

Cours Machine Learning

Chapitre 09 Reinforcement Learning

Objectifs d'apprentissage :

- Comprendre le paradigme du Reinforcement Learning
- Maîtriser Q-Learning et Deep Q-Network (DQN)
- Découvrir Policy Gradient et Actor-Critic
- Appliquer le RL à des environnements pratiques

Prérequis : Chapitres 06 (MLP)

Durée estimée : 6-8 heures

Table des matières

1	Introduction au Reinforcement Learning	2
1.1	Paradigme RL vs Supervised Learning	2
1.2	Composants du RL	2
2	Q-Learning	2
2.1	Q-Function	2
2.2	Algorithme Q-Learning	3
2.3	Implémentation	3
3	Deep Q-Network (DQN)	3
3.1	Motivation	3
3.2	Innovations DQN	3
3.3	Algorithme	4
3.4	Implémentation PyTorch	4
4	Policy Gradient	6
4.1	Policy directe	6
4.2	REINFORCE	6
5	Actor-Critic	6
5.1	A3C (Asynchronous Advantage Actor-Critic)	7
6	Applications	7
7	Résumé	7
7.1	Points Clés	7
7.2	Formules Essentielles	7
8	Pour Aller Plus Loin	7
8.1	Lectures	7
8.2	Environnements	8
9	Notebooks Pratiques	8

1 Introduction au Reinforcement Learning

1.1 Paradigme RL vs Supervised Learning

	Supervised Learning	Reinforcement Learning
Données	Labels explicites	Récompenses différées
Feedback	Immédiat	Delayed
Objectif	Prédire y depuis x	Maximiser récompense cumulative
Exemples	Classification, Régression	Jeux, Robotique

1.2 Composants du RL

Définition

Markov Decision Process (MDP) Un MDP est défini par (S, A, P, R, γ) :

- S : États (states)
- A : Actions
- $P(s'|s, a)$: Probabilités de transition
- $R(s, a, s')$: Fonction de récompense
- $\gamma \in [0, 1]$: Facteur de discount

Objectif : Apprendre une **policy** $\pi(a|s)$ qui maximise la récompense cumulative :

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (1)$$

2 Q-Learning

2.1 Q-Function

Définition

Q-Function La Q-function représente la récompense cumulative attendue en prenant l'action a dans l'état s puis en suivant la policy π :

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \quad (2)$$

Équation de Bellman :

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a') \quad (3)$$

2.2 Algorithme Q-Learning

Algorithm 1 Q-Learning

Require: Environnement, learning rate α , discount γ , exploration ϵ

- 1: Initialiser $Q(s, a) = 0$ pour tout s, a
 - 2: **for** chaque épisode **do**
 - 3: Initialiser état s
 - 4: **repeat**
 - 5: Choisir action $a : \epsilon$ -greedy sur $Q(s, \cdot)$
 - 6: Exécuter a , observer r, s'
 - 7: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
 - 8: $s \leftarrow s'$
 - 9: **until** s terminal
 - 10: **end for**
-

2.3 Implémentation

Listing 1 – Q-Learning simple

```

1 import numpy as np
2
3 class QLearning:
4     def __init__(self, n_states, n_actions, alpha=0.1, gamma=0.99,
5                  epsilon=0.1):
6         self.Q = np.zeros((n_states, n_actions))
7         self.alpha = alpha
8         self.gamma = gamma
9         self.epsilon = epsilon
10
11    def choose_action(self, state):
12        if np.random.rand() < self.epsilon:
13            return np.random.randint(self.Q.shape[1]) # Explore
14        return np.argmax(self.Q[state]) # Exploit
15
16    def update(self, state, action, reward, next_state):
17        target = reward + self.gamma * np.max(self.Q[next_state])
18        self.Q[state, action] += self.alpha * (target - self.Q[state, action])

```

3 Deep Q-Network (DQN)

3.1 Motivation

Pour des espaces d'états larges/continus (images), stocker $Q(s, a)$ dans une table devient impossible. Solution : approximer Q avec un réseau de neurones.

3.2 Innovations DQN

1. **Experience Replay** : Stocker transitions dans buffer, échantillonner mini-batches

2. **Target Network** : Réseau cible gelé pour stabiliser l'entraînement

3.3 Algorithme

Algorithm 2 Deep Q-Network (DQN)

- 1: Initialiser réseau $Q(s, a; \theta)$ et réseau cible $Q(s, a; \theta^-)$
 - 2: Initialiser replay buffer D
 - 3: **for** chaque épisode **do**
 - 4: Initialiser état s_1
 - 5: **for** $t = 1, \dots, T$ **do**
 - 6: $a_t = \epsilon\text{-greedy}(s_t)$
 - 7: Exécuter a_t , observer r_t, s_{t+1}
 - 8: Stocker (s_t, a_t, r_t, s_{t+1}) dans D
 - 9: Échantillonner mini-batch de D
 - 10: Calculer target : $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$
 - 11: Gradient descent sur $(y - Q(s, a; \theta))^2$
 - 12: Mettre à jour θ^- périodiquement
 - 13: **end for**
 - 14: **end for**
-

3.4 Implémentation PyTorch

Listing 2 – DQN avec PyTorch

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import random
5 from collections import deque
6
7 class DQN(nn.Module):
8     def __init__(self, state_dim, action_dim, hidden_dim=128):
9         super(DQN, self).__init__()
10        self.fc = nn.Sequential(
11            nn.Linear(state_dim, hidden_dim),
12            nn.ReLU(),
13            nn.Linear(hidden_dim, hidden_dim),
14            nn.ReLU(),
15            nn.Linear(hidden_dim, action_dim)
16        )
17
18        def forward(self, x):
19            return self.fc(x)
20
21 class ReplayBuffer:
22     def __init__(self, capacity=10000):
23         self.buffer = deque(maxlen=capacity)
24
25     def push(self, state, action, reward, next_state, done):

```

```

26         self.buffer.append((state, action, reward, next_state, done))
27
28     def sample(self, batch_size):
29         return random.sample(self.buffer, batch_size)
30
31     def __len__(self):
32         return len(self.buffer)
33
34 class DQNAgent:
35     def __init__(self, state_dim, action_dim):
36         self.policy_net = DQN(state_dim, action_dim)
37         self.target_net = DQN(state_dim, action_dim)
38         self.target_net.load_state_dict(self.policy_net.state_dict())
39
40         self.optimizer = optim.Adam(self.policy_net.parameters(), lr=1e
41                                     -3)
42         self.memory = ReplayBuffer()
43         self.gamma = 0.99
44         self.epsilon = 1.0
45         self.epsilon_decay = 0.995
46         self.epsilon_min = 0.01
47
48     def select_action(self, state):
49         if random.random() < self.epsilon:
50             return random.randint(0, self.policy_net.fc[-1].out_features
51                                   - 1)
52
53         with torch.no_grad():
54             state_tensor = torch.FloatTensor(state).unsqueeze(0)
55             q_values = self.policy_net(state_tensor)
56             return q_values.argmax().item()
57
58     def train(self, batch_size=64):
59         if len(self.memory) < batch_size:
60             return
61
62         batch = self.memory.sample(batch_size)
63         states, actions, rewards, next_states, dones = zip(*batch)
64
65         states = torch.FloatTensor(states)
66         actions = torch.LongTensor(actions)
67         rewards = torch.FloatTensor(rewards)
68         next_states = torch.FloatTensor(next_states)
69         dones = torch.FloatTensor(dones)
70
71         # Q(s, a)
72         q_values = self.policy_net(states).gather(1, actions.unsqueeze
73                                                 (1))
74
75         # Target: r + gamma * max Q(s', a')
76         with torch.no_grad():

```

```

74         next_q_values = self.target_net(next_states).max(1)[0]
75         targets = rewards + self.gamma * next_q_values * (1 - dones)
76
77     # Loss et backprop
78     loss = nn.MSELoss()(q_values.squeeze(), targets)
79
80     self.optimizer.zero_grad()
81     loss.backward()
82     self.optimizer.step()
83
84     # Decay epsilon
85     self.epsilon = max(self.epsilon_min, self.epsilon * self.
86                         epsilon_decay)
87
88     def update_target_network(self):
89         self.target_net.load_state_dict(self.policy_net.state_dict())

```

4 Policy Gradient

4.1 Policy directe

Au lieu d'apprendre Q puis dériver policy, apprendre directement $\pi_\theta(a|s)$.

Définition

Policy Gradient Theorem Le gradient de l'objectif $J(\theta) = \mathbb{E}_\pi[G_t]$ est :

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\theta \log \pi_\theta(a|s) \cdot G_t] \quad (4)$$

4.2 REINFORCE

Algorithm 3 REINFORCE

-
- 1: Initialiser policy network π_θ
 - 2: **for** chaque episode **do**
 - 3: Générer épisode $\{s_1, a_1, r_1, \dots, s_T, a_T, r_T\}$ en suivant π_θ
 - 4: **for** $t = 1, \dots, T$ **do**
 - 5: $G_t \leftarrow \sum_{k=t}^T \gamma^{k-t} r_k$
 - 6: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t$
 - 7: **end for**
 - 8: **end for**
-

5 Actor-Critic

Combiner policy gradient (actor) et value function (critic).

- **Actor** : Policy $\pi_\theta(a|s)$
- **Critic** : Value function $V_\phi(s)$

Advantage :

$$A(s, a) = Q(s, a) - V(s) \approx r + \gamma V(s') - V(s) \quad (5)$$

5.1 A3C (Asynchronous Advantage Actor-Critic)

Version parallélisée avec plusieurs agents explorant en parallèle.

6 Applications

Exemple

Domaines d'application

- **Jeux** : AlphaGo, Atari (DQN), Dota 2, StarCraft
- **Robotique** : Manipulation d'objets, locomotion
- **Recommandation** : Systèmes adaptatifs
- **Finance** : Trading algorithmique
- **Ressources** : Optimisation énergétique, data centers

7 Résumé

7.1 Points Clés

- **MDP** : États, actions, récompenses, transitions
- **Q-Learning** : Apprentissage de $Q(s, a)$ par équation de Bellman
- **DQN** : Approximation neuronale + replay buffer + target network
- **Policy Gradient** : Optimisation directe de la policy
- **Actor-Critic** : Combinaison policy + value function

7.2 Formules Essentielles

Formules à retenir

Q-Learning Update :

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

Policy Gradient :

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi}[\nabla_{\theta} \log \pi_{\theta}(a|s) \cdot G_t]$$

8 Pour Aller Plus Loin

8.1 Lectures

- Sutton & Barto - *Reinforcement Learning : An Introduction*
- Mnih et al. (2015) - "Human-level control through deep RL" (DQN)
- Silver et al. (2016) - "Mastering the game of Go with deep neural networks" (AlphaGo)

8.2 Environnements

- OpenAI Gym : gym.openai.com
- Stable Baselines3 : Implémentations RL

9 Notebooks Pratiques

Ce chapitre est accompagné des notebooks suivants :

- `09_demo_qlearning.ipynb` : Implémentation de Q-Learning sur FrozenLake
 - Algorithme Q-Learning from scratch
 - Environnement OpenAI Gym FrozenLake
 - Construction et visualisation de la Q-table
 - Analyse de la convergence
- `09_demo_dqn.ipynb` : Deep Q-Network (DQN) sur CartPole
 - Architecture DQN avec PyTorch
 - Experience Replay et Target Network
 - Entraînement et évaluation
 - Visualisation des performances