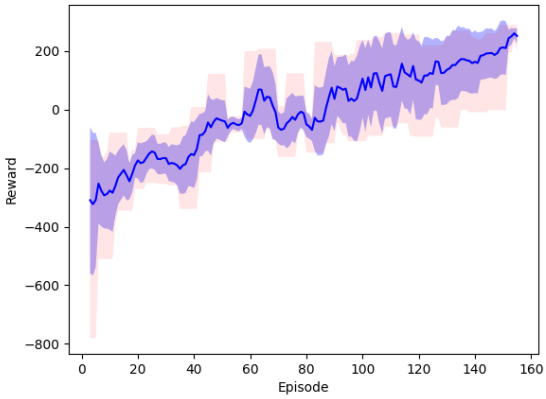

Figure 6: Rewards 2 TD3

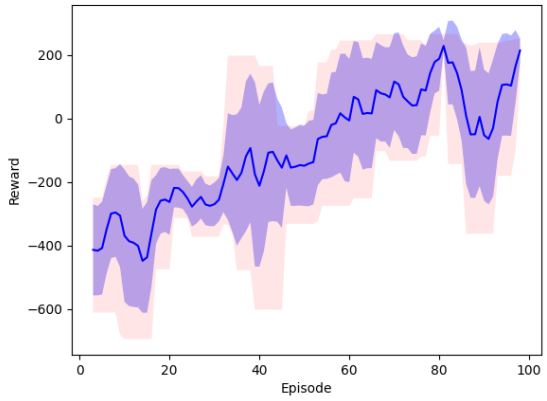

Figure 7: Rewards 3 TD3

4 Comparació de resultats

Modifica aquest text afegint les coses que consideris necessaries. Vull que el resultat sigui un text complet que expliqui sense errades un analisi un es comparen els gràfics dels algorismes TD3 i DDPG. Has de comportarte com un alumne de 10 amb coneixements molt clars i avançants en aprenentatge per reforç:

Tal com podem veure en els gràfics afegits, existeixen notables diferències entre els 2 algoritmes pel que fa al reforç obtingut per episodi al llarg de l'entrenament. A continuació destacarem els 2 aspectes més importants que podem diferenciar entre els dos algorismes.

En primer lloc, la primera diferència que podem observar, és la rapidesa per convergir, ja que en les 3 execucions fetes, l'algorisme DDPG ha necessitat de mitjana 516 episodis mentre que TD3 n'ha necessitat 120. Això és un clar indicador que TD3 és més eficient aprenent de les dades generades per la política.

En segon lloc, podem veure com la corba d'aprenentatge en el cas del DDPG és molt menys estable que a TD3, això ens indica que DDPG és un algorisme molt més sensible a hiperparàmetres i que, en canvi, TD3 és més robust i, per tant, menys sensible a hiperparàmetres. A més, durant l'entrenament d'ambdós algoritmes ens hem trobat que perquè DDPG convergeixi ha calgut utilitzar els paràmetres exactament usats al paper "Addressing Function Approximation Error in Actor-Critic Methods", si no aquest algorisme podia acabar no convergint. En canvi, en el cas de TD3, inicialment vam fer servir hiperparàmetres diferents dels del paper i malgrat tardar més, acabava convergint en un temps raonable igualment.

La solidesa de TD3 es destaca per la seva capacitat d'abordar la sobreestimació utilitzant dues xarxes crítiques per estimar el valor de Q, a diferència de DDPG, que utilitza només una. Aquesta diferència és crítica, ja que quan una sola xarxa Critic sobreestima, DDPG pot caure en un bucle de sobreestimació, mentre que les dues xarxes Critic de TD3 mitiguen aquest risc seleccionant sempre aquella estimació més pessimista.

A més a més, a TD3 s'incorpora un mecanisme addicional per millorar la seva estabilitat. L'ús d'un retard en les actualitzacions de la xarxa Actor respecte a les xarxes Critic disminueix la probabilitat de repetir actualitzacions amb un Critic inalterat. Aquesta estratègia resulta en actualitzacions de política menys freqüents, utilitzant estimacions de valor amb menor variància, i, teòricament, conduint a actualitzacions de política de major qualitat. Fet que es tradueix en un aprenentatge més robust i estable, com es reflecteix en els gràfics presentats.

En conclusió, els resultats observats en els gràfics destaquen que TD3 és un algorisme que permet un entrenament més ràpid i més estable, fets que fan que sigui una opció millor que DDPG en el problema del LunarLander amb un entorn continu.

5 Explicació de la implementació

A continuació s'explicarà per sobre la implementació feta per cada algorisme.

5.1 DDPG

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 8: Algorisme DDPG

En la primera fase de la implementació, vam realitzar la inicialització de les xarxes Critic, Actor, Critic' i Actor', traslladant-les a la GPU. Posteriorment, vam copiar els pesos de les xarxes originals a les xarxes target mitjançant la funció 'copy_target'.

En segon lloc, vam desenvolupar la funció 'choose_action', encarregada de rebre l'estat actual en cada timestep i retornar l'acció que la xarxa Actor prediu com aquella que maximitzarà la recompensa. Per aconseguir-ho, vam convertir l'estat en un tensor, el vam enviar a la GPU i el vam utilitzar com a paràmetre a la xarxa Actor. Després, vam transferir la sortida de la xarxa a la CPU, la vam desvincular i la vam convertir en un array de numpy.

En tercer lloc, vam crear la funció 'compute_critic_loss'. Aquesta funció rep un batch de transicions seleccionades aleatòriament del replay buffer i retorna la pèrdua que s'utilitzarà per actualitzar els pesos de la xarxa Critic. En cada transició, tenim 6 arrays de numpy de mida batch_size que representen l'estat actual, l'acció presa, el següent estat, la recompensa immediata, si l'estat està truncat i si l'estat ha finalitzat. A continuació, convertim tots aquests arrays de numpy a tensors i els passem a la GPU perquè la xarxa Critic pugui treballar amb ells. Seguidament, definim Q' com la sortida de la xarxa Critic' en passar-li com a paràmetres el següent estat i l'acció que prediu Actor' en rebre el següent estat. Posteriorment, calculem y_i (target) fent $recompensa + (1 - finalitzat) * gamma * Q'$, de manera que si estem en un estat terminal, a target se li assigni només el valor de la recompensa immediata. A més, fem detach del target perquè a aquest no se li apliqui backpropagation. Després, li assignem a Q (valor de Q esperat) l'estimació de la xarxa Critic en rebre com a paràmetres l'estat actual i l'acció presa. Finalment, li assignem a la pèrdua el valor de la mitjana de la suma dels errors quadràtics entre cada valor de Q i de 'y' i retornem la pèrdua.

En quart lloc, vam crear la funció 'compute_actor_loss' que funciona igual que 'compute_critic_loss' amb la diferència que només ens cal extreure el batch d'estats actuals, passar-lo a tensor i enviar-lo a la GPU i, posteriorment, calcular la pèrdua fent el negatiu de la mitjana dels valors estimats de Q. Aquests valors de Q els estima la xarxa Critic que rep com a paràmetres el batch d'estats actuals i la predicció de la xarxa Actor en rebre com a paràmetre el batch d'estats actuals. Finalment, retorna la pèrdua.

En cinquè lloc, dins del bucle d'aprenentatge, quan tenim més mostres en el replay buffer que les establertes en la mida del batch, hem seleccionat un batch de mostres de mida batch size, hem computat la pèrdua per la xarxa Critic i Actor amb les funcions explicades en els 2 passos anteriors i mitjançant els gradients hem actualitzat els pesos de les dues xarxes. Per a això, a PyTorch ha calgut cridar per a cada xarxa al mètode '.zero_grad()' de l'optimitzador corresponent, al mètode '.backward()' de la pèrdua i al mètode '.step()' de l'optimitzador.

Finalment, per a cada timestep hem afegit que es faci una petita actualització de les xarxes target mitjançant la funció 'soft_copy' per la xarxa Critic i Actor amb els seus targets corresponents.

5.2 TD3

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ
with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

Figure 9: Algorisme TD3

La implementació de TD3 és molt similar a la de DDPG, salvant les diferències que explicarem a continuació.

En primer lloc, el primer canvi que hem hagut de fer ha estat afegir soroll gaussià a l'acció seleccionada a cada timestep en lloc d'OUNoise. A més a més, dins de la funció 'compute_critic_loss', a l'hora de calcular Q', hem afegit soroll gaussià, clipjeat entre -0.5 i 0.5, a l'acció 'next_action' que rep com a segon paràmetre Crític' per estimar el valor de Q'. També hem clipejat l'acció més el soroll afegit entre -1 i 1. Cal destacar que per afegir el soroll a l'acció estimada per la xarxa Actor' i clipejar-la, hem hagut de passar l'output de la xarxa Actor' a la CPU, fer detach i passar-ho a un array de numpy. Posteriorment, per estimar Q', ha calgut tornar a convertir aquesta acció a tensor i enviar-ho a la GPU per poder passar-la com a segon paràmetre a la xarxa Critic'.

En segon lloc, per evitar la sobreestimació dels valors de Q, hem afegit una nova xarxa Critic2 i Critic2'. Per fer-ho, hem hagut d'inicialitzar-les com en els altres casos i afegir un nou optimitzador per aquesta nova xarxa. Dins de la funció 'compute_critic_loss', hem duplicat el procés de calcular Q' de manera que, a l'hora de calcular el target 'y', utilitzarem aquella Q'_i més petita. Seguidament, calcularem Q (expectació de Q) per les dues xarxes Critic i calcularem la pèrdua per les dues xarxes amb els seus respectius valors de Q i els valors de 'y'. Finalment, retornarem les dues pèrdues.

En tercer lloc, en el bucle d'entrenament, ara computarem els gradients per les dues xarxes Critic i actualitzarem els seus pesos fent ús dels seus optimitzadors corresponents.

Finalment, donat que en les arquitectures Actor-Critic, la xarxa Actor està guiada per les estimacions de la xarxa Critic, ens interessa que la xarxa Critic aprengui més ràpid que l'Actor. Per aquesta raó, hem afegit un retard d'aprenentatge a l'hora de computar els gradients i actualitzar els pesos de la xarxa Actor. Aquest retard l'hem afegit fent que en lloc de cada 1, cada 2 timesteps s'actualitzin els pesos.