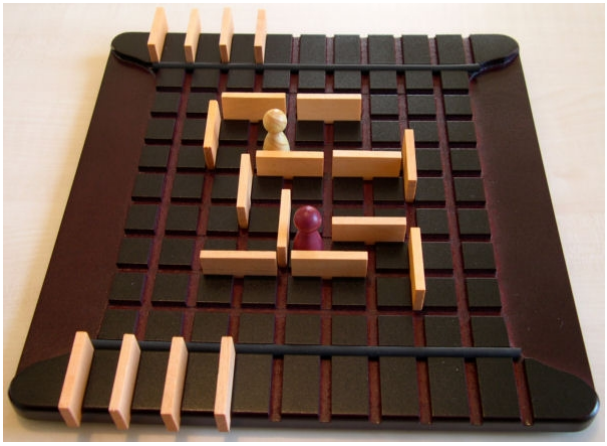
Info-F-106 : Projet d'Informatique
Une intelligence artificielle pour Quoridor

Jérôme De Boeck Gwenaël Joret Charlotte Nachtegael Arnaud Pollaris
Cédric Ternon

version du 28 mars 2019



Présentation générale

Le projet en trois phrases

L'objectif du projet est de réaliser une intelligence artificielle (IA) en Python 3 pour le jeu Quoridor, un jeu de plateau deux joueurs aux règles très simples. Une particularité de cette IA est qu'elle ne connaîtra que les règles du jeu (c-à-d les mouvements possibles), aucune stratégie basée sur les connaissances des joueurs humains ne sera implémentée. L'IA apprendra à jouer en jouant de manière répétée contre elle-même, en utilisant les méthodes *d'apprentissage par renforcement* en combinaison avec des *réseaux de neurones*.

Les étapes de développement

La réalisation du projet est découpée en quatre parties, chacune s'étalant sur environ un mois. Voici un résumé de ce qui sera développé dans chacune de celles-ci :

- Partie 1 : Implémentation d'une IA simpliste basée sur l'apprentissage par renforcement, les états du jeu sont codés explicitement.
- Partie 2 : Amélioration de l'apprentissage grâce à une représentation implicite des états à l'aide d'un réseau de neurones.
- Partie 3 : Réalisation d'une interface graphique pour le jeu à l'aide de la librairie `PyQt`.
- Partie 4 : Différentes améliorations de l'IA au choix de l'étudiant(e), *créativité encouragée*.

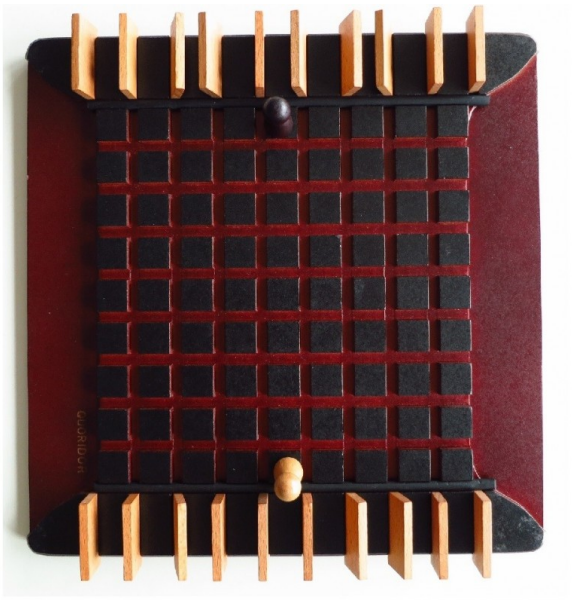
Le jeu Quoridor

Quoridor est un jeu de plateau deux joueurs [1] créé par Mirko Marchesi et édité par la société Gigamic. Le jeu a l'avantage d'avoir des règles très simples et à la fois de faire preuve d'une étonnante profondeur

1. Il y a également une variante quatre joueurs mais celle-ci ne sera pas considérée dans ce projet

de jeu.

Le plateau de jeu est un plateau 9×9, chaque case est appelée *tuile*. Il y a un joueur blanc (qui commence) et un joueur noir. La situation de départ est illustrée ci-dessous :



Voici un très bref aperçu de la mécanique du jeu : Le but de chaque joueur est d'arriver le premier de l'autre côté du plateau. Pour empêcher l'adversaire d'atteindre son but, chaque joueur dispose également de 10 murs qui peuvent être placés sur le plateau entre les tuiles. À chaque tour, un joueur peut soit bouger son pion sur une tuile adjacente (4 directions), soit poser un mur. Le joueur adverse doit toujours avoir un chemin vers la rangée du plateau qu'il doit atteindre, il ne peut donc être complètement emmuré. Le pion d'un joueur peut également sauter au-dessus du pion adverse selon une règle précise. Les règles complètes du jeu sont disponibles au lien suivant : `https://www.gigamic.com/files/catalog/products/rules/quoridor-classic-fr.pdf`, nous vous invitons à les lire avant de continuer à lire ce document.

Dans ce projet, nous ferons varier deux paramètres du jeu : nous considérons un plateau $N \times N$ où chaque joueur possède M murs. Le Quoridor classique correspond donc à $N = 9$ et $M = 10$. Ceci nous permettra de considérer des versions plus petites du jeu lors des premières parties du projet, comme par-exemple un plateau 5×5 avec 6 murs (un *baby Quoridor*), afin de simplifier la tâche d'apprentissage de notre IA.

L'apprentissage par renforcement

Expliquons l'idée générale de l'apprentissage par renforcement directement sur le jeu Quoridor. Un état du jeu est simplement une situation sur le plateau de jeu. Cependant il y a deux types d'états lors d'une partie : le plateau après un coup du joueur blanc, et le plateau après un coup du joueur noir. Pour simplifier, supposons que notre IA soit le joueur blanc, nous allons donc prendre le point de vue du joueur blanc dans ce qui suit. Lorsque blanc doit jouer, il doit choisir parmi les différents mouvements possibles (bouger ou poser un mur). Chaque mouvement possible mène à un nouvel état sur le plateau ; ce sont ces états-là qui vont nous intéresser. Appelons-les *états-après-coup* (*afterstates* dans la littérature anglophone).

Imaginons que pour chacun de ces états-après-coup, notre IA ait une estimation de la probabilité de gagner à partir de cet état. Une stratégie simple pour notre IA est alors de choisir le mouvement qui mène à l'état ayant la plus grande probabilité de gagner. C'est ce qu'on appelle une stratégie *gloutonne* (*greedy*). Il nous faut donc juste obtenir (apprendre) une bonne estimation de ces probabilités de gagner, ce qui est exactement l'objectif de l'apprentissage par renforcement.

L'apprentissage par renforcement utilise différentes techniques pour estimer ces probabilités sur base de nombreuses parties jouées. L'idée la plus simple pour estimer ces probabilités est la suivante : pour chaque état-après-coup s, nous retenons deux nombres n_s et w_s : le premier est le nombre de parties où

l'état s est apparu, et le second est le nombre de ces parties qui ont été gagnées par blanc. Une estimation de la probabilitité de gagner à partir de l'état s pour blanc est alors donnée par le rapport w_s/n_s. Plus l'état s est rencontré lors de différentes parties, plus notre estimation de gagner à partir de s devient précise.

Même si ceci donne une bonne idée du mécanisme d'apprentissage par renforcement, en pratique la méthode ci-dessus prend longtemps avant de converger vers de bonnes estimations, nous verrons des méthodes d'estimation plus astucieuses et efficaces lors de la partie 1 du projet.

Organisation

Pour toute question portant sur ce projet, n'hésitez pas à rencontrer le titulaire du cours ou la personne de contact de la partie concernée.

Titulaire. Gwenaël Joret – gjoret@ulb.ac.be – O8.111

Assistants.

Partie 1 : Cédric Ternon – cternon@ulb.ac.be – N8.217

Partie 2 : Arnaud Pollaris – arnaud.pollaris@ulb.ac.be – N8.21u6

Partie 3 : Charlotte Nachtegael – charlotte.nachtegael@ulb.ac.be – N8.213

Partie 4 : Jérôme De Boeck – jdeboeck@ulb.ac.be – N3.207

Consignes générales

- L'ensemble du projet est à réaliser en `Python 3`.
- Le projet est organisé autour de quatre grandes parties
- En plus de ces quatre parties à remettre, chaque étudiant devra remettre un rapport final et préparer une *présentation orale* d'environ 8 minutes, ces présentations se dérouleront fin avril.
- Chacune des quatre parties du projet compte pour 20 points. Le rapport et la présentation comptent également pour 20 points en tout, ce qui fait un total de 100 points.
- Chacune des quatres grandes parties devra être remise sur GitHub Classroom.
- Le rapport final devra être remis *sur papier* au secrétariat étudiants du Département d'Informatique.
- Après chacunes des trois premières parties, un correctif sera proposé. Vous serez libre de continuer la partie suivante sur base de ce correctif mais nous vous conseillons de plutôt continuer avec votre travail en tenant compte des remarques qui auront été faites.
- Les « Consignes de réalisation des projets » (cf. `http://www.ulb.ac.be/di/consignes_projets_INFO1.pdf`) sont d'application pour ce projet individuel. *(Exception : Ne tenez pas compte des consignes de soumission des fichiers, des consignes précises pour la soumission via GitHub Classroom seront données).* Vous lirez ces consignes en considérant chaque partie de ce projet d'année comme un projet à part entière. Relisez-les régulièrement !
- Si vous avez des questions relatives au projet (incompréhension sur un point de l'énoncé, organisation, *etc.*), n'hésitez pas à contacter le titulaire du cours ou la personne de contact de la partie concernée, et non votre assistant de TP.
- **Il n'y aura pas de seconde session pour ce projet !**

Veuillez noter également que le projet vaudra **zéro** sans exception si :

- le projet ne peut être exécuté correctement via les commandes décrites dans l'énoncé ;
- les noms de fonctions (et de vos scripts) sont différents de ceux décrits dans cet énoncé, ou ont des paramètres différents ;
- à l'aide d'outils automatiques spécialisés, nous avons détecté un plagiat manifeste (entre les projets de plusieurs étudiants, ou avec des éléments trouvés sur Internet). Insistons sur ce dernier point car l'expérience montre que chaque année une poignée d'étudiants pensent qu'un petit copier-coller d'une fonction, suivi d'une réorganisation du code et de quelques renommages de variables

passera inaperçu... Ceci sera sanctionné d'une note nulle pour toutes les personnes impliquées, sans discussion possible. Afin d'évitez ce genre de situations, veillez en particulier à ne pas partager de bouts de codes sur forums, Facebook, etc.

Soumission de fichiers

La soumission des fichiers se fait via un repository individuel par partie du projet sur la plateforme GitHub Classroom. Le lien pour s'inscrire au projet sur GitHub Classroom ainsi que des explications concernant l'utilisation de l'outil Git sont disponibles sur **les slides de présentation de Git / GitHub Classroom** sur l'UV.

Une remarque concernant les retards : Contrairement à la remise de projets pour d'autres de vos cours d'informatique, il n'est ici pas possible de remettre une de vos parties en retard. Le système de versioning offert par Git vous permet de constamment mettre à jour la version de votre code sur le serveur de GitHub Classroom. Vous êtes d'ailleurs **fortement encouragé** à le faire régulièrement lorsque vous travaillez sur une partie. Cela vous permet à vous de garder une copie de chaque version intermédiaire de votre travail, et cela nous permet à nous, en tant qu'enseignants, d'avoir une idée de la régularité de votre travail. Pour l'evaluation d'une partie, vous ne devez pas indiquer quelle est la version "finale" de votre code, nous prendrons simplement la dernière version uploadée sur le repository avant la date et heure limite (toute version uploadée après sera ignorée).

Objectifs pédagogiques

Ce projet *transdisciplinaire* permettra de solliciter vos compétences selon différents aspects.

- Des connaissances vues aux cours de programmation, langages, algorithmique ou mathématiques seront mises à contribution, avec une vue à plus long terme que ce que l'on retrouve dans les divers petits projets de ces cours. L'ampleur du projet requerra une analyse plus stricte et poussée que celle nécessaire à l'écriture d'un projet d'une page, ainsi qu'une utilisation rigoureuse des différents concepts vus aux cours.
- Des connaissances non vues aux cours seront nécessaires, et les étudiants seront invités à les étudier par eux-mêmes, aiguillés par les *tuyaux* fournis par l'équipe encadrant le cours. Il s'agit entre autres d'une connaissance de base des interfaces graphiques en `Python 3`.
- Des compétences de communication seront également nécessaires : à la fin de la partie 4, les étudiants remettront un rapport expliquant leur analyse, les difficultés rencontrées et les solutions proposées. Une utilisation correcte des outils de traitement de texte (utilisation des styles, homogénéité de la présentation, mise en page, *etc.*) sera attendue de la part des étudiants. Une orthographe correcte sera bien entendu exigée.
- En plus d'un rapport, les étudiants prépareront une présentation orale, avec *transparents* ou *slides* (produits par exemple avec LaTeX, `LibreOffice Impress`, `Microsoft PowerPoint`), ainsi qu'une démonstration du logiciel développé. À nouveau, on attendra des étudiants une capacité à présenter et à vulgariser leur projet (c'est-à-dire rendre compréhensible leur présentation pour des non informaticiens, ou en tout cas pour des étudiants ayant eu le cours de programmation mais n'ayant pas connaissance de ce projet-ci).

En résumé, on demande aux étudiants de montrer qu'ils sont capables d'appliquer des concepts vus aux cours, de découvrir par eux-mêmes des nouvelles matières, et enfin de communiquer de façon scientifique le résultat de leur travail.

Conseil concernant la rédaction du rapport

- Il n'est pas obligatoire d'utiliser LaTeX pour votre rapport et vos slides mais c'est encouragé car c'est un outil que vous devrez maîtriser par la suite (au minimum pour écrire votre mini-mémoire de bachelier et votre mémoire de master) et qui demande un certain temps d'apprentissage.

Bon travail !

1 Partie 1 : Apprentissage par renforcement

Le but de cette première partie est de se familiariser avec les principes de bases de l'apprentissage par renforcement. Afin de rendre ce premier contact le plus simple possible, nous allons faire deux hypothèses simplificatrices pour cette partie-ci : (1) les joueurs ne peuvent poser de mur, ils peuvent juste se déplacer, et (2) le plateau est un plateau 5×5. Ces deux hypothèses permettront d'avoir un petit nombre d'états possibles sur le plateau de jeu, ce qui aide beaucoup dans l'apprentissage. Dès la partie 2 nous lèverons ces hypothèses et considérerons le jeu complet avec pose des murs ; nous verrons des techniques pour gérer un grand nombre d'états possibles.

Puisque les joueurs ne peuvent poser de mur, le but est simplement d'arriver de l'autre côté le plus vite possible. (NB : On pourrait croire que chaque joueur joue ici complètement indépendamment de l'autre mais ce n'est pas vrai car un joueur peut sauter au-dessus de son adversaire ; la position du joueur adverse est donc une information importante). Afin de considérer tout de même différents niveaux de difficultés pour notre IA, nous allons considérer différents plateaux de jeu, avec des murs déjà placés.

Concrètement, vous trouverez sur l'UV un fichier `main_partie1.py` qui implémente le jeu considéré lors de cette première partie. Lors du lancement de ce programme, vous pourrez choisir le plateau de jeu dans une liste, ainsi que le type d'apprentissage par renforcement pour chaque joueur (voir plus loin), et le nombre de parties à joueur. Votre tâche sera d'écrire le code implémentant l'IA dans un fichier `IA_partie1.py`, selon les consignes décrites dans cet énoncé.

1.1 Etats du jeu

Pour rappel, nous nous intéressons aux états-après-coup, c-à-d aux états du plateau de jeu après un mouvement de notre IA. A chaque état s, notre IA va associer un nombre p_s entre 0 et 1 qui représente l'estimation de la probabilité de gagner à partir de s. Avant la première partie, l'IA n'a encore aucune connaissance concernant ces probabilités ; elle initialisera simplement ses estimations en mettant $p_s = 0.5$ pour chaque état-après-coup s du jeu qu'elle n'a pas rencontré dans une partie précédente.

1.2 Choix du mouvement

Comme décrit dans l'introduction, une stratégie évidente pour notre IA est de simplement choisir le mouvement qui mène à un état s ayant la plus haute probabilité de gagner. L'IA ne connaît pas les valeurs exactes de ces probabilités mais elle possède ses propres approximations de ces probabilités, les nombres p_s. La stratégie *greedy* consiste alors à simplement choisir le mouvement qui mène à un état s tel que p_s soit maximum. S'il y a plusieurs choix possibles pour s, l'IA choisit un des mouvements correspondants au hasard. Ce détail est important : lors de la première partie l'IA a par-exemple la même estimation $p_s = 0.5$ pour chaque état s, et il semble intuitivement clair qu'il vaut mieux explorer au hasard plutôt que de faire un choix déterministe arbitraire.

Un léger inconvénient de *greedy* est qu'il n'encourage pas l'exploration. Dans tout apprentissage par renforcement, il faut trouver un équilibre entre *exploitation* (choisir le coup qui semble optimal) et *exploration* (choisir un coup peut-être sous-optimal mais qui pourrait nous faire découvrir de nouvelles stratégies par après). Une variante de *greedy* qui favorise plus l'exploration est la stratégie ϵ-*greedy*, avec ϵ un nombre réel fixé entre 0 et 1 : Lorsque l'IA doit choisir un mouvement, elle prend d'abord un nombre r au hasard entre 0 et 1. Si $r \geq \epsilon$, alors elle suit la stratégie *greedy* décrite ci-dessus. Si par-contre $r < \epsilon$, alors elle choisit simplement un mouvement au hasard. Plus ϵ est elevé, plus cette stratégie favorisera l'exploration. (Remarquez que la stratégie *greedy* correspond à prendre $\epsilon = 0$).

Lors de cette partie 1, nous vous demandons d'implémenter ϵ-*greedy* pour le choix du mouvement, et nous vous encourageons à tester différentes valeurs pour ϵ par vous-même.

1.3 Stratégies d'apprentissage

La qualité de notre IA se résume à la qualité de ses estimations p_s des probabilités de gagner à partir d'un état-après-coup s. Une première méthode d'estimation a été décrite dans l'introduction : choisir $p_s = w_s/n_s$ où n_s est le nombre de parties précédentes où l'état s est apparu, et w_s est le nombre de ces parties qui ont été gagnées par l'IA (si jamais s n'a pas encore été rencontré dans une partie précédente alors p_s est initialisé arbitrairement à $p_s = 0.5$). Cette approche a le mérite d'être très simple mais est

peu efficace en pratique, nous ne l'utiliserons donc pas dans ce projet (mais rien ne vous empêche de la tester par vous-même !). Le défaut principal de cette approche est que chaque état s est mis à jour de la même manière, alors qu'intuitivement il semble clair que les états apparus vers la fin d'une partie ont plus d'impacts sur le résultat de la partie que ceux du début. En d'autres mots, il manque le facteur *temps* : après une victoire ou défaite, nous devrions modifier plus nos estimations p_s pour les états s rencontrés récemment que pour ceux apparus en début de partie. Les stratégies que nous allons considérer dans ce projet prennent toutes en compte ce facteur temps dans leur mise à jour des nombres p_s.

Les stratégies appartiennent en général à deux grandes familles : les stratégies *off-line* et les stratégies *on-line*. Les premières mettent à jour les nombres p_s seulement à la fin de chaque partie, sur base de la partie jouée et du résultat (gagné ou perdu). Les secondes mettent à jour les nombres p_s *après chaque coup*, et donc continuellement pendant une partie.

Pour cette première partie du projet, nous vous demandons d'implémenter une stratégie off-line appelée *Monte Carlo*, et deux stratégies on-line appelées *TD(0)* et *Q-learning*, afin de vous familiariser avec différents types de stratégies. Ces stratégies sont paramétrées par un nombre réel α fixé entre 0 et 1, appelé *facteur d'apprentissage* (*learning rate*).

1.3.1 Monte Carlo

À la fin d'une partie, Monte Carlo considère la séquence $s_1, s_2, \ldots, s_k$ des états-après-coup rencontrés lors de la partie, dans *l'ordre chronologique inverse*. Donc s_1 est le dernier état-après-coup rencontré, s_2 l'avant-dernier, etc. Monte Carlo met alors à jour les nombres p_{s_i} selon la formule suivante :

$$p_{s_i} \leftarrow (1 - \alpha^i) \cdot p_{s_i} + \alpha^i \cdot r$$

où r représente le résultat de la partie : 1 pour victoire, 0 pour défaite. Remarquons que pour $\alpha < 1$, le facteur α^i décroit très vite (exponentiellement) vers 0, et $(1 - \alpha^i)$ croît très vite vers 1. Les états proches de la fin de partie (i petit) seront fort influencés par le résultat de la partie (r), alors que ceux plus loin dans le temps le seront de moins en moins. C'est une façon standard de prendre en compte la dimension temps dans l'apprentissage.

1.3.2 TD(0)

TD(0) [2] fait partie d'une famille de stratégies on-line appelées *Temporal Difference learning*. Considérons un mouvement de l'IA lors d'une partie et soit s l'état-après-coup correspondant. L'idée de TD(0) est d'aller modifier l'estimation $p_{s'}$ de la probabilité associée à l'état-après-coup s' rencontré *juste avant* s (c-à-d que s' est l'état juste après le coup précédent de l'IA), sur base de p_s :

$$p_{s'} \leftarrow (1 - \alpha) \cdot p_{s'} + \alpha \cdot p_s$$

Notons qu'aucune mise à jour n'est effectuée après le premier coup de l'IA, car l'état s' n'est alors pas défini.

Une fois la partie terminée, une dernière mise à jour est effectuée comme suit : soit s' le dernier état-après-coup rencontré, c-à-d l'état juste après le dernier coup de l'IA. Il reste à mettre à jour $p_{s'}$, selon la formule suivante :

$$p_{s'} \leftarrow (1 - \alpha) \cdot p_{s'} + \alpha \cdot r$$

où r représente le résultat de la partie : 1 pour victoire, 0 pour défaite.

1.3.3 Q-learning

La stratégie du Q-learning est presque la même que celle de TD(0), reprenons donc les mêmes notations que ci-dessus. La seule différence entre le Q-learning et TD(0) est la suivante : lorsque nous mettons à jour l'estimation $p_{s'}$, nous allons utiliser p_{s^*} à la place de p_s, où s^* est l'état-après-coup correspondant au choix de mouvement glouton, c-à-d celui maximisant p_{s^*}. La formule devient donc

2. la signification du 0 entre parenthèses vient du fait que c'est en fait la stratégie TD(λ) pour $\lambda = 0$, nous rencontrerons TD(λ) plus tard dans ce projet.

$$p_{s'} \leftarrow (1-\alpha) \cdot p_{s'} + \alpha \cdot p_{s^*}$$

(A nouveau, aucune mise à jour n'est effectuée après le premier coup de l'IA car l'état s' n'est alors pas défini). Remarquons que si notre IA effectue un choix de mouvement glouton alors $p_{s^*} = p_s$, et la mise à jour est identique à celle de TD(0). C'est seulement lorsque l'IA fait un choix non-glouton (ce qui arrive avec probabilité ϵ dans ϵ-greedy) que les deux formules diffèrent. L'idée du Q-learning est que c'est une bonne idée d'explorer des coups sous-optimaux de temps en temps mais que cela ne doit pas impacter la façon dont nous estimons les probabilités : celles-ci doivent être mises à jour comme si on jouait localement de manière optimale. En d'autres termes, on *découple* le choix du mouvement et la mise à jour des estimations des probabilités, le choix du mouvement peut être aléatoire (pour explorer) mais $p_{s'}$ doit être mis à jour selon notre meilleure connaissance actuelle. C'est une idée simple mais peut-être contre-intuitive à première vue, ce qui explique probablement pourquoi elle n'a été découverte que tardivement, en 1989 par Watkins.

Une fois la partie terminée, une dernière mise à jour est effectuée, cette fois-ci exactement comme pour TD(0) (puisqu'il n'y a plus de mouvement à choisir) : soit s' le dernier état-après-coup rencontré, alors

$$p_{s'} \leftarrow (1-\alpha) \cdot p_{s'} + \alpha \cdot r$$

où r représente le résultat de la partie : 1 pour victoire, 0 pour défaite.

1.4 Fonctions à implémenter

Nous vous demandons d'implémenter les fonctions suivantes dans le fichier `IA_partie1.py`, qui sera utilisé par `main_partie1.py`. Veillez à respecter scrupuleusement la signature des fonctions, c-à-d le nom (attention miniscules vs majuscules !) et l'ordre des paramètres. Ceci est indispensable pour le bon fonctionnement de votre programme. Un code ne respectant pas une des signatures sera sanctionné d'une note nulle (même si c'est dû juste à une faute de frappe), soyez donc extrêmement vigilant sur ce point !

`makeMove(M, last, strategy, eps, alpha) : return move`

`M` est la liste des listes de la forme `[s, p]` avec `s` un mouvement possible pour le joueur (c-à-d un état-après-coup possible) et `p` l'estimation actuelle de la probabilité de gagner à partir de cet état. Remarque : si l'état `s` n'a pas encore été rencontré par le joueur, alors la boucle principale du jeu (`main_partie1.py`) initialisera `p` avec la valeur arbitraire `p=0.5`.
`last` est la liste `[s, p]` avec `s` l'état après le précédent coup joué par le joueur, et `p` l'estimation actuelle de la probabilité de gagner à partir de cet état. Si jamais le joueur n'a pas encore joué de coup lors de la partie en cours, alors `last` vaut `None`.
`strategy` est une chaîne de caractères indiquant la stratégie d'apprentissage : `'Monte Carlo'`, `'TD(0)'`, ou `'Q-learning'`.
`eps` est la valeur de ϵ pour ϵ-greedy.
`alpha` est la valeur de α (*learning rate*) dans l'apprentissage.

Etant donnés les paramètres ci-dessus, cette fonction devra choisir le mouvement à jouer selon la stratégie ϵ-greedy. Dans le cas d'une stratégie d'apprentissage on-line, elle devra également mettre à jour l'estimation de la probabilité associée à l'état-après-coup précédent dans la liste `last`. La fonction renvoie le mouvement choisi (uniquement l'état-après-coup `s` correspondant, sans la probabilité `p` associée).

`endGame(won, history, strategy, alpha) : return None`

`won` est un booléen indiquant si le joueur a gagné (`True`) ou perdu (`False`) la partie.
`history` est la liste des listes `[s, p]` avec `s` un état-après-coup joué par le joueur lors de la partie et `p` l'estimation actuelle de la probabilité de gagner à partir de cet état, par ordre chronologique.
`strategy` et `alpha` sont comme ci-dessus.

Cette fonction est appelée en fin de partie. Etant donnés les paramètres ci-dessus, cette fonction devra mettre à jour les estimations des probabilités associées aux états-après-coup dans la liste `history` selon la stratégie d'apprentissage considérée.

Vous êtes bien sûr libre d'implémenter des fonctions supplémentaires ; suivez les bonnes pratiques vues au cours de Programmation. Pour effectuer les choix aléatoires, vous aurez besoin du module `random`, que vous pouvez donc utiliser. Pour cette partie 1, vous ne pouvez utiliser d'autres librairies.

Remarques concernant l'utilisation du fichier `main_partie1.py` : Si vous lancez le programme en tapant

```
python3 main_partie1.py
```

le programme choisira des valeurs par défauts pour ϵ et α : $\epsilon = 0.1$ et $\alpha = 0.3$. Vous pouvez également spécifier ces deux paramètres via des arguments de la ligne de commande, par-exemple :

```
python3 main_partie1.py 0.01 0.8
```

pour choisir $\epsilon = 0.01$ et $\alpha = 0.8$. Vous êtes encouragés à tester différentes valeurs pour ces deux paramètres.

1.5 Consignes de remise

Lien GitHub Classroom pour la partie 1 : `https://classroom.github.com/a/yTNI1p6M`

L'ensemble de votre code doit être contenu dans un fichier `IA_partie1.py` qui sera soumis via votre repository sur GitHub Classroom. **Veillez à indiquer vos nom, prénom, et matricule en commentaires au début de votre code.** Il n'est pas nécessaire d'inclure dans votre repository le fichier `main_partie1.py` de la boucle principale de jeu donné sur l'UV, l'assistant en charge des corrections utilisera sa propre version du fichier afin d'effectuer différents tests sur votre code.

Remise : Deadline sur GitHub Classroom : dimanche 18 novembre 2018 à 22 heures. Notez qu'il n'y a aucun retard possible (tout commit passé cette heure ne sera pas pris en compte). Conseil : **uploadez régulièrement votre code** sur le serveur ! (`git commit` + `git push`).

Personne de contact : Cédric Ternon – cternon@ulb.ac.be – N8.217

2 Partie 2 : Utilisation d'un réseau de neurones

L'objectif de cette partie 2 du projet est de réaliser une IA qui apprenne par elle-même à jouer à *Baby Quoridor* : Quoridor sur un plateau 5x5. Pour ce faire, nous allons utiliser un *réseau de neurones* pour évaluer la probabilité de gagner à partir d'un état donné. Celui-ci permettra à l'IA de reconnaître efficacement des situations de jeu semblables à des situations déjà rencontrées précédemment. En d'autres mots, ce réseau de neurones permettra à l'IA de *généraliser* ses connaissances, et ainsi de réagir efficacement face à une situation nouvelle. Un réseau de neurones est spécifié par sa matrice de *poids* (voir plus loin) ; ces poids seront entraînés afin que le réseau donne, pour un état du jeu, une prédiction sur la probabilité de gagner qui soit la plus juste possible.

Nous vous fournissons sur l'UV une boucle de jeu (`main_partie2.py`) qui vous permettra de jouer contre votre IA. Il sera également facile de sauver l'état d'apprentissage d'une IA car il suffira simplement de sauver les poids de son réseau de neurones, ce qui est très compact, typiquement entre 1000 et 10000 nombres pour Baby Quoridor (pour comparaison, il y a déjà plus de 10^9 états possibles dans Baby Quoridor, sauver explicitement une estimation de probabilité de victoire par état comme fait lors de la partie 1 ne serait donc pas envisageable).

Puisque les poids d'un réseau de neurones sont faciles à stocker, vous pourrez facilement vous les échanger entre étudiants (notez que ces poids sont juste une collection de nombres réels, ceux-ci ne donnent aucune information sur le code que vous aurez réalisé pour entraîner votre IA). Vous pourrez ainsi faire jouer votre IA contre une autre et comparer leurs forces respectives. Nous vous fournissons également sur l'UV quelques fichiers poids d'IAs déjà entraînées par nos soins, à vous d'essayer de produire des IAs qui jouent aussi bien, voire mieux !

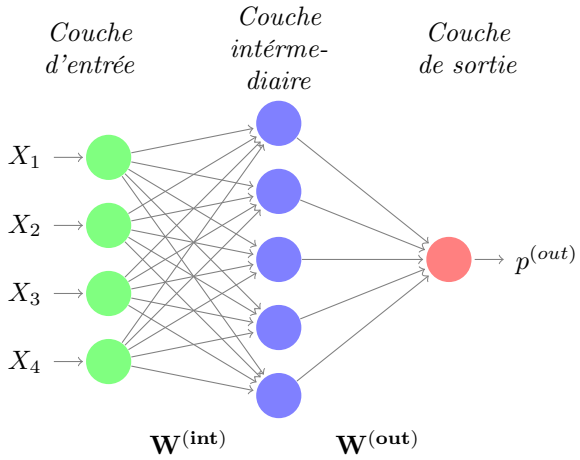
Outre l'utilisation d'un réseau de neurones, une seconde nouveauté de cette partie 2 est l'utilisation d'une nouvelle stratégie d'apprentissage appelée TD(λ). Celle-ci généralise les stratégies TD(0) ($\lambda = 0$) et Monte Carlo ($\lambda = 1$) vues lors de la partie 1, elle est décrite en détail plus loin dans ce document. Lors de cette partie 2 vous devrez implémenter cette stratégie TD(λ) ainsi qu'un Q-learning pour comparaison. Ces deux stratégies, Q-learning et TD(λ), font parties des méthodes d'apprentissage par renforcement les plus utilisées aujourd'hui.

2.1 Description d'un réseau de neurones

L'objectif de cette partie reste d'écrire un code qui, étant donné un état du jeu de Quoridor (incluant pour rappel la position des pions des deux joueurs, la position des murs horizontaux, des murs verticaux, ainsi que les murs non encore placés pour chaque joueur), effectue une prédiction (estimation) quant à la probabilité de gagner. La stratégie adoptée ici consistera à utiliser un réseau de neurones recevant en input l'état actuel (codé sous forme de vecteur binaire) et fournissant en output une estimation de la probabilité de gagner à partir de cet état.

2.2 Perceptron multicouche

Un perceptron multicouche est un modèle de réseau de neurones constitué de plusieurs couches de neurones. Nous considérons dans cette partie-ci un réseau avec trois couches : couche d'entrée, couche intermédiaire et couche de sortie. (Notons au passage que les réseaux ayant de nombreuses couches intermédaires (*deep networks* / *deep learning*) ne seront pas considérés dans le cadre de ce projet afin de limiter la complexité de votre travail). L'architecture que vous devrez implémenter aura la forme suivante :



La couche d'entrée possède un neurone d'entrée par composante X_j du vecteur X en entrée ($n = 4$ dans l'exemple ci-dessus). La couche d'entrée est connectée avec une couche intermédiaire. Le nombre H de neurones de cette couche intermédiaire est un des hyperparamètres du modèle. Vous serez invités dans le cadre de cette partie du projet à réaliser différents tests afin de déterminer quel pourrait être le nombre idéal de neurones à utiliser dans cette couche intermédiaire. Les poids des connexions entre les deux premières couches sont encodés sous forme d'une matrice $\mathbf{W^{(int)}}$ ayant H lignes et n colonnes. Les poids des connexions entre la couche intermédiaire et l'unique neurone de sortie sont encodés dans la matrice $\mathbf{W^{(out)}}$ ayant H lignes et une colonne.

La fonction d'activation proposée, qui s'applique à la fois aux neurones des couches intermédiaires et au neurone de sortie est une fonction non linéaire, nommée sigmoïde, dont la formulation analytique est la suivante :

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

La sigmoïde est souvent utilisée comme fonction d'activation à cause de certaines propriétés mathématiques avantageuses, comme par exemple :

$$\frac{\partial}{\partial x}\sigma(x) = (1 - \sigma(x))\sigma(x) \tag{2}$$

et

$$0 < \sigma(x) < 1 \qquad \forall x \in \mathbb{R} \tag{3}$$

La propriété (2) permet de rendre plus efficace le calcul de la dérivée première de la fonction d'activation, opération récurrente dans la procédure de backpropagation. La propriété (3) permet d'interpreter la sortie d'un neurone comme une probabilité. En particulier, la sortie $p^{(out)}$ du neurone de la couche de sortie sera interpretée comme une estimation de la probabilité (selon le réseau) que l'état codé en input X permette de mener à la victoire. Cette estimation de la probabilité est calculée comme suit :

$$p^{(out)} = \sigma\left(\sum_{i=1}^{H} \mathbf{W_i^{(out)}} \cdot \sigma\left(\sum_{j=1}^{n} \mathbf{W_{ij}^{(int)}} X_j\right)\right) \tag{4}$$

2.3 Codage des états sous forme d'un vecteur binaire

Comme indiqué ci-dessus, le réseau de neurones reçoit en input un vecteur à partir duquel une probabilité de gagner peut être estimée. Ce vecteur correspond à un état du jeu et reprendra à ce titre différentes informations :

— La case du plateau sur laquelle se trouve le pion blanc (N^2 cases possibles).
— La case du plateau sur laquelle se trouve le pion noir (à nouveau N^2 cases possibles).

— La position des murs horizontaux posés sur le plateau de jeu. Dans ce projet, nous prenons comme convention de référencer un mur par la position de sa case "Sud-Ouest" sur le plateau. Sur un plateau vierge de Quoridor de taille $N * N$, cela offre un total de $(N-1)^2$ cases possibles pour la pose d'un mur horizontal.
— La position des murs verticaux posés sur le plateau de jeu. A nouveau, c'est la case "Sud-Ouest" qui sert de référence pour la position du mur sur le plateau. Ici encore, sur un plateau vierge de taille $N * N$, cela offre un total de $(N-1)^2$ cases possibles pour la pose d'un mur vertical.
— le nombre de murs que le joueur blanc et le joueur noir peuvent respectivement encore poser sur le plateau. Si chaque joueur dispose de N_w murs en début de partie, on totalise ici $2*(N_w+1)$ cas de figure pour le codage (le $+1$ dans la formule vient du fait que l'on souhaite également signaler spécifiquement la situation où le joueur ne dispose plus d'aucun mur à placer).

L'encodage d'un état prend alors la forme d'un vecteur binaire ayant $2*N^2+2*(N-1)^2+2*(N_w+1)$ bits, 90 bits par-exemple pour Baby Quoridor avec $N_w = 3$ murs par joueur, où chaque information ci-dessus est encodée en mettant un bit à 1 au bon endroit dans le vecteur. A partir de cette partie-ci du projet, seul ce nouveau codage des états sera utilisé. Comme vous le remarquerez en lisant la suite de ce document, vous n'avez en fait pas besoin de connaître la façon précise dont un état est encodé pour réaliser votre partie 2, vous devez juste savoir à ce stade-ci qu'un état est représenté sous forme d'un vecteur binaire d'une longueur fixée. Nous reviendrons sur l'encodage précis des états dans la partie 3.

2.4 Q-learning avec un réseau de neurones

Pour l'apprentissage des probabilités de victoire, il vous est demandé d'utiliser deux stratégies différentes, la stratégie Q-learning déjà rencontrée lors de la partie 1, et la stratégie TD(λ). Dans les deux cas, lorsqu'il y a une erreur d'estimation, c-à-d lorsque la probabilité de victoire associée au nouvel état diffère de celle associée à l'état précédent, cette dernière est mise à jour ("corrigée") en modifiant légèrement les poids du réseau de neurones. Nous allons d'abord expliquer comment faire ceci dans le cadre du Q-learning, la stratégie TD(λ) est décrite dans la section suivante.

Soit $p^{(out)}(s, \mathbf{W^{(int)}}, \mathbf{W^{(out)}})$ l'estimation de probabilité de victoire faite par le réseau pour l'état s, c-à-d la valeur de $p^{(out)}$ calculée pour le vecteur binaire X encodant l'état s avec les matrices de poids $\mathbf{W^{(int)}}, \mathbf{W^{(out)}}$. Pour rappel, dans le cas du Q-learning nous comparons toujours cette estimation pour l'état précédent s' à celle du meilleur état s^* que nous pouvons choisir actuellement, même si nous ne choisissons pas nécessairement d'aller dans ce meilleur état (c.f. ϵ-greedy). La formule décrivant la différence δ entre les estimations des deux probabilités est fournie dans l'équation suivante 5.

$$\delta \leftarrow p^{(out)}(s', \mathbf{W^{(int)}}, \mathbf{W^{(out)}}) - p^{(out)}(s^*, \mathbf{W^{(int)}}, \mathbf{W^{(out)}}) \tag{5}$$

Cet écart permet ensuite de mettre à jour les poids du réseau de neurones, comme nous allons l'expliquer. Les deux algorithmes qui seront utilisés, forward pass et backpropagation, sont détaillés aux Algorithmes 1 et 2, respectivement. Mais, pour débuter l'explication, définissons au préalable des quantités intermédiaires, $X_i^{(int)}$ et $P_i^{(int)}$, respectivement pour l'entrée et la sortie du $i^{\text{ème}}$ neurone intermédiaire, et $x^{(out)}$ et $p^{(out)}$, respectivement pour l'entrée et la sortie du neurone de sortie :

$$X_i^{(int)} = \sum_j \mathbf{W_{ij}^{(int)}} X_j$$

$$P_i^{(int)} = \sigma(X_i^{(int)})$$

$$x^{(out)} = \sum_i \mathbf{W_i^{(out)}} P_i^{(int)}$$

$$p^{(out)} = \sigma(x^{(out)})$$

La définition de ces quantités nous permet d'écrire de manière élégante les formules utilisées pour les calculs des gradients :

$$grad^{(out)} = \frac{\partial p^{(out)}}{\partial x^{(out)}} = \sigma(x^{(out)})(1 - \sigma(x^{(out)})) = p^{(out)}(1 - p^{(out)}) \tag{6}$$

$$grad_i^{(int)} = \frac{\partial P_i^{(int)}}{\partial X_i^{(int)}} = \sigma(X_i^{(int)})(1 - \sigma(X_i^{(int)})) = P_i^{(int)}(1 - P_i^{(int)}) \tag{7}$$

Algorithme 1 Pseudocode algorithme forward pass

for chaque neurone i **de la couche intermédiaire do**
  $X_i^{(int)} \leftarrow \sum_j \mathbf{W_{ij}^{(int)}} X_j$
  $P_i^{(int)} \leftarrow \sigma(X_i^{(int)})$
end for
$x^{(out)} \leftarrow \sum_i \mathbf{W_i^{(out)}} P_i^{(int)}$
$p^{(out)} \leftarrow \sigma(x^{(out)})$

Algorithme 2 Pseudocode algorithme backpropagation Q-learning

$\delta \leftarrow p^{(out)}(s', \mathbf{W^{(int)}}, \mathbf{W^{(out)}}) - p^{(out)}(s^*, \mathbf{W^{(int)}}, \mathbf{W^{(out)}})$
for chaque neurone i **de la couche intermédiaire do**
  $\Delta_i^{(int)} \leftarrow grad^{(out)} \cdot \mathbf{W_i^{(out)}} \cdot grad_i^{(int)}$
  for chaque neurone j **de la couche d'entrée do**
    $\mathbf{W_{ij}^{(int)}} \leftarrow \mathbf{W_{ij}^{(int)}} - \alpha \cdot \delta \cdot \Delta_i^{(int)} \cdot X_j$
  end for
  $\mathbf{W_i^{(out)}} \leftarrow \mathbf{W_i^{(out)}} - \alpha \cdot \delta \cdot grad^{(out)} \cdot P_i^{(int)}$
end for

N.B. :

1. Les pseudocodes ci-dessus sont donnés à titre explicatifs. Ceci étant, à l'implémentation, c'est à vous qu'il revient le soin de les optimiser pour l'exécution.
2. Dans l'algorithme relatif à la backpropagation 2, vous aurez probablement relevé la présence d'un paramètre α. Celui-ci correpond au "learning-rate" déjà rencontré dans la partie 1. Vous êtes invités à le faire varier en lui assignant des valeurs entre 0 et 1. Lorsqu'$\alpha = 0$, vous constaterez que la mise à jour des poids n'est jamais effectuée.

Initialisation des poids. Etant donné la fonction d'activation utilisée, une initialisation des poids du réseau de neurones avec des valeurs nulles empêche en pratique, pour des raisons d'approximation numérique, la convergence vers des valeurs optimales des poids. La solution standard pour éviter ces problèmes numériques est d'initialiser les poids du réseau avec des valeurs tirées au hasard selon une distribution normale avec moyenne nulle et petit écart-type. Dans le cas présent, il vous est proposé d'utiliser 0.0001 comme valeur pour l'écart-type. [3]

2.5 La stratégie TD(λ)

Comme mentionné précédemment, TD(λ) est à la fois une généralisation de TD(0) lorsque λ vaut 0 et de Monte-Carlo lorsque λ vaut 1. L'idée sous-jacente dans la stratégie TD(λ) consiste, en parallèle aux différents poids $\mathbf{W^{(int)}}, \mathbf{W^{(out)}}$ du réseau de neurones, à utiliser une "eligibility trace"—codée ici sous la forme d'un couple de deux matrices $\mathbf{Z^{(int)}}, \mathbf{Z^{(out)}}$—pour la mise à jour des valeurs des poids. L'idée de l'"eligibility trace" consiste à intervenir, dans les cas où $0 < \lambda \leq 1$, comme une trace en mémoire à court terme. Autrement dit, suite au passage d'un état à un autre, le système apprendra de ses erreurs et ce non seulement sur base du dernier changement d'état (TD(0)) mais aussi, dans une moindre mesure, à partir des transitions entre les états précédents (comme dans la stratégie Monte-Carlo). La valeur λ est donc un hyperparamètre compris entre 0 et 1 pour lequel vous êtes invité à chercher une valeur

3. Vous êtes encouragés à tester vous-même l'importance de bien choisir ce paramètre : qu'observez-vous avec un écart-type trop grand ? Et à l'opposé avec un écart-type quasi nul ?

satisfaisante pour favoriser un apprentissage efficace du jeu Quoridor.

Les deux matrices d'"eligibility trace" $\mathbf{Z^{(int)}}, \mathbf{Z^{(out)}}$, dont les dimensions sont respectivement les mêmes que celles de $\mathbf{W^{(int)}}, \mathbf{W^{(out)}}$ et dont tous les éléments sont initialisés par des 0 **au début de chaque partie**, seront mises à jour à chaque étape.

Dans le cas du TD(λ), nous nous inscrivons dans le cas d'une stratégie online où la probabilité estimée de l'état précédent sera mise à jour sur base de la probabilité estimée de l'état qui vient d'être observé.

Comme indiqué à l'équation (8), on calculera donc une valeur δ qui représente l'écart entre les deux probabilités estimées de gagner. Pour rappel, ici encore il est important que ces deux probabilités soient estimées à partir du même réseau de neurones, avec les mêmes poids mais à partir de deux états successifs s' et s, où l'état s est l'état choisi par ϵ-greedy et s' est l'état précédent.

$$\delta \leftarrow p^{(out)}(s', \mathbf{W^{(int)}}, \mathbf{W^{(out)}}) - p^{(out)}(s, \mathbf{W^{(int)}}, \mathbf{W^{(out)}}) \tag{8}$$

A présent, nous allons décrire l'algorithme de la backpropagation utilisé dans le cadre de la stratégie TD(λ) 3. Dans la mesure où l'algorithme de forward pass est utilisé pour estimer une probabilité de victoire—peu importe la stratégie d'apprentissage employée pour la mise à jour des poids dont il se servira ensuite—il reste inchangé, que l'on applique le Q-learning ou le TD(λ). La définition des quantités intermédiaires $X_i^{(int)}$ et $P_i^{(int)}$, respectivement pour l'entrée et la sortie d'un neurone intermédiaire, et $x^{(out)}$ et $p^{(out)}$, respectivement pour l'entrée et la sortie du neurone de sortie reste également d'application. Il en va de même pour les formules destinées à calculer les gradiants $grad_i^{(int)}$ et $grad^{(out)}$. Attirons l'attention sur le fait que, si les "eligibilty traces" sont initialement nulles, alors les formules de mise à jour des poids $\mathbf{W_{ij}^{(int)}}$ et $\mathbf{W_i^{(out)}}$ sont alors équivalentes à celles de l'algorithme de backpropagation du Q-learning.

Algorithme 3 Pseudocode algorithme backpropagation TD(λ)

$\delta \leftarrow p^{(out)}(s', \mathbf{W^{(int)}}, \mathbf{W^{(out)}}) - p^{(out)}(s, \mathbf{W^{(int)}}, \mathbf{W^{(out)}})$
for chaque neurone i **de la couche intermédiaire do**
  $\Delta_i^{(int)} \leftarrow grad^{(out)} \cdot \mathbf{W_i^{(out)}} \cdot grad_i^{(int)}$
  for chaque neurone j **de la couche d'entrée do**
    $\mathbf{Z_{ij}^{(int)}} \leftarrow \lambda \cdot \mathbf{Z_{ij}^{(int)}} + \Delta_i^{(int)} \cdot X_j$
    $\mathbf{W_{ij}^{(int)}} \leftarrow \mathbf{W_{ij}^{(int)}} - \alpha \cdot \delta \cdot \mathbf{Z_{ij}^{(int)}}$
  end for
  $\mathbf{Z_i^{(out)}} \leftarrow \lambda \cdot \mathbf{Z_i^{(out)}} + grad^{(out)} \cdot P_i^{(int)}$
  $\mathbf{W_i^{(out)}} \leftarrow \mathbf{W_i^{(out)}} - \alpha \cdot \delta \cdot \mathbf{Z_i^{(out)}}$
end for

Initialisation des poids L'initiation des poids se fait de la même manière que celle décrite pour le réseau de neurones apprenant avec la stratégie Q-learning.

2.6 Phase d'entrainement et phase de test

Dans le cadre de cette partie 2, comme le réseau de neurones de l'IA doit apprendre en jouant (via la backpropagation), nous opérerons une distinction entre la phase d'entraînement et la phase de test qui permettra d'évaluer ensuite les progrès engrangés. Il est possible d'entraîner une IA dotée d'un réseau de neurones contre un joueur humain (mais ce serait très long compte tenu du nombre de parties à jouer pour que des progrès non négligeables soient observés) ou contre une autre IA (par exemple une IA qui choisit chaque coup de manière aléatoire). Mais il est également possible d'entraîner une IA en la faisant jouer directement contre elle-même ! C'est ce que nous vous proposons à présent...

Pour ce faire, définissons une phase d'entraînement constituée d'un certain (grand) nombre de parties, par-exemple 10000, 100000, voire même 1000000 parties. Tout à tour, l'IA jouera un coup en tant que blanc puis un coup en tant que noir. Elle apprendra en jouant contre-elle-même et améliorera

simultanément son niveau en tant que blanc et en tant que noir.

Peut-être penserez-vous que l'alternance des couleurs peut poser des problèmes pour généraliser l'apprentissage. Ceci étant, la difficulté peut être astucieusement contournée en utilisant une approche proposée par Tsinteris & Wilson dans le cadre du jeu de Backgammon. [4] En effet, nous pouvons convenir que quand le joueur blanc joue le meilleur coup possible, c'est-à-dire celui qui maximise sa probabilité de gain, il s'agit également de la situation qui minimise la probabilité de gain du joueur noir. Et inversement : le meilleur coup joué par le joueur noir est celui qui minimise la probabilité de gain du joueur blanc. Pour simplifier, toutes les parties peuvent être alors envisagées du point de vue du joueur blanc avec tantôt l'objectif de maximiser la probabilité de gain lors du passage vers un nouvel état (quand c'est au tour du joueur blanc), tantôt avec l'objectif de minimiser cette probabilité de gain lors du passage vers un nouvel état (quand c'est au tour du joueur noir). Ainsi, comme le même réseau de neurones peut être utilisé pour proposer une prédiction du meilleur et du pire coup, il est possible de l'utiliser tant pour jouer en tant que blanc qu'en tant que noir. C'est exactement ce que nous allons faire pour cette partie-ci du projet.

Insistons sur la conséquence suivante de cette approche : lorsque l'IA est sur le point de choisir un nouvel état s, par-exemple en tant que blanc, l'état précédent s' considéré lors de la mise à jour des poids du réseau de neurones est **l'état actuel** du jeu, et non plus l'état après le coup précédent de blanc, comme c'était le cas lors de la partie 1. Concrètement, puisque toutes les probabilités estimées sont calculées du point de vue de blanc :

- Quand c'est au tour de blanc de jouer, s' est l'état actuel du jeu (juste après le dernier coup de noir), blanc choisit ensuite (avec probabilité $1 - \epsilon$) le meilleur coup possible s, et la mise à jour des poids est faite sur base de la paire s', s.
- Quand c'est au tour de noir de jouer, s' est l'état actuel du jeu (juste après le dernier coup de blanc), noir choisit ensuite (avec probabilité $1 - \epsilon$) le *pire* coup possible s, et la mise à jour des poids est à nouveau faite sur base de la paire s', s.

Après une phase d'entraînement, on peut s'attendre à ce que l'IA ait progressé. Il sera alors intéressant de la faire jouer dans une phase dite de test. Au cours de celle-ci, l'IA ne peut plus apprendre et elle sera confrontée à une autre IA (qui ne peut également pas apprendre au cours de cette phase), par-exemple une copie de notre IA juste avant l'entraînement, ou une autre IA dont le niveau sert de référence (voir les IAs données sur l'UV). Ainsi, si par exemple 1000 parties sont alors jouées (avec alternance des couleurs et une valeur ϵ petite mais non nulle pour favoriser l'exploration de différents coups joués par les deux IA, par-exemple $\epsilon = 0.05$), on s'intéressera au pourcentage de parties gagnées par l'IA qui vient d'être entraînée.

Comme vous l'observerez, le nombre de parties jouées par l'IA contre elle-même en phase d'entraînement a une influence cruciale sur le développement des capacités de l'IA. Il s'agira d'un paramètre que vous serez amené à faire varier et, pourquoi pas, en comparant différentes stratégies d'apprentissage ?

2.7 Fonctions à implémenter

Pour effectuer les choix aléatoires, vous aurez besoin du module `random`, que vous pouvez donc utiliser. De même, afin de faciliter votre travail en ce qui concerne les calculs à effectuer, il vous est demandé d'utiliser la librairie `numpy` de calcul scientifique. En particulier, les vecteurs et matrices rencontrées dans cet énoncé seront tous codés sous forme de `numpy array`. Pour cette partie 2, vous ne pouvez pas utiliser d'autres librairies.

Nous vous demandons d'implémenter les fonctions suivantes dans le fichier `IA_partie2.py`, qui sera utilisé par `main_partie2.py`. Pour rappel, veillez à respecter scrupuleusement la signature des fonctions, c-à-d le nom (attention miniscules vs majuscules !) et l'ordre des paramètres. Ceci est indispensable pour le bon fonctionnement de votre programme.

`makeMove(moves, s, color, NN, eps, learning_strategy=None) : return new_s`

`moves` est la liste des actions possibles pour le joueur, où chaque action est représentée par l'état du plateau de jeu (vecteur binaire) correspondant.

4. `https://www.cs.cornell.edu/boom/2001sp/Tsinteris/gammon.htm`

`s` est un vecteur binaire correspondant l'état courant du plateau de jeu.
`color` est un entier qui vaut 0 lorsque c'est au tour de blanc de jouer, 1 lorsque c'est au tour de noir de jouer.
`NN` est le réseau de neurones qui servira à estimer la probabilité de gain pour le joueur blanc. C'est également ce réseau de neurones qu'il conviendra de modifier par backpropagation lorsque la fonction est appelée dans le cadre d'une phase d'entraînement. `NN` est le couple `(W_int, W_out)` des deux matrices reprenant les différents poids du réseau de neurones.
`eps` est la valeur de ϵ pour ϵ-greedy.
`learning_strategy` correspond à la stratégie d'apprentissage utilisée pour mettre à jour les poids. Par défaut, la valeur est `None`. Autrement, il peut s'agir d'un tuple de la forme `('Q-learning', alpha)` où `alpha` est une valeur de "learning rate" comprise entre 0 et 1 inclus ou encore d'un tuple de la forme `('TD-lambda', alpha, lamb, Zint, Zout)` où `alpha` est une valeur de "learning rate" comprise entre 0 et 1 inclus, `lamb` est un réel compris entre 0 et 1 inclus représentant la valeur de λ du TD(λ), et `Zint, Zout` sont les deux matrices de l'"eligibility trace".
La fonction renvoie l'action choisie, c-à-d le nouvel état `new_s` du plateau de jeu correspondant.

Selon qu'il s'agisse du tour du joueur blanc ou noir, votre fonction déterminera, en utilisant l'algorithme de forward pass du réseau de neurones du joueur IA la liste des meilleurs ou des pires coups à jouer du point de vue du joueur blanc.

Etant donnés les paramètres ci-dessus, cette fonction devra choisir le mouvement à jouer selon la stratégie ϵ-greedy. Cela signifie qu'il y a toujours une probabilité ϵ de réaliser un mouvement d'exploration de manière totalement aléatoire. Dans le cas contraire (probabilité $1 - \epsilon$), un mouvement sera choisi au hasard dans la liste des meilleurs (ou des pires) mouvements.

Si une stratégie d'apprentissage a été choisie, il faudra encore procéder à la backpropagation pour la mise à jour des poids du réseau de neurones selon cette stratégie.

`endGame(s, won, NN, learning_strategy) : return None`
`s` est le dernier état de la partie (donc le pion blanc ou le pion noir a atteint sa ligne d'arrivée).
`won` est un booléen indiquant si le joueur blanc a gagné (`True`) ou perdu (`False`) la partie.
`NN` et `learning_strategy` sont comme dans la fonction `makeMove()` décrite ci-dessus.

Cette fonction est appelée en fin de partie. Elle n'apportera des modifications au réseau de neurones que dans le cas où l'IA concernée est en phase d'entraînement. Dans ce cas, elle estimera à l'aide de son réseau de neurones la probabilité de gain (pour le joueur blanc) du dernier état rencontré avant la fin de la partie et effectuera par backpropagation une dernière mise à jour en se basant sur la différence entre cette probabilité estimée et la valeur du booléen won (0 ou 1).

Si la stratégie d'apprentissage est TD(λ), la fonction prendra soin également de remettre à zéro les "eligibility traces" en prévision de la partie suivante.

`sigmoid(x) : return p`
Cette fonction permet de calculer $\sigma(x)$ pour un réel x donné.
Conseil : utilisez la fonction `exp()` de la librairie numpy.

`createNN(n_input, n_hidden) : return NN`
`n_input` est un entier positif indiquant la taille du vecteur (binaire) d'input que prendra le réseau de neurones.
`n_hidden` est un entier positif indiquant le nombre de neurones que comportera la couche intermédiaire.

Cette fonction est destinée à la création d'un réseau de neurones. En sortie, elle fournira un couple `NN` contenant deux matrices de poids $\mathbf{W^{(int)}}, \mathbf{W^{(out)}}$. Les différents poids seront initialisés sur base d'une loi normale avec des valeurs proches de 0 (cfr supra pour plus de précisions).

`forwardPass(s, NN) : return p_out`
`s` est un vecteur binaire contenant l'input (état du plateau de jeu).
`NN` est un réseau de neurones présenté sous la forme d'un couple de 2 matrices de poids. La fonction réalise le calcul de la forward pass et renvoie la probabilité estimée $p^{(out)}$.

`backpropagation(s, NN, delta, learning_strategy=None) : return None`
`s` est un vecteur binaire contenant l'input (état du plateau de jeu)
`NN` est un réseau de neurones présenté sous la forme d'un couple de 2 matrices de poids. `delta` est la différence entre les probabilités de gain estimées respectivement pour l'état précédent et pour l'état qui vient d'être choisi.
`learning_strategy` correspond à la stratégie d'apprentissage utilisée pour mettre à jour les poids, décrite comme dans la fonction `makeMove()`. Avec cette fonction de backpropagation, aucune valeur spécifique n'est renvoyée mais différentes variables sont mises à jour durablement dans le réseau de neurones : en l'occurrence : $\mathbf{Z}^{(\mathbf{int})}, \mathbf{Z}^{(\mathbf{out})}, \mathbf{W}^{(\mathbf{int})}, \mathbf{W}^{(\mathbf{out})}$.

Vous êtes bien sûr libre d'implémenter des fonctions supplémentaires ; suivez les bonnes pratiques vues au cours de Programmation.

Remarque importante : Ainsi que vous pouvez le voir ci-dessus, vous devez dans un certain nombre de cas *modifier* directement un `numpy array` donné, par-exemple `NN[0], NN[1]`, ou les "eligibility traces". Il est donc primordial d'utiliser des opérations qui modifient directement le `numpy array` sans en créer une copie au préalable. Les opérateurs `+=`, `-=`, `*=`, et `/=` ont cette propriété, il vous faudra donc les utiliser. Par-exemple, si `x` et `y` sont deux `numpy array` de même dimension, l'instruction

```
x += y
```

modifiera directement `x` en mémoire. A l'opposé, l'instruction

```
x = x + y
```

va d'abord créer un *nouveau* `numpy array` contenant le résultat de cette somme, et ensuite `x` pointera vers ce nouveau `numpy array`. Le `numpy array` qui était pointé par `x` avant reste lui par-contre inchangé !

Dans le même ordre d'idées, un `numpy array x` peut être remis à 0 via un appel à la méthode `x.fill(0)`, ce qui vous sera utile lorsque vous devrez remettre les "eligibility traces" à 0.

2.8 Reporting graphique

Pour cette partie, vous êtes confrontés à toute une série d'hyperparamètres que nous vous encourageons à faire varier. Parmi ceux-ci, il y avait le nombre de cases N que compte un côté du plateau de jeu, le nombre de murs que peut placer chaque joueur en début de partie N_w, l'écart-type (toujours fixé à 0.0001 dans le cadre de cette partie du projet) de la distribution normale (Gaussienne) qui sert à initialiser les poids, la taille de la couche intermédiaire du réseau de neurones, le paramètre α (learning rate) relatif à la vitesse de mise à jour des poids, le paramètre λ de la stratégie TD(λ) et le paramètre favorisant l'exploration des états ϵ.

Sur l'UV, nous vous fournissons un réseau de neurones dont les poids ont été fixés. Il vous est demandé d'entraîner un nouveau réseau de neurones (en jouant contre lui-même) avec les valeurs suivantes : $N = 5$, $N_w = 1$, $H = 40$, $\alpha = 0.4$ et $\epsilon = 0.3$. Vous répèterez cette opération séparément pour un Q-learning et pour un TD(λ) avec λ fixé à 0.9. Tant pour le TD(λ) que pour le Q-learning, il vous est demandé d'entraîner votre IA contre elle-même sur 10000 parties, sur 20000, sur 30000, ... et ainsi de suite par pas de 10000 jusqu'à atteindre 100000 parties. [5].

Pour chacun de ces $2 \cdot 10 = 20$ cas de figures, il vous sera demandé de noter le pourcentage de victoires obtenues par l'IA que vous avez entraînée contre celle qui vous a été fournie sur une phase de

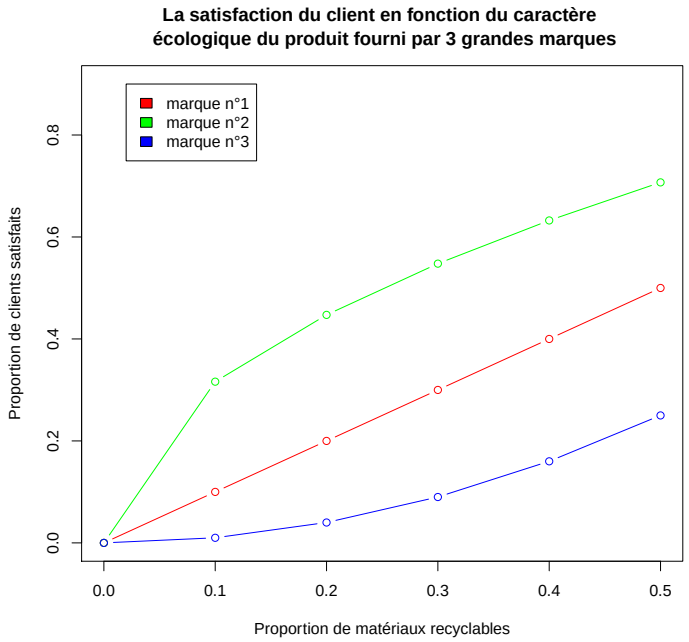
5. Une astuce, pour gagner du temps, consiste, après avoir entraîné et évalué votre IA sur 10000 parties, à l'entraîner sur 10000 parties additionnelles pour atteindre le total de 20000 demandé et ensuite à l'entraîner de 10000 parties supplémentaires pour atteindre 30000, puis 40000, ... parties ; et ce jusqu'à 100000.

test de 1000 parties, avec "eps" fixé à 0.3 pour les deux IA.

A l'aide de ces 20 scores, il vous est demandé de construire un graphique en 2D. En abscisse figurera l'expérience de l'IA, c'est-à-dire le nombre total de parties jouées en phase d'entraînement (l'unité de l'axe sera 1 partie et votre axe ira de 0 à 100000 inclus (vous pouvez simplement placer une graduation toutes les 10000 parties). En ordonnées figurera sur le graphique le % de victoires (votre axe débutera à 0 et ira jusqu'à 60 par pas de 10%. Utilisez une couleur différente pour chacune des 2 stratégies d'apprentissage testées et reliez à l'aide de segments de droite (également colorés) sur le graphique les différents points issus de tests impliquant la même stratégie. N'oubliez pas de légender dans un coin du graphique le code couleur utilisé pour chacune des 2 stratégies, de nommer les différents axes avec un nom évocateur (pas "X" et "Y") et de placer en haut du graphique un titre décrivant ce qu'il représente. Sous votre graphique, un court texte d'une ou deux lignes (maximum) figurera pour décrire ce que vous concluez de la variation des paramètres "nombre de parties jouées en entraînement" et "stratégie d'apprentissage".

Vous êtes libres d'utiliser le logiciel de votre choix pour la réalisation de votre graphique. Ceci étant, une impression de ce dernier sous un format pdf que vous joindrez à votre code est demandée.

Ci-après, vous trouverez un exemple (fictif) de graphique réalisé avec le logiciel R dont vous pouvez vous inspirer :



2.9 A propos du fichier main_partie2.py

Lancez simplement le programme en tapant

```
python3 main_partie2.py
```

Celui-ci chargera les fonctions de votre fichier `IA_partie2.py`. Une aide apparaîtra à l'écran expliquant les différentes commandes.

Une fois familiarisé avec les différentes commandes du programme, il est utile de remarquer qu'on peut également écrire des scripts (des successions de commandes) à l'avance et les faire éxecuter par la bouche principale. Par-exemple, imaginez que vous écriviez le script suivant dans un fichier `commands.txt` :

```
new 5 3 100
eps 0.2
TD-lambda
alpha 0.5
lambda 0.8
train 100000
save IA.npz
quit
```

Ce script crée donc une nouvelle IA pour plateau 5×5 avec 3 murs par joueur avec un réseau de neurones ayant 100 neurones sur la couche interne, il choisit ensuite $\epsilon = 0.2$, la stratégie d'apprentissage TD(λ) avec $\alpha = 0.5$ et $\lambda = 0.8$, entraîne l'IA sur 100000 parties, sauve le résultat dans un fichier `IA.npz`, et enfin quitte le programme. Pour l'exécuter, il suffit de taper :

```
cat commands.txt | python3 main_partie2.py
```

Astuce : si vous souhaitez créer, entraîner, et sauver des IAs pour une série de valeurs différentes des paramètres d'apprentissage, vous pouvez simplement écrire un programme Python qui génère le script correspondant !

2.10 Consignes de remise

Lien GitHub Classroom pour la partie 2 : `https://classroom.github.com/a/uPX9zHiv`

L'ensemble de votre code doit être contenu dans un fichier `IA_partie2.py` qui sera soumis via votre repository sur GitHub Classroom. **Veillez à indiquer vos nom, prénom, et matricule en commentaires au début de votre code.** Il n'est pas nécessaire d'inclure dans votre repository le fichier `main_partie2.py` de la boucle principale de jeu donné sur l'UV, l'assistant en charge des corrections utilisera sa propre version du fichier afin d'effectuer différents tests sur votre code.

Remise : Deadline sur GitHub Classroom : mardi 18 décembre 2018 à 22 heures. Notez qu'il n'y a aucun retard possible (tout commit passé cette heure ne sera pas pris en compte). **Uploadez régulièrement votre code** sur le serveur ! (`git commit` + `git push`), une bonne habitude est de le faire systématiquement à la fin de chaque session de travail.

Personne de contact : Arnaud Pollaris – Arnaud.Pollaris@ulb.ac.be – N8.216

3 Partie 3 : Interface graphique

La troisième partie du projet consiste en la mise en œuvre d'une *interface graphique* (*Graphical User Interface* en anglais, d'où GUI) pour votre logiciel qui permettra à l'utilisateur d'interagir avec votre programme de manière plus conviviale. Dans ce but, nous vous demandons d'utiliser la bibliothèque graphique `PyQt5`[6] (notons que d'autres bibliothèques graphiques pour `Python` existent également, telles que `TkInter`, `PyGTK` ou encore `wxPython`). Par ailleurs, nous vous demandons qu'au terme de cette troisième partie, votre code gère les erreurs produites lors de l'exécution à l'aide de gestion d'exceptions comme vu au cours de programmation.

3.1 Structure et fonctionnement d'une GUI

Une GUI est typiquement composée d'un ensemble d'éléments nommés *widgets* agencés ensembles. Une fenêtre, un bouton, une liste déroulante, une boîte de dialogue ou encore une barre de menus sont des exemples de widgets typiques. Il peut aussi s'agir d'un *conteneur* qui a pour rôle de contenir et d'agencer d'autres widgets.

Une GUI va typiquement être structurée, de manière interne, sous la forme d'un *arbre* de widgets. Dans `PyQt5`, tout programme doit avoir un widget « racine », qui est une instance de la classe `QMainWindow`, créant une fenêtre principale, dans laquelle on peut ensuite ajouter des widgets.

Réaliser la structure visuelle d'une GUI (aussi appelée *maquette*) ne constitue que la moitié du travail, il faut que votre code des parties précédentes puisse être également lié à cette interface. Votre programme ne va donc plus être réduit à se lancer, effectuer des opérations et se terminer. À l'exécution, une GUI va réagir aux *événements* (par exemple, un clic d'un bouton). `PyQt5` offre notamment la possibilité de lier le clic d'un bouton à l'appel d'une fonction donnée. Donc, le principe d'une GUI est d'exécuter une boucle qui va traiter les événements jusqu'á la terminaison de l'application. Il vous faudra utiliser intelligemment ces possibilités pour mettre à jour l'affichage au besoin.

3.2 Exemple de maquette et contraintes

La Figure 1 vous donne un exemple de maquette. Il doit être possible de fermer votre programme en cliquant sur le bouton de fermeture de la fenêtre. Notez que la créativité est encouragée ; vous n'êtes pas tenu de reproduire au pixel près cet exemple. Veillez à ce que votre interface graphique soit ergonomique, agréable visuellement et simple d'utilisation.

Remarque sur la performance. Si vous liez une fonction qui nécessite un certain temps de calcul à un bouton de votre interface, celle-ci restera « gelée » après l'événement (clic) jusqu'à ce que la fonction termine son exécution. Dans le cadre de ce projet, nous nous contenterons de ce comportement (et ne le pénaliserons certainement pas lors de la notation). Pour éviter ce phénomène, il faut exploiter le concept de *multithreading* (qui sort du cadre de ce projet) enseigné aux cours INFO-F201 (Systèmes d'exploitation) et INFO-F202 (Langages de programmation 2). Ici, vous pouvez utiliser la fonction `QApplication.processEvents()` pour forcer le relancement de votre GUI.

3.3 Comportement de la GUI

Votre GUI est composée de deux parties : la partie où vous pouvez sélectionner les paramètres de votre jeu (type d'IA, taille du plateau, nombre de murs, etc.) et une partie avec le plateau du jeu.

La partie supérieure doit contenir tous les paramètres que vous aviez implémentés pour la partie 2. Pensez bien aux limites imposées par les paramètres et les valeurs possibles que ceux-ci peuvent prendre lors de votre implémentation de la GUI (donc au choix du type de la boîte : menu déroulant, spinbox, etc.) Des *Boutons* vous permettront de lancer le jeu avec les valeurs des paramètres entrées dans votre GUI, de créer une nouvelle IA ou d'entraîner votre IA.

La seconde partie contient le plateau du jeu sur lequel va se dérouler la partie du quoridor avec les paramètres sélectionnés dans la première partie. Cette partie remplace la visualisation que vous aviez précédemment dans votre terminal. Cela veut donc dire que vous allez devoir lire, comprendre et adapter la fonction `main`, `display` et `playGame` du code qui vous est fourni. Un exemple de représentation vous

6. `https://riverbankcomputing.com`

est montré à la Figure 1. Votre plateau fera au maximum une taille de 9x9 cases avec un maximum de 10 murs.

Veillez à bien penser à gérer les exceptions et les cas où votre programme ne pourrait pas fonctionner.

3.4 Utilisation des classes

L'utilisation des classes pour structurer votre code est plus que conseillée et fera l'objet d'une évaluation. Nous vous conseillons entre autre d'implémenter le plateau du jeu dans une classe qui remplacera essentiellement la boucle main du fichier `main_partie2.py`. N'hésitez pas à adapter les fonctions du code de la partie précédente au besoin.

Vous pouvez structurer vos classes dans différents fichiers. Il vous suffit alors d'importer les fonctions créées dans un autre fichier (par exemple le fichier `FileName.py`) grâce à la ligne `from FileName import *` dans votre script python. Veillez simplement à ce que votre GUI finale se lance grâce à la ligne de commande : `python3 partie3.py`. Cela veut donc dire que votre script principal sera nommé **partie3.py**. Tout manquement à cette règle résultera en une note nulle. Veillez également à commenter les fonctions de votre code à l'aide de docstrings, ainsi que d'indiquer vos nom et prénom dans un docstring au début de chacun de vos scripts.

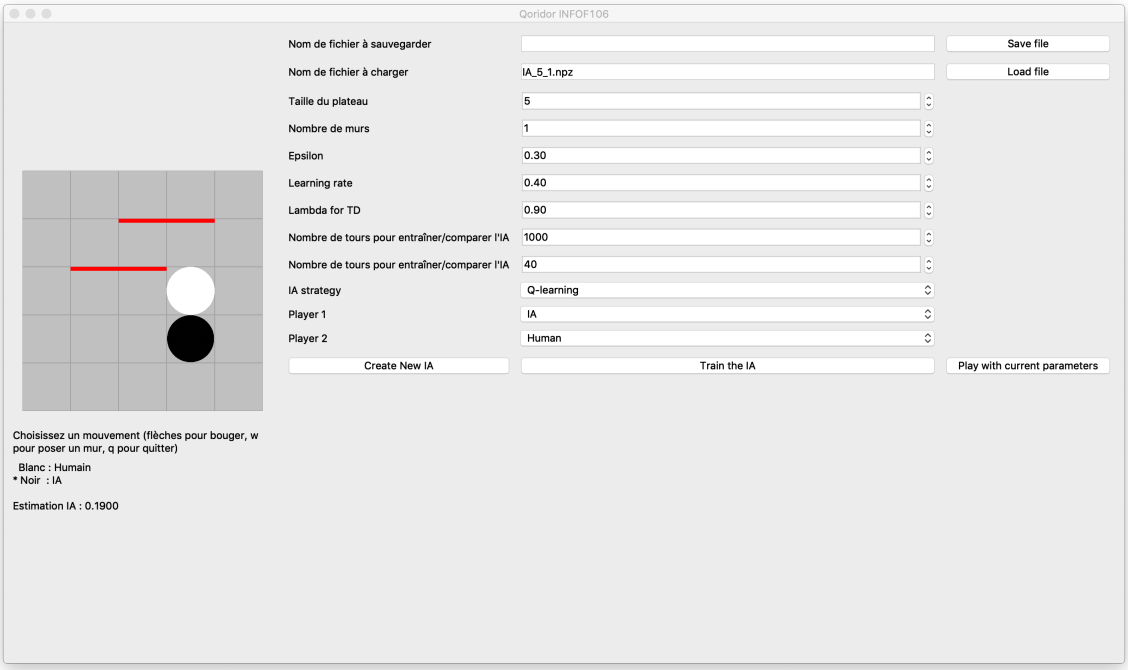


FIGURE 1 – Exemple d'une simulation du jeu quoridor avec un joueur humain et une IA.

3.5 Dessin sur un canevas PyQt5

La Figure 2 donne un exemple de code PyQt5 créant une fenêtre de message qui reçoit un signal et fait une action en fonction de la réponse.

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QMessageBox
from PyQt5.QtGui import QIcon
from PyQt5.QtCore import pyqtSlot


class App(QWidget):

    def __init__(self):
        super().__init__()
        self.title = 'PyQt5 messagebox - pythonspot.com'
        self.left = 10
        self.top = 10
        self.width = 320
        self.height = 200
        self.initUI()

    def initUI(self):
        self.setWindowTitle(self.title)
        self.setGeometry(self.left, self.top, self.width, self.height)

        buttonReply = QMessageBox.question(self, 'PyQt5 message', "Do you like PyQt5?",
                                           QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
        if buttonReply == QMessageBox.Yes:
            print('Yes clicked.')
        else:
            print('No clicked.')

        self.show()


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = App()
    sys.exit(app.exec_())
```

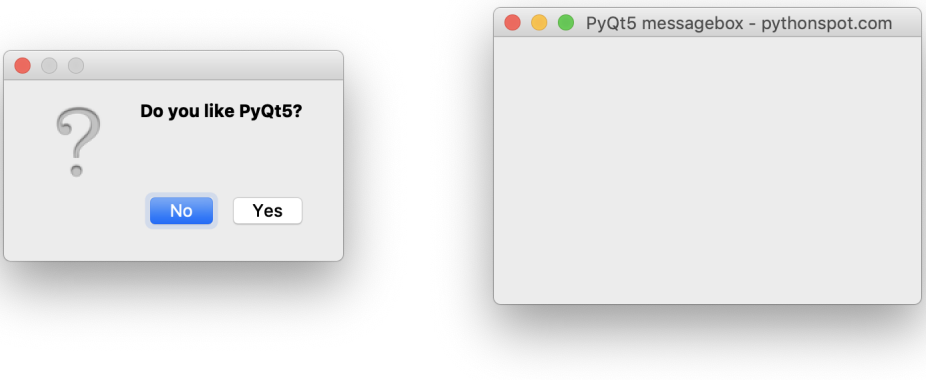


FIGURE 2 – Exemple de maquette *PyQt5*.

De la même manière, on peut ajouter des objets de types `Button`, `Label`, `Menu`, `Text` (et beaucoup d'autres) sur une `Window`. Nous vous invitons à explorer les différents éléments graphiques qui existent dans PyQt5 afin de choisir les éléments qui conviennent le mieux à votre interface. N'hésitez pas à utiliser des couleurs dans votre interface graphique.

3.6 Références utiles

La documentation sur le module `PyQt5` est disponible à l'adresse suivante et pourra vous aider à développer votre interface graphique : `https://wiki.python.org/moin/PyQt`. De plus, vous avez la possibilité d'utiliser `Qt Designer` si vous le souhaitez, un outil d'aide à la conception d'interfaces graphiques. Sa documentation est disponible à l'adresse suivante : `http://pyqt.sourceforge.net/Docs/PyQt5/designer.html`. Un tutoriel simple est disponible à l'adresse suivante : `https://www.tutorialspoint.com/pyqt` et sur `https://build-system.fman.io/pyqt5-tutorial`.

3.7 Rendu

Personne de contact : Charlotte Nachtegael – charlotte.nachtegael@ulb.ac.be – N8.213

À remettre : Les scripts et les autres fichiers nécessaires au fonctionnement du projet (par exemple des images ou autres fichiers texte ou scripts python que vous utiliseriez), dont le fichier `partie3.py` :

— Deadline sur GitHub Classroom : Dimanche 3 mars 2019 à 22 heures. Conseil : **uploadez régulièrement votre code** sur Github Classroom ! Lien pour obtenir l'accès au repository Github : `https://classroom.github.com/a/FPLrqbmz`

4 Partie 4 : Rapport et ajout de fonctionnalités supplémentaires

La quatrième partie du projet d'année consiste à améliorer votre IA selon différentes méthodes proposées ci-dessous. Vous êtes bien évidemment encouragés à implémenter plusieurs de ces propositions si vous avez le temps et l'envie. L'ensemble des nouvelles fonctionnalités devront être intégrées à l'interface graphique développée lors de la partie 3.

Un rapport de 4 à 6 pages motivant les choix d'implémentation est également à rédiger. Ce rapport peut par exemple contenir des graphiques comparatifs des différentes méthodes testées pour améliorer votre IA, mettant en évidence l'influence des différents paramètres. Des conseils sur comment rédiger un rapport scientifique vous seront donnés lors d'une présentation sur un temps de midi (date à préciser).

Remarque concernant la cotation : la quatrième partie du projet est sur 20 points, 10 points sont attribués au code et 10 points au rapport.

Outre la partie implémentation et le rapport, vous aurez également l'occasion de participer à une compétition entre IAs pour des points bonus, voir plus loin pour les détails.

4.1 Amélioration de l'IA

Voici une série de suggestions de variantes à tester. Il n'est nullement obligatoire de les implémenter toutes mais veillez à en implémenter au moins quelques unes.

4.1.1 Variantes de ϵ-greedy

Jusqu'à présent, nous avons utilisé ϵ-greedy pour le choix de l'action par une IA. C'est un bon compromis entre exploitation des connaissances (jouer le meilleur mouvement selon nos connaissances actuelles) et exploration (en jouant au hasard de temps en temps). Cependant, il existe d'autres méthodes qui réalisent ce compromis entre exploration et exploitation. Nous vous proposons ici quelques idées à explorer.

4.1.2 ϵ-greedy avec ϵ décroissant

La variante la plus simple de ϵ-greedy est de ne pas garder ϵ constant durant toutes les parties de l'entraînement mais de commencer avec une valeur $\epsilon_{\text{début}}$ relativement large pour ϵ (ex : $\epsilon_{\text{début}} = 0.5$) et de faire décroitre ϵ vers une valeur finale proche de 0 (ex : $\epsilon_{\text{fin}} = 0.1$) au fil du temps. Ici il faut non seulement bien choisir les valeurs de $\epsilon_{\text{début}}$ et ϵ_{fin} mais aussi la vitesse à laquelle ϵ décroit, ni trop vite ni trop lentement.

4.1.3 ϵ-greedy avec du bruit

Une autre variante intéressante de ϵ-greedy est d'ajouter un petit *bruit* à chaque nombre renvoyé par le réseau de neurones : Si nous notons p_s l'évaluation de la probabilité que blanc gagne à partir de l'état s selon notre réseau de neurones, lorsque ϵ-greedy fait un choix glouton (c-à-d choisir le meilleur état suivant possible), au lieu de comparer les valeurs p_s pour chaque état s candidat, il compare les valeurs $p'_s = p_s + n_s$, où n_s est un bruit pour l'état s : un nombre aléatoire pris selon une distribution normale de moyenne nulle et d'écart-type σ. Si σ est assez petit mais pas trop, cela encouragera glouton à choisir parmi les différents états presque optimaux, au lieu de toujours choisir la meilleure option. En d'autres-mots, quand ϵ-greedy fait un choix glouton il est quand même encouragé à explorer mais seulement parmi les options qui semblent les plus prometteuses. La valeur de l'écart-type σ est évidemment cruciale ici : si elle est trop grande, ϵ-greedy jouera complètement aléatoirement, et si elle trop proche de 0, alors il n'y aura pas de différence avec le ϵ-greedy standard.

Remarque : Il est évidemment aussi possible de combiner cette idée avec la précédente : Vous pourriez faire décroître ϵ et/ou σ au fil du temps.

4.1.4 Soft-Max exponentiel

Une méthode différente de ϵ-greedy consiste à *toujours* faire un choix aléatoire (au lieu de de temps en temps) mais à choisir l'état suivant en utilisant une distribution de probabilités *Soft-Max exponentiel*, qui favorise les états plus prometteurs : Si, comme ci-dessus, nous notons p_s l'évaluation de la probabilité que

blanc gagne à partir de l'état s selon notre réseau de neurones, et si nous notons $\mathcal{S}$ l'ensemble des états parmi lesquels nous devons choisir, alors en tant que blanc nous choisirons l'état $s \in \mathcal{S}$ avec probabilité

$$\frac{e^{p_s}}{\sum_{t \in \mathcal{S}} e^{p_t}},$$

où $e \approx 2.71828$ est la constante d'Euler. Par-exemple, si notre joueur blanc n'a que quatre mouvements haut-bas-gauche-droite possibles, et que le réseau de neurones donne des probabilités de victoire de respectivement

$$0.2 \quad 0.5 \quad 0.1 \quad 0.8$$

aux états résultants de ces mouvements, selon Soft-Max exponentiel les mouvements seront choisis avec les probabilités suivantes :

$$0.197 \quad 0.266 \quad 0.178 \quad 0.359$$

Remarque : Si c'est au tour de noir de jouer, il faut évidemment d'abord complémenter les probabilités p_s calculées par le réseau de neurones, puisqu'il évalue toujours la probabilité de victoire de blanc : il faut donc simplement utiliser les probabilités $p'_s = 1 - p_s$ à la place des p_s.

Vous pouvez expérimenter l'utilisation de Soft-Max exponentiel à la place de ϵ-greedy, ou même faire un hybride des deux : le premier pendant les x premières parties de l'apprentissage, et le second ensuite par exemple.

4.1.5 La stratégie d'apprentissage $Q(\lambda)$ de Watkins

Watkins a proposé une variante de $TD(\lambda)$ qui tente d'intégrer l'idée du Q-learning tout en gardant les "eligibility traces". Sa variante, appelée $Q(\lambda)$, consiste à suivre exactement la stratégie $TD(\lambda)$ sauf que chaque fois qu'un mouvement sous-optimal est choisi dans ϵ-greedy, les eligibility traces sont remises à 0. La philosophie est donc qu'une fois un mouvement non-glouton choisi, nous ne devrions plus modifier notre évaluation des états rencontrés précédemment.

Remarque : ici il faut faire attention au fait que quand ϵ-greedy décide de faire un choix au hasard, ce mouvement choisi aléatoirement pourrait être le mouvement glouton (ceci arrive avec probabilité 1/4 s'il y a quatre mouvements possibles par exemple). Si c'est le cas, il ne faut pas remettre les eligibility traces à 0. Dans $Q(\lambda)$ les eligibility traces sont remises à 0 seulement si le mouvement choisi n'est pas le mouvement glouton.

4.1.6 Fonctions d'activation

Durant la partie 2, vous avez utilisé la sigmoïde comme fonction d'activation du réseau de neurones :

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

On peut travailler avec d'autres fonctions d'activation et comparer leurs performances. Bien qu'historiquement la sigmoïde était une fonction d'activation prisée, ces dernières années la fonction d'activation la plus souvent utilisée est la fonction $ReLU$ (rectified linear unit) :

$$f(x) = max(0, x)$$

Cette fonction remplace donc simplement toute valeur négative par 0. Elle possède de nombreuses propriétés qui en font une fonction d'activation de choix.

Voici d'autres exemples de fonctions d'activation qui sont parfois utilisées :

— Fonction tangente hyperbolique : $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

— Leaky ReLU : $f(x) = max(0.1x, x)$

— SWISH : $f(x) = \frac{x}{1+e^{-x}}$

D'autres fonctions d'activation sont également disponibles sur internet. Nous vous encourageons à tester les différentes fonctions et à voir l'impact de leur utilisation sur vos résultats.

Important : Si vous changer la fonction d'activation, vous devrez adapter les formules des gradients utilisées lors de la back-propagation car la dérivée de la fonction d'activation changera également ! (La dérivée de la sigmoïde $\sigma(x)$ était $\frac{\partial}{\partial x}\sigma(x) = (1 - \sigma(x))\sigma(x)$; les autres fonctions d'activations ci-dessus ont leurs propres dérivées).

4.1.7 Autres variantes et ajouts

Vous pouvez, et êtes encouragés, à développer vos propres idées et variantes personnelles. Pour cette partie 4, la créativité est encouragée !

Vous pouvez également implémenter d'autres idées non mentionnées ici provenant de la littérature. Nous vous demandons alors de contacter l'assistant au préalable afin de vous assurer que c'est adéquat pour ce projet.

4.1.8 Lectures

Ceci n'est *aucunement nécessaire* pour réaliser cette partie 4 mais si jamais vous souhaitez en apprendre davantage sur l'apprentissage par renforcement, nous vous conseillons alors l'excellent livre *Reinforcement Learning : An Introduction* par Richard S. Sutton et Andrew G. Barto, très complet et pédagogique dans sa présentation. Il est disponible sous format pdf à l'adresse suivante : `https://drive.google.com/open?id=1opPSz5AZ_kVa1uWOdOiveNiBFiEOHjkG`

4.2 Rapport

En plus du code, nous vous demandons de rédiger un rapport faisant entre 4 et 6 pages (page de garde, annexes et table des matières non comprises). Le rapport devra porter uniquement sur les nouvelles fonctionnalités de la partie 4 du projet (celui-ci peut bien sûr évoquer les fonctionnalités de base implémentées à la partie 1 et 2). L'interface graphique et sont implémentation ne doivent pas y être décrites. Ce rapport devra contenir les éléments suivants :

- une page de garde qui reprend vos coordonnées, la date, le titre,... ;
- une table des matières ;
- une introduction et une conclusion ;
- des sections décrivant ce qui a été réalisé, les comparaisons de performance, etc. ;
- des exemples d'exécution de votre code ;
- éventuellement des références bibliographiques si applicable.

Le rapport doit détailler les éventuelles difficultés rencontrées, l'analyse et les solutions proposées. Il doit également contenir des comparaisons de performances des différentes méthodes et valeurs de paramètres utilisés. L'utilisation de graphiques est conseillée. Si vous prenez part à la compétition, veillez à également décrire vos choix de méthodes d'apprentissage et les valeurs des paramètres utilisés pour cette compétition.

Le rapport doit être clair et compréhensible pour un étudiant imaginaire qui aurait suivi le cours INFO-F-101 (Programmation) mais n'aurait pas fait le projet ; ni trop d'informations ni trop peu. Expliquez toutes les notions et terminologie que vous introduisez.

Le rapport peut être écrit soit en LaTeX, soit à l'aide des logiciels de traitement de texte `LibreOffice Writer`, `OpenOffice Writer` ou `Microsoft Word`. Il est obligatoire d'utiliser les outils de gestion automatique des styles, de la table de matière et de numérotation des sections. Nous vous conseillons d'utiliser le système LaTeX, très puissant pour une mise en page de qualité. Il est évident qu'une bonne orthographe sera exigée.

Le rapport sera remis sous format papier uniquement. L'échéance pour la remise du rapport est fixée à 5 jours après l'échéance du code (voir ci-dessous), ce qui vous laissera le temps de peaufiner votre rapport une fois le code fini.

4.3 Tournoi entre IAs

Nous allons organiser une compétition entre les IAs des étudiants. La participation est facultative mais encouragée ; tous les étudiants y prennant part avec un code fonctionnel auront **un point bonus** sur 100, et les 16 premiers auront davantage de points bonus, voir ci-dessous.

Afin de participer, vous devez ajouter dans votre repository un fichier `tournoi.py` qui s'exécutera uniquement en ligne de commande (*pas de GUI*!). Une fois executé, ce programme devra automatiquement (sans interaction avec l'utilisateur) lancer la création et l'entraînement d'une nouvelle IA pour un **plateau** 5×5 **avec** 3 **murs** par joueur sur un nombre n de parties spécifié via la ligne de commande, et ensuite sauver l'IA résultante dans un fichier `.npz` dont le nom aura également été spécifié via la ligne de commande, et enfin s'arrêter. Par-exemple,

```
python3 tournoi.py 100000 mon_fichier.npz
```

produira un fichier `mon_fichier.npz` dans le dossier courant contenant le réseau de neurones `NN = [W^int,W^out]` obtenu après 100000 parties d'entraînement. Votre fichier `tournoi.py` fera donc appel aux fonctions développées lors des différentes parties du projet mais ici vous devrez vous même spécifier dans votre code tous les paramètres liés à l'apprentissage de votre IA, afin de produire la meilleure IA possible.

Il est primordial que les conventions d'utilisation de `tournoi.py` ci-dessus soient observées à la lettre. En effet, nous entraînerons les IAs de tous les participants nous même durant les vacances de printemps de manière automatique via un script, d'où la nécessité qu'il n'y ait pas de GUI, et plus généralement aucune interaction avec l'utilisateur. Si votre fichier ne s'exécute pas comme attendu, vous ne serez pas pénalisé mais ne prendrez simplement pas part à la compétition. Le nombre n de parties d'entraînement que nous utiliserons pour chaque IA sera précisé ultérieurement, n sera probablement entre 200000 et 1000000.

La compétition se déroulera comme suit. Chaque partie sera jouée en **greedy pur**, c'est-à-dire ϵ-greedy avec $\epsilon = 0$, afin que les IAs jouent le mieux possible. Il y aura deux phases, une phase de **préselection** et une phase **finale**. Lors de la préselection, nous allons nous-même faire jouer l'un contre l'autre chaque paire d'IAs sur deux parties, afin que chacun puisse jouer blanc et noir. [7]. Nous totaliserons ensuite le nombre de victoires de chaque IA. Ensuite, nous sélectionnerons les 16 IAs ayant les plus grands scores [8]. La phase finale consistera ensuite en un tournoi entre ces 16 IAs. Ce tournoi sera public, nous l'organiserons sur un temps de midi après les vacances de printemps (date à préciser). Nous utiliserons un système à élimination directe avec huitième de finale, quart de finale, demi-finale et une finale pour terminer. Lorsque deux IAs sont mises en compétition, chacune jouera une fois en blanc et une fois en noir. Une IA remporte la manche si elle gagne les deux parties, c-à-d en blanc et en noir. En cas d'égalité, nous recommencerons sur un autre plateau de jeu avec certains murs déjà placés par nos soins, afin de les confronter à une situation nouvelle (notez que, peu importe le plateau, c'est toujours égal puisque chaque IA joue en blanc et en noir). Et s'il y a encore égalité, nous passerons alors à un autre plateau de jeu, etc. jusqu'à ce qu'une IA gagne.

Les points bonus seront ajoutés à la note totale du projet sur 100 comme suit :

$1^{\text{ère}}$ place	+10 points
$2^{\text{ème}}$ place	+8 points
$3^{\text{ème}}$-$4^{\text{ème}}$ place	+6 points
$5^{\text{ème}}$-$8^{\text{ème}}$ place	+4 points
$9^{\text{ème}}$-$16^{\text{ème}}$ place	+3 points

Tous les autres étudiants ayant participés au tournoi avec un code fonctionnel auront 1 point bonus.

4.4 Consignes de remise

Les consignes de remises du code sont semblables à celles de la partie 3 : Vous pouvez structurer vos classes dans différents fichiers, veillez simplement à ce que votre GUI se lance cette fois-ci grâce à la ligne de commande : `python3 partie4.py`. Cela veut donc dire que votre script principal sera nommé **partie4.py**. Tout manquement à cette règle résultera en une note nulle. Veillez également à commenter les fonctions de votre code à l'aide de docstrings, ainsi que d'indiquer vos nom et prénom dans un docstring au début de chacun de vos scripts.

À remettre : Les scripts et les autres fichiers nécessaires au fonctionnement du projet (par exemple des images ou autres fichiers texte ou scripts python que vous utiliseriez), dont le fichier `partie4.py`. Le fichier `tournoi.py` est facultatif et ne doit être remis que si vous souhaitez participer au tournoi.

Deadline sur GitHub Classroom : Dimanche 7 avril à 22h. Lien GitHub Classroom pour la partie 4 : `https://classroom.github.com/a/rovZjIwv`

7. Notez qu'en greedy pur, les IAs jouent de manière déterministe, il n'y a donc pas besoin de jouer plus de deux parties pour les évaluer.

8. s'il y a égalité entre le 16ème et 17ème (et peut-être plus), ce qui pourrait tout à fait arriver, nous utiliserons une autre manière de départager ces IAs que nous expliquerons le cas échéant.

Remise du rapport : Sous format papier uniquement, au secrétariat : Mardi 23 avril avant 16h.

Notez qu'il n'y a aucun retard possible, autant pour la remise du code que du rapport. Comme toujours, **uploadez régulièrement votre code** sur le serveur ! (`git commit` + `git push`), une bonne habitude est de le faire systématiquement à la fin de chaque session de travail.

Personne de contact : Jérôme De Boeck – jdeboeck@ulb.ac.be – N3.204