FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

UNIVERSITÉ LIBRE DE BRUXELLES ULB

INFO-F-311: Intelligence Artificielle

Projet 4: Reinforcement Learning

Yannick Molinghen **Pascal Tribel** **Tom Lenaerts**

1 Préambule

Dans ce projet, vous allez implémenter des algorithmes d'apprentissage par renforcement (reinforcement learning). On vous fournit des fichiers de base pour le projet que vous pouvez les télécharger sur l'université virtuelle.

1.1 Poetry

Le projet nécessite Python ≥ 3.10 et utilise le système de build Poetry (https://python-poetry.org/docs/#installation).

Après avoir installé Poetry, ouvrez un terminal dans le répertoire du projet et lancez les commandes:

```
poetry shell
poetry install
```

Ces commandes vont automatiquement créer un environnement virtuel puis installer les dépendances du projet. Vous pouvez aussi utiliser `pip` pour installer les dépendences du projet en regardant les dépendences à installer dans le fichier `pyproject.toml`.

1.2 Tests automatiques

Vous pouvez tester vos fonctionnalités avec la commande `pytest`. Pour lancer uniquement les tests d'un fichier en particulier, vous pouvez le donner en paramètre. Il y a un fichier de test pour chacune des tâches que vous devez remplir.

Note: Il est très difficile de tester des algorithmes de RL. Par conséquent, il n'y a des tests que pour les exercices de la Section 3 et de la Section 4.

2 Introduction

Intuitivement, un processus de décision markovien (Markov Decision Process, MDP) est composé d'états dans lesquels un agent effectue des actions qui ont une certaine probabilité d'atterrir dans un autre état.

Après chaque transition, l'agent reçoit une récompense qui donne une indication sur la qualité de l'action: plus la récompense est élevée, meilleure est l'action. Le but d'un agent est de maximiser la somme des récompenses au cours d'un épisode (c'est-à-dire une partie).

Par conséquent, pour chaque état s, on peut calculer sa valeur $V(s)$ qui correspond à la meilleure somme des récompenses possible à partir de s grâce à l'**équation de Bellman** présentée dans l'Equation 1.

$$V(s) = \max_{a \in A} \sum P(s, a, s')[R(s, a, s') + \gamma V(s')] \tag{1}$$

3 Value iteration

L'algorithme de «value iteration» a pour but d'approximer itérativement la valeur $V(s)$ de chaque état s à l'aide de l'Equation 2.

$$V_{k+1} = \max_a \sum_{s'} P(s,a,s') \left[R(s,a,s') + \gamma V_{k(s')} \right] \tag{2}$$

L'algorithme de value iteration prend en paramètre un entier k qui détermine combien d'itération effectuer. Implémentez l'algorithme *value iteration* dans le fichier `value_iteration.py`.

Conseils:

- Quand vous implémentez l'algorithme, faites attention à vous baser sur V_k pour calculer V_{k+1} et à ne pas modifier V_k durant l'itération.
- N'oubliez pas que la valeur d'un état terminal est 0 par définition.

4 Trouvez les bons paramètres

Dans cet exercice, on vous demande de trouver les bons paramètres du MDP et de *value iteration* pour induire un certain comportement à l'agent.

Le MDP considéré est celui présenté dans la Figure 1. Il s'agit d'un `lle.World` dont les rewards ont été modifiées et qui comprend sept états terminaux:

- Cinq "ravins" en bas de la carte. L'agent meurt s'il s'y rend et reçoit une "récompense" de −10;
- Une sortie proche qui rapporte une récompense de 1;
- Une sortie lointaine qui rapporte une récompense de 10.

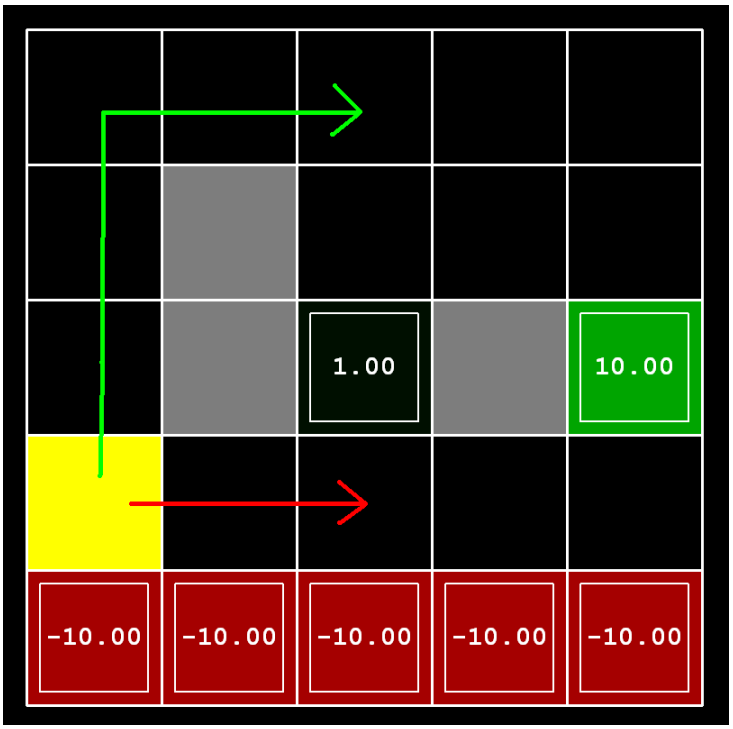

Figure 1: MDP pour lequel trouver les bons paramètres

Le but de cet exercice est d'induire les comportements suivants:

1. Préférer la sortie proche (+1) en longeant la falaise.
2. Préférer la sortie proche (+1) en évitant la falaise.
3. Préférer la sortie distante (+10) en longeant la falaise.
4. Préférer la sortie distante (+10) en évitant la falaise.
5. Eviter de terminer le jeu (l'épisode ne se termine jamais).

Pour ce faire, choisissez des valeurs adéquates de chaque comportement dans la fonction correspondante pour `reward_live` (la récompense à chaque étape pour continuer le jeu), `gamma` (le *discount factor*) et `noise` (la probabilité de prendre une action aléatoire) dans le fichier `analysis.py`.

5 Q-learning

L'algorithme de Q-learning consiste à interagir avec l'environnement pour mettre à jour une fonction d'évaluation $Q(s, a)$. Contrairement aux exercices précédents, on évalue ici la valeur d'une action dans un état $Q(s, a)$ et pas la valeur de l'état $V(s)$.

Une fois que votre agent est entraîné, vous pouvez exploiter la stratégie apprise en prenant l'action ayant la plus haute q-value $\max_{a \in A} Q(s, a)$ dans chaque état.

Pour entraîner votre agent, il est nécessaire d'explorer l'environnement. Pour ce faire, vous devez implémenter la stratégie d'exploration dite «**ε-greedy**» qui consiste à choisir une action aléatoire avec une probabilité ε, et à prendre la meilleure action avec une probabilité $1 - \varepsilon$.

Note: Il n'y a quasiment pas de tests pour les exercices liés au q-learning car ceux-ci sont très difficiles à tester. On vous demande par contre d'analyser vos résultats dans le rapport (voir Section 6).

5.1 Q-learning tabulaire

Pour entraîner la fonction $Q(s, a)$, on emploie une méthode adaptée de l'équation de Bellman (Equation 1) illustrée dans l'Equation 3, où α est le *learning rate*.

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[R(s, a, s') + \gamma V(s')] \tag{3}$$

Implémentation

Implémentez l'algorithme de Q-learning dans le fichier `qlearning.py`. Faites en sorte que cette implémentation utilise un dictionnaire pour stocker les qvalues de chaque état.

Conseils:

- Pour hasher des tableaux `numpy`, nous vous conseillons d'utiliser `hash(array.tobytes())`.
- Pour favoriser l'exploration, initialisez vos $Q(s, a)$ à 1 et non à 0.

Entraînement et rapport

Entraînez votre algorithme sur les niveaux 1, 3 et 6 de LLE. Dans votre rapport, créez un graphique pour chacun de ces trois niveaux qui montre le score (c'est-à-dire la somme des *rewards* par épisode) au cours de l'entrainement. Voir Section 6 pour plus d'informations sur le rapport.

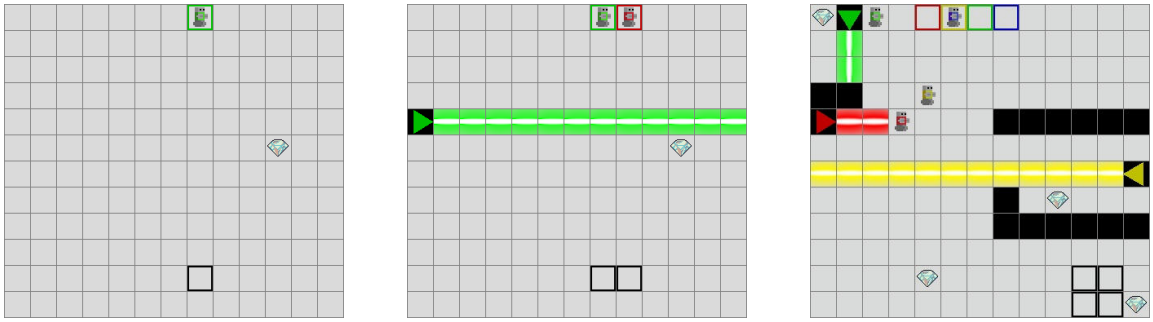

Table 1: De gauche à droite: niveau 1, niveau 3 et niveau 6 de LLE

Exemple

L'extrait de code ci-dessous montre ce à quoi pourrait ressembler votre boucle d'entrainement. Il n'y a aucune obligation de suivre ce canvas.

```
from lle import LLE
from rlenv.wrappers import TimeLimit

env = TimeLimit(LLE.level(1), 80) # Maximum 80 time steps
agents = [QAgent(lr, gamma, ...), QAgent(lr, gamma, ...), ...]
observation = env.reset()
done = truncated = False
score = 0
while not (done or truncated):
  actions = [a.choose_action(observation) for a in agents]
  next_observation, reward, done, truncated, info = env.step(actions)
  for a in agents:
    a.update(...)
  score += reward
  ...
```

Lorsque vous récupérez une observation, vous pouvez accéder à son contenu avec `observation.data` qui contient un tableau `numpy` dont la forme est `(n_agents, ...)`.

Conseil: LLE est un environnement multi-agent et utilise la librairie `rlenv` prévue à cet effet. N'oubliez pas de bien prendre en compte le caractère multi-agent dans la représentation de vos données.

5.2 Approximate Q-learning

Le but de cet exercice est d'implémenter l'Approximate Q-Learning (AQL) pour votre agent. Pour rappel, AQL suppose l'existence d'une fonction $f(s)$ qui associe à chaque état (et pour chaque agent) un vecteur de features $[f_1(s), f_2(s), ..., f_n(s)]$ avec $f_k(s) : S \rightarrow \mathbb{R}$.

Concrètement, dans le cas de LLE, $f_1(s)$ pourrait être le nombre de gemmes non collectées, $f_2(s)$ la distance (en lignes) à la gemme la plus proche, $f_3(s)$ la distance (en colonnes) à la gemme la plus proche, $f_4(s)$ la présense (1) ou l'absence (0) d'un laser nord de l'agent, etc.

Dans AQL, la fonction $Q(s, a)$ d'un agent n est définie comme indiqué dans l'Equation 4.

$$Q_n(s, a) = \sum_{i=1}^{k} f_i(s) w_{a,i} \tag{4}$$

Dans le Q-learning, on définit la TD-error δ (temporal difference error) comme la différence entre la Q-value estimée et la Q-value effective (estimée avec l'équation de Bellman).

$$\delta = [R(s, a, s') + \gamma V(s')] - Q(s, a) \tag{5}$$

Dans AQL, le principe est de mettre à jour les poids w de $Q_n(s, a)$ de sorte à diminuer la TD-error δ conformément à l'Equation 6

$$w_{a,i} \leftarrow w_{a,i} + \alpha\delta f(s) \tag{6}$$

Implémentation

Dans le fichier `approximate_qlearning.py`:

1. Définissez une fonction `feature_extraction(world)` qui prend en entrée le `World` et renvoie pour chaque agent les *features* que vous aurez choisies.
2. Implémentez l'algorithme de Approximate Q-Learning.

Entrainement et rapport

Entraînez votre algorithme AQL sur les niveaux 1, 3 et 6 de LLE. Dans votre rapport, créez un graphique pour chacun de ces trois niveaux qui montre le score (c'est-à-dire la somme des *rewards* par épisode) au cours de l'entrainement. Voir Section 6 pour plus d'informations sur le rapport.

6 Rapport

6.1 Value iteration

- Quelle limitation voyez-vous à l'algorithme de *Value Iteration* ? Dans quelle mesure est-il applicable à un niveau avec un nombre plus important d'agents ou avec une carte plus grande ?
- Expliquez pourquoi *Value Iteration* n'est pas un algorithme de RL mais plutôt un planificateur hors-ligne.
- Appliquez l'algorithme de *Value Iteration* sur le niveau 1 de LLE jusqu'à ce que la *policy* se stabilise. Ensuite, montrez graphiquement la stratégie découverte par l'algorithme.

6.2 Expériences avec le Q-learning

Le but ici est de mesurer les performances des deux algorithmes (Q-learning et AQL) dans les niveaux évoqués précedemment. Vous devez ensuite présenter ces données dans des graphiques adaptés (type de graphique, échelle, axes, titre).

Métrique

La métrique que vous devez tracer est le score, c'est-à-dire la somme des récompenses obtenues au cours d'un épisode.

Méthode

Pour chaque expérience, donnez les paramètres que vous avez utilisés:

- le *discount factor* γ
- le *learning rate* α
- la fonction de *feature extraction* $f(s)$
- le niveau dont il est question
- la limite de temps que vous avez utilisée
- la manière dont évolue votre ε au cours du temps
- ...

Graphiques

Nous vous conseillons la librairie `matplotlib` pour tracer des graphiques. Cette librairie est installée automatiquement si vous avez utilisé poetry.

Les graphiques que vous devez produire sont ceux qui indiquent le score moyen au cours du temps, calculé sur minimum 10 entrainements. Vos graphiques doivent aussi montrer la déviation standard en plus de la moyenne, comme dans ce deuxième exemple de la documentation de matplotlib.

Remise

Le livrable de ce projet se présente sous la forme d'un fichier zip contenant vos sources python ainsi que votre rapport en PDF. Nous vous encourageons à utiliser un outil tel que Typst ou Latex (par exemple avec Overleaf) pour rédiger votre rapport.

Ce travail est **individuel** et doit être rendu sur l'Université Virtuelle pour le 10/12/2023 à 23:59.

7 Annexes

7.1 Précisions sur `lle.LLE`

La classe `lle.LLE` est directement dédiée au RL multi-agents (MARL) coopératif. Par soucis de généralité, `LLE` utilise le vocabulaire des *observations* plutôt que des *états* afin de différencier ce que les agents voient de l'état de l'environnement.

Dans `LLE`, une observation peut se représenter couche par couche, comme montré dans la Figure 2.

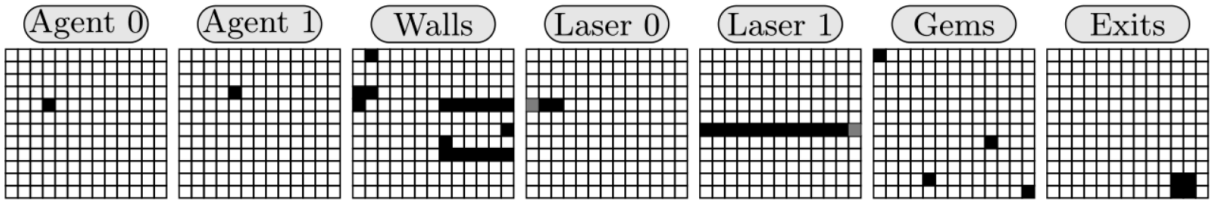

Figure 2: Représentation d'une observation correspondant à la situation de l'imade de droite dans la Table 1. Les cases noires représentent des 1, les cases grises des −1 et les cases blanches des 0. Par soucis de concision, certaines couches (agent 2, agent 3, laser 2, laser 3) ont été ommise.

Le type d'observation indiqué dans la Figure 2 peut être utilisé en utilisant `ObservationType.LAYERED`. Dans ce cas, chaque agent reçoit une observation `(x, height, width)` où `x` dépend du nombre d'agents.

```
from lle import LLE, ObservationType
env = LLE.level(6, ObservationType.LAYERED)
print(env.observation_shape) # (12, 12, 13)
```

Cependant, par défaut (et pour pouvoir être utilisées dans des réseaux de neurones linéaires), `LLE` utilise le type d'observation `ObservationType.FLATTENED`, qui encodent exactement les mêmes informations que `ObservationType.LAYERED`, mais *applaties* sur une seule dimension.

```
from lle import LLE, ObservationType
env = LLE.level(6, ObservationType.FLATTENED) # On peut ommettre le second paramètre
print(env.observation_shape) # Affiche (1872,), càd 12 * 12 * 13
```

Notez que chaque observation possède aussi des informations sur l'état du `World` dans `observation.state`, à la différence que le `WorldState` auquel vous êtes habitués a lui aussi été transformé en tableau numpy pour être utilisable dans le contexte du deep MARL.