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

Cours ML - Sandbox-ML
Version 1.0 - 2026

Table des matières

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 :

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

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 :

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

Équation de Bellman :

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

2.2 Algorithme Q-Learning

Algorithm 1 Q-Learning

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

Initialiser $Q(s, a) = 0$ pour tout s, a
for chaque episode **do**
  Initialiser état s
  repeat
    Choisir action a : ϵ-greedy sur $Q(s, \cdot)$
    Exécuter a, observer r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  until s terminal
end for

2.3 Implémentation

Listing 1 – Q-Learning simple

```python
import numpy as np

class QLearning:
    def __init__(self, n_states, n_actions, alpha=0.1, gamma=0.99,
        epsilon=0.1):
        self.Q = np.zeros((n_states, n_actions))
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon

    def choose_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.randint(self.Q.shape[1])  # Explore
        return np.argmax(self.Q[state])  # Exploit

    def update(self, state, action, reward, next_state):
        target = reward + self.gamma * np.max(self.Q[next_state])
        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)

Initialiser réseau $Q(s, a; \theta)$ et réseau cible $Q(s, a; \theta^-)$
Initialiser replay buffer D
for chaque episode **do**
  Initialiser état s_1
  for $t = 1, \ldots, T$ **do**
    $a_t = \epsilon$-greedy(s_t)
    Exécuter a_t, observer r_t, s_{t+1}
    Stocker (s_t, a_t, r_t, s_{t+1}) dans D
    Échantillonner mini-batch de D
    Calculer target : $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$
    Gradient descent sur $(y - Q(s, a; \theta))^2$
    Mettre à jour θ^- périodiquement
  end for
end for

3.4 Implémentation PyTorch

Listing 2 – DQN avec PyTorch

```python
import torch
import torch.nn as nn
import torch.optim as optim
import random
from collections import deque

class DQN(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(DQN, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim)
        )

    def forward(self, x):
        return self.fc(x)

class ReplayBuffer:
    def __init__(self, capacity=10000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
```

```
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

    def __len__(self):
        return len(self.buffer)

class DQNAgent:
    def __init__(self, state_dim, action_dim):
        self.policy_net = DQN(state_dim, action_dim)
        self.target_net = DQN(state_dim, action_dim)
        self.target_net.load_state_dict(self.policy_net.state_dict())

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=1e
            -3)
        self.memory = ReplayBuffer()
        self.gamma = 0.99
        self.epsilon = 1.0
        self.epsilon_decay = 0.995
        self.epsilon_min = 0.01

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, self.policy_net.fc[-1].out_features
                - 1)

        with torch.no_grad():
            state_tensor = torch.FloatTensor(state).unsqueeze(0)
            q_values = self.policy_net(state_tensor)
            return q_values.argmax().item()

    def train(self, batch_size=64):
        if len(self.memory) < batch_size:
            return

        batch = self.memory.sample(batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        states = torch.FloatTensor(states)
        actions = torch.LongTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)
        dones = torch.FloatTensor(dones)

        # Q(s, a)
        q_values = self.policy_net(states).gather(1, actions.unsqueeze
            (1))

        # Target: r + gamma * max Q(s', a')
        with torch.no_grad():
```

```
            next_q_values = self.target_net(next_states).max(1)[0]
            targets = rewards + self.gamma * next_q_values * (1 - dones)

        # Loss et backprop
        loss = nn.MSELoss()(q_values.squeeze(), targets)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Decay epsilon
        self.epsilon = max(self.epsilon_min, self.epsilon * self.
            epsilon_decay)

    def update_target_network(self):
        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 :

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

4.2 REINFORCE

Algorithm 3 REINFORCE

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

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

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 :

— $09_demo_qlearning.ipynb : Implmentation de Q-Learning sur Frozen Lake$

— Algorithme Q-Learning from scratch
— Environnement OpenAI Gym FrozenLake
— Construction et visualisation de la Q-table
— Analyse de la convergence

$09_demo_dqn.ipynb : Deep Q-Network (DQN) sur Cart Pole$

`Architecture DQN avec PyTorch`

`Experience Replay et Target Network`

`Entraînement et évaluation`

`Visualisation des performances`