Introduction au reinforcement Learning

Machine Learning : apprentissage non supervisé :

obj : découvrir les structures sous-jacentes à des données non étiquetées

Données : Données non étiquetées 0 1

Exp : détectation d'anomalie

Machine Learning : Apprentissage supervisé

obj : apprendre des fonctions de prédiction à partir des données annotées

Données : Données étiquetées

Exp : prédiction d'une valeur dans des problèmes de régression Classif/Regression

Reinforcement Learning : Apprentissage par renforcement

obj : Laisser l'algo apprendre de ses propres erreurs.
choix aléa au début → se trompe → puni
↳ sinon → récompense

Données : Etats et Actions

Obj de l'app supervisé et non supervisé :
↳ La minimisation de l'erreur

Obj de l'apprentissage par renforcement :
↳ maximiser la récompense

Les principaux composants de l'app par renforcement :

- Agent : l'entité qui prend des décisions dans l'environnement.
- Etat : Représentation de la situation Actuelle de l'agent dans l'env : $S = \{$position, vitesse, ...$\}$
- Action : La décision prise par l'agent à un moment donné $A = \{Act1, ...\}$
- Récompense : $\forall t, \forall s_t, \forall a_t$, il reçoit une r_t ... si l'agent se comporte bien et $r_t < 0$ sinon.

En RL : un agent procède à des observations et réalise des actions au sein d'un env. Il reçoit en retour des récompenses. 14

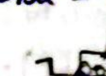
Exemple :

Finance :
Agent : observe les prix des actions
Décide le nbr d'act° à acheter ou vendre
Récompenses : dépendent des gains et des pertes

Politique : stratégie à suivre pour atteindre un obj

$\pi : S \to A$: associe à chaque état s une action a

Exemple : Aspirateur
→ Récomp : volume de poussière ramassé en 30min
→ Politique : avancer avec proba $= p$
tourner avec proba $= 1 - p$

Exemple : CartPole 
→ Politique : déclenche une accélération vers la gauche lorsque le bâton penche vers la gauche et vice versa.

Evaluation des actions :

Dans $s_t \in S$, l'agent émet l'action $a_t \in A$ et reçoit une récomp r_{t+1}

seq de récompense : $r_{t+1}, r_{t+2}, ...$

→ la fonction de récompense :

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

$\gamma \in [0,1]$: paramètre de rabais
↳ détermine la valeur présente des récompenses futures

$\gamma = 0 \Rightarrow$ agent myope : maximise les récompenses immédiates

$\gamma \to 1 \Rightarrow$ agent avec un horizon de plus en plus lointain

OpenAI Gym 1 : c'est un défis qui a un environnement fonctionnel pour

entraîner un agent, il fournit divers env simulés tq les jeux Atari, des jeux de société, des simulations physique 2D ou 3D qui permet de entraîner, comparer et développer des algorithmes d'apprentissage par renforcem

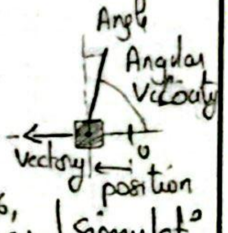
Cartpole environnement I

↳ utilisé pour tester et développer des algo

↳ obj : maintenir un poteau en équilibre aussi longtemps que possible en déplaçant un chariot horizontalement.

```
import gym
env = gym.make("CartPole-v1")
obs = env.reset()
obs → array([-0,0127, -0,00156,
             0,042 , -0,0018])
```

↳ renvoie l'env passé en paramètre



gym.make : crée un environnement

env.reset() : réinitialise l'env, et renvoie une obs initiale   création → réinitialisat°

→ Les obs dépendent du type d'env

l'env = Cartpole, alors les obs sont 4 réels :

l'a position horizontale du chariot, (centre 0,0)

sa vitesse (≥ 0 vers la droite, < 0 vers la g)

l'angle du poteau (0,0 vertical)

Vitesse Ang du poteau   Angle orienté

env.render() : affiche une fenêtre contenant rendu à chaque nouvelle action prise à chaque étape.

env.render()   → affiche l'environnement

>> True

env.action_space : pour définir les caract de l'espace d'action de l'env : on peut définir si l'espace d'act° est continu ou discret, définir les valeurs min et max des actions, ...

env.action_space

>> Discrete(2) → càd les actions possibles sont 0 et 1 : l'accélération vers la gauche (0) ou vers la droite (1).

→ peut avoir des autres act° supp pour les autres env, ou d'autres type d'action

env.observation_space

>> Box(4,) vecteur à 4 dim de nbr réels

env.observation_space.low } découvrir
env.observation_space.high } la plage

de chaque variable d'observation

↳ Les valeurs min et max

env.step() : effectue une act° à chaque étape. l'act° est spécifiée en tant que param

env.step() : retourne 4 params (obs, récompense, done et info)

Le poteau → droite alors obs(2) = vitesse ≥ 0

l'acc est vers la droite, action = 1

```
obs, reward, done, info = env.step(action)
```

[...] ↓ → récomp réel l'échelle varie selon l'env

done ↓ booléen → indique s'il est temps de réinitialiser l'env à nouveau

info → des info diagnostiques

Ecrire un Script qui teste une politique aléat dans l'env CartPole

```
import gym
import random
env = gym.make("CartPole-v1")
def tester_polit_alea(env, nb_ep=5):
  for e in range(nb_ep):
     obs = env.reset()
     total_reward = 0
     while True:
        env.render()
        act = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
```

```
if done:
    print('epis', e+1, 'Rec', totalrew)
env.close()
```

Processus de Decision de Markov:

- hypothèse de Markov: $\forall t$, la proba de passer de s à s' en faisant une act° a
  $P(s_{t+1} = s' | s_t = s, a_t = a)$ → ne dépend pas des états précédents

Un problème de RL est défini comme un PDM (chaîne de Markov) où T et R sont inconnus

$T : S \times A \times S \rightarrow [0,1]$: fonct° de transition qui donne la proba d'arriver en s'

$T(s,a,s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$

$R : S \times A \rightarrow \mathbb{R}$: fonct° de récompense

$R(s,a) = E(r_{t+1} | s_t = s, a_t = a)$

Fonction valeur d'un état s:

$V^{\pi}(s) = E(R_t | s_t = s)$
$= E(\sum \gamma^k r_{t+k+1} | s_t = s)$

Fonction état-valeur:

$Q^{\pi}(s,a) = E(R_t | s_t = s, a_t = a)$
$= E(\sum \gamma^k r_{t+k+1} | s_t = s, a_t = a)$

Fonction valeur optimale

$V^*(s) = \max_a [R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^{\pi}(s')]$

→ Si l'agent agit de façon optimale

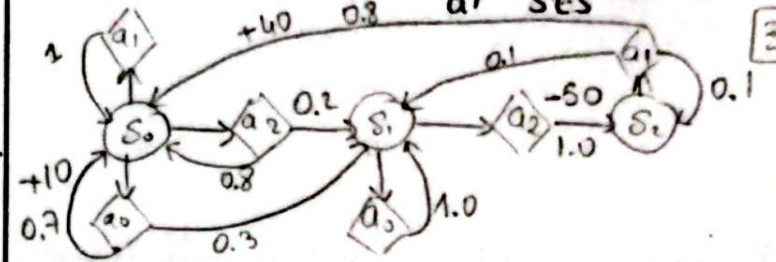
$V_{op}(\text{état courant}) = R(a_{optim}) + \sum V_{op}(\text{état suiv})$

$V_{k+1}(s) = \max_a [R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_k(s')]$

Equation d'optimalité de Bellman pour estimer la fonct° état-valeur optimale:

$Q^*(s,a) = R(s,a) + \gamma \max_{a'} (\sum_{s' \in S} T(s,a,s') Q^*(s',a'))$

$Q_{k+1}(s,a) = R(s,a) + \gamma \max_{a'} (\sum_{s' \in S} T_{ss'}^{a} Q_k(s',a'))$



3 états, 3 actions,

S_0 → choisir a_1 → reste dans S_0 avec certitude et sans aucune récompense
→ choisir a_0 → 70% chance obtenir récomp +10 et rester dans S_0
→ choisir a_2 → se trouve dans S1 avec proba de 0,2

reste immobile ← a_0 / a_2 → -50 يخسر
a_1 ← S2

Q-value iteration algo

→ c'est une méthode utilisée pour déterminer la fonction d'action-valeur optimale d'un proc de décision markovien

Q : la récompense cumulative attendue de prendre une action spécifique dans un état particulier.

1/ Définir le MDP
- → transitions_proba
- → rewards
- → possible_actions

2/ $Q_{k+1}(s,a)$ $\forall (s,a)$

3/ gamma = 0.9

```
for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s,a] = np.sum([
                transition_proba[s][a][sp] *
                (rewards[s][a][sp] +
                gamma * np.max(Q_prev[sp]))
            for sp in range(3)])
```

→ Q_values result: array([[,], [,] 0, 17, -inf... → [,]])

Q values: 18,92 17,03 13,62
0 -∞ -4.88
-∞ 50.13 -∞

(S0) → (a1)
+17 ← rewards

Pour chaque état, l'action qui a la Q-value la plus élevé:

np.argmax(Q_values, axis=1)
array([0,0,1])

Catégorisation des techniques de RL:

- RL basé sur la valeur:
→ valeur Q: Estimation des Q-values représentant la qualité des actions dans un état donné.
Q-Learning / SARSA
→ Valeur V: Estimer la valeur d'un état particulier Dynamic programming
- RL basé sur la politique:
→ Politique déterministe: qui mappe les états aux actions directement
Policy Gradient Methods
→ Politique stochastique: les politiques sont traitées comme des distributions de proba sur les actions
Trust Region Policy Optimization
- Méthodes basée sur la modélisation:
→ Model-based: utilisent des modèles internes pour simuler l'env et planifier des actions NPC
→ Model-free: ne mémorise pas de modèle explicite de l'env.
Q-learning / Policy Gradient Methods
- Exploration et exploitation:
→ Exploration: essayer de nouvelles Actions pour découvrir de nouvelles actions et états pour améliorer la connaissance de l'agent sur l'env.
Epsilon Greedy
→ Exploitation: utilisation de la connaissance actuelle pour prendre des décisions optimales
Greedy Policy
borne
Récompense

- Deep Learning:
→ Réseaux de neurones: intégration de réseaux de neurone pour présenter et apprendre des politiques ou des valeurs plus complexes. Deep Q-Networks (DQN)
Actor Critic, DDQN
Deep Policy Network, NPC

=> Ces techniques sont utilisées pour entraîner des agents intelligents à prendre des décisions dans un env donné pour maximiser une récompense cumulative au fil du temps.