Model-free RL

Model-free → sans modèle → Des situat° où l'agent n'a pas des connaissances de proba de transition entre les états et des récompenses associés. + Cas d'actions discrètes → Modélisation par PDN (Naïka)

commence sans info → explorer l'env pour acquérir ces informations

Agent: teste chaque état et chaque transition pour découvrir les récompenses.
↓
Il faut plusieurs fois pour pouvoir estimer les proba de transitions

Q-Learning: Model-free (sans modèle)
↓
apprend directement de la fonct° Q à partir de l'expérience sans avoir besoin d'un modèle de l'env
Le initialement: l'agent connaît les états et les act° possibles

• Approche de Monte Carlo pour estimer Q:
→ l'agent apprend à partir d'épisodes complets d'interactions avec l'env.
→ MC attend la fin de l'épisode pour mettre à jour les valeurs des états.
→ Après chaque épisode, l'agent attribue des récompenses réelles à chaque état visité pour obtenir une estimation de la valeur de chaque état.

→ mise à jour de la valeur: en fonct° des récompenses réelles obtenues.
Episode: état visité → mise à jour ?
$V_{new} = \cdot val + \frac{1}{m}(R - Val)$

• TD Learning:
→ Apprentissage par différence temporelle
→ met à jour les valeurs des états à chaque pas de temps : l'agent apprend plus rapidement et de manière incrémentielle.
→ Estimat° de la valeur de l'état actuel: en utilisant R, la val de l'état suiv pondérée par un facteur d'apprentissage.

• Exploration / Exploitation
Code: Greedy

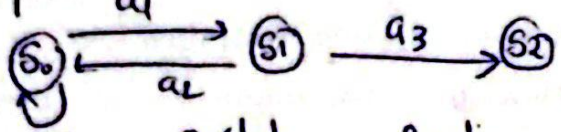
```
class EpsilonGreedyPolicy:
  def __init__(self, epsilon):
      self.epsilon = epsilon
  def select_action(self, q_values):
    p = random.uniform(0,1)
    if p < self.epsilon:
      action = random.choice(range(len(q_values)))
    else:
      action = max(range(len(q_values)), key=
      lambda a: q_values[a])
    return action
Epsilon_value = 0,1
Epsilon_greedy_policy = EpsilonGreedyPolicy(
                epsilon_value)
q_values_ex = [0,2 , 0.8, 0.1, 0.3]
select_act = Epsilon_greedy_policy.select_acti
          (q_values_ex)
```

ChoixEpsilon → Cree instance de class → selec une act° en fonct° de Q_values
remplacé par les Q_val de l'env

• SARSA:
→ C'est un algorithme qui met à jour les valeurs Q en utilisant une stratégie de politique ε-greedy. Voir le code

• Q-Learning: env + Act° → Q-learning
→ C'est un algo qui apprend les Q_values pour chaque paire état-action dans un env.

Exemple:



3 états 2 actions

$R(S_0) = -0.1$ $R(S_1) = -0.1$ $R(S_2) = 1$

$\alpha = 0.5$ $\gamma = 0.5$

au dem: l'agent ens S_0 et choisit a_2

La formule de la mise à jour de la Q-table

$Q(s,a) = Q(s,a) + \alpha [R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

- $(S_0)_{-0.1}$

Act° à prendre $\pi(S_0) = \text{argmax}(Q(S_0,a_1), Q(S_0,a_2))$

$= \text{argmax}(0,0) = a_2$

- $(S_1)_{-0.1} \xrightarrow{a_2} (S_0)_{-0.1}$

$Q(S_0,a_2) = Q(S_0,a_2) + \alpha (R(S_0) + \gamma \max\{-,-\} - Q(S_0,a_2))$

$= 0 + 0.5 (-0.1 + 0.5 \max(0,0) - 0) = -0.05$

$\pi(S_0) = \text{argmax}\{\underbrace{Q(S_0,a_1)}_{=0}, \underbrace{Q(S_0,a_2)}_{=-0.05}\} = a_1$

- $(S_0)_{-0.1} \xrightarrow{a_2} (S_0)_{-0.1} \xrightarrow{a_1} (S_1)_{-0.1}$

$Q(S_0,a_1) = -0.05$

$\pi(S_1) = a_2$

- $(S_0)_{-0.1} \xrightarrow{a_2} (S_0)_{-0.1} \xrightarrow{a_1} (S_1)_{-0.1} \xrightarrow{a_2} (S_0)_{-0.1}$

$Q(S_1,a_2) = -0.0625$

$\pi(S_0) = a_1$

- $(S_0)_{-0.1} \xrightarrow{a_2} (S_0)_{-0.1} \xrightarrow{a_1} (S_1)_{-0.1} \xrightarrow{a_2} (S_0)_{-0.1} \xrightarrow{a_1} (S_1)_{-0.1}$

$Q(S_0,a_1) = -0.075$

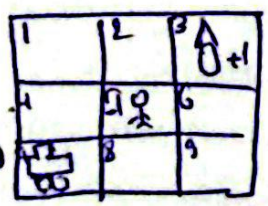
$\pi(S_1) = a_3 \longrightarrow (S_2)_1$ Etat terminal

$Q(S_2, \text{None}) = 1$

$Q(S_1,a_3) = 0.2$

Etat	Act°
S_0	$Q(S_0,a_1) = -0.05$ $Q(S_0,a_2) = -0.05$
S_1	$Q(S_1,a_2) = -0.062$ $Q(S_1,a_3) = 0.2$

Exemple:



- l'agent initialement à 7

Les déplacements: droite, gauche, haut, bas

Les récompenses: -1 en 5, 1 en 3, 0 partout

→ Epoques = 100 $\alpha = 0.1$ $\gamma = 0.9$

Etat 1 → 9 $Q = (10)_{9 \times 4}$

$a=0$ (↑) $a=1$ (↓) $a=2$ (←) $a=3$ (→)

La fonction take_action détermine l'act° à adopter:

```
def take_action (st, Q, eps):
    if random.uniform(0,1) < eps:
        action = randint (0, 3)
    else:
        action = mp.argmax (Q[st])
    return action
```

Mise à jour de la Q-table:

```
while not env.is_finished():
    at = take_action (st, Q, 0.4)
    stp1, r = env. step (at)
    atp1 = take_action (stp1, Q, 0.0)
    Q[st][at] = Q [st] [at] + 0.1 * (r + 0.9 *
    Q [stp1] [atp1] - Q [st] [at]
    st = stp1
```

- Deep Q-Learning:

→ utilise les réseaux de neurones profonds pour représenter et estimer la fonct° Q

→ Architecture:

- Estimer la cible y_i et $Q(s,a)$ à l'aide de deux réseaux neuronaux distincts.

Target Network Q-Network Séparat° des réseaux

- θ_{i-1}: (poids, biais) ∈ Reseau cible

θ_i ∈ Reseau Q

- s: état actuel

reseau Q: calcul les val d'act° $Q(s,a)$

reseau cible: calcul les $Q(s',a)$ en utilisant l'état suivant pour coms bruit la cible TD.

Experience Replay:

→ Utilisé pour améliorer l'efficacité et la stabilité de l'entraînement: l'agent stocke les expériences dans une mémoire de répétition et échantillonne un lot d'exp de manière aléatoire pour choisir l'action d'entraî

Replay memory :

→ Structure de données utilisée pour stocker les expériences passées en vue de l'entraînement ultérieur.

→ tampon qui contient un nombre fixe des expériences les plus récentes

→ Chaque exp est représentée par un tuple (s, a, r, s') → l'état suivant

état actuel ← act° effectuée → recomp obtenue

Deep reinforcement Learning

Hector Critic

Le critique : méthode qui évalue chaque action que l'agent prend dans l'env avec une scalaire >0 ou <0 (Q-learn DQN...)

Acteur : politique paramétrée qui définit comment les actions sont sélectionnées

Policy Gradient : paramètre la fonct° de politique et mettre à jour les params dans la direction qui augmente le rendement attendu.

→ Le critique évalue les act° choisies par l'acteur

→ L'acteur est mise à jour dans la direction suggérée par le critique ce qui stabilise l'entraînement et réduit la variance

Policy Gradient Methods : optimisent directement la politique

Actor-Critic Methods : utilise une fonct° de valeur pour guider les mise à jour de la politique.

Politique : stratégie qui dicte comment un agent devrait prendre des décisions dans un env donné.

PPO algorithms :

→ appartient à la famille policyGradient

→ Résout certains problèmes de stabilité et de convergence associés à d'autres algo de Policy Gradient.

- initialise un réseau neuronal pour représenter la politique $\pi_\theta(a|s)$ où θ sont les params du réseau
- initialise un réseau neuronal pour représenter la valeur $V_\phi(s)$ où ϕ sont les params du réseau

- interagit avec l'env pour collecter des états / act° / récomp
- Calculez les avantages $A(s,a)$ en utilisant les récompenses observées et les estim de val
- Calcule la fonct° objective POO = combin de l'obj politique L^{clip} et l'obj de valeur L^{VF}
  - maximise la proba des act° qui ont été prises
  - minimise la diff quadratique moyenne entre les estim et les avantages calculés de valeur
- utilise un algo d'optimisation pour mettre à jour les params du réseaux de politique et du reseau de valeur en minimisant l'obj POO.

$$r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

- >1 : more likely in the current policy than the old policy
- $0 < \cdot < 1$: less likely for the current policy than for the old one

The objectif function :

$$L^{clip}(\theta) = \hat{E}\left[\min\left(r_t(\theta)\cdot\hat{A}_t,\ clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\cdot\hat{A}_t\right)\right]$$