RAPPORT FINAL DE PFE

“Stratégies d’Optimisation de WebCrawlers par Reinforcement Learning”

“ Traçage d'empreintes de navigateur en utilisant des techniques avancées d'apprentissage automatique”

Etudiant : Naif Mehanna
Tuteurs : Walter Rudametkin / Romain Rouvoy / Antoine Vastel
Responsable de projets IMA5 : Thomas Vantroys

Sommaire

Introduction **3**

Présentation du contexte **4**
- Description et cahier des charges 4
- Le reinforcement learning 6
  - Vue générale du machine learning 6
  - Reinforcement learning 7
- Les bots informatiques 9

Réalisation en environnement de test **10**
- Environnement de test 11
- Modèles et algorithmes 13
  - Modèle orienté sites web 13
  - Modèle orienté crawler 14
  - Modèle mixte 14
  - Algorithmes implémentés 15
- Analyse des résultats 16
  - Résultats du modèle orienté sites web 17
  - Résultats du modèle orienté crawlers 18
  - Résultats du modèle mixte 18

Réalisation en environnement réel **19**
- Modèle et implémentation 19
  - Environnement et modèle 19
  - Cas particulier de gestion du mouvement de souris 23
- Extension à plusieurs machines 25
  - Méthode séquentielle 25
  - Méthode parallèle 25
- Analyse des résultats 29

Conclusion **32**

Bibliographie **33**

Introduction

Dans le cadre du *projet de fin d'études* (PFE), je réalise un projet au sein d'Inria, portant sur le traçage d'empreintes de navigateur en utilisant des techniques avancés d'apprentissage automatique. Les empreintes de navigateurs étant utilisés de manière de plus en plus répandue dans la détection et le blocage des bots informatiques, l'utilisation des techniques d'apprentissage automatique s'inscrit dans le but d'éviter la détection à ces bots en modifiant leur empreinte de façon entièrement automatisée.

Les bots informatiques sont omniprésents sur le web, et disposent d'autant de points positifs que négatifs. Cependant, les bots ne provenant pas d'acteurs informatiques tels que les moteurs de recherches, qui utilisent ces derniers afin d'indexer leur contenu, se retrouvent souvent considérés comme mauvais et sont bloqués malgré une conception dans un but académique. Ainsi, l'utilisation du *machine learning*, et plus précisément, du reinforcement learning, qui est une branche du machine learning aux côtés du supervised learning et de l'unsupervised learning, pourra permettre aux bots informatiques d'apprendre à adopter la configuration la plus optimale de manière automatique, dans le but d'éviter la détection par les sites web, et donc le blocage.

Le travail est ainsi réalisé dans un cadre de recherche, et est encadré par M. Rudametkin, M. Rouvoy ainsi que M. Vastel.

Je présenterai dans un premier temps le contexte du projet, en introduisant les notions principales utilisés en termes de *reinforcement learning* et de bots informatiques, pour passer à la description de l'environnement de test, correspondant à la partie initiale du projet, puis enfin, présenter la réalisation et les résultats en environnement réel.

I. Présentation du contexte

Depuis quelques années, et plus précisément, depuis près d'une décennie, l'utilisation du *machine learning* ne cesse de se généraliser, et ses domaines d'utilisations sont toujours plus vastes. Bien que les algorithmes les plus performants existent depuis les années 70 pour la plupart, nous assistons depuis quelques temps à une véritable révolution dans le taux d'articles de recherche publiés dans le domaine.

Le *machine learning*, autrefois perçu comme inaccessible, a pu s'imposer en raison de l'augmentation de la puissance des ordinateurs personnels, permettant l'exécution d'algorithmes qui étaient dans un premier temps uniquement théoriques. Les domaines du traitement d'image, de la reconnaissance et du traitement vocal, ou encore, de la traduction instantanée, sont autant de domaines que le *machine learning* a su révolutionner après des années de stagnation relative.

Le travail réalisé dans le cadre du PFE allie ainsi l'utilisation du *machine learning* au cas des bots informatiques, et plus précisément, des *crawlers*, qui se chargent de récupérer des informations publiquement affichées sur des sites webs.
L'état de l'art dans le domaine de l'utilisation du *machine learning* au profit des *crawlers* n'est pas assez développé avec simplement quelques recherches alliant la reconnaissance d'image pour un crawling efficace basée sur une analyse visuelle [1] , ou encore dans le cas de l'utilisation du *reinforcement learning* pour une exploration optimale du *deep web [2].* Finalement, un nombre plus conséquent de papiers à été mis en avant sur l'utilisation du *reinforcement learning* dans le choix optimal des liens à visiter [3].

Ainsi, dans le cadre de notre projet, nous introduisons également la notion de *bots informatiques,* et plus précisément, des *crawlers,* en présentant leur caractéristiques, les différentes méthodes pour altérer leur comportement, ainsi que différents frameworks possibles utilisés pour les mettre en place.

a. Description et cahier des charges

Le but de ce projet est ainsi d'utiliser les techniques de *reinforcement learning* (RL) afin de permettre à des *crawlers*, dont le but est de récupérer des informations publiques sur les pages webs, de réaliser les actions pour lesquelles ils ont été programmés sans se faire détecter par les sites web. En d'autres mots, nous cherchons à optimiser de manière automatique le parcours des *crawlers* sur le web afin d'éviter leur blocage.
En effet, une grande majorité des sites web bloquent ou limitent l'accès aux *bots*, en raison notamment de la haute bande passante consommée par ces derniers, entraînant un ralentissement du site en question.

Le blocage d'un *crawler* par un site web entraîne des conséquences sérieuses pour les acteurs derrière ces bots informatiques : il est ainsi courant pour ces derniers de louer un nombre d'adresses IPs résidentielles, ou propres. Le blocage de ceux-ci engendre ainsi des pertes d'argent parfois conséquentes.
Une perte de temps conséquente est également engendrée par le blocage des IPs utilisés par un *crawler* : certains sites webs bloquent des IPs de manière non définitive, pour une durée de temps fixée. Ainsi, dans le cas où toutes les adresses IPs ont été bloqués, l'attente jusqu'au déblocage de ces

derniers est limitante.
Notre but est ainsi de mettre en place une série de techniques utilisant le *reinforcement learning* permettant d'éviter ces conséquences en s'attaquant directement à la raison du blocage : nos *crawlers* devront ainsi éviter le blocage en analysant chaque site web visité et en extrayant des informations permettant de guider ceux-ci de manière totalement automatisé.

L'application du *reinforcement learning* permettra ainsi aux *bots* d'apprendre par eux même des informations sur les sites web que nous chercherons à *crawler*, et ainsi de déterminer la configuration et le comportement optimal pour éviter la détection, tout en privilégiant la rapidité d'exécution que l'utilisation des *bots* informatiques confèrent.

Comme il est courant lors de la mise en place de nouveaux modèles dans le domaine du *machine learning*, il est nécessaire d'implémenter une phase de test dans un environnement factice, modélisé sur la base d'hypothèses faites sur l'environnement réel, avant de passer dans un environnement de production. Un autre avantage de mettre en place un environnement de test, spécifique à notre cas, est d'éviter que nos IPs ne se fassent blacklister sur une parties des sites webs de nos bases de données, en raison des nombreux essais/erreurs qui seront faits lors de la phase de test.

La procédure de test consistera ainsi à modéliser une série d'environnements dans leur totalité, en générant des éléments factices (abstraction de sites webs, abstraction de *crawlers*…) de la manière la plus fidèle possible. Chaque modèle mis en place sera doté de caractéristiques particulières. Nous appliquerons ces environnement à plusieurs algorithmes de RL que nous choisirons selon leur caractéristiques et leur viabilité vis à vis du but du projet.

Le déploiement en environnement réel se fera sur une sélection basée sur les résultats des modèles mis en place en environnement de test. Il s'agira ainsi d'implémenter un véritable environnement, basé sur des véritables sites webs, et utilisant un véritable *crawler*, et lancer la procédure d'apprentissage face à ces derniers. Le modèle sera ensuite amélioré au fur et à mesure des tests.

Il s'agira ensuite de pouvoir étendre le déploiement de notre modèle sur plusieurs machines, à savoir, l'exécution de plusieurs bots informatiques de manière séquentielle, puis en parallèle.

Enfin, au terme de la mise en place de l'environnement en conditions réelles, nous nous focaliserons sur un aspect plus particulier de la détection des bots : certains sites webs utilisent des techniques avancées permettant d'analyser les mouvements de la souris et de déterminer si ceux-ci proviennent d'un bot ou d'un humain. Il est ainsi intéressant de pouvoir mettre en place une solution permettant de générer des séquences de mouvements de la souris reproduits par le bot qui seront classifiés comme "humains" afin d'éviter de se faire détecter par cette technique particulière. La solution se centrera ainsi sur l'utilisation du *deep learning* et des réseaux de neurones afin de parvenir à nos fins.

Ainsi, le cahier des charges du projet est le suivant :

- Modéliser plusieurs environnements factices et les exécuter sur plusieurs algorithmes, dont nous analyserons les résultats et effectuons par la suite une sélection du meilleur modèle;
- Suite à la sélection, le meilleur modèle sera implémenté en conditions réelles et sera soumis à une phase d'apprentissage;
- Le modèle sera amélioré dans la mesure du possible, puis sera étendu à plusieurs machines pour un déploiement de plusieurs bots informatiques, de manière séquentielle dans un premier temps, puis enfin, de manière parallèle. Le modèle sera soumis à une série d'algorithmes dont les résultats seront analysés.

- Enfin, une solution sera mise en place, utilisant le *deep learning*, afin de générer des séquences de mouvements de souris “humains”.

b. Le reinforcement learning

1. Vue générale du machine learning

L’*Intelligence Artificielle (IA)* est définie comme “*une série de techniques mises en oeuvre pour permettre aux machines d’imiter une forme d’intelligence réelle” (Larousse[1]).* Cette définition à le mérite de laisser à entendre que l’IA ne correspond pas à un domaine limité, mais plutôt à un ensemble de domaines.

Ainsi, le *machine learning* correspond à un sous domaine de l’IA, car il nous permet de disposer d’algorithmes permettant de résoudre des tâches qui étaient préalablement uniquement résolues par l’humain. Le *deep learning* quant à lui, est un sous-domaine du *machine learning* : l’état de l’art en terme de traitement d’images, de reconnaissance vocale, et de génération (de texte, d’image…) étant atteint en très grande majorité par des techniques propres au *deep learning,* dont la spécificité est d’utiliser des réseaux de neurones construits sur une abstraction du fonctionnement du cerveau humain.

Enfin, le *machine learning* se divise en trois catégories distinctes :

- Le *supervised learning* (apprentissage supervisé), qui consiste à utiliser des données préalablement annotées (labellisées), afin de permettre au modèle d’apprendre une fonction de prédiction, qui va permettre soit de classifier des données dans des catégories (tels que la détermination de l'espèce d’une plante à partir de ses caractéristiques), soit d'effectuer une régression (tels que la prédiction du prix d’une maison par exemple);

- L’*unsupervised learning* (apprentissage non supervisé), qui quant à lui, consiste à utiliser des données non annotées, et dont le but est majoritairement d’extraire des classes d’individus présentant des caractéristiques communes, soit, de généraliser des caractéristiques présentes dans le jeu de données;[2]

- Le *reinforcement learning* (apprentissage par renforcement)*,* que l’on notera RL par la suite, représente une classe d’algorithmes de machine learning consistant à fortifier la connaissance d’un agent de son environnement à travers un processus d’essai/erreur, permettant à l’agent de récolter des informations sur les actions à prendre dans le futur.

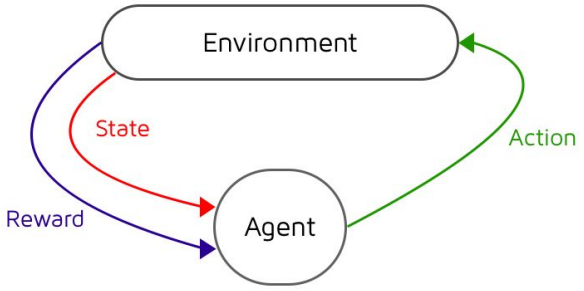
Nous nous intéressons lors de ce projet en priorité au *reinforcement learning,* ainsi qu’à une sous famille de ce dernier se nommant le *deep reinforcement learning*, qui allie l’utilisation de réseaux de neurones aux algorithmes classiques du *reinforcement learning.*

[1] https://www.larousse.fr/encyclopedie/divers/intelligence_artificielle/187257
[2] https://fr.wikipedia.org/wiki/Apprentissage_non_supervis%C3%A9

2. Reinforcement learning

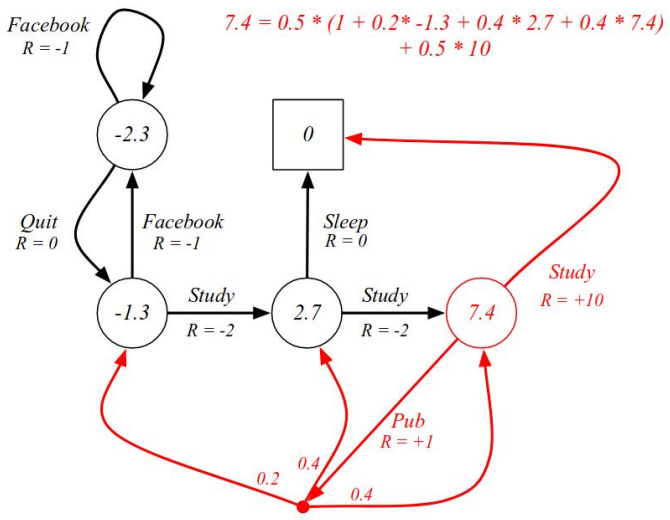
Nous pouvons rapprocher le comportement des algorithmes de RL à la phase d'apprentissage d'un humain : une analogie très pratique à la compréhension de ce domaine est l'expérimentation de plats. Supposons qu'un humain se présente pour la première fois dans un restaurant, dont il ne connaît absolument pas la cuisine. Ce dernier décide d'expérimenter plusieurs plats : après avoir goûté le premier plat, il décide que celui-ci n'est pas à son goût, et ne le reprendra plus par la suite. Cependant, le second plat est totalement à son goût et il décide alors que lors d'un retour potentiel, c'est ce plat qu'il choisira.

Le RL fonctionne de la sorte, un agent est placé dans un environnement modélisé. Cet agent dispose d'un jeu d'actions qu'il peut effectuer, agissant directement sur l'environnement en question. Si l'action provoque une conséquence positive, une récompense est donnée à l'agent, tandis que si l'action engendre une conséquence négative, un malus est distribué. L'agent aura ainsi tendance, au fil du temps, à privilégier les actions ayant engendrés des récompenses. La séquence d'états-actions privilégiée par l'agent se nomme *la politique*.



Afin de comprendre plus en détail le fonctionnement des algorithmes de RL, et leur utilité, il est nécessaire de rentrer dans les détails techniques de ces derniers : le RL est basé sur les *Processus de Markov,* les *Markov Decision Process (MDP),* qui correspondent à un modèle markovien où l'état dans lequel se trouve l'agent est probabiliste et dépend de l'action prise à l'état précédent. Les *MDP* sont une extension des *chaînes de Markov,* dont la seule différence est le nombre multiple d'actions et les différences de récompenses. En tant qu'extension des *chaînes de Markov*, les MDPs héritent ainsi de la propriété de Markov : "*l'information utile à la prédiction du futur n'est contenue qu'uniquement dans l'état actuel et ne dépend aucunement des états précédents".*[3]

[3] https://fr.wikipedia.org/wiki/Cha%C3%AEne_de_Markov



Exemple de Markov Decision Process[4]

Les algorithmes de RL sont très diversifiés et reposent sur diverses techniques. Il existe plusieurs classes d'algorithmes reposant sur des techniques différentes, le livre "*Introduction to Reinforcement Learning"* de Sutton & Barto[5] recense quatre familles d'algorithmes principales :

1. Les algorithmes basés sur le *dynamic programming* (programmation dynamique), qui correspond à une technique d'optimisation des algorithmes basés sur la récursion et dont l'idée est de simplement stocker les résultats des sous problèmes résolus par l'algorithme. Le *dynamic programming* induit la détermination d'une *value function* optimale. La *value function* correspond aux récompenses estimés en partant d'un certain état et en suivant une certaine politique, et repose sur *l'équation de Bellman.* Le désavantage de ces algorithmes est qu'ils sont basés sur un modèle parfait, dont la table de transition est connue et dont le modèle est fini, il ainsi est nécessaire d'avoir une connaissance parfaite de l'environnement dans lequel l'agent évolue, ce qui peut être un frein à l'apprentissage dans le cas d'une modélisation partielle;

2. Les algorithmes basés sur les méthodes de *Monte-Carlo,* et sont tournés vers les problèmes dont l'environnement ne peut être complètement modélisé, à savoir, vers l'environnement dont nous ne disposons pas des tables de transitions entre les états, ou dont les conséquences sur le long terme (à la suite de plusieurs actions) ne sont pas connus avec exactitude. La *value function* est ainsi apprise au fur et à mesure que l'on visite les états. Les méthodes de *Monte-Carlo* introduisent également la notion de *Q-value* qui correspond à une valeur associée à une paire état-action, analogue à la table de transition utilisée en *dynamic programming*, avec la différence qu'elle est ici entièrement apprise. Ils introduisent également les notions d'algorithmes *on-policy* et *off-policy*, qui correspondent respectivement à la prédiction de la meilleure action en suivant une politique précise, et la prédiction de la meilleure action en suivant une certaine politique, tout en estimant une politique différente;

[4] https://cdn-images-1.medium.com/max/1200/1*-fXYPGPNfdVy94l19uzKeQ.png
[5] http://www.incompleteideas.net/book/the-book-2nd.html

3. Les algorithmes basés sur le *temporal-difference,* qui regroupent les méthodes utilisées en *dynamic programming* et *Monte-Carlo.* Ces algorithmes adressent une des limitations de *Monte-Carlo*, qui exigent de visiter tous les états d'une séquence avant de pouvoir mettre à jour la *value function* ou/et les *Q-values.* Ici, nous avons besoin de regarder simplement un nombre limité d'états plus loin afin de mettre à jour les fonctions. Le *temporal-difference* introduit les algorithmes *SARSA* et *Q-Learning,* qui ont permis un grand bond en avant dans le domaine du RL;
4. Finalement, les algorithmes basés sur les méthodes du *Policy Gradient*, cassant avec les méthodes précédentes en raison du fait que nous tentons d'estimer la meilleure politique, plutôt que les Q-Values (ou V-Values).

Enfin, ces différentes classes d'algorithmes disposent eux même de plusieurs variations, tels que l'utilisation de réseaux de neurones pour approximer les Q/V-Values.

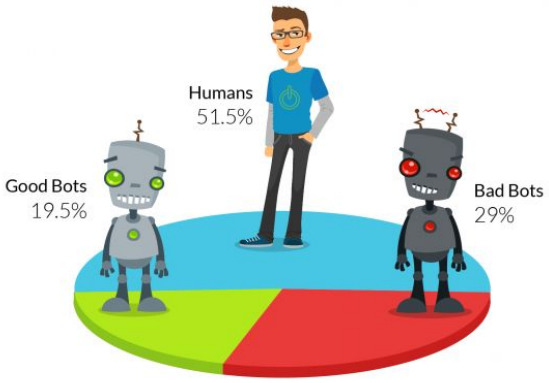
c. Les bots informatiques

Les bots informatiques correspondent à des scripts écrits dans le but de réaliser des actions prédéfinies sur des pages webs en supprimant la nécessité de l'interaction humaine. Divers types de bots existent, et leurs applications sont très variés. Des exemples de domaines où les bots informatiques sont efficaces sont :

- Le *web scraping*, consistant au parcours de pages web afin d'en récupérer des données. Ces bots sont communément appelés *scrapers* ou *crawlers.*

- Les tests automatisés : il peut être utile, lors de la réalisation d'un site web, de tester certaines fonctionnalités de celui-ci et d'analyser le résultat d'une action avant le passage en environnement de production.

Les *bots informatiques* sont très présents sur internet, bien que peu visibles : ils représentent près de 20% du trafic internet pour les bots non malicieux, et près de 30% pour les bots représentant une menace (tels que les *botnets*). Cela représente près de 50% du trafic internet total en 2015.[6]



https://www.incapsula.com/blog/bot-traffic-report-2015.html

[6] https://www.incapsula.com/blog/bot-traffic-report-2015.html

Les bots informatiques qui nous intéressent se contentent uniquement de récupérer des informations publiques, sans disposer d'un comportement "nocif" pour le site web.
Certaines personnes voient en ces bots une menace à la confidentialité[7] en raison du fait qu'ils récoltent un nombre élevé d'informations sans consentement préalable. La question se pose alors concernant la moralité de récupérer et stocker des informations ayant été avancés pour ou par un site web particulier.

Ainsi, certains gestionnaires de sites web, ou leur security providers, procèdent à un blocage de bots abusif, à savoir, ils ne permettent à aucun bot informatique de se maintenir sur leur site web, dès lors de leur détection.

Les bots informatiques peuvent être implémentés de plusieurs façons. Ils peuvent soit être écrits à partir de zéro, en utilisant les librairies de *networking* du langage utilisé, soit implémentés à travers l'utilisation de frameworks écrits dans ce but. Ainsi, il existe de nombreuses librairies permettant de mettre en place des *crawlers* très rapidement. Les plus anciennes, et donc les moins à jour et les plus facile à détecter par les sites webs se nomment PhantomJS et CasperJS. Ces librairies disposent l'avantage d'être très répandues mais leur ancienneté joue en leur défaveur car leur détection est devenue banale. Des librairies plus avancées ont vu le jour dernièrement, et répondent au nom de Selenium et Puppeteer. L'avantage de Selenium est de pouvoir lancer des véritables navigateurs et d'orchestrer leur comportement. La détection est beaucoup plus complexe en raison du fait que l'empreinte des navigateurs est similaire à celle d'un utilisateur humaine. Puppeteer, quant à lui, se contente de fonctionner avec une instance *headless* de Chromium (sans GUI - *Graphical User Interface*). Une instance headless de Chromium est très difficilement détectable en raison du fait que son empreinte est similaire à une empreinte de navigateur Chromium classique. Puppeteer à été développé et est maintenu par Google, et dispose d'un large panel de customisation et permet ainsi une implémentation très flexible de notre *crawler.* Pour ces raisons, nous choisirons *Puppeteer* et *Chrome Headless* lors de l'implémentation des environnement réels.

II. Réalisation en environnement de test

La décision de réaliser un environnement de test à été pris après discussions avec mes tuteurs de projets : de nombreux modèles plausibles avaient émergés du lot et étaient considérés comme viables. Ainsi, il était nécessaire de réaliser un environnement de test, à la fois pour pouvoir évaluer les performances de nos modèles, mais également pour éviter un blacklist des adresses IPs à notre disposition.

Il est nécessaire de passer en revue quelques termes avant de rentrer dans le vif du sujet. Un *modèle* correspond ici à un environnement complet, et inclus la définition d'un *agent*, qui choisira les actions à prendre selon son expérience, la définition des états, qui correspondent à la situation dans laquelle l'agent se retrouvera à l'instant présent. Le *modèle* induit également la définition des *récompenses* (ou malus) que l'*agent* recevra à la suite d'une action. Ces différentes parties des modèles mis en place seront décrites en détail par la suite.

Il est également à noter que l'environnement sera modélisé sur la base d'hypothèses. En effet, durant la phase de test, nous n'utilisons pas de véritable *crawler,* mais une abstraction de celui-ci : nous

[7] https://snatchbot.me/blog/64/the-challenge-of-data-protection-in-the-era-of-bots

effectuerons des hypothèses concernant son comportement, et tenterons de les reproduire. De même pour les sites web à visiter, des hypothèses seront effectuées sur la nature de ceux-ci, tels que leur tendance à bloquer les bots, leur tendance à vérifier certains attributs, où leur tendance à blacklister une IP par exemple.

Ces hypothèses seront faites à la suite de documentation et de recherche concernant les attributs principaux des *crawlers* et des sites web.

Les environnements de tests seront réalisés sous **OpenAI Gym[8],** en utilisant le langage **Python**. OpenAI Gym est un *toolkit* permettant le développement et le test d'algorithmes de *reinforcement learning*. Il a été mis en place dans le but de standardiser les environnements de test, afin de simplifier l'implémentation des algorithmes issues de papiers de recherche. L'utilisation d'OpenAI permet une intégration simplifiée dans les algorithmes de *reinforcement learning.*

a. Environnement de test

Avant de pouvoir détailler les modèles utilisés, il est important de revoir en détail les éléments qui le composeront, et que nous définirons en détail pour chaque modèle mis en place. Ainsi, nos modèles comprendront les éléments suivants :

- **L'agent** correspond à un contrôleur du/des *crawlers*. Ce choix a été préféré au fait de définir l'agent comme étant le bot lui même en raison de la flexibilité que cela nous apporte : nous pourrons ainsi contrôler un unique bot, ou l'étendre à plusieurs bots sans devoir changer le modèle tout entier;
- Les **récompenses** seront définies précisément pour chaque *modèle* particulier. Cependant, de manière générale, les récompenses seront positives en cas de non détection, tandis qu'une détection entraînera une récompense négative (pénalisation);
- Les **actions** et les **états** seront propres à chaque modèle que nous décrirons par la suite.

Le but du projet étant de faire en sorte que les *crawlers* ne se fassent pas détecter à l'aide du *reinforcement learning,* il est nécessaire de recenser les différentes raisons pouvant amener à la détection du bot. Après un travail de documentation, une majorité des raisons pouvant entraîner la détection ont été recensés. Le tableau suivant décrit ces attributs et présente des façons de contrer ces méthodes de détection[9] :

Techniques de détection	Solution(s) possible(s)
Durée de la session : elle est plus courte pour les bots en raison de leur rapidité.	Temporiser la session, soit, patienter avant de passer à un autre site web.
Nombre de requêtes réalisés : un bot tend à réaliser un nombre élevé de requêtes en raison du fait que ce dernier tend à maximiser le nombre de pages visitées, tandis qu'un humain à généralement un but prédéfini avant la visite d'un site web.	Cibler les ressources nécessaires au préalable afin d'éviter un parcours chaotique du site web en question.

[8] https://gym.openai.com/
[9] https://antoinevastel.com/bot%20detection/2018/01/17/detect-chrome-headless-v2.html

Pourcentage de requêtes sans la présence du header HTTP Referer, qui contient l'URL de la page précédente visitée. Cette technique de détection est mitigée car ce header peut ne pas être défini également pour un utilisateur humain.	Mettre en place le header HTTP Referer lors de la navigation.
Pourcentage d'images et de ressources CSS chargés par le bot. Le bot tend à minimiser le chargement des ressources inutiles.	L'algorithme devrait pouvoir déterminer lorsque le chargement d'images ou des ressources CSS est nécessaire à la non-détection.
Pourcentages d'erreurs 4xx. Le bot tend à visiter les liens présents sur la page, dont certains mènent vers des pages inexistantes. Certains sites webs mettent en place des liens cachés qui font office de piège pour les bots.	Visiter uniquement les pages webs voulues.
Pourcentage de requêtes HTTP séquentielles. Les humains tendent à disposer d'un pourcentage plus élevé en raison d'une demande moindre pour des ressources de type script, images ou document, tandis que les bots accèdent à des liens donnant directement sur ce type de ressources et cassent ainsi le parcours séquentiel.	Limiter le chargement des ressources média (PDF, images, etc)
Le fingerprinting[10], ou les empreintes de navigateurs, qui proviennent d'une activité douteuse sont sauvegardés. Si la même empreinte est détectée, l'adresse IP de provenance est bloquée.	Altérer l'empreinte pour chaque visite dans le cas d'une détection.
L'user-agent d'un utilisateur classique diffère de celui utilisé par une instance *Chrome Headless.*	Utiliser un user-agent commun puisé d'une base de donnée.
Une **utilisation intensive d'une adresse IP** peut mener au blocage de celle-ci	L'algorithme devra être capable d'alterner entre les IPs à sa disposition avec parcimonie.
L'analyse des **mouvements de la souris et des entrées claviers** : le bot aura tendance à effectuer des mouvements du curseur très direct.	Nous pourrons mettre en place une solution générant des mouvements du curseurs de façon plus humaine en observant le comportement d'un humain face à des pages webs.

Après le travail de documentation réalisé, et la revue de la théorie des méthodes utilisés en *reinforcement learning,* nous avons pu déterminer tout au long de la phase de test **trois** modèles principaux qui paraissaient être viables.

Le premier modèle est centré sur les caractéristiques des sites webs, que nous analyserons par la suite. Le second modèle est quant à lui centré sur le bot informatique, tandis que le troisième et dernier modèle mis en place met au centre de l'attention des caractéristiques du bot mais également des sites web.

[10] https://naifmehanna.com/2018-09-16-browser-fingerprints-in-a-nutshell/

b. Modèles et algorithmes

1. Modèle orienté sites web

Le premier modèle mis en place met au centre de l'environnement les sites webs. Ainsi, l'action de l'agent dépendra uniquement des attributs du site web dans lequel il sera présent à l'instant t. Nous représentons alors les sites webs par une série de *features* (caractéristiques) exploitables, qui dans ce cas, correspondent au fait que le site web bloque les bots ou non, dispose d'un *security provider*, le nombre de visite du *security provider* et du site web, et l'utilisation du fingerprinting.

Un *security provider* correspond à une entité gérant la sécurité de plusieurs sites webs, et améliorant ainsi les capacités de détection du site qu'elle protège.

Étant en environnement de test, nous ne visitons pas de véritables sites webs mais une abstraction de ceux-ci. Il en est de même pour les *security providers*. Ainsi, les sites webs sont représentés par des attributs qui sont distribué de manière probabiliste : si 10% des sites webs réels analysés lors de la phase de documentation bloquent les bots, l'attribut permettant de bloquer les *crawlers* dans la classe représentant les sites webs sera activé avec une probabilité de 0.1. Chaque site web factice est identifié par un **UUID** unique, qui fait abstraction à l'URL. Le site web factice garde également un historique des adresses IPs visités et des User-agents. Si une IP à été classifiée comme appartenant à un bot, elle sera ainsi bloquée par la suite. Le *security provider* quant à lui dispose d'attributs permettant d'évaluer sa performance, et recense la liste des UUIDs des sites webs qu'il protège. Le *bot* quant à lui est également représenté par une série d'attributs, représentant son user-agent, son adresse IP ou son proxy.

Nous associons également aux sites webs un attribut *value qui* correspond à la valeur que le site web donne à un visiteur (bot ou non) : plus cette valeur est élevée, plus le site web factice à tendance à classifier le visiteur en tant que bot.
Ainsi, le modèle orienté site webs est représenté de la façon suivante :

- Les **états** correspondent aux *features* exploitables des sites webs. Cela inclut toutes les *features* représentants un site web, en plus d'un intervalle représentant le nombre de fois que celui-ci à été visité, et une autre intervalle similaire pour le *security provider.* Ainsi des sites présentants des attributs similaires représenteront le même état. Cependant, nous pourrons également avoir des états vides, car aucun site web factice généré ne correspond aux caractéristiques que l'état-dit représente;

Fingerpriting
Bloque les bots
Security provider
Visites security prov.
Visites du site web

- Les **actions** ici sont définis par la visite d'un état particulier, le changement de l'adresse IP et le changement d'user-agent dans un premier temps;
- L'une des actions étant la visite d'un état particulier, cela entraîne la possibilité de visiter un état qui ne correspond à aucun site web généré. Ainsi, les **récompenses** sont positives dans le cas

d'un *crawl* réussi, négatives en cas de détection, et très faiblement négative en cas de visite d'un état vide.

Les algorithmes de *reinforcement learning* fonctionnent soit par épisodes, soit continuellement. Dans notre cas, nous avons un environnement discret donc un fonctionnement épisodique. Ainsi, dans le cas de notre modèle, un **épisode** correspond au *crawl* d'un nombre de site webs fixés, ou, plus communément, un épisode correspond à un nombre d'étapes, ou chaque étape correspond au *crawl* d'un site web factice. Les paramètres du modèle sont mis à jour à la fin de chaque épisode, et les états sont ré-attribués aux sites webs à chaque fin de *crawl (*le compteur du nombre de visites étant mis a jour, un site web peut ne plus représenter l'état dans lequel il était précédemment*).*

2. Modèle orienté *crawler*

À partir du modèle précédent, et à travers l'analyse des résultats (présentés lors de la partie II.c), nous avons pu en déduire que malgré des résultats concluants, nous pouvions obtenir un taux de réussite supérieur en mettant en place un nouveau modèle.

Ainsi, le second modèle centre son environnement autours du *crawler* et de sa configuration :

- Les **états** représentent la configuration du bot, soit, *l'user-agent* utilisé, le proxy/IP choisi, l'utilisation du header REFERER, l'utilisation du header HEAD, le taux de chargement d'images et le taux de chargement des ressources CSS et des documents. Le nombre d'état pour cette configuration est élevé en raison du fait que nous comptabilisons toutes les combinaisons possibles de la liste énoncée précédemment;

- Les **actions** sont représentées par le changement de l'user-agent, le changement de l'IP/Proxy, la mise en place des headers REFERER et HEAD, le passage d'un site web à un autre, et l'augmentation ou la diminution du taux de chargement d'image;

- Les **récompenses** incluent un score positif en cas de *crawl* sans détection et un malus dans le cas contraire.

Les épisodes et les sites webs factices sont toujours représentés de la même manière. Il est également à noter que dans ce modèle, le blocage d'un bot factice par un site web est basée sur la mise en place d'un score prenant en compte les caractéristiques du bot en question. Ainsi, par exemple, si le bot dispose d'un header REFERER erroné, un user-agent déjà detecté plusieurs fois, et une IP "blacklistée", le score sera élevé. Ce score est ensuite transformé en une probabilité de blocage, donc normalisé entre 0 et 1.

Une autre différence avec le modèle précédent est que les sites webs stockent toujours les adresses IPs et les user-agents les ayant visités, mais nous introduisons la notion d'oubli. Au fur et à mesure que nous avançons dans le temps, le site web oubliera ainsi les IPs qu'il a blacklisté par le passé.

3. Modèle mixte

Avant de passer à l'implémentation en environnement réel, le choix s'est porté sur la mise en place d'un modèle alliant les points forts des deux modèles précédents en raison des bons résultats de ces derniers. Cette proposition s'est révélée intéressante en raison du fait que certains attributs des modèles précédents étaient difficilement reproductible en environnement réel : nous pouvons ainsi

utiliser les attributs les plus accessibles et les plus significatifs des deux modèles précédents et les fusionner en un unique modèle.

Ainsi, pour ce modèle, la représentation est la suivante :

- Les **états** sont représentés ici par les *features* des sites webs présenté lors du premier modèle, à l'exception de la feature concernant les *security providers,* qui est éliminée, agrémentés d'intervalles du nombre d'utilisation d'un certain user-agent et de certaines IPs par un même bot. La décision de ne plus utiliser des *security providers* est motivée par le fait qu'il est très difficile de savoir quel *security provider* gère quel site web en environnement réel;

- Les **actions** consistent ici à changer d'user-agent, de proxy/IP, de passer d'un site web à un autre, ou de ne rien faire, ce qui consistera à garder la configuration actuelle sur le même site.

La définition d'un épisode ne change pas, et les récompenses restent les même que pour le second modèle. Il est également important de noter que les trois modèles sont simplifiés afin d'éviter de complexifier l'environnement de test.

4. Algorithmes implémentés

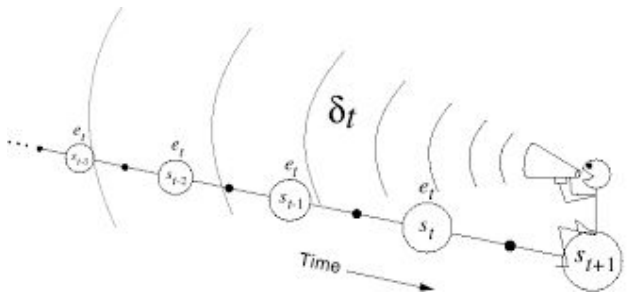
Afin de pouvoir évaluer les performances de chaque modèle, il est nécessaire d'implémenter les algorithmes de *reinforcement learning* que nous considérons les plus adaptés à la tâche.
Les algorithmes que nous choisissons d'implémenter correspondent à la classe des algorithmes utilisés en *temporal difference learning*, qui ont été décrit lors de la partie I.b.
Nous choisissons dans un premier temps de partir sur des algorithmes classiques, à savoir :

- Le **Q-Learning**[11]**,** qui correspond à un algorithme off-policy, qui va chercher à optimiser deux politiques d'action différentes et à suivre la meilleure des deux. Le Q-Learning cherche à optimiser la Q-Value, ce qui le rend plus intéressant dans notre cas, car nous souhaitons évaluer les conséquences du choix d'une action en fonction de l'état présent;

- L'algorithme **SARSA**[12], qui est très similaire au Q-Learning à l'exception qu'il s'agit d'un algorithme on-policy, qui va chercher à optimiser une unique politique d'action. Le choix d'utiliser SARSA à été pris dans le but de confronter deux algorithmes on-policy et off-policy;

- L'algorithme **SARSA(λ)**[13]**,** qui correspond à une version améliorée de l'algorithme **SARSA**. L'amélioration réside dans le fait que **SARSA(λ)** utilise les *eligibility traces*, qui permettent aux actions prises à un instant t d'influer sur le choix d'une action dans un instant futur t+n :

[11] http://researchers.lille.inria.fr/~munos/master-mva/lecture02.pdf
[12] http://laid.delanover.com/reinforcement-learning-sarsa-algorithm-a-practical-case/
[13] http://incompleteideas.net/book/ebook/node72.html



Fonctionnement des eligibility traces

- **N-step SARSA[14],** qui correspond encore à une variation de l'algorithme **SARSA**, mais qui dispose de l'avantage de mettre à jour les hyperparamètres (paramètres définissant le comportement de l'algorithme) après chaque N-étapes, au lieu d'attendre la fin de l'épisode. Cet algorithme est intéressant dans notre cas en raison du fait qu'il peut être pratique que l'algorithme s'adapte plus rapidement à son environnement;

- L'algorithme **A2C[15] (Advantage Actor-Critic),** correspondant à la catégorie des algorithmes d'**Actor-critic**, qui utilise le plus souvent des réseaux de neurones. Il a été dérivé de l'algorithme A3C, que nous verrons lors de la partie III.b. L'A2C dispose généralement de deux réseaux de neurones, l'acteur, qui permet de prédire une politique à suivre et prend en entrée l'état, et la critique, qui permet de prédire la valeur de l'état., et cherche à optimiser la politique de l'algorithme. Ces réseaux sont entraînés au fur et à mesure que l'agent interagit avec son environnement. Cet algorithme sera revu avec plus de détail par la suite.

Ces algorithmes ont tous été écrits en utilisant le langage Python, et les librairies adaptées au machine learning, tel que *Pandas ou Numpy,* qui est une librairie de calcul matriciel.

c. Analyse des résultats

Une fois les algorithmes mis en place, nous pouvons évaluer les performances des différents modèles et ainsi sélectionner celui qu'on implémente en environnement réel.

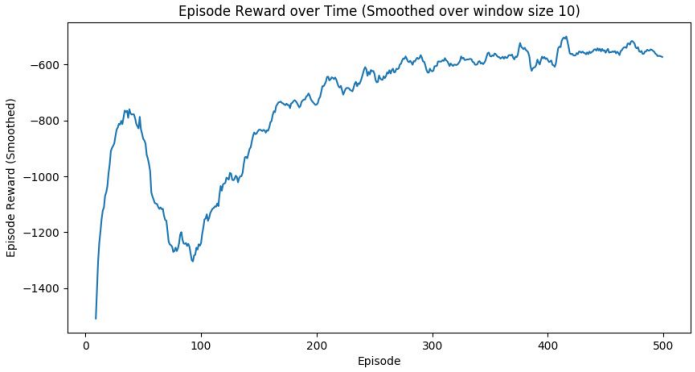
Afin d'estimer les performances sur chaque algorithme, nous sauvegarder les récompenses accumulés sur chaque épisode, que nous traçons lors de la fin de l'exécution de l'algorithme : plus les retours accumulés sont élevés, mieux l'algorithme performe sur notre modèle.

À noter également que les algorithmes tournent **généralement** dans les même conditions, cependant, il est parfois utile de modifier les valeurs des récompenses/malus, ou le nombre d'étapes par épisode (1000 par défaut).

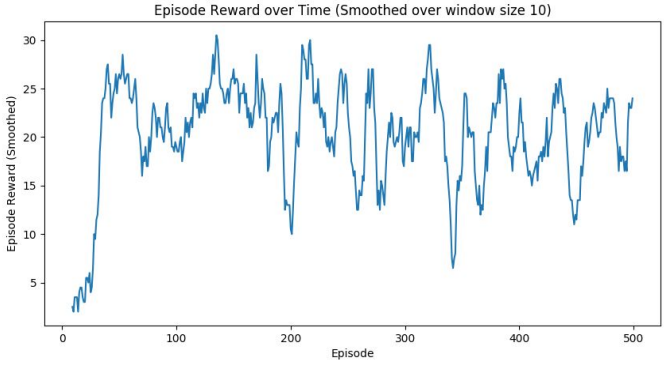
La caractéristique du graphique important le plus étant l'évolution de la courbe. Nous cherchons à maximiser les récompenses au fil du temps, la courbe doit ainsi progresser verticalement.
Les tracés représentent ainsi les récompenses cumulés lors d'un épisode sur l'ordonnée, et l'épisode en question en abscisse.

[14] https://michaeloneill.github.io/RL-tutorial.html
[15] https://danieltakeshi.github.io/2018/06/28/a2c-a3c/

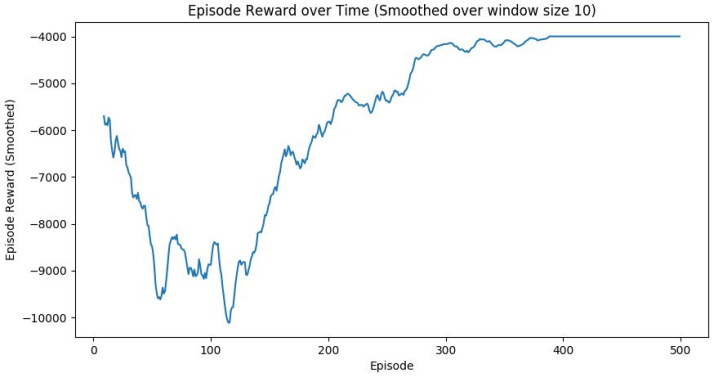
1. Résultats du modèle orienté sites web



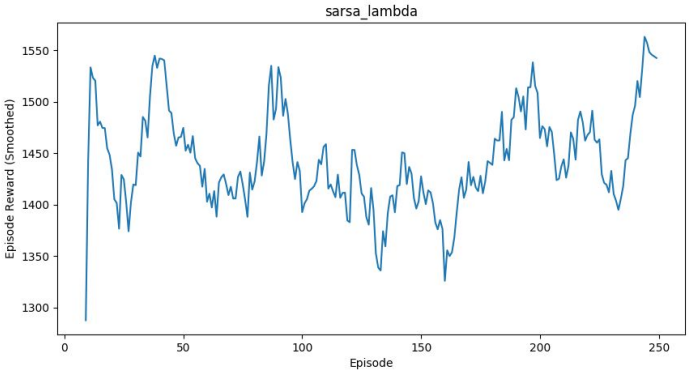
Algorithme SARSA



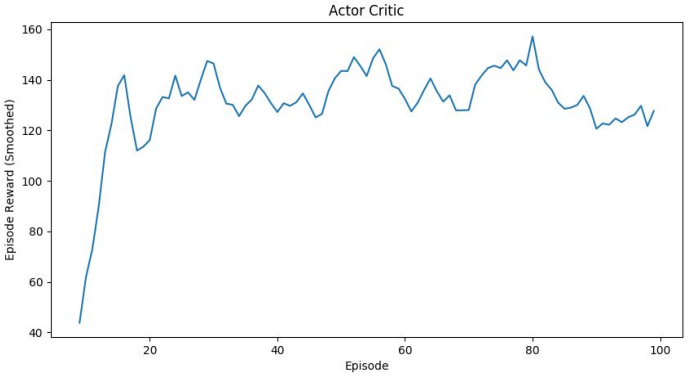
Algorithme Q-Learning



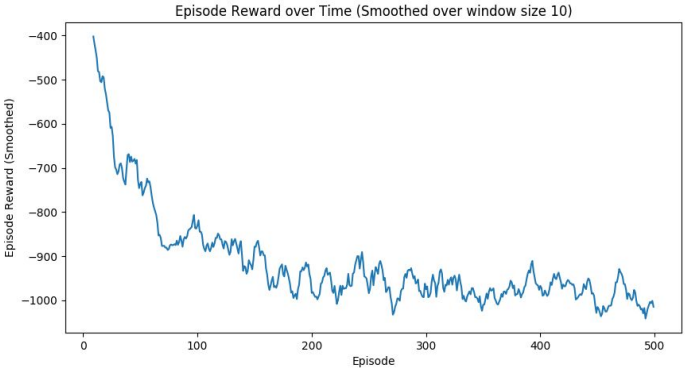
Algorithme n-step SARSA (8000 étapes/épisodes)



SARSA(λ)



A2C



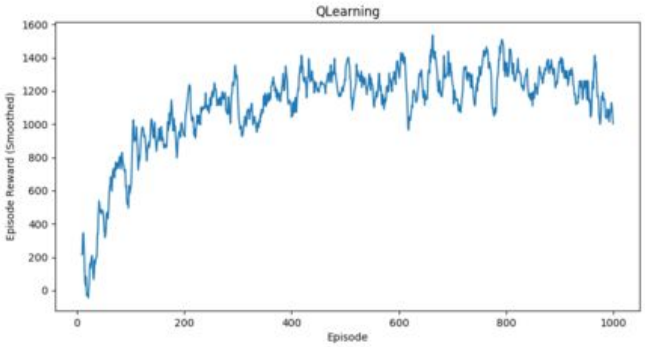
Au hasard pour comparaison

Nous observons ainsi ici une nette amélioration des performances du modèle au fur et à mesure que l'algorithme progresse dans le temps. En comparant aux choix d'actions entièrement au hasard, nous observons clairement que le choix d'actions par les algorithmes se fait selon une politique qui tend à

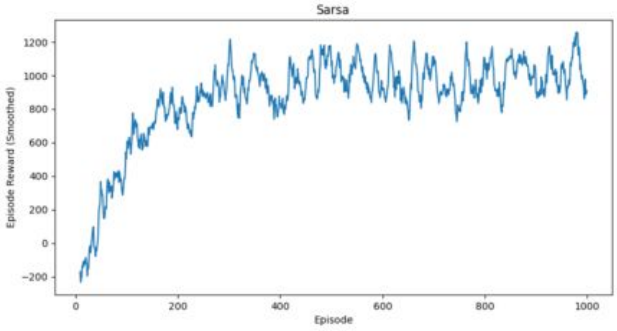
maximiser les récompenses. Trois algorithmes se démarquent cependant : SARSA, N-Step SARSA et l'A2C.

2. Résultats du modèle orienté *crawlers*

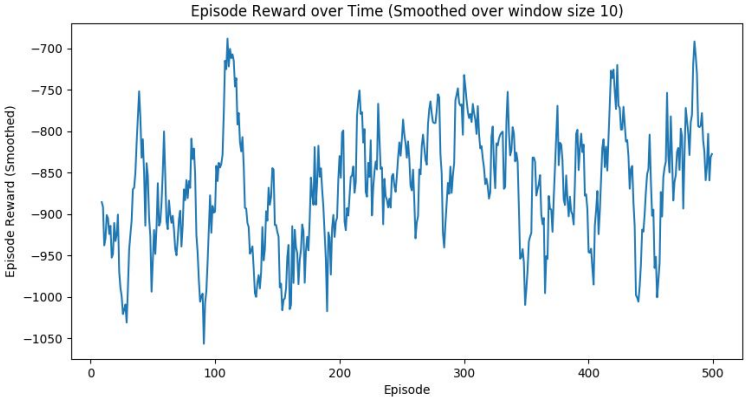
Nous procédons de la même manière pour évaluer les performances de ce modèle. Cependant, les seuls algorithmes que nous retenons ici sont les algorithmes **SARSA** et **Q-Learning** en raison des performances erratiques des autres algorithmes.



Q-Learning



SARSA



Random

Nous pouvons ici observer que les deux algorithmes SARSA et Q-Learning disposent de résultats très similaires, cependant, les performances du Q-Learning se démarquent légèrement de ceux de l'algorithme SARSA en raison d'un retour moyen plus élevé.

Les résultats de ce modèle ne sont cependant pas très positifs, en raison du fait que seulement deux algorithmes sont parvenus à réaliser des performances acceptables mais tournent cependant autours des 60% de réussite seulement.

3. Résultats du modèle mixte

Les résultats du modèle mixte ne sont pas présentable en raison de performances très mitigés. Cependant, ce modèle est celui qui se rapproche le plus à ce que l'on pourrait adapter en environnement réel : nous pensons ainsi que les mauvais résultats de ce modèle proviennent du fait que l'on est dans une modélisation.

Ainsi, il a été décidé d'améliorer ce dernier modèle et de l'implémenter en conditions réelles afin d'évaluer ses performances. Le modèle amélioré et ses résultats seront ainsi présentés par la suite.

III. Réalisation en environnement réel

Le passage en environnement réel nécessitait un travail plus conséquent, notamment en raison du fait que nous n'avions plus aucune hypothèse comme pour l'environnement de test : tous les éléments étaient implémentés sur toute la ligne.

Comme annoncé lors de la partie I, le *crawler* à été implémenté avec *Puppeteer* et *Chrome Headless*. L'implémentation d'un *crawler* induit la manipulation du navigateur, et dans notre cas, la manipulation de son empreinte et l'implémentation des techniques d'évasion, que nous verrons par la suite. Le langage utilisé par le framework *Puppeteer* est le Javascript, sous NodeJs. Cela motive ainsi le choix d'utiliser le Javascript tout au long de la mise en place de l'environnement réel. Cependant, ce choix ne vient pas sans limitation : NodeJs ne permet pas le multithreading et privilégie surtout l'asynchronisme, ce qui sera à prendre en compte.

Enfin, de nombreuses librairies utilisés sous Python, tels que Numpy pour le calcul matriciel, ne sont pas présentes sous NodeJs, ce qui m'a poussé à développer mes propres outils lorsque le besoin se présentait. NodeJS présente également l'avantage d'être significativement plus rapide que le langage Python (hors librairies)[16].

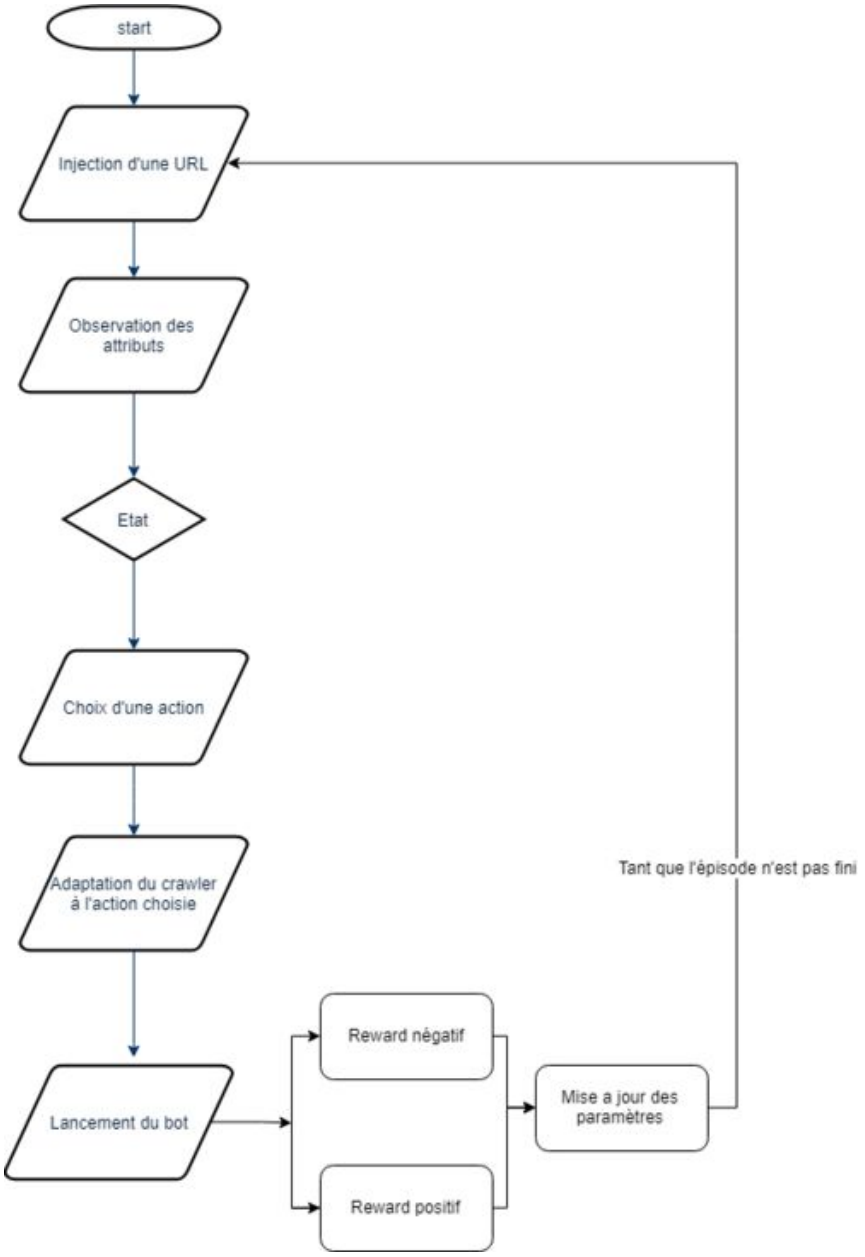
Le programme réalisé est organisé en modules et tente d'être le plus modulaire possible afin de permettre l'ajout et la suppression de fonctionnalité en toute facilité. Un utilitaire de *logging* et de tests unitaires est également mis en place afin de pouvoir traquer les erreurs.

a. Modèle et implémentation

1. Environnement et modèle

Le modèle choisi repose sur le modèle testé lors de la troisième phase de test, et est grandement amélioré et adapté à une production en environnement réel. Dans un premier temps, le programme est réalisé pour uniquement un *crawler*, qui s'exécute sur la même machine. Cette configuration n'est pas idéale, car dans un cas d'utilisation en production, il serait souhaitable d'accélérer la cadence, et de lancer plusieurs *crawlers* en même temps. Cependant, en raison de la complexité que présente la communication entre différents crawlers s'exécutant parallèlement et partageant les résultats de leur expérience fait que l'on choisit une approche séquentielle avant de mettre en place un parallélisme.
Le graphique suivant représente le fonctionnement du programme principal :

[16] https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/node-python3.html



Avant de pouvoir décrire en détail ce graphique, il convient de présenter le modèle et ses caractéristiques :

Etats	Représentés par les attributs observables des sites webs : ● L'utilisation du fingerprinting ● La vérification des conflits de taille de l'écran (taille réelle vs taille retournée) ● Les conflits dans la liste de plugins disponibles (aucun plugins pour une instance *Headless*) ● Le nombre de visites ● Le nombre d'utilisation d'une IP/Proxy ● Le nombre d'utilisation d'un user-agent ● La vérification des proxies/IPs utilisés et des user-agents ● La vérification du mouvement de la souris et des touches du clavier[17]

[17] http://cs229.stanford.edu/proj2009/Winslow.pdf

	● Le blocage des bots ● La présence du webdriver (résultat erroné dans le cas d'un bot)
Actions	● Changer l'user-agent ● Changer de proxy/IP ● Ne rien faire (garder la configuration actuelle) ● Charger les ressources (images, documents) ● Charger le contenu CSS ● Altérer la taille de la fenêtre retournée ● Altérer la liste des plugins ● Altérer le résultat de la vérification du webdriver
Récompenses	● Positive dans le cas d'un *crawl* réussi ● Négative dans le cas d'une détection
Agent	L'agent sera représenté par un contrôleur qui communiquera aux *crawlers* la configuration que ces derniers doivent adopter face au site web à visiter.
Episode	Un épisode est ici décrit comme le *crawl* de toutes les pages de la base de donnée appartenant à un même domaine.

Le modèle étant défini, nous pouvons ainsi décrire en détail le fonctionnement du programme principal. Les sites webs à visiter sont ainsi listés avant le lancement de l'algorithme de *reinforcement learning* et leur attributs sont déterminés à l'aide d'une phase de *pré-processing* basé sur le travail d'Antoine, et qui permet de vérifier quels attributs du bot ont été accédés par le site web visité : le programme réécrit ainsi toutes les fonctions appelées par les sites webs pour effectuer leur vérification en ajoutant des lignes de code permettant de notifier lorsqu'un attribut particulier à été accédé. Par exemple, un site web désirant vérifier la liste des plugins d'un visiteur appellera la fonction `show_plugins()`, le script de pré-processing altère le fonctionnement de cette fonction et stocke ainsi le nombre de fois que cette fonction à été appelée.

Afin d'accélérer l'écriture du programme, Antoine m'a fourni une base de donnée sous MongoDB contenant près de 10.000 sites webs et leurs attributs, ce qui était largement assez pour la procédure d'apprentissage. Il suffisait ainsi de charger les éléments de la base de donnée, de filtrer les attributs utiles, puis de stocker les informations constituant le site web dans un fichier JSON que le programme principal utilisera pour construire les états. En plus des attributs précédents, un *sitemap* de chaque site web est généré et constitue l'ensemble des liens (ou l'ensemble des étapes) à visiter lors de chaque épisode (correspond à la visite de l'ensemble des liens du même domaine).

Une fois la phase de pré-processing réalisée, et les états constitués, nous pouvons lancer notre environnement. L'ordre des sites webs à visiter est ainsi établi par le programme et les épisodes sont définis en regroupant les sites webs de même domaine. L'algorithme choisit une action pour le *crawler,* qui s'empresse de s'adapter. Les actions définies plus haut peuvent être combinées : nous pouvons ainsi décider de changer le proxy/IP **ET** charger le contenu CSS par exemple. Le nombre d'action augmente ainsi de 9, à près de 2047 actions au total, augmentant la dimensionnalité et la complexité du programme.

L'agent construit dans un premier temps un objet décrivant les caractéristiques du *crawler*. Cet objet est ensuite traité, et génère le *crawler* dans la configuration voulue. Le choix d'utiliser un agent

contrôlant le *crawler* est encore une fois motivé par la volonté d'étendre l'environnement à plusieurs *crawlers*.

Une fois le bot lancé et ayant effectué son action, il est nécessaire de déterminer si celui-ci à été bloqué ou si son action à été réalisée avec succès. Pour cela, j'ai dû observer le résultat de plusieurs cas de blocage afin de savoir comment déterminer le résultat d'un *crawl*. A ce jour, il n'existe pas de méthode fiable à 100% permettant de savoir si le bot a été bloqué ou pas. Dans mon cas, j'ai pu extraire plusieurs données des cas où le *crawler* ne parvenait pas à accomplir son but :

- Le renvoi de codes HTTP 4xx ou 5xx, mais qui restent insuffisant à eux seuls, car ils peuvent réellement représenter une ressource non disponible ou une erreur de serveur;
- La récurrence du mot "Captcha" sur la page;
- La taille du *screenshot* : en général, elle tend à être inférieure à 100 Ko dans les cas de blocages, en raison du très peu de ressources chargés;
- Certains sites webs utilisent CloudFare pour éviter les attaques DDoS. CloudFare fait également office de détecteur de bots informatiques : ainsi, la présence de phrases récurrentes utilisés par les scripts CloudFare est un bon indicateur de détection.

La combinaison des méthodes précédentes permet de détecter un blocage dans la majorité des cas, mais n'exclut cependant pas la possibilité d'un faux positif, ou d'un faux négatif.

Finalement, les récompenses sont distribués de manière assez classique avec une récompense positive dans le cas d'un *crawl* réussi et une récompense négative dans le cas d'un blocage. Une particularité à cependant été mise en place : afin d'éviter une utilisation de toutes les actions élémentaires à tous les coups, j'induis un malus sur la récompense dans le cas où trop d'actions auraient été selectionnés. J'induis également un malus si des actions inutiles ont été prises, telles que "Charger les ressources CSS" et "Ne pas charger les ressources CSS", qui au final, ne change rien à la configuration du bot. Nous verrons par la suite que les récompenses seront tenus d'évoluer.

L'algorithme met ensuite à jour les paramètres du modèle à chaque fin d'épisode. Afin d'éviter de reprendre de zéro à chaque session d'entraînement, la table des Q-Values est systématiquement sérialisée dans un fichier JSON à chaque fin d'épisode et, si l'option est activée, est rechargée lors de la reprise de l'algorithme dans le futur.

La gestion des erreurs à été vue en profondeur en raison du fait que cet aspect devait être très judicieusement abordé dans le cas de Puppeteer : les erreurs sont fréquentes, notamment en raison d'un crash du navigateur dû à des raisons internes, à une mauvaise gestion de certains éléments d'une page web qui la font charger indéfiniment, où encore, aux processus zombies que Puppeteer laisse à chaque exécution. Chaque aspect à du être abordé individuellement : les erreurs internes au navigateur, ne s'exécutant pas sur la même pile que le programme principal devraient être redirigés vers ce dernier et capturées. Ainsi, un crash du navigateur devrait simplement entraîner l'interruption de l'étape actuelle de l'algorithme et le passage à l'étape suivante (une récompense de 0 est renvoyée dans ce cas).

Un *timeout* de 50 secondes est mis en place dans le cas où le chargement de la page tourne en boucle. A l'issue des 50 secondes, l'étape est considérée comme abandonnée. Enfin, les instances zombies ont été limitée en détruisant correctement les instances *Chrome Headless* lancés.

Les algorithmes ayant été utilisés pour évaluer le modèle sont similaires à ceux utilisés en phase de test à l'exception du retrait de l'algorithme Sarsa(λ).

2. Cas particulier de gestion du mouvement de souris

L'analyse des techniques de détections de plusieurs sites webs à permis de mettre en avant une méthode de détection basée sur le mouvement du curseur[18] sur une page web. Nous avons ainsi décidé de nous focaliser sur cet aspect en fin de projet, qui constitue un sujet assez différent du sujet principal.
L'idée est ainsi de mettre en place une solution entièrement autonome de génération de mouvements du curseur de façon "humaine", afin de que les sites webs ne puissent détecter un *crawler* à travers cette méthode. La solution mise en place passe par l'utilisation du *deep learning*, qui dispose d'outils adaptés pour notre problème.

De nombreuses solutions permettent la génération d'une série de donnée, tels que les Generative Adversarial Networks (GANs) qui restent cependant très complexes à implémenter.
Notre problème nécessitant l'introduction d'une dimension temporelle, nous avions besoin d'une solution qui prennent en compte les liens temporels lors de la génération : les *Recurrent Neural Networks* sont ainsi spécialisés dans le traitement de tâches liés dans le temps.

Leur fonctionnement est le suivant : une séquence de vecteurs (représentant les données) est introduite dans le réseau de neurones et une séquence est récupérée en sortie. La différence des RNNs par rapport aux réseaux de neurones classiques est le fait que la sortie n'est pas uniquement par l'entrée que nous venons d'introduire, mais également par les entrées que nous avons introduits pas le passé : nous introduisons une notion de mémoire dans le réseau, qui va influer sur les séquences de sortie. Les RNNs classiques ne performent cependant pas très bien face aux longues séquences fortement liés dans le temps. On retrouve alors dans l'état de l'art une variante plus puissante du RNN : le LSTM *(Long Short Term Memory)* répond aux limitations du RNN. Notre choix se porte ainsi sur la mise en place d'un LSTM.

Le LSTM à ainsi été implémenté à la fois à l'aide du framework Tensorflow, qui malgré sa complexité, permet une plus grande flexibilité et une plus grande main-mise sur l'implémentation. Cependant, afin de privilégier la rapidité et éviter un débuggage conséquent, le LSTM a également été implémenté sous Keras, qui permet de construire l'architecture du réseau en quelques lignes. L'architecture choisie est ainsi la suivante :

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 10, 256)           265216
_________________________________________________________________
lstm_2 (LSTM)                (None, 10, 256)           525312
_________________________________________________________________
time_distributed_1 (TimeDist (None, 10, 2)             514
=================================================================
Total params: 791,042
Trainable params: 791,042
Non-trainable params: 0
_________________________________________________________________
```

[18] http://cs229.stanford.edu/proj2009/Winslow.pdf - Bot Detection via Mouse Mapping, E.Winslow

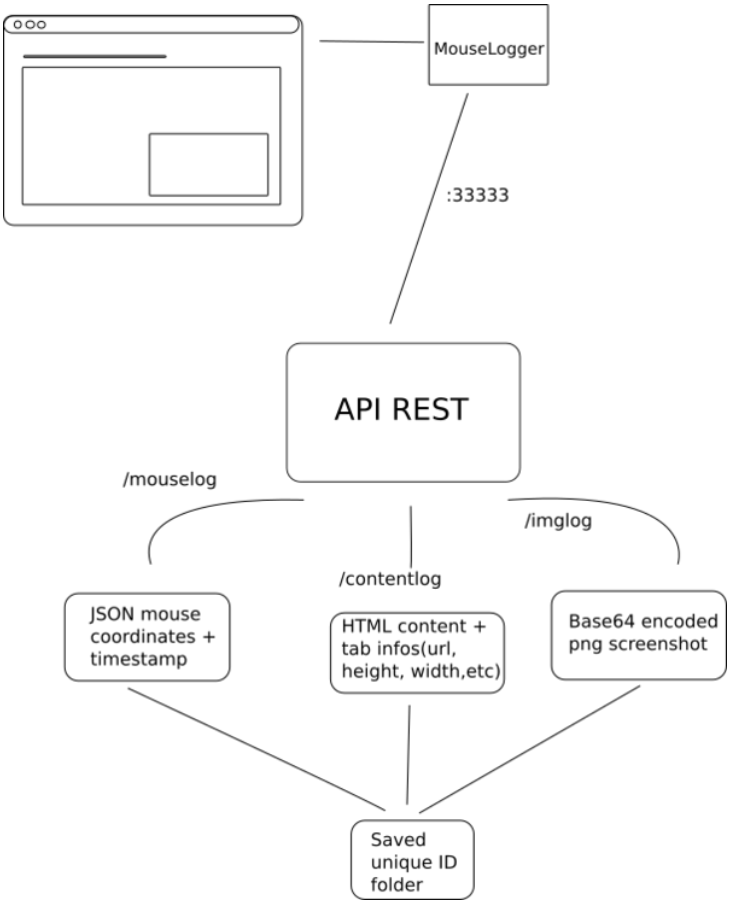
Nous avons ainsi deux couches de LSTM, composées chacunes de 256 cellules et acceptant des séquences de 10 éléments temporels à deux dimensions. Nous cherchons en effet à obtenir des coordonnées en sortie, et voulons ainsi obtenir l'abscisse et l'ordonnée, ce qui correspond à deux dimensions.

Après que le choix de l'architecture fût réalisé, nous devions nous pencher sur la récupération des données d'entraînement et le pré-processing de ces données avant l'introduction dans le réseau de neurones.

Afin de récupérer les données d'entraînement, nous devions enregistrer nos habitudes d'utilisation de la souris lors de l'interaction avec une page web. Une extension web à ainsi été écrite dans ce but, et permet de :

- Récupérer des cycles de souris, où un cycle est défini du moment où à clic à eu lieu jusqu'au prochain clic,
- Récupérer les tailles véritables et relatives de la page, avec la taille véritable incluant l'ensemble de la page, *scroll* inclus, tandis que la page relative renvoi plutôt la taille de la fenêtre,
- Récupérer le contenu HTML de la page afin de situer où le clic à été fait et enregistrer un screenshot de la page.

Le fonctionnement général de l'extension est représenté par le graphique suivant :



Programme de récolte de données

Les coordonnées récupérées devront ensuite être normalisés entre 0 et 1, représentant un pourcentage par rapport à la taille de la page avant d'être introduits dans le réseau. Par exemple, si un clic à été effectué aux coordonnées (10, 30) et que la taille de la page est de (1000, 1000), la normalisation renverra (0.01, 0.03). La normalisation est nécessaire à la convergence de la phase d'entraînement [19].

L'option de générer des coordonnées "humaines" à traver un réseau de neurones pourra ainsi être ajoutée à la liste des actions possibles du modèle.

b. Extension à plusieurs machines

1. Méthode séquentielle

L'extension à plusieurs machines a dans un premier temps été décidée en raison du fait que l'action permettant de changer l'IP du *crawler* était implémentée au départ en utilisant une listes de proxies. Cependant, nous avions la limitation que les proxies récupérés provenaient de sources gratuites et étaient souvent, soit de mauvaise qualitée, soit ne fonctionnaient pas, provoquant un timeout du navigateur *Headless*. Ainsi, il a été décidé d'utiliser plusieurs serveurs sur des machines différentes, et ainsi, avec des IPs différentes pour lancer les crawlers.

La mise en place de cette solution a été réalisée en mettant en place une liaison *WebSocket* entre les machines virtuelles à notre disposition et l'algorithme principal. Ainsi, dans le cas où nous devons changer d'adresse IP, une requête est envoyée par l'algorithme principale au serveur WebSocket s'exécutant sur la VM dont l'IP à été choisie, contenant la configuration du *crawler* qui à été décidée par l'algorithme, sous format JSON. Ce dernier exécute son action, récolte les différentes données permettant d'établir si un blocage à eu lieu ou non, et renvoi le tout à l'algorithme principal, qui continue son exécution de la même manière que si le *crawler* s'exécutent sur la même machine.

Le choix de travailler avec des WebSockets à été pris en raison du fait que la liaison établie est maintenue jusqu'à fermeture par le client ou le serveur.


```
{
"useragent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36",
"url":"naifmehanna.com",
"loadpictures":false,
"runcss":true,
"plugins":true,
"webdriver":true
}
{"fileSize":173163,"captchaOccurence":0,"cloudflareOccurence":4,"pleaseAllowOccurence":0,"responseCode":200}
```


Exemple de communication client-serveur (orange: configuration du crawler, noir: résultat du crawl)

2. Méthode parallèle

Finalement, l'extension à plusieurs machines de manière séquentielle n'apporte aucun gain de vitesse, car nous exécutons de toute manière un unique *crawler* à la fois. Il serait intéressant de pouvoir exécuter plusieurs crawlers, afin d'obtenir un apprentissage bien plus rapide, et pouvoir ainsi explorer la majorité de l'espace d'états. L'idée d'adapter l'environnement à plusieurs *crawlers* en

[19] https://www.quora.com/Why-do-we-normalize-the-data

parallèle à dans un premier temps été écartée, en raison du peu de machines virtuelles à notre disposition. J'ai cependant tenu à la réaliser et tenter l'exécution malgré la contrainte.

À noter que l'environnement utilisé reste le même que celui mis en place précédemment, de manière séquentielle, la différence réside ici dans l'algorithme de *reinforcement learning* gérant l'environnement, qui va se charger de les paralléliser, comme nous le verrons par la suite.

L'obstacle que présente la parallélisation des *crawlers* est notamment la communication constante entre ces derniers afin de pouvoir échanger leur expériences et en bénéficier sur le moment.

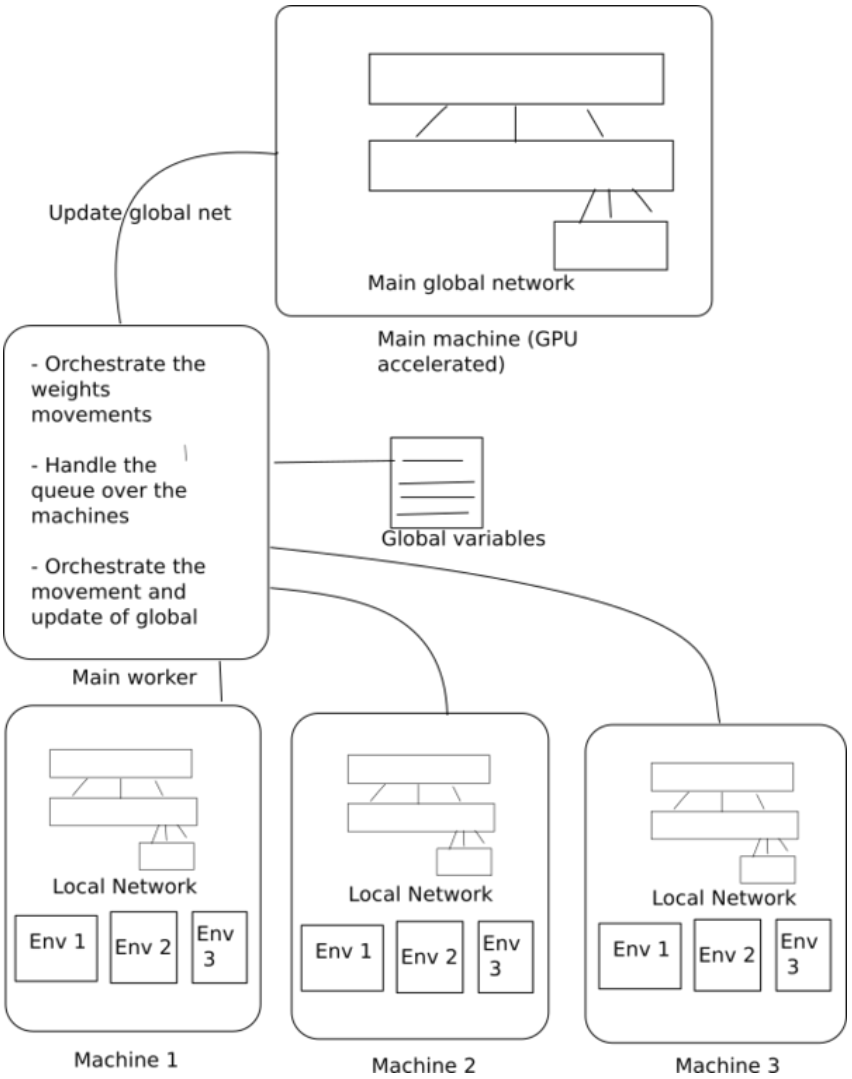
L'algorithme A3C (*Asynchronous Advantage Actor Critic)*[20], appartenant à la classe d'algorithmes *Actor-Critic* en deep reinforcement learning, a été publié par les équipes de Google's DeepMind en 2016. L'A3C se démarquait des algorithmes de l'état de l'art du moment en deep reinforcement learning, tels que les DQN (*Deep Q-Network)*[21], en raison de sa rapidité, de sa simplicité et de sa robustesse. L'A3C définissait alors l'état de l'art en deep reinforcement learning en écrasant les précédents benchmarks. L'algorithme à pu s'imposer comme l'algorithme utilisé *de-facto* en deep reinforcement learning.

L'une des nouveautés qu'introduit l'A3C, comme l'indique son nom, est son asynchronisme : A3C utilise de multiples incarnations des réseaux de neurones représentant l'acteur et la critique afin d'apprendre plus efficacement. L'A3C introduit ainsi un réseau de neurone global (on merge l'acteur et la critique pour plus de simplicité de description) et plusieurs *workers* (travailleurs), disposant d'une copie du réseau de neurones principal. Les travailleurs agissent avec leur propre copie du réseau à tout moment et gèrent plusieurs copies de l'environnement. L'asynchronisme provient du fait que chaque travailleur s'exécute en parallèle.

Le terme *Advantage* fait référence quant à lui à la règle de mise à jour des récompenses cumulés au fil d'un épisode, permettant d'indiquer à l'agent quelles actions étaient bonnes et lesquelles étaient mauvaises. Cela nous permet de déterminer non seulement si l'action choisie étaient bonne, mais aussi de combien elle était meilleure que ce que l'on attendait.

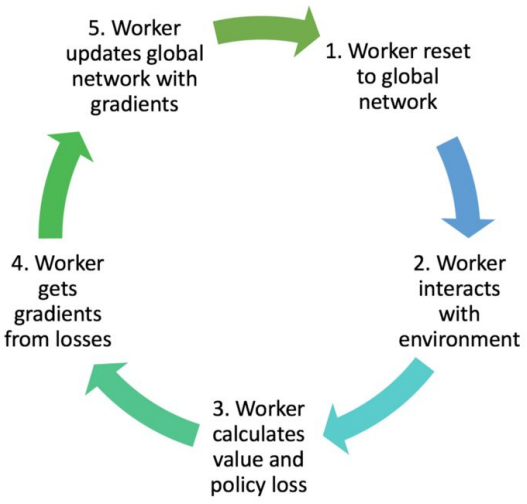
Finalement, le terme *Actor-Critic* a été décrit lors de la partie I.b. Une représentation graphique de l'organisation de l'A3C est donnée par la suite :

[20] https://arxiv.org/pdf/1602.01783.pdf
[21] https://arxiv.org/abs/1812.02632



Dans notre cas, nous avons décidés de prendre une approche différente de celle spécifiée dans le papier original, qui utilise le *multi-threading* afin de mettre en place le parallélisme des travailleurs : notre approche utilise ainsi plusieurs machines différentes au lieu de plusieurs threads. Afin d'améliorer encore plus les performances de notre approche face à l'A3C, nous gardons la possibilité d'exécuter plusieurs travailleurs sur une unique machine. Ainsi, nous avons un parallélisme introduit par des machines entièrement indépendantes, mais également un parallélisme au sein d'une même machine permettant de gérer un nombre plus élevé de travailleurs.

La phase d'entraînement de l'algorithme se déroule de la façon suivante :



Phase d'entraînement pour chaque travailleur en A3C[22]

La particularité de notre approche est ainsi de mettre en place la communication des différents travailleurs avec le réseau de neurone principal, et de maintenir la communication entre ces derniers. En effet, l'algorithme original préconise de maintenir des variables globales partagées entre les *threads*, et représentant le nombre d'épisodes cumulés par les travailleurs, le meilleur score obtenu par tous les travailleurs au niveau où ils en sont, ainsi qu'une queue (FIFO) que chaque travailleur mettra à jour au fil et à mesure de son exécution avec les récompenses cumulées au fil d'un épisode. Cette queue sera consommée par le processus gérant le réseau principal, afin de tracer l'évolution des récompenses cumulés au fil du temps.

Ainsi, ne pouvant pas mettre en place des variables globales car notre approche s'exécutant sur plusieurs machines, ces variables sont remplacées par des fichiers qui sont mis à jour à travers des requêtes HTTP bloquantes afin d'éviter des conflits d'écriture. La queue est également représentée par un fichier, qui, si il est vide, bloque jusqu'à l'insertion d'une nouvelle valeur ou l'envoi d'un signal d'arrêt.

Enfin, les paramètres des réseaux de neurones sont transmis au réseau de neurone principal à travers un transfert de fichier binaire en HTTP.

Les réseaux de neurone sont quant à eux implémentés en utilisant le framework Tensorflow.Js[23], qui diffère de son homologue Tensorflow (sous Python, C++) notamment par sa simplification et sa capacité à s'exécuter directement dans le navigateur en utilisant le moteur graphique WebGL.
Sous nodeJS, TensorflowJs utilise directement le noyau de Tensorflow et utilise le GPU de la machine si il est présent pour réaliser les calculs matriciels.

Le papier original préconise la mise en place d'un unique réseau de neurones composé de deux sorties représentant l'acteur et le critique. Dans notre approche, nous décidons de mettre en place deux réseaux de neurones différents pour l'acteur et le critique afin de simplifier le calcul des pertes. L'une des difficultés majeure de l'implémentation du réseau de neurone à résidé dans la mise en place de la fonction de calcul des pertes du modèle. Le calcul des pertes permet par la suite de déterminer les

[22] https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2
[23] https://js.tensorflow.org/

dérivées partielles (par rapport aux poids du réseau) qui vont servir à modifier les hyperparamètre de ce dernier. En effet, le calcul des dérivées partielles est aisé sous Tensorflow (API Python), en raison de son approche qui est *graph-oriented* : les opérations sont construites dans un premier temps en un *graph*, et les valeurs sont introduites par la suite. TensorflowJs fonctionne différement, de façon séquentielle : toutes les opérations s'exécutent au moment où elles ont été déclarés, ce qui rend plus complexe de retrouver les variables ayant contribués au calcul des pertes.

Ainsi, le calcul des dérivées partielles est plus complexe en raison du fait que nous ne gardons pas en mémoire les variables qui agissent sur la fonction de calcul des pertes. Après des sessions de documentation, je suis parvenu à mettre en place une alternative au calcul de la perte de façon classique, en analysant les formules mathématiques utilisés dans le papier original : l'acteur (*policy loss)* utilise comme fonction le *softmax cross-entropy (entropie croisée),* tandis que le critique (*value loss)* utilise le *mean squared error* (*moyenne de l'erreur au carré)*.

$$Value\ Loss: L = \Sigma(R - V(s))^2$$

$$Policy\ Loss: L = -log(\pi(s)) * A(s)$$

V(s) correspond ici à la sortie du critique, tandis que R correspond aux récompenses décomptées au fil du temps lors de l'épisode (en les multipliant par un paramètre nommé le *discount factor*). A(s) correspond à l'*advantage*, qui à été expliqué plus tôt tandis que π(s) correspond à la sortie de l'acteur.

Une fois ces fonctions implémentés, nous avons ainsi pu tester et résoudre les derniers obstacles qui résidaient en grande majorité dans la communication entre les différents travailleurs. Nous obtenons au final une nouvelle approche face à un algorithme réputé du *deep reinforcement learning*, dont le fonctionnement est particulièrement adapté à notre projet.

c. Analyse des résultats

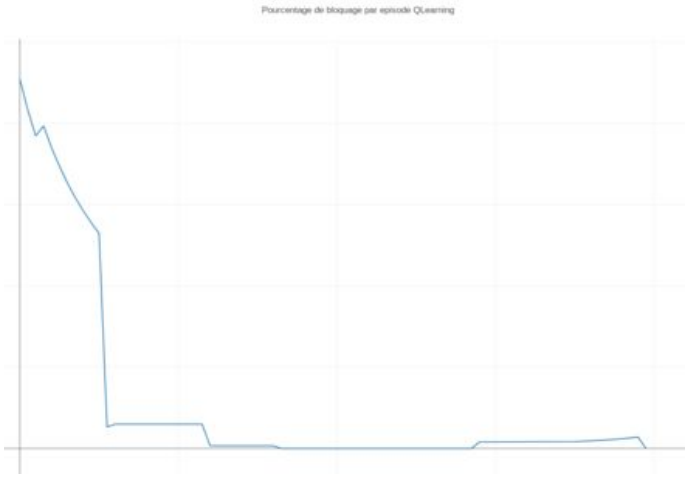
L'exécution des algorithmes est très lente et est confrontée, et ralentie, par des erreurs occasionnelles qui arrêtent la progression de la phase d'apprentissage. Des problèmes de mémoire au sein des machines virtuelles provoquent également des interruptions du programme.
Finalement, dans le cas où les conditions sont réunies pour une bonne exécution de nos algorithmes, l'approche séquentielle permet d'exécuter près de 500 épisodes en 72h.

En raison du manque de temps, nous limitons certains algorithmes à un nombre réduit d'épisodes, et observons la progression des performances de ces derniers dans le temps.

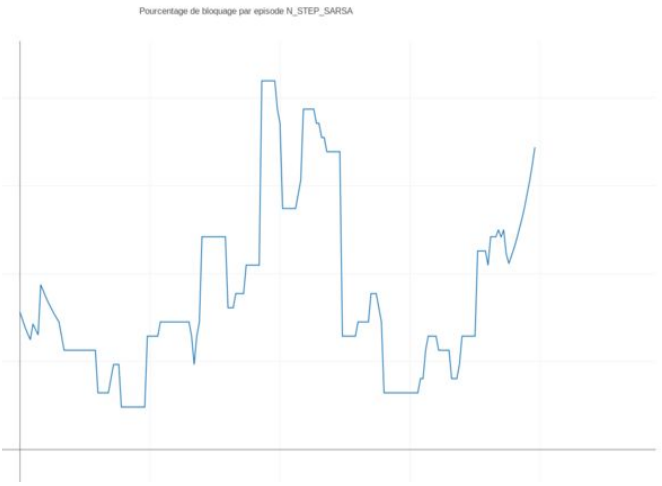
Les graphiques tracés représentent le **pourcentage de *crawl* réussi** (en ordonnée) pour chaque **épisode** (abscisse). Ce choix est motivé par le fait que chaque épisode dispose d'un nombre d'étapes différentes. Ainsi, il serait absurde de tracer le nombre de réussite ou d'échec : dans le cas où un épisode dispose d'une uniquement étape, si elle est réussie, constituerait un taux de réussite de 100% tandis que si nous traçons le nombre de réussite, elle serait insignifiante sur le graphique, comparé à d'autre épisodes disposant de plusieurs centaines d'étapes.

Il est également à noter que les résultats sont lissés afin d'éviter des pics et plutôt faire ressortir la tendance.

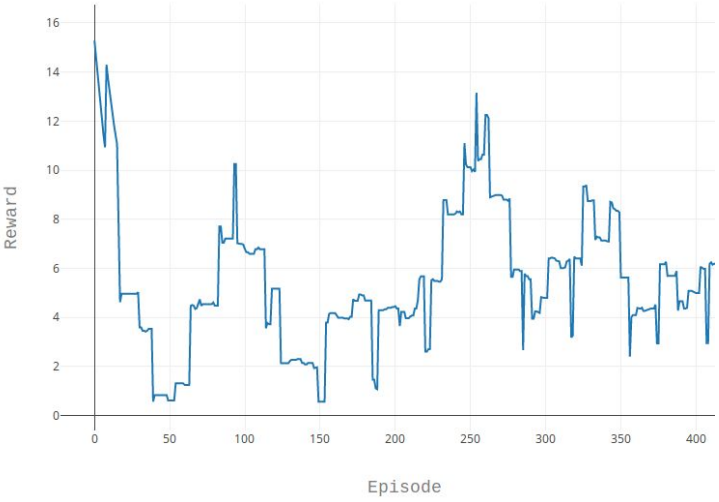
Tous les algorithmes sont exécutés dans les mêmes conditions, avec la même base de sites web, et dans le même ordre. L'A2C a cependant été exécuté sur mon ordinateur personnel (les *crawlers* étant cependant sur des machines virtuelles sur des serveurs distants) en raison du fait que cet algorithme nécessite la présence d'un GPU. Les résultats des algorithmes classiques sont représentés dans le tableau suivant :



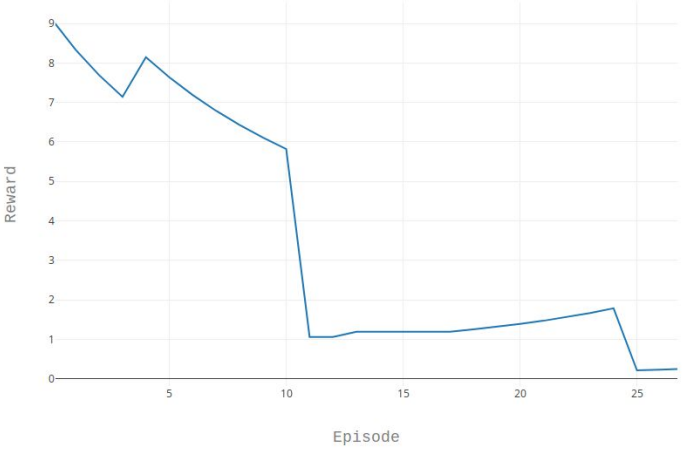
Q-Learning sur 80 épisodes



N-Step SARSA sur 180 épisodes



SARSA sur près de 420 épisodes



A2C sur près de 30 épisodes

Nous pouvons ainsi clairement observer une tendance pour les algorithmes SARSA, Q-Learning, et A2C, qui tendent bien à réduire le taux de blocage du bot au fil du temps. Le comportement du N-Step SARSA est cependant plus erratique et est ainsi à écarter. L'agent tend ainsi soit anticiper les cas de blocage et s'adapter en conséquent, soit à détecter un cas de blocage en cours, et ainsi de prendre les mesures nécessaires pour changer son empreinte de telle sorte que le bot ne soit plus reconnu comme celui s'étant fait bloquer. Cela peut se voir notamment dans le fait que nous atteignons rarement 100%

de blocage. Je suis cependant convaincu qu'en exécutant l'algorithme sur un nombre plus élevé d'épisodes, nous pourrons disposer de données plus significatives à analyser.

Conclusion

Le cahier des charges du projet à été rempli au terme de ce PFE. Les différents points à aborder l'ont été dans leur totalité, incluant les points optionnels ayant été présentés lors du rapport de mi-projet : nous avons ainsi pu mettre en place un modèle permettant aux crawlers de s'adapter aux conditions de son environnement, dans un premier temps de manière séquentielle, soit, en contrôlant un unique crawler. Nous avons ensuite étendu cette méthode à plusieurs crawlers sur des machines différentes, dans un premier temps toujours de manière séquentielle, avant de paralléliser ces derniers à l'aide d'algorithmes de l'état de l'art en *deep reinforcement learning*.
Nous avons enfin mis en place une méthode permettant de générer des mouvements de curseur réalistes et "humains" à travers l'utilisation de réseaux de neurones.
Au final, notre agent parvient à généraliser sur le comportement à adopter face à un site web particulier, afin d'éviter la détection.

Ce PFE orienté recherche m'a également permis de m'immerger dans le monde de la recherche au sein de l'équipe Spirals, et au contact de mes tuteurs. Les nouveaux concepts auxquels j'ai pu faire face m'ont permis de développer mes connaissances dans un large panel de technologies, et d'étendre ma connaissance et mon expérience au contact du *machine-learning.*

Dans la continuité de mon travail, il serait ainsi intéressant de pouvoir introduire le *reinforcement learning* directement au coeur des *crawlers*, dictant ainsi leurs choix lors de leur exécution, en optimisant les données récupérés par exemple.

Bibliographie

[1] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6047804/

[2] Efficient Deep web crawling using reinforcement learning - Jiang Wu & Al https://www.semanticscholar.org/paper/Efficient-Deep-Web-Crawling-Using-Reinforcement-Jiang-Wu/92dbaf21b19ce14dd236993455ddc9a387b66a52

[3] Personalized and Focused Web Spiders - Michael Chau https://link.springer.com/chapter/10.1007/978-3-662-05320-1_10

Mnih, V., Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K. (2019). *Asynchronous Methods for Deep Reinforcement Learning*. [online] arXiv.org. Available at: https://arxiv.org/abs/1602.01783

Medium. (2019). *Deep Reinforcement Learning: Playing CartPole through Asynchronous Advantage Actor Critic (A3C)….* [online] Available at: https://medium.com/tensorflow/deep-reinforcement-learning-playing-cartpole-through-asynchronous-advantage-actor-critic-a3c-7eab2eea5296

Sutskever, I., Vinyals, O. and Le, Q. (2019). *Sequence to Sequence Learning with Neural Networks*. [online] arXiv.org. Available at: https://arxiv.org/abs/1409.3215

Clemente, A., Castejón, H. and Chandra, A. (2019). *Efficient Parallel Methods for Deep Reinforcement Learning*. [online] arXiv.org. Available at: https://arxiv.org/abs/1705.04862

Vinyals, O., Toshev, A., Bengio, S. and Erhan, D. (2019). *Show and Tell: A Neural Image Caption Generator*. [online] arXiv.org. Available at: https://arxiv.org/abs/1411.4555

Sangaline, E. (2019). Making Chrome Headless Undetectable. Retrieved from https://intoli.com/blog/making-chrome-headless-undetectable/