



école supérieure de  
génie informatique

# RAPPORT DEEP REINFORCEMENT LEARNING 4IABD2

TOUZART Antoine Max Vincent  
BOLOORIAN Alan  
HAMMACHI Selim  
4 IABD 2

25 JUILLET 2024

# 2

## SOMMAIRE

### 1. Algorithmes Implémentés

- Policy Iteration
- Value Iteration
- Monte Carlo ES
- Monte Carlo On Policy
- Monte Carlo Off Policy
- Sarsa
- Q-Learning
- Dyna-Q

### 2. Environnements Testés

- LineWorld
- GridWorld
- SecretEnv0
- SecretEnv1
- SecretEnv2
- SecretEnv3

### 3. Notebooks

# 3

## POLICY ITERATION

La Policy Iteration est un algorithme d'apprentissage par renforcement utilisé pour trouver la politique optimale dans un environnement donné. Il alterne entre deux étapes : l'évaluation de la politique actuelle et l'amélioration de cette politique. L'algorithme continue jusqu'à ce que la politique soit stable, c'est-à-dire qu'il n'y a plus de changements dans les actions recommandées par la politique.

Paramètres :

- `env` : L'environnement où l'agent évolue, définissant les états, actions, récompenses, et probabilités de transition.
- `gamma` : Le facteur de réduction pour les récompenses futures (entre 0 et 1), influençant l'importance des récompenses à long terme.
- `theta` : Le seuil de convergence pour l'évaluation de la politique, déterminant la précision des valeurs de fonction d'état. L'algorithme s'arrête lorsque la variation maximale entre les itérations est inférieure à ce seuil.
- `max_iter` : Le nombre maximum d'itérations pour les étapes d'évaluation de la politique et d'amélioration de la politique, pour éviter les boucles infinies.

Fonctionnement :

1. Évaluation de la Politique : Calcule la fonction de valeur des états sous la politique actuelle. Cela se fait en itérant jusqu'à ce que les valeurs de fonction d'état convergent (c'est-à-dire que les changements sont inférieurs à `theta`).
2. Amélioration de la Politique : Met à jour la politique en choisissant l'action qui maximise la valeur attendue pour chaque état, en utilisant la fonction de valeur calculée lors de l'évaluation.

# 4

## POLICY ITERATION

Méthodes :

- `policy_iteration` : Exécute les étapes d'évaluation et d'amélioration de la politique jusqu'à ce que la politique devienne stable.
- `policy_evaluation` : Évalue la politique actuelle en calculant les valeurs de fonction d'état jusqu'à convergence.
- `expected_value` : Calcule la valeur attendue d'un état donné sous l'action choisie par la politique actuelle.
- `greedy_policy` : Améliore la politique en choisissant l'action qui maximise la valeur attendue pour un état donné.
- `expected_value_for_action` : Calcule la valeur attendue d'un état pour une action spécifique.

En résumé, l'algorithme de Policy Iteration améliore de manière itérative la politique en alternant entre l'évaluation de la politique actuelle et l'amélioration basée sur les valeurs d'état calculées. Il continue jusqu'à ce que la politique n'évolue plus, assurant ainsi que la politique optimale a été trouvée.

# 5

## VALUE ITERATION

Le Value Iteration est un algorithme d'apprentissage par renforcement utilisé pour calculer la politique optimale en déterminant la fonction de valeur optimale des états. Contrairement à la méthode de Policy Iteration, qui alterne entre l'évaluation et l'amélioration de la politique, Value Iteration met à jour les valeurs de fonction d'état directement jusqu'à convergence, puis en extrait la politique optimale.

Paramètres :

- `env` : L'environnement dans lequel l'agent évolue, incluant les états, actions, récompenses, et probabilités de transition.
- `gamma` : Le facteur de réduction pour les récompenses futures (entre 0 et 1), influençant l'importance des récompenses à long terme par rapport aux récompenses immédiates.
- `theta` : Le seuil de convergence pour les valeurs de fonction d'état, déterminant la précision des valeurs d'état. L'algorithme s'arrête lorsque la variation maximale entre les itérations est inférieure à ce seuil.
- `max_iter` : Le nombre maximum d'itérations pour l'algorithme de mise à jour des valeurs de fonction d'état, pour éviter les boucles infinies.

Fonctionnement :

1. Mise à Jour des Valeurs de Fonction d'État : Met à jour les valeurs des états en utilisant l'équation de Bellman pour toutes les actions possibles dans chaque état, en se basant sur les valeurs actuelles des états suivants. Cela se fait jusqu'à ce que la variation entre les valeurs de fonction d'état dans les itérations consécutives soit inférieure à `theta` ou que le nombre maximum d'itérations soit atteint.
2. Extraction de la Politique : Une fois que les valeurs de fonction d'état ont convergé, la politique optimale est extraite en choisissant, pour chaque état, l'action qui maximise la valeur attendue selon la fonction de valeur optimale.

# 6

## VALUE ITERATION

Méthodes :

- `value_iteration` : Effectue les mises à jour de la fonction de valeur des états et extrait la politique optimale. Arrête l'itération lorsque les valeurs de fonction d'état convergent.
- `compute_value` : Calcule la valeur maximale d'un état en évaluant toutes les actions possibles et en sélectionnant celle avec la valeur la plus élevée.
- `expected_value_for_action` : Calcule la valeur attendue d'un état pour une action spécifique en utilisant les probabilités de transition et les récompenses.
- `extract_policy` : Extrait la politique optimale en choisissant, pour chaque état, l'action avec la valeur attendue maximale basée sur les valeurs de fonction d'état optimales.
- `get_available_actions` : Récupère les actions disponibles pour un état donné (ou pour tous les états si aucune spécification n'est faite).

En Résumé :

Le Value Iteration calcule la politique optimale en mettant à jour les valeurs des états jusqu'à convergence et en extrayant la politique qui maximise ces valeurs. C'est un algorithme efficace pour trouver la politique optimale dans un environnement donné en utilisant une approche directe basée sur la mise à jour des valeurs d'état.

## 7

## MONTE CARLO ES

Le MonteCarloES est un algorithme d'apprentissage par renforcement basé sur la méthode de Monte Carlo pour améliorer les politiques de décision. Il utilise les épisodes de simulation pour estimer les valeurs des états et actions, puis ajuste la politique en fonction de ces estimations.

Paramètres :

- env : L'environnement dans lequel l'agent évolue, définissant les états, actions, récompenses, etc.
- gamma : Le facteur de réduction pour les récompenses futures (entre 0 et 1), influençant la priorité des récompenses à long terme par rapport aux récompenses immédiates.
- episodes : Le nombre total d'épisodes pour entraîner l'agent, c'est-à-dire combien de fois l'algorithme va générer et évaluer des épisodes pour apprendre.

En résumé, le MonteCarloES améliore la politique en estimant la valeur des actions basées sur des épisodes simulés et en ajustant les politiques pour maximiser la récompense attendue.

# 8

## MONTE CARLO ON POLICY

Le MonteCarloOnPolicy est un algorithme d'apprentissage par renforcement qui apprend une politique tout en l'exploitant pour générer des épisodes. Contrairement aux méthodes off-policy, cette approche ajuste la politique actuelle en se basant sur les épisodes générés avec cette même politique, ce qui permet de rendre l'apprentissage plus cohérent avec la politique adoptée.

Paramètres :

- env : L'environnement où l'agent évolue, définissant les états, actions, récompenses, etc.
- gamma : Le facteur de réduction pour les récompenses futures (entre 0 et 1), qui détermine l'importance des récompenses à long terme par rapport aux récompenses immédiates.
- epsilon : Le paramètre d'exploration pour la politique, permettant à l'agent d'explorer des actions non optimales avec une probabilité d'epsilon. Une valeur de 0.1 signifie que l'agent explore 10% du temps.
- episodes : Le nombre total d'épisodes pour entraîner l'agent, c'est-à-dire combien de fois l'algorithme génère et évalue des épisodes pour apprendre.

En résumé, le MonteCarloOnPolicy apprend et améliore une politique en utilisant directement les épisodes générés par cette politique, en équilibrant exploration et exploitation pour maximiser la récompense.



## 9

## MONTE CARLO OFF POLICY

Le MonteCarloOffPolicy est un algorithme d'apprentissage par renforcement qui apprend une politique cible tout en utilisant une politique comportementale différente pour générer les épisodes. Cela permet d'explorer de manière plus flexible en utilisant des politiques de comportement qui peuvent être plus variées que la politique cible.

Paramètres :

- env : L'environnement dans lequel l'agent évolue, définissant les états, actions, récompenses, etc.
- gamma : Le facteur de réduction pour les récompenses futures (entre 0 et 1), qui évalue l'importance des récompenses à long terme par rapport aux récompenses immédiates.
- epsilon : Le paramètre d'exploration pour la politique comportementale, permettant à l'agent d'explorer des actions non optimales avec une probabilité d'epsilon. Une valeur de 0.1 signifie que l'agent explore 10% du temps.
- episodes : Le nombre total d'épisodes pour entraîner l'agent, c'est-à-dire combien de fois l'algorithme génère et évalue des épisodes pour apprendre.

En résumé, le MonteCarloOffPolicy utilise une politique comportementale pour générer des épisodes tout en apprenant une politique cible. Il ajuste la politique cible en fonction des récompenses obtenues et des poids d'importance, ce qui permet d'explorer de manière plus variée tout en cherchant à optimiser la politique cible.

## 10

# SARSA

Sarsa (State-Action-Reward-State-Action) est un algorithme de renforcement qui utilise une approche on-policy pour apprendre la valeur des politiques dans un environnement. Contrairement à Q-learning, qui utilise la meilleure action possible pour estimer la valeur des actions, Sarsa utilise la action effectivement choisie pour mettre à jour les valeurs d'état-action.

Paramètres :

- length : La taille de l'environnement, définissant les états possibles.
- alpha : Le taux d'apprentissage, déterminant la vitesse à laquelle l'agent met à jour ses valeurs d'état-action.
- gamma : Le facteur de réduction pour les récompenses futures, influençant l'importance des récompenses à long terme.
- epsilon : Le taux d'exploration, contrôlant la probabilité que l'agent choisisse une action aléatoire plutôt que la meilleure action connue.

Fonctionnement :

- 1.Choix de l'Action : L'agent choisit une action pour un état donné en utilisant une stratégie  $\epsilon$ -greedy, qui explore aléatoirement avec une probabilité  $\epsilon$  et choisit l'action avec la valeur  $Q$  la plus élevée avec une probabilité  $1 - \epsilon$ .
- 2.Mise à Jour des Valeurs  $Q$  : Les valeurs  $Q$  sont mises à jour en utilisant l'équation de mise à jour de Sarsa, qui prend en compte la récompense reçue et la valeur de la prochaine action choisie (non optimale, comme dans Q-learning).
- 3.Entraînement : L'agent effectue un certain nombre d'épisodes d'entraînement où il interagit avec l'environnement, choisissant des actions, recevant des récompenses, et mettant à jour les valeurs  $Q$  en conséquence.

# 11

## SARSA

Méthodes :

- `get_next_state(state, action)` : Détermine le prochain état en fonction de l'état actuel et de l'action choisie.
- `get_reward(state)` : Retourne la récompense associée à un état donné. Dans cet environnement, la récompense est 1 pour l'état -2 et 0 pour les autres états.
- `is_terminal(state)` : Vérifie si l'état est terminal (dans ce cas, l'état -2 est terminal).
- `choose_action(state)` : Choisit une action à partir de l'état actuel en utilisant une politique  $\epsilon$ -greedy.
- `update_q_value(state, action, reward, next_state, next_action)` : Met à jour la valeur  $Q$  pour l'état-action actuel en utilisant la récompense reçue et la valeur  $Q$  pour l'état-action suivant.
- `train(episodes)` : Entraîne l'agent en effectuant un nombre donné d'épisodes, mettant à jour les valeurs  $Q$  à chaque étape.
- `get_best_action(state)` : Obtient la meilleure action pour un état donné, basée sur les valeurs  $Q$  apprises.
- `print_q_table()` : Affiche la table  $Q$  pour tous les états.
- `print_optimal_policy()` : Affiche la politique optimale en termes d'actions pour chaque état.

En Résumé :

L'algorithme de Sarsa apprend une politique en mettant à jour les valeurs  $Q$  basées sur les actions réellement choisies pendant l'apprentissage, en utilisant une approche  $\epsilon$ -greedy pour équilibrer l'exploration et l'exploitation. Il est utile pour les environnements où il est important d'apprendre des politiques basées sur les actions effectivement prises, et peut être utilisé pour des tâches d'apprentissage par renforcement où une approche on-policy est souhaitée.

## 12

## Q-LEARNING

Q-Learning est un algorithme de renforcement off-policy qui vise à apprendre la valeur des actions dans un environnement, afin de déterminer la meilleure politique possible pour maximiser la récompense cumulée. Il est largement utilisé pour les problèmes de contrôle dans les environnements discrets.

Paramètres :

- rows et cols : Dimensions de la grille de l'environnement.
- start\_state : État de départ de l'agent.
- goal\_state : État objectif que l'agent doit atteindre.
- epsilon : Taux d'exploration pour choisir des actions aléatoires.
- alpha : Taux d'apprentissage, déterminant combien les nouvelles informations sont incorporées.
- gamma : Facteur de réduction pour les récompenses futures.
- q\_table : Table Q pour stocker les valeurs d'état-action.

Fonctionnement :

1. Choix de l'Action : L'agent choisit une action en utilisant une politique  $\epsilon$ -greedy. Avec une probabilité  $\epsilon$ , il choisit une action aléatoire (exploration), et avec une probabilité  $1-\epsilon$ , il choisit l'action avec la valeur Q maximale (exploitation).
2. Mise à Jour des Valeurs Q : Après avoir pris une action et observé la récompense et le prochain état, l'agent met à jour la valeur Q pour l'état-action actuel en utilisant l'équation de mise à jour de Q-Learning :

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

où  $s's'$  est le prochain état et  $a'a'$  est l'action dans ce prochain état.

3. Entraînement : L'agent effectue un certain nombre d'épisodes, interagissant avec l'environnement, en mettant à jour les valeurs Q à chaque étape, et en suivant la politique  $\epsilon$ -greedy pour choisir les actions.

# 13

## Q-LEARNING

Méthodes :

- `get_next_state(state, action)` : Détermine le prochain état basé sur l'état actuel et l'action choisie, en respectant les limites de la grille.
- `is_terminal(state)` : Vérifie si l'état actuel est l'état objectif (terminal).
- `is_penalty(state)` : Vérifie si l'état actuel est un état de pénalité.
- `getReward(state)` : Retourne la récompense pour l'état donné. Les récompenses sont 10 pour l'état objectif, -1 pour l'état de pénalité, et 0 pour les autres états.
- `choose_action(state)` : Choisit une action en utilisant une politique  $\epsilon$ -greedy.
- `update_q_value(state, action, reward, next_state)` : Met à jour la valeur  $Q$  pour l'état-action actuel en utilisant la récompense reçue et la valeur  $Q$  du prochain état.
- `get_best_action(state)` : Obtient la meilleure action pour un état donné, basée sur les valeurs  $Q$  apprises.
- `train(episodes)` : Entraîne l'agent en effectuant un certain nombre d'épisodes, mettant à jour les valeurs  $Q$  à chaque étape.
- `print_optimal_policy()` : Affiche la politique optimale pour chaque état dans la grille.
- `print_q_table()` : Affiche la table  $Q$  pour tous les états dans la grille.

En Résumé :

L'algorithme de Q-Learning apprend la valeur des actions en utilisant une approche off-policy. Cela signifie qu'il évalue les actions selon la politique optimale même si les actions choisies pendant l'apprentissage peuvent être aléatoires. Cela permet à l'agent de découvrir la meilleure politique possible en se basant sur les expériences accumulées dans l'environnement.

# 14

## DYNA-Q

Le DynaQAgent est un algorithme d'apprentissage par renforcement qui combine l'apprentissage par essais et erreurs avec une planification interne. Il utilise un modèle interne pour simuler des expériences supplémentaires, ce qui accélère l'apprentissage en intégrant à la fois des interactions réelles avec l'environnement et des mises à jour basées sur des simulations.

Paramètres :

- **environment** : L'environnement où l'agent évolue, incluant les états, actions, et récompenses.
- **gamma** : Le facteur de réduction pour les récompenses futures (entre 0 et 1), influençant l'importance des récompenses à long terme.
- **alpha** : Le taux d'apprentissage, déterminant la rapidité avec laquelle les valeurs  $Q$  sont mises à jour en fonction des nouvelles informations (entre 0 et 1).
- **epsilon** : Le paramètre d'exploration pour la sélection d'actions, permettant à l'agent de choisir des actions aléatoires avec une probabilité d'epsilon pour explorer de nouvelles stratégies.
- **n** : Le nombre de mises à jour de planification effectuées à chaque étape, déterminant la fréquence des mises à jour simulées de la fonction de valeur.

Fonctionnement :

- **Apprentissage en ligne** : L'agent apprend en interagissant directement avec l'environnement et en mettant à jour ses valeurs  $Q$  basées sur les récompenses reçues.
- **Modélisation** : L'agent maintient un modèle interne de l'environnement qui prédit les récompenses et les états suivants pour chaque état-action.
- **Planification** : En plus de l'apprentissage direct, l'agent utilise le modèle pour effectuer des mises à jour supplémentaires sur les valeurs  $Q$  à l'aide de simulations basées sur les transitions enregistrées dans le modèle.

En résumé, le DynaQAgent améliore l'apprentissage en combinant l'expérience réelle avec des simulations internes pour accélérer la convergence vers une politique optimale.

## 15

## LINE WORLD

-Sarsa, le goal state se trouve tout à gauche et l'algorithme montre bien que le meilleur path pour atteindre le goal state:

```

Desktop/Workspace/ESGI/M1/S2/ 1 from QLearning import QLearning
                                2 from SarsaLearning import SarsaLearning
                                3
                                4 #q = QLearning(3, 3)
                                5 q = SarsaLearning(4)
                                6 q.train(1000)
                                7 q.print_q_table()
                                8 q.print_optimal_policy()

by
ssoles

bolorian_1/Desktop/Workspace/ESGI/M1/S2/DeepR/DeepRProject/.venv/bin/python /Users/Alanbolorian_1/Desktop/Workspace/ESGI/M1/S2/DeepR/DeepRP
': 0.0, 'RIGHT': 0.0}, -1: {'LEFT': 0.0, 'RIGHT': 0.0}, 0: {'LEFT': 0.0, 'RIGHT': 0.0}, 1: {'LEFT': 0.0, 'RIGHT': 0.0}, 2: {'LEFT': 0.0, 'RIGH
'LEFT': 0.0, 'RIGHT': 0.0}
'LEFT': 0.9999999999999996, 'RIGHT': 0.7622367306942268}
LEFT': 0.8803564383387369, 'RIGHT': 0.6830646176856154}
LEFT': 0.7954672482997713, 'RIGHT': 0.038360034547884655}
oal
est action = LEFT
st action = LEFT
st action = LEFT

```

## 16

## LINE WORLD

Policy Iteration sur environnement personnel

```
Desktop/Workspace/ESGI/M1/S2/ 1 from QLearning import QLearning
                                2 from SarsaLearning import SarsaLearning
                                3
                                4 #q = QLearning(3, 3)
                                5 q = SarsaLearning(4)
                                6 q.train(1000)
                                7 q.print_q_table()
                                8 q.print_optimal_policy()
```

by  
nsols

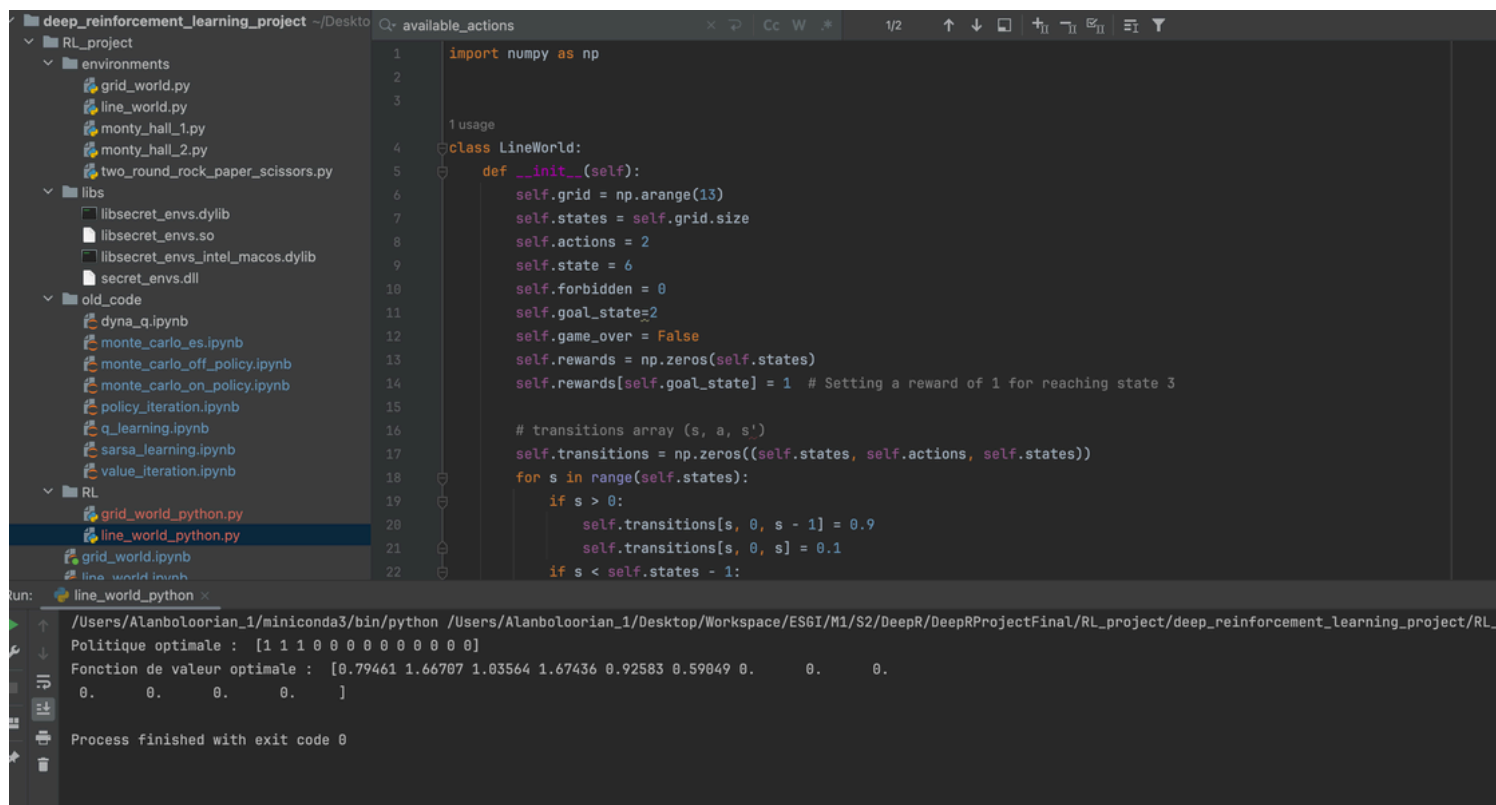
```
Alanbloorian_1/Desktop/Workspace/ESGI/M1/S2/DeepR/DeepRProject/.venv/bin/python /Users/Alanbloorian_1/Desktop/Workspace/ESGI/M1/S2/DeepR/DeepRPr
': 0.0, 'RIGHT': 0.0}, -1: {'LEFT': 0.0, 'RIGHT': 0.0}, 0: {'LEFT': 0.0, 'RIGHT': 0.0}, 1: {'LEFT': 0.0, 'RIGHT': 0.0}, 2: {'LEFT': 0.0, 'RIGH
'LEFT': 0.0, 'RIGHT': 0.0}
'LEFT': 0.9999999999999996, 'RIGHT': 0.7622367306942268}
LEFT': 0.8803564383387369, 'RIGHT': 0.6830646176856154}
LEFT': 0.7954672482997713, 'RIGHT': 0.038360034547884655}
oal
est action = LEFT
st action = LEFT
st action = LEFT
```



## 17

## LINE WORLD

Policy Iteration sur environnement demandé



The screenshot displays a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure for 'deep\_reinforcement\_learning\_project' with subfolders 'environments', 'libs', 'old\_code', and 'RL'. The 'environments' folder contains 'grid\_world.py', 'line\_world.py', 'monty\_hall\_1.py', 'monty\_hall\_2.py', and 'two\_round\_rock\_paper\_scissors.py'. The 'RL' folder contains 'grid\_world\_python.py', 'line\_world\_python.py', 'grid\_world.ipynb', and 'line\_world.ipynb'. The code editor shows the implementation of the 'LineWorld' class in 'line\_world.py'. The class has an 'init' method that sets up the environment parameters and a 'transitions' array. The output window at the bottom shows the execution of 'line\_world\_python.py' and the results of the policy iteration algorithm.

```
1 import numpy as np
2
3
4 1 usage
5 class LineWorld:
6     def __init__(self):
7         self.grid = np.arange(13)
8         self.states = self.grid.size
9         self.actions = 2
10        self.state = 6
11        self.forbidden = 0
12        self.goal_state = 2
13        self.game_over = False
14        self.rewards = np.zeros(self.states)
15        self.rewards[self.goal_state] = 1 # Setting a reward of 1 for reaching state 3
16
17        # transitions array (s, a, s')
18        self.transitions = np.zeros((self.states, self.actions, self.states))
19        for s in range(self.states):
20            if s > 0:
21                self.transitions[s, 0, s - 1] = 0.9
22                self.transitions[s, 0, s] = 0.1
23            if s < self.states - 1:
```

run: line\_world\_python x

```
/Users/Alanbolloorian_1/miniconda3/bin/python /Users/Alanbolloorian_1/Desktop/Workspace/ESGI/M1/S2/DeepR/DeepRProjectFinal/RL_project/deep_reinforcement_learning_project/RL_
Politique optimale : [1 1 1 0 0 0 0 0 0 0 0 0 0]
Fonction de valeur optimale : [0.79461 1.66787 1.03564 1.67436 0.92583 0.59849 0. 0. 0.
0. 0. 0. 0. ]
Process finished with exit code 0
```

## 18

## GRID WORLD

- Policy Iteration, ici S est le départ, la valeur de reward la plus haute est le goal state. On peut voir que quand on change la position du départ ou la valeur des rewards pour chaque état il va s'adapter et essayer de trouver le meilleur chemin.

The screenshot shows a Jupyter Notebook interface with a file explorer on the left, a code editor in the center, and a console output at the bottom.

**File Explorer:** The left sidebar shows a directory structure with files like `cars.csv`, `main.py`, `MCdemo.py`, `mdp.py`, `offpolicy.py`, `pca.py`, `QTutorial.py`, `RLModelBased.py`, `RLModelFree.py`, `SarsaLearning.py`, `svd.py`, and `Tp0Demo.py`.

**Code Editor:** The main area shows Python code for setting up a Grid World environment. The code includes comments and function calls like `Parameters`, `Environment`, and `Agent`. The environment is a 3x3 grid with a start state 'S' at (0,1), a goal state 'G' at (2,2), and a reward of -5 at (1,1). The code sets `gamma = 1.0`.

```

10 # | S | 0 | 0 |
11 # +-----+
12 # | -5 | 0 | 0 |
13 # +-----+
14 # | 0 | 0 | 10 |
15 # +-----+
16 param = Parameters( size: (3, 3), goal_state: 8, rewards: {8: 10, 3: -5}, gamma: 1.0)
17 # Initialize the Environment
18 environment = Environment(param.size, param.goal_state, param.rewards)
19 # Initialize the Agent
20 agent = Agent(environment, param)
21

```

**Console Output:** The bottom section shows the execution results. It displays the state values as a 3x3 matrix, the policy path, and the best path from state 0 to the goal.

```

/Users/khosrov/Desktop/projects/python/playground/.venv/bin/python /Users/khosrov/Desktop/projects/python/playground/main.py
State Values:
[[ 7.  8.  9.]
 [ 8.  9. 10.]
 [ 9. 10.  0.]]

Policy:
right down down
down down down
right right 6

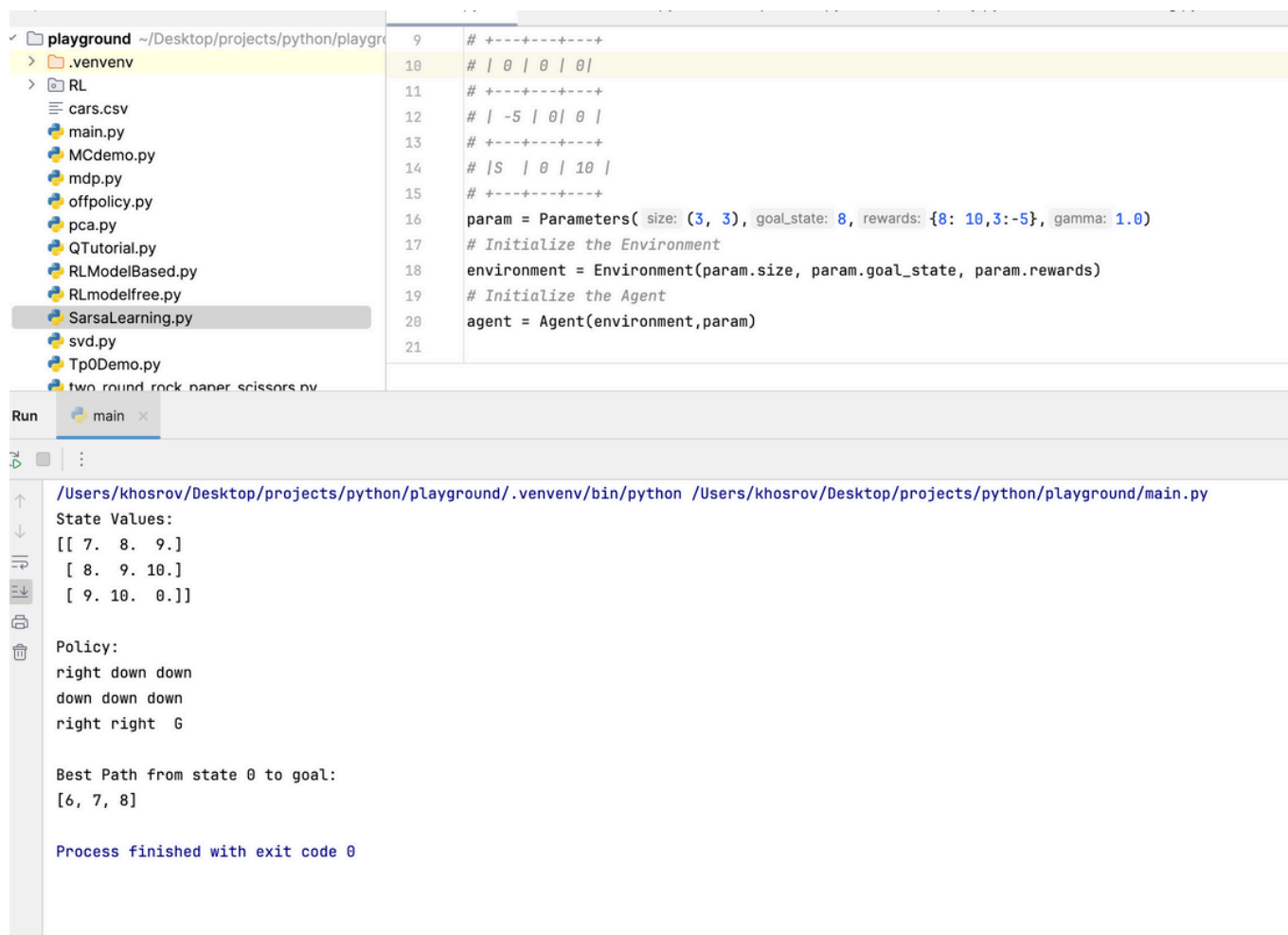
Best Path from state 0 to goal:
[0, 1, 4, 7, 8]

Process finished with exit code 0

```

## 19

## GRID WORLD



The screenshot displays a code editor with a file explorer on the left and a terminal at the bottom. The file explorer shows a project named 'playground' with a subdirectory '.venv' and a file 'SarsaLearning.py'. The code editor shows the following Python code:

```
9 # +-----+
10 # | 0 | 0 | 0 |
11 # +-----+
12 # | -5 | 0 | 0 |
13 # +-----+
14 # | S | 0 | 10 |
15 # +-----+
16 param = Parameters( size: (3, 3), goal_state: 8, rewards: {8: 10, 3: -5}, gamma: 1.0)
17 # Initialize the Environment
18 environment = Environment(param.size, param.goal_state, param.rewards)
19 # Initialize the Agent
20 agent = Agent(environment, param)
21
```

The terminal output shows the execution of the code:

```
Run main x
/Users/khosrov/Desktop/projects/python/playground/.venv/bin/python /Users/khosrov/Desktop/projects/python/playground/main.py
State Values:
[[ 7.  8.  9.]
 [ 8.  9. 10.]
 [ 9. 10.  0.]]
Policy:
right down down
down down down
right right G
Best Path from state 0 to goal:
[6, 7, 8]
Process finished with exit code 0
```

20

## GRID WORLD

```
✓ playground ~/Desktop/projects/python/playgr
> .venv
> RL
  cars.csv
  main.py
  MCdemo.py
  mdp.py
  offpolicy.py
  pca.py
  QTutorial.py
  RLModelBased.py
  RLModelFree.py
  SarsaLearning.py
  svd.py
  Tp0Demo.py
  two_round_rock_paper_scissors.py
  ValueIteration.py
> External Libraries
  Scratches and Consoles

9 # +-----+
10 # | 0 | 0 | 0 |
11 # +-----+
12 # | -5 | 0 | 0 |
13 # +-----+
14 # | S | -10 | 10 |
15 # +-----+
16 param = Parameters( size: (3, 3), goal_state: 8, rewards: {8: 10, 3: -5, 7: -10}, gamma: 1.0)
17 # Initialize the Environment
18 environment = Environment(param.size, param.goal_state, param.rewards)
19 # Initialize the Agent
20 agent = Agent(environment, param)
21
22 # Perform policy iteration
23 agent.policy_iteration()
24
25 # Print the state values and policy
26 print("State Values:")
7

Run main
/Users/khosrov/Desktop/projects/python/playground/.venv/bin/python /Users/khosrov/Desktop/projects/python/playground/main.py
State Values:
[[ 7.  8.  9.]
 [ 8.  9. 10.]
 [ 3. 10.  0.]]

Policy:
right down down
right right down
up right 6

Best Path from state 0 to goal:
[6, 3, 4, 5, 8]
```

21

GRID WORLD

QLearning, ici il faudrait créer la fonction qui permet de faire le tri sur les plus hautes valeurs pour garder le meilleur path

RL

cars.csv

main.py

MCdemo.py

mdp.py

offpolicy.py

pca.py

QTutorial.py

RLModelBased.py

RLmodelfree.py

SarsaLearning.py

svd.py

Tp0Demo.py

two\_round\_rock\_paper\_scissors.py

ValueIteration.py

External Libraries

Scratches and Consoles

3

# +---+-----+-----+

4

# | S | 0 | 0 |

5

# +---+-----+-----+

6

# | 0 | -5 | 0 |

7

# +---+-----+-----+

8

# | 0 | 0 | 10 |

9

# +---+-----+-----+

10

# Define the grid world environment

1 usage

11

class GridWorldOffPolicy:

12

def \_\_init\_\_(self):

13

self.grid\_size = 3

14

self.start\_state = (0, 0)

15

self.goal\_state = (2, 2)

16

self.state = self.start\_state

17

2 usages (1 dynamic)

GridWorldOffPolicy > \_\_init\_\_()

Run

offpolicy

/Users/khosrov/Desktop/projects/python/playground/.venv/bin/python /Users/khosrov/Desktop/projects/python/playground/offpolicy.py

State (0, 0): {'U': 5.109326301711805, 'D': 3.516100708601847, 'L': 5.067283597264415, 'R': 6.732889999999976}

State (0, 1): {'U': 5.877018825610061, 'D': 2.7733821726609027, 'L': 5.518853794494394, 'R': 7.810999999999998}

State (0, 2): {'U': 6.728829609870805, 'D': 8.899999999999984, 'L': 6.172690861896233, 'R': 7.5053614946683105}

State (1, 0): {'U': -0.6719585934917103, 'D': 6.286361568120909, 'L': -0.5985019985005999, 'R': -0.95}

State (1, 1): {'U': 0.583288037847052, 'D': 0.775218760500567, 'L': -0.23089790208085514, 'R': 8.8389763643813}

State (1, 2): {'U': 7.3985763409476, 'D': 9.999999999999993, 'L': 2.7367840608436107, 'R': 8.479609889531822}

State (2, 0): {'U': 0.33229843067993386, 'D': 0.432473274846956, 'L': 0.33583910999336203, 'R': 8.340241574710264}

State (2, 1): {'U': -0.13024766891319234, 'D': 1.443915071556538, 'L': 0.7555761217441317, 'R': 9.892247363356942}

State (2, 2): {'U': 0.0, 'D': 0.0, 'L': 0.0, 'R': 0.0}

22

# GRID WORLD

## Policy Iteration

deep\_reinforcement\_learning\_project ~/Desktop

RL\_project

environments

grid\_world.py

line\_world.py

monty\_hall\_1.py

monty\_hall\_2.py

two\_round\_rock\_paper\_scissors.py

libs

libsecret\_envs.dylib

libsecret\_envs.so

libsecret\_envs\_intel\_macos.dylib

secret\_envs.dll

old\_code

dyna\_q.ipynb

monte\_carlo\_es.ipynb

monte\_carlo\_off\_policy.ipynb

monte\_carlo\_on\_policy.ipynb

policy\_iteration.ipynb

q\_learning.ipynb

sarsa\_learning.ipynb

value\_iteration.ipynb

RL

grid\_world\_python.py

line\_world\_python.py

grid\_world.ipynb

line\_world.ipynb

```
1 import numpy as np
2
3
4 1 usage
5 class GridWorld:
6     def __init__(self):
7         self.grid = np.array([
8             [0, 1, 2, 3, 4],
9             [5, 6, 7, 8, 9],
10            [10, 11, 12, 13, 14],
11            [15, 16, 17, 18, 19],
12            [20, 21, 22, 23, 24]
13        ])
14        self.state = 0
15        self.actions = np.array([0, # up
16                                1, # down
17                                2, # left
18                                3 # right
19                                ])
20        self.rewards = np.array([-1, 0, 1])
21        self.forbidden = np.array([1])
22        self.game_over = False
23
24    def num_states(self) -> int:
```

Run: grid\_world\_python x

/Users/Alanbooorian\_1/miniconda3/bin/python /Users/Alanbooorian\_1/Desktop/Workspace/ESGI/M1/S2/DeepR/DeepRProjectFinal/RL\_project/deep\_reinforcement\_learning\_project/RL\_

Politique optimale : [1 3 3 3 1]

Fonction de valeur optimale : [0.4782969 1.0097379 1.6002279 2.2563279 2.9853279 1.0097379 1.6002279

2.2563279 2.9853279 3.7953279 1.6002279 2.2563279 2.9853279 3.7953279

4.6953279 2.2563279 2.9853279 3.7953279 4.6953279 5.6953279 2.9853279

3.7953279 4.6953279 5.6953279 5.6953279]

Process finished with exit code 0

## 23

## SECRET ENV 0

Malheureusement, sur cet environnement secret, chaque algorithme testé a pris énormément de temps, rendant l'obtention de résultats extrêmement complexe. Cependant, nous ne sommes pas trop déçus, car après un important refactoring du code global, nous avons au moins réussi à lancer les algos sur ces environnements.

```
[_] 1 env = SecretEnv0()
    2 policy_iteration = PolicyIteration(env)
    3 policy, value_function = policy_iteration.policy_iteration()
    4 print("Politique optimale : ", policy)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv0()
    2 value_iteration = ValueIteration(env)
    3 value, value_function = value_iteration.value_iteration()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv0()
    2 monte_carlo_es = MonteCarloES(env, episodes=10)
    3 value, value_function = monte_carlo_es.monte_carlo_es()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv0()
    2 monte_carlo_on_policy = MonteCarloOnPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_on_policy.monte_carlo_on_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv0()
    2 monte_carlo_off_policy = MonteCarloOffPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_off_policy.monte_carlo_off_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv0()
    2 dyna_q = DynaQAgent(env)
    3 value, value_function = dyna_q.dyna_q()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)
```

## 24

## SECRET ENV 1

Malheureusement, sur cet environnement secret, chaque algorithme testé a pris énormément de temps, rendant l'obtention de résultats extrêmement complexe. Cependant, nous ne sommes pas trop déçus, car après un important refactoring du code global, nous avons au moins réussi à lancer les algos sur ces environnements.

```
[_] 1 env = SecretEnv1()
    2 policy_iteration = PolicyIteration(env)
    3 policy, value_function = policy_iteration.policy_iteration()
    4 print("Politique optimale : ", policy)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv1()
    2 value_iteration = ValueIteration(env)
    3 value, value_function = value_iteration.value_iteration()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv1()
    2 monte_carlo_es = MonteCarloES(env, episodes=10)
    3 value, value_function = monte_carlo_es.monte_carlo_es()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv1()
    2 monte_carlo_on_policy = MonteCarloOnPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_on_policy.monte_carlo_on_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv1()
    2 monte_carlo_off_policy = MonteCarloOffPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_off_policy.monte_carlo_off_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv1()
    2 dyna_q = DynaQAgent(env)
    3 value, value_function = dyna_q.dyna_q()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)
```



## 25

## SECRET ENV 2

Malheureusement, sur cet environnement secret, chaque algorithme testé a pris énormément de temps, rendant l'obtention de résultats extrêmement complexe. Cependant, nous ne sommes pas trop déçus, car après un important refactoring du code global, nous avons au moins réussi à lancer les algos sur ces environnements.

```
[_] 1 env = SecretEnv2()
    2 policy_iteration = PolicyIteration(env)
    3 policy, value_function = policy_iteration.policy_iteration()
    4 print("Politique optimale : ", policy)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv2()
    2 value_iteration = ValueIteration(env)
    3 value, value_function = value_iteration.value_iteration()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv2()
    2 monte_carlo_es = MonteCarloES(env, episodes=10)
    3 value, value_function = monte_carlo_es.monte_carlo_es()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv2()
    2 monte_carlo_on_policy = MonteCarloOnPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_on_policy.monte_carlo_on_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv2()
    2 monte_carlo_off_policy = MonteCarloOffPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_off_policy.monte_carlo_off_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv2()
    2 dyna_q = DynaQAgent(env)
    3 value, value_function = dyna_q.dyna_q()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)
```

## 26

## SECRET ENV 3

Malheureusement, sur cet environnement secret, chaque algorithme testé a pris énormément de temps, rendant l'obtention de résultats extrêmement complexe. Cependant, nous ne sommes pas trop déçus, car après un important refactoring du code global, nous avons au moins réussi à lancer les algos sur ces environnements.

```
[_] 1 env = SecretEnv3()
    2 policy_iteration = PolicyIteration(env)
    3 policy, value_function = policy_iteration.policy_iteration()
    4 print("Politique optimale : ", policy)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv3()
    2 value_iteration = ValueIteration(env)
    3 value, value_function = value_iteration.value_iteration()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv3()
    2 monte_carlo_es = MonteCarloES(env, episodes=10)
    3 value, value_function = monte_carlo_es.monte_carlo_es()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv3()
    2 monte_carlo_on_policy = MonteCarloOnPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_on_policy.monte_carlo_on_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv3()
    2 monte_carlo_off_policy = MonteCarloOffPolicy(env, episodes=10)
    3 value, value_function = monte_carlo_off_policy.monte_carlo_off_policy()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)

[.] 1 env = SecretEnv3()
    2 dyna_q = DynaQAgent(env)
    3 value, value_function = dyna_q.dyna_q()
    4 print("Politique optimale : ", value)
    5 print("Fonction de valeur optimale : ", value_function)
```

# 27

## NOTEBOOKS

L'ensemble des notebooks représentant chacun des environnements sont disponibles dans cet archive, ainsi que sur le dépôt GitHub suivant :

[https://github.com/aboloorian/deep\\_reinforcement\\_learning\\_project](https://github.com/aboloorian/deep_reinforcement_learning_project)