Reinforcement Learning

Cours 3 : Policy Iteration and Value Iteration

Pour trouver une politique optimale, il existe deux grandes familles d'algorithmes : la programmation dynamique (résoudre le problème en le décomposant récursivement en plus petits problèmes) et les simulations de Monte-Carlo (faire des expériences pour estimer les distributions de probabilités).

Dans ce TP, nous étudions deux types d'algorithme utilisant la programmation dynamique : les itérations sur les valeurs et les itérations sur la politique.

RAPPEL : 1/4 de la note finale est liée à la mise en forme :

- pensez à nettoyer les outputs inutiles (installation, messages de débuggage, ...)
- soignez vos figures : les axes sont-ils faciles à comprendre ? L'échelle est adaptée ?
- commentez vos résultats : vous attendiez-vous à les avoir ? Est-ce étonnant ? Faites le lien avec la théorie.

Ce TP reprend l'exemple d'un médecin et de ses vaccins. Vous allez comparer plusieurs stratégies et trouver celle optimale. Un TP se fait en groupe de 2 à 4. Aucun groupe de plus de 4 personnes.

Vous allez rendre le TP dans une archive ZIP. L'archive ZIP contient ce notebook au format `ipynb`, mais aussi exporté en PDF & HTML. L'archive ZIP doit aussi contenir un fichier txt appelé `groupe.txt` sous le format:

```
Nom1, Prenom1, Email1, NumEtudiant1
Nom2, Prenom2, Email2, NumEtudiant2
Nom3, Prenom3, Email3, NumEtudiant3
Nom4, Prenom4, Email4, NumEtudiant4
```

Un script vient extraire vos réponses : ne changez pas l'ordre des cellules et soyez sûrs que les graphes sont bien présents dans la version notebook soumise.

In [149…

```python
import matplotlib.pyplot as plt
import torch
import networkx as nx
```

I. Estimation de la fonction de valeur d'un gridword

Nous avons vu en cours que :

$$v_\pi(s) = \mathbb{E}_\pi\left(G_t|s\right) = \sum_{s'} p(s'|s,a)\left[r + \gamma v_\pi(s')\right]$$

Dans le cas où les dynamiques de l'environnement sont entièrement connus, $p(s'|s,a)$ peut s'exprimer sous la forme d'un tensor et l'équation précédente aboutit à un système d'équations linéaires. Le problème est donc résolvable, mais la résolution risque d'être longue si l'environnement est grand.

On cherche plutôt une résolution itérative qui applique le principe de la programmation dynamique. Concrètement, on part d'une fonction de valeur arbitraire v_0 (par exemple nulle partout), puis on y applique à chaque étape l'équation de Bellman :

$$v_{k+1}(s) = \sum_{s'} p(s'|s,a)\left[r + \gamma v_k(s')\right]$$

Lorsque l'algorithme a convergé vers un point fixe v_∞, nous avons fini d'évaluer v_π, puisque ce dernier est l'unique point fixe de la fonction de valeur.

Cet algorithme est appelé l'**évaluation itérative de la politique**.

On considère par la suite le "gridworld" suivant :

gridworld

Les cases grisées sont terminales et la récompense est de -1 sur toutes les transitions. La taille du gridworld est une constante `CUBE_SIDE`.

Q1: évaluez la fonction de valeur de la politique aléatoire à l'aide d'un algorithme itératif. Arrêtez l'algorithme lorsque les valeurs n'ont pas évolué de plus de 1e-2.

In [150…

```python
from tabulate import tabulate
import typing as t
from dataclasses import dataclass, field
import random
import torch
import matplotlib.pyplot as plt

Action = t.Literal["L", "R", "U" , "D"]
CUBE_SIDE = 6

@dataclass
class State:
    """
    It represents any cell in the world
    """
    cell: int
    value: int = 0

    def __post_init__(self):
        self.bounds = {
            'L': self.cell - self.cell % CUBE_SIDE,
            'R': self.cell - self.cell % CUBE_SIDE + (CUBE_SIDE - 1),
            'U': self.cell % CUBE_SIDE,
            'D': self.cell % CUBE_SIDE + CUBE_SIDE * (CUBE_SIDE - 1),
        }
        self.neighbors = [self.act(a) for a in "LRUD"]
        assert all(i >= 0 and i < CUBE_SIDE*CUBE_SIDE for i in self.neighbors)

    def is_termination(self):
        return self.cell in {0, CUBE_SIDE * CUBE_SIDE - 1}

    def act(self, a: Action):
        """
        Get next state
        """
        if a == 'L':
            return min(self.bounds['R'], max(self.bounds['L'], self.cell - 1))
        if a == 'R':
            return min(self.bounds['R'], max(self.bounds['L'], self.cell + 1))
        if a == 'U':
            return min(self.bounds['D'], max(self.bounds['U'], self.cell - 4))
        if a == 'D':
            return min(self.bounds['D'], max(self.bounds['U'], self.cell + 4))
        raise ValueError('Unexpected action')


def init_states():
    return [State(i) for i in range(CUBE_SIDE * CUBE_SIDE)]


@dataclass
class Env:
    states: t.List[State] = field(default_factory=init_states)

def policy_evaluation(env, pi, gamma=0.9, theta=1e-2, r=-1):
    """
    Policy evaluation function
    """
    inter = 1
    while True:
        delta = 0
        env_copy = Env(states=env.states.copy())

        for s, s_copy in zip(env.states, env_copy.states):
            if s.is_termination():
                continue
            v = s.value
            s.value = 0

            for i, index_state in enumerate(s_copy.neighbors):
                s.value += pi[s.cell][i] * (r + gamma * env_copy.states[index_state].value)

            delta = max(delta, abs(v - s.value))

        # Check if we stop iterating
        if delta < theta:
                break
        inter += 1
    print("Number of iterations: ", inter)
    return env

def print_states(states):
    for i in range(CUBE_SIDE):
        print([states[i*CUBE_SIDE + j].value for j in range(CUBE_SIDE)])
env = Env()
print("Initial state values")
print_states(env.states)

# The probability of taking the action is 0.25 because we have 4 actions
pol_eval = policy_evaluation(env, [[0.25, 0.25, 0.25, 0.25] for _ in range(CUBE_SIDE * CUBE_SIDE)])

print("\nState values after policy evaluation")
print_states(pol_eval.states)
```

```
Initial state values
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
Number of iterations:  17

State values after policy evaluation
[0, -3.2879383610186155, -4.44555114854303, -5.00589407010135, -4.994933400824104, -4.233009887479573]
[-4.666149237691077, -5.752817907690666, -5.9498007272112545, -6.0743710383323, -6.109557938542456, -5.51475980
2492964]
[-5.190389376976314, -6.274827946280725, -6.467204718420831, -6.585900914917751, -6.413431263750833, -5.7246285
531996035]
[-5.341258015743366, -6.444051839317244, -6.570493937301101, -6.622970332441267, -6.394948276774256, -5.7086114
97306631]
[-5.287249963492066, -6.310811811652834, -6.313233609208383, -6.403270833598547, -6.181134103802791, -5.3870449
57210603]
[-4.511604206204862, -5.817761496164923, -5.954749934720489, -5.367132997082065, -3.891576941655959, 0]
```

La politique gloutonne cherche uniquement à exploiter, sans aucune exploration. A chaque instant, elle choisit l'action qui permet de maximiser la fonction de valeur :

$$\pi(s) = \text{argmax}_a \sum_{s'} p(s'|s,a)[r + \gamma V(s')]$$

Q2: calculez la politique ainsi obtenue. Vérifiez qu'il s'agit de la politique optimale. Combien d'itérations ont été nécessaires pour obtenir ce résultat ?

In [151…

```python
def compute_policy(env):
    pi = [ [0, 0, 0, 0] for _ in range(CUBE_SIDE * CUBE_SIDE)]
    for s in env.states:
        if s.is_termination():
            continue
        for i, index_state in enumerate(s.neighbors):
            pi[s.cell][i] = env.states[index_state].value

        max_value = max(pi[s.cell])

        # Count number of max values to divide the probability
        count = 0
        for i in range(len(pi[s.cell])):
            if pi[s.cell][i] == max_value:
                count += 1

        for i, value in enumerate(pi[s.cell]):
            if value != max_value:
                pi[s.cell][i] = 0
            else:
                pi[s.cell][i] = 1 / count
    return pi

policy = compute_policy(pol_eval)


print("\nPolicy:")
for i in range(CUBE_SIDE):
    print([policy[i*CUBE_SIDE + j] for j in range(CUBE_SIDE)])
```

```
Policy:
[[0, 0, 0, 0], [1.0, 0, 0, 0], [1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 1.0, 0, 0], [0, 0.5, 0.5, 0]]
[[0, 0, 1.0, 0], [1.0, 0, 0, 0], [0, 0, 1.0, 0], [0, 0, 1.0, 0], [0, 0, 1.0, 0], [0, 1.0, 0, 0]]
[[1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 0, 0, 1.0], [0, 0, 1.0, 0], [0, 0, 1.0, 0], [0, 1.0, 0, 0]]
[[1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 0, 0, 1.0], [0, 0, 1.0, 0], [0, 0, 1.0, 0], [0, 1.0, 0, 0]]
[[1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 0, 0, 1.0], [0, 0, 1.0, 0], [0, 0, 1.0, 0], [0, 0, 0, 1.0]]
[[0.5, 0, 0, 0.5], [1.0, 0, 0, 0], [0, 1.0, 0, 0], [0, 1.0, 0, 0], [0, 1.0, 0, 0], [0, 0, 0, 0]]
```

Si on suit les probabilités trouvées on va bien se diriger de façon optimale vers la destination. Le nombre d'itérations effectué est 17.

II. Algorithme *policy iteration*

Une amélioration de l'algorithme consiste 1) à évaluer la fonction de valeur sur un petit nombre d'itérations (on testera en Q3 avec une seule itération), puis 2) à mettre à jour la politique, puis à recommencer l'étape 1). On peut arrêter l'entraînement lorsque la politique a convergé.

Q3: implémentez cet algorithme. Est-il plus rapide ?

In [152…

```python
def improved_policy_evaluation(env, pi, iter_max, gamma=0.9, theta=1e-2, r=-1, iter=1):
    """
    Improved policy evaluation function
    """
    delta = 0
    env_copy = Env(states=env.states.copy())

    for s, s_copy in zip(env.states, env_copy.states):
        if s.is_termination():
            continue
        v = s.value
        s.value = 0

        for i, index_state in enumerate(s_copy.neighbors):
            s.value += pi[s.cell][i] * (r + gamma * env_copy.states[index_state].value)

        delta = max(delta, abs(v - s.value))

    # Check if we stop iterating
    if delta < theta or iter == iter_max:
        print("Number of iterations: ", iter, "\n")
        return env

    pi_new = compute_policy(env)
    return improved_policy_evaluation(env, pi_new, iter_max, gamma, theta, r, iter = iter + 1)

env = Env()
print("Initial state values")
print_states(env.states)

pol_eval = improved_policy_evaluation(env, [[0.25, 0.25, 0.25, 0.25] for _ in range(CUBE_SIDE * CUBE_SIDE)], 10

print("\nState values after policy evaluation")
print_states(pol_eval.states)

policy = compute_policy(pol_eval)
print("\nPolicy:")
for i in range(CUBE_SIDE):
    print([policy[i*CUBE_SIDE + j] for j in range(CUBE_SIDE)])
```

```
Initial state values
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
Number of iterations:  5


State values after policy evaluation
[0, -1.0, -1.9, -2.71, -2.71, -3.439]
[-1.0, -1.9, -1.9, -2.71, -1.9, -2.71]
[-1.0, -1.9, -1.9, -2.71, -1.9, -2.71]
[-1.0, -1.9, -1.9, -2.71, -1.9, -2.71]
[-1.0, -1.9, -2.1025, -2.71, -1.9, -2.71]
[-1.225, -2.1025, -2.71, -1.9, -1.0, 0]

Policy:
[[0, 0, 0, 0], [1.0, 0, 0, 0], [0.5, 0, 0, 0.5], [0.5, 0, 0, 0.5], [0, 0, 0, 1.0], [0.5, 0, 0, 0.5]]
[[1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 0, 0, 1.0], [0.3333333333333333, 0.3333333333333333, 0, 0.333333333333333
3], [0, 0, 1.0, 0], [0.5, 0, 0.5, 0]]
[[1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 0, 0, 1.0], [0.3333333333333333, 0.3333333333333333, 0, 0.333333333333333
3], [0, 0, 1.0, 0], [0.5, 0, 0.5, 0]]
[[1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 0, 0, 1.0], [0.3333333333333333, 0.3333333333333333, 0, 0.333333333333333
3], [0, 0, 1.0, 0], [0.5, 0, 0.5, 0]]
[[1.0, 0, 0, 0], [1.0, 0, 0, 0], [0, 0, 0, 1.0], [0, 1.0, 0, 0], [0, 0, 1.0, 0], [0.3333333333333333, 0, 0.3333
333333333333, 0.3333333333333333]]
[[0.5, 0, 0, 0.5], [1.0, 0, 0, 0], [0, 0.5, 0.5, 0], [0, 1.0, 0, 0], [0, 1.0, 0, 0], [0, 0, 0, 0]]
```

On peut le voir grâce au nombre d'itération. Le premier algo est à 17 itérations alors qu'ici on est à 5 itérations.

III. Algorithme *value iteration*

Une autre variante conserve la politique aléatoire tout en long de l'entraînement, mais met à jour la fonction de valeur avec l'équation suivante :

$$v_{k+1}(s) = \max_a \sum_{s'} p(s'|s,a)\left[r + \gamma v_k(s')\right]$$

Une fois que la fonction de valeur a convergé, on calcule la politique avec :

$$\pi(s) = argmax_a \sum_{s'} p(s'|s,a)[r + \gamma V(s')]$$

Q4: implémentez cet algorithme.

In [160…

```python
def greedy_policy_evaluation(env, pi, gamma=0.9, theta=1e-2, r=-1, iter=1):
    """"
    Greddy policy evaluation function
    """
    delta = 0
    env_copy = Env(states=env.states.copy())

    for s, s_copy in zip(env.states, env_copy.states):
        if s.is_termination():
            continue
        v = s.value
        s.value = -float('inf')

        for i, index_state in enumerate(s_copy.neighbors):
            s.value = max(s.value, pi[s.cell][i] * (r + gamma * env_copy.states[index_state].value))

        delta = max(delta, abs(v - s.value))

    # Check if we stop iterating
    if delta < theta:
        print("Number of iterations: ", iter, "\n")
        return env

    return greedy_policy_evaluation(env, pi, gamma, theta, r, iter = iter + 1)

env = Env()
print("Initial state values")
print_states(env.states)

pol_eval = greedy_policy_evaluation(env, [[0.25, 0.25, 0.25, 0.25] for _ in range(CUBE_SIDE * CUBE_SIDE)])

print("\nState values after policy evaluation")
print_states(pol_eval.states)
```

```
Initial state values
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
Number of iterations:  4


State values after policy evaluation
[0, -0.25, -0.30625, -0.31890625, -0.32175390625, -0.32175390625]
[-0.31890625, -0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625]
[-0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625]
[-0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625]
[-0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625, -0.32175390625, -0.31890625]
[-0.32175390625, -0.32175390625, -0.31890625, -0.30625, -0.25, 0]
```

On peut voir avec le nombre d'itération que cette dernière implémentation semble être la plus rapide. Cependant, la politique reste aléatoire.

Q5: Quel algorithme vous paraît le plus judicieux ?

L'algorithme `Policy Iteration` parait être le meilleur car il est optimisé et les résultats sont cohérents. En effet on est sûr de s'arrêter au bon moment et d'avoir des résultats plus qu'acceptable dans la prise de décision.