Le but de ce TP est de créer une IA jouant au morpion grâce à un apprentissage par renforcement basé sur la méthode du Q-learning, à l'aide de l'équation de Bellman. Avant de commencer il convient de récupérer les fichiers Python `.py` fournis, de les placer dans un même dossier et de les ouvrir au fur et à mesure du déroulement du TP.

1 L'environnement de jeu.

Pour créer une IA jouant au morpion, il faut tout d'abord un environnement de jeu. Le fichier `morpion.py`, dont les premières lignes de codes sont reproduites ci-dessous, fournit cet environnement.

```
import numpy as np

class Morpion:
    def __init__(self, markerJ1 = "X", markerJ2 = "O"):
        self.__grille = None
        self.__cases_vides = None
        self.__grille_joueurs = None
        self.__grille_points = None
        self.__done = False
        self.current_player = 0
        self.__markers = [markerJ1, markerJ2]
        self.__case2point = {c:(c//3,c%3) for c in range(9)}
        self.__info = {}
        self.reset()
```

Ce script Python utilise la programmation orientée objet (POO). Il permet de créer une classe `Morpion` comprenant des attributs et des méthodes. Par exemple, la valeur `0` est affectée à l'attribut `current_player`.

1.1 Prise en main rapide.

Après avoir interprété le code, on travaille dans la console pour découvrir l'environnement de jeu et la manière d'utiliser les méthodes en programmation objet. On crée tout d'abord une instance de la classe `Morpion`, nommée `env` :

```
>>>env=Morpion()
```

Il n'est pas nécessaire de préciser les arguments `markerJ1` et `markerJ2` qui sont définis par défaut. On peut aussi écrire `env=Morpion("A","B")` si on préfère jouer avec les symboles `A` et `B` plutôt qu'avec des `X` et des `O`...

On peut alors vérifier la valeur de l'attribut `current_player` qui indique quel joueur (`0` ou `1`) a la main :

```
>>> env.current_player
0
```

On peut désormais utiliser les différentes méthodes à notre disposition. Par exemple la méthode `render` permet d'afficher la grille (qui est vide pour l'instant) :

```
>>> env.render()
- - -
- - -
- - -
```

La méthode `obs` renvoie la description de l'environnement pour chacun des deux joueurs, à savoir deux tableaux composés chacun :

- d'un tableau indiquant les cases vides (1 si vide, 0 si occupée),
- d'un tableau indiquant les cases occupées par le joueur (1 si occupée, 0 sinon),
- d'un tableau indiquant les cases occupées par le joueur adverse (1 si occupée, 0 sinon).

```
>>> env.obs()
array([[[1, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0]],

       [[1, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0]]])
```

La méthode `info` renvoie des informations sur l'état du jeu, à savoir un dictionnaire contenant quatre couples clés/valeur :

- `"legal_actions"` : la liste des cases qu'on peut jouer au coup suivant,
- `"current_player"` : le numéro du joueur qui a la main (`0` ou `1`),
- `"win_actions"` : la liste des cases gagnant la partie immédiatement, c'est-à-dire créant un alignement,
- `"def_actions"` : la liste des cases contrant un gain immédiat, c'est-à-dire empêchant un alignement.

```
>>> env.info()
{"legal_actions": [0, 1, 2, 3, 4, 5, 6, 7, 8], "current_player": 0, "win_actions":
    [], "def_actions": []}
```

Pour l'instant, toutes les cases, numérotées de 0 à 8, sont vides et c'est au joueur `0` de jouer : il n'a pas de coup gagnant la partie immédiatement et il n'a pas de coup de défense évident.

Pour jouer, on appelle la méthode `step` en fournissant un numéro de case (entre 0 et 8) :

```
>>> env.step(4)
(array([[[1, 1, 1, 1, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0]],

        [[1, 1, 1, 1, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 1, 0, 0, 0, 0]]]), [0, 0], False, {"legal_actions": [0, 1, 2,
    3, 5, 6, 7, 8], "current_player": 1, "win_actions": [], "def_actions": []})
```

On constate que la la méthode `step` renvoie beaucoup d'informations... Avant de les détailler, vérifions que le coup a bien été joué :

```
>>> env.render()
- - -
- X -
- - -
```

La méthode `step` renvoie :

- les mêmes informations que la méthode `obs` (cases vides, cases occupées par les différents joueurs...)
- une liste de deux valeurs parmi :
  - `[1,-1]` si le joueur qui a commencé la partie a gagné,
  - `[-1,1]` si le joueur qui n'a pas commencé la partie a gagné,
  - `[0,0]` sinon,
- un booléen : `True` si la partie est terminée, `False` sinon.
- les mêmes informations que la méthode `info` (coups possibles, joueur qui a la main, coups gagnants, coups de défense).

Exercice 1 :

1. Si, pour continuer la partie, on tapait, dans la console, les instructions `env.step(1)` puis `env.step(0)`, que renverrait `env.render()` ? Que renverrait `env.info()` ?
2. Vérifier.
3. On tape dans la console `env.step(8)` puis `env.step(3)` puis `env.step(2)`. Que renvoie `env.info()` ?
4. Jouer un coup gagnant puis afficher la grille. Vérifier ce que renvoient `env.obs()` et `env.info()`.

Exercice 2 :

On commence une nouvelle partie en tapant `env.step(0)`, par exemple, dans la console.

1. Parcourir le code pour déterminer ce qui se passerait si on jouait dans une case déjà jouée (en l'occurence la case `0`) ?
2. Parcourir le code pour déterminer ce qui se passerait si on jouait dans une case non existante (par exemple la case `12`) ?
3. Vérifier.

1.2 Deux joueurs jouant de manière aléatoire.

Comme il a été vu précédemment, la méthode `step` renvoie toutes les informations nécessaires à la boucle de jeu. On les récupère de la manière suivante, si on souhaite jouer dans la case numéro 3 par exemple :

```
obs, reward, done, info = env.step(3)
```

Ainsi,

- la variable `obs` contient les mêmes informations que la méthode `obs` (cases vides, cases occupées par les différents joueurs...)
- la variable `reward` contient une liste de deux valeurs parmi :
  - `[1,-1]` si le joueur qui a commencé la partie a gagné,
  - `[-1,1]` si le joueur qui n'a pas commencé la partie a gagné,
  - `[0,0]` sinon
- la variable `done` contient `True` si la partie est terminée, `False` sinon.
- la variable `info` est un dictionnaire contenant les mêmes informations que la méthode `info` (coups possibles, joueur qui a la main, coups gagnants, coups de défense). Ainsi, `info["legal_action"]` contient la liste de toutes les cases disponibles.

Le fichier `jeu_aleatoire_vs_aleatoire.py`, reproduit ci-dessous, permet de faire jouer l'un contre l'autre deux joueurs jouant de manière aléatoire :

```
import morpion, random

# Création de l'environnement
env = morpion.Morpion() # La classe Morpion vient du fichier morpion.py importé à la
    1ere ligne


# Initialisation de l'environnement
obs = env.reset()
info = env.info()
done = False  # La partie n'est pas terminée !

while not done: # Boucle de jeu
    env.render()  # Affichage de la grille
    print("")
    #print(info) (utile pour débuggage)
    case = random.choice(info["legal_actions"]) # Choix d'une case aléatoirement
   parmi les cases disponibles
    obs, reward, done, info = env.step(case)  # Coup joué

env.render()
```

Exercice 3 :

1. Tester le script ci-dessus.
2. Recopier au bon endroit et compléter, dans le fichier `jeu_aleatoire_vs_aleatoire.py`, les lignes 3, 5 et 8 du code ci-dessous pour que le gagnant éventuel soit affiché :

```
1 # Affichage du gagnant
2
3 if reward[...] == ... :
4     print("Partie nulle.")
5 elif ..... :
6     print("Le gagnant est le joueur qui a commencé la partie.")
7 else:
8     .......
```

Exercice 4 :

Compléter et modifier le programme précédent afin qu'il joue un nombre donné de parties (par exemple 1000) et qu'il affiche les pourcentages :

- de parties gagnées par le joueur ayant commencé la partie,
- de parties gagnées par l'autre joueur,
- de parties nulles.

1.3 Jouer contre un joueur jouant de manière aléatoire.

Le fichier `humain_vs_aleatoire.py`, reproduit ci-dessous, doit permettre à un joueur humain de faire une partie contre un joueur jouant de manière aléatoire (commençant la partie) :

```
1  import morpion, random
2
3  # Création de l'environnement
4  env = morpion.Morpion()  # La classe Morpion vient du fichier morpion.py importé à
       la 1ere ligne
5  # Initialisation de l'environnement
6  obs = env.reset()
7  info = env.info()
8  done = False
9
10 while not done: # Boucle de jeu
11     env.render()  # Affichage de la grille
12     print("")
13
14     #print(info) (utile pour débuggage)
15     if env.current_player == 0:
16         case = random.choice(info["legal_actions"]) # Choix d'une case aléatoirement
       parmi les cases disponibles
17     else:
18         case = -1 # Choix d'une case interdite
19         while case ..... :
20             case=int(input("Coup ? (Entre 0 et 8)"))
21
22     obs, reward, done, info = env.step(case)  # Coup joué
23
24 env.render()
25
26 if reward[0] == 0:
27     print("Partie nulle.")
28 elif reward[0] == 1:
29     print("Vous avez perdu.")
30 else:
31     print("Vous avez gagné.")
```

Exercice 5 :

1. Compléter la ligne 20 du code.
2. Tester le jeu.

2 L'Intelligence Artificielle.

On implémente une IA dont l'apprentissage se fait par renforcement grâce à la méthode du Q-learning, basée sur l'équation de Bellman. On va entraîner cette IA en la faisant jouer contre différents types de joueur et mettre en place un sytème de récompenses pour les bons coups, ce qui va faire progresser le niveau de jeu de l'IA.

En Q-learning, on cherche à déterminer une fonction de qualité Q prenant deux paramètres :

- l'état de l'environnement s,
- l'action qu'on veut effectuer a.

Ainsi, $Q(s, a)$ représente la récompense potentielle de l'action a dans l'état s de l'environnement. Si l'on connaît cette fonction, alors il est possible de déterminer l'action à effectuer en fonction de l'état de l'environnement actuel, en choisissant l'action a qui maximise la valeur de $Q(s, a)$.

Il faut voir cette fonction Q comme un tableau à double entrée :

-	État s_1	État s_2	...	État s_n
Action a_1	$Q(s_1, a_1)$	$Q(s_2, a_1)$	...	$Q(s_n, a_1)$
Action a_2	$Q(s_1, a_2)$	$Q(s_2, a_2)$	...	$Q(s_n, a_2)$
...	...	...	...	...
Action a_p	$Q(s_1, a_p)$	$Q(s_2, a_p)$	...	$Q(s_n, a_p)$

À un état donné, on choisit l'action qui correspond à la récompense maximale.

Pour le jeu du morpion, le nombre d'états est relativement faible et le nombre d'actions limité : on peut créer un tableau initialisé avec des valeurs nulles. Ce n'est pas toujours le cas...

Pour remplir le tableau, on fait "jouer" l'IA en utilisant le tableau, et on met à jour le tableau à chaque étape avec la formule d'apprentissage suivante (dite équation de Bellman) :

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(R_{t+1} + \gamma \times \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

ou encore :

$$Q_{t+1}(s_t, a_t) = (1 - \alpha)Q_t(s_t, a_t) + \alpha \times R_{t+1} + \alpha \times \gamma \times \max_a Q_t(s_{t+1}, a))$$

où :

- Q_t désigne la fonction de qualité au temps t,
- s_t et a_t désignent l'état de l'environnement et l'action choisie au temps t,
- R_{t+1} représente la récompense obtenue suite à l'action a_t choisie,
- γ est un paramètre d'apprentissage,
- α est un paramètre de vitesse d'apprentissage.

Remarque(s) :

$\max_a Q_t(s_{t+1}, a)$ est donc le meilleur score pour l'état suivant.

2.1 Le joueur avec IA Q-learning.

Le fichier `Joueur_QL.py` définit une classe `Joueur` qui permettra de jouer des parties contre différents joueurs et de remplir le tableau Q. Sans entrer dans le détail du code, on peut remarquer la méthode `recompenser` qui utilise l'équation de Bellman, l'attribut `Q` étant le tableau qu'on cherche à remplir :

```
def recompenser(self, state, action, reward, new_state, learning_rate=0.85,
    actualisation_factor=0.99):
        self.__Q[state,action] = self.__Q[state,action] + learning_rate * (reward +
    actualisation_factor * np.max(self.__Q[new_state,:]) - self.__Q[state,action])
```

2.2 Les adversaires pour l'entraînement.

Le fichier `Joueurs.py`, dont les premières lignes sont reproduites ci-dessous, définit trois classes correspondant à trois types de joueurs. On s'intéresse pour l'instant à deux types de joueurs :

```
"""
Joueurs pré-programmés
Chaque joueur implémente une méthode "jouer" qui prend en paramètre "info"
"""

import random

"""
----------------
Joueur Aléatoire
----------------
Joue complètement aléatoirement
"""

class Joueur_Aleatoire():
    def __init__(self):
        pass

    def jouer(self, info):
        return random.choice(info["legal_actions"])

"""
-------------
Joueur Simple
-------------
Lorsqu'un coup gagnant existe, il est joué (attaque)
Lorsqu'un coup empêchant un alignement adverse existe, il est joué (défense)
Sinon, un coup aléatoire est joué
"""

class Joueur_Simple():
    def __init__(self):
        pass

    def jouer(self, info):
        pass
```

Comme son nom l'indique, le `Joueur_Aleatoire` joue complètement aléatoirement. Le `Joueur_Simple` doit jouer un peu mieux...

Exercice 6 :

Implémenter la méthode `jouer` du `Joueur_Simple` afin :

- qu'il joue un gagnant quand il y en a un,
- sinon qu'il joue un coup, quand il y en a un, empêchant un alignement adverse,
- sinon qu'il joue aléatoirement.

La variable `info` passée en argument de la méthode `jouer` correspond à la variable `info` vue jusqu'à présent (coups possibles, joueur qui a la main, coups gagnants, coups de défense). Ainsi, `info["legal_action"]` contient la liste de toutes les cases disponibles.

Le troisième joueur, le `Joueur_Avance` joue comme le `Joueur_Simple` sauf qu'il joue son premier coup dans un coin quand il commence ou au centre quand il joue en deuxième (et que c'est possible). Il prend, en outre, en argument un paramètre permettant d'introduire une dose d'aléatoire, afin que ce ne soit pas toujours les mêmes parties qui soient jouées pendant l'apprentissage.

Exercice 7 :

Implémenter une classe `Joueur_Simple_Intermediaire` qui joue comme le `Joueur_Simple` sauf qu'il joue son premier coup systématiquement au centre. S'il commence, il peut toujours le faire ; s'il joue en deuxième et que la case centrale est occupée il joue aléatoirement. On pourra s'inspirer du code du `Joueur_Avance`.

2.3 Entraînement de l'IA et performances.

Le fichier `Entrainements&Stats.py`, dont un extrait est reproduit ci-dessous, permet d'entraîner l'IA en jouant contre les différents types de joueurs implémentés précédemment, puis de lui faire jouer des parties pour mesurer ses performances :

```
import morpion, random
from tqdm import tqdm
import Joueur_QL as JQL
import Joueurs
import numpy as np
import matplotlib.pyplot as plt

EPISODES_ENTRAINEMENT = 5000
EPISODES_STATS = 1000
NOM_IA = "IA_QL"

# Environnement

env = morpion.Morpion()  # La classe Morpion vient du fichier morpion.py importé à
    la 1ere ligne

joueur_Aleatoire = Joueurs.Joueur_Aleatoire() # Les classes des différents joueurs
    proviennent du fichier Joueurs.py importés en début de code
joueur_Simple = Joueurs.Joueur_Simple()
joueur_Avance = Joueurs.Joueur_Avance()
# joueur_Intermediaire = Joueurs.Joueur_Intermediaire()

liste_adversaires_entrainement = [joueur_Aleatoire, joueur_Simple, joueur_Avance]

liste_adversaires_tests = [joueur_Aleatoire, joueur_Simple, joueur_Avance]
```

`EPISODES_ENTRAINEMENT` est le nombre de parties d'entraînement de l'IA pour chacun des adversaires de la liste `liste_adversaires_entrainement`. Une fois que l'IA est entraînée, on la fait jouer `EPISODES_STATS` parties contre chacun des types joueurs de la liste `liste_adversaires_tests` pour évaluer les performances de l'IA et on regarde les pourcentages de victoires/défaites/nulles.

Ainsi, avec le code actuel du fichier `Entrainements&Stats.py`, l'IA va être entraînée 5000 parties contre le `joueur_Aleatoire`. Quand cet entraînement est terminé, on fait jouer à l'IA 1000 parties contre le `joueur_Aleatoire`, 1000 parties contre le `joueur_Simple` et 1000 parties contre le `joueur_Avance` et on affiche les statistiques.

Quand on ferme la fenêtre de statistiques, l'IA est alors entraînée 5000 parties contre le `joueur_Simple`. Quand cet entraînement est terminé, on fait à nouveau jouer à l'IA 1000 parties contre le `joueur_Aleatoire`, 1000 parties contre le `joueur_Simple` et 1000 parties contre le `joueur_Avance` et on affiche les statistiques.

Quand on ferme la fenêtre de statistiques, l'IA est finalement entraînée 5000 parties contre le `joueur_Avance`. Quand cet entraînement est terminé, on fait à nouveau jouer à l'IA 1000 parties contre le `joueur_Aleatoire`, 1000 parties contre le `joueur_Simple` et 1000 parties contre le `joueur_Avance` et on affiche les statistiques.

Exercice 8 :

Tester le code du fichier `Entrainements&Stats.py`.

Remarque(s) :

Si on relance une session d'entraînement, on entraîne davantage l'IA nommée `"IA_QL"`, ce qui peut être pertinent. Si on souhaite "repartir à zéro" il faut supprimer le fichier `"IA_QL"` du répertoire de travail ou le renommer pour l'archiver.

Exercice 9 :

1. Faire varier la valeur de `EPISODES_ENTRAINEMENT` pour déterminer l'influence du nombre de parties d'entraînement sur les performances de l'IA.

2. Modifier la liste `liste_adversaires_entrainement` pour essayer d'optimiser les performances de l'IA. On pourra, entre autres, changer l'ordre des joueurs pour l'entraînement, ajouter le `joueur_intermediaire`, rajouter en fin d'entraînement un joueur avec lequel l'IA s'est déjà entraînée...
   On pourra également ajouter le `joueur_intermediaire` à la liste `liste_adversaires_tests` pour évaluer les performances de l'IA.

Exercice 10 :

Tester le fichier `jeu_IA_vs_humain` qui permet de jouer contre l'IA entraînée nommée `"IA_QL"`.