UNILIM

ANALYSE ET DÉVELOPPEMENT LOGICIEL
PREMIER SEMESTRE
RAPPORT

IA des jeux, PPO et μRTS

Élèves :
RASSAT Marian
PAINCHAULT Emmanuel
TRABELSI Ayoub

Enseignant :
Tristan VACCON

8 janvier 2023

Table des matières

1 Introduction **2**

2 Acronymes et abréviations **2**

3 L'environnement μRTS **4**
3.1 Règles du jeu . . . 4
3.1.1 Les différentes unités . . . 4
3.1.2 Personnalisation des parties . . . 5
3.2 Options de lancement de μRTS . . . 6

4 L'environnement $\mu RTS - Py$ **7**
4.1 Organisation des données . . . 8

5 Le wrapper **9**

6 L'apprentissage par renforcement **10**
6.1 Les réseaux de neurones artificiels . . . 11
6.2 Le RL via Python . . . 11
6.2.1 TensorFlow . . . 12
6.2.2 PyTorch . . . 12
6.3 Notations et concepts du RL . . . 13
6.3.1 Les Actions et les Etats . . . 14
6.3.2 Les policies (stochastiques) . . . 14
6.3.3 Les trajectoires et les transitions . . . 14
6.3.4 Le principe de rewards et de return . . . 15
6.3.5 La maximisation du retour dans l'espace des policies . . . 15
6.3.6 Les fonctions de valeurs et les fonctions d'avantage . . . 16

7 L'algorithme PPO **16**
7.1 Les principes de PPO . . . 17
7.2 Résumé rapide de TRPO . . . 17
7.3 La divergence de Kullback-Leibler . . . 18
7.4 PPO . . . 18
7.5 Algorithme principal . . . 18

8 Sources **19**

Introduction

Le sujet que nous avons choisi est en rapport avec l'utilisation de l'intelligence artificielle dans un jeu vidéo, MicroRTS, qui est une version simpliste de ce qu'un jeu de stratégie en temps réel peut proposer. L'intelligence artificielle en question doit reposer sur les algorithmes PPO qui, qui sont une branche des algorithmes de Reinforcement Learning.

Il a donc été demandé dans un premier temps de se familiariser avec les algorithmes PPO, les réseaux de neurones, MicroRTS, et finalement l'environnement MicroRTS-Py pour l'entraînement dans MicroRTS. Avec tout cela, le but est d'entraîner et d'obtenir des IA ayant des comportements intéressants dans le jeu.

L'objectif ici sera donc à l'aide de ce qui à déjà été implémenté dans MicroRTS-Py de concevoir un wrapper Python pour générer, entraîner et tester des IA à l'aide des algorithmes PPO.

Acronymes et abréviations

AI : Artificial Intelligence

ML : Machine Learning

PPO : Proximal Policy Optimization

RL : Reinforcement Learning

RTS : Real Time Strategy Games

L'environnement μRTS

MicroRTS

MicroRTS (µRTS) est un jeu de stratégie en temps réel, dans lequel peuvent s'affronter deux joueurs. Il a été développé de manière minimaliste afin de permettre les tests d'algorithmes d'apprentissage par renforcement en laissant s'affronter deux intelligences artificielles l'une contre l'autre. Si ces tests se veulent concluants, les algorithmes peuvent alors être portés sur des RTS plus complexes afin d'entrer potentiellement en compétition avec des humains.

Règles du jeu

µRTS étant personnalisable, il n'est pas nécessaire de rentrer dans les détails, une vue d'ensemble du jeu et de ses mécaniques suffira.

Les différentes unités

µRTS se voulant minimaliste, il ne disposent que de quatre types d'unités mobiles :

 Les "Light"

 Les "Heavy"

 Les "Ranged"

 Les "Workers"

Chacune de ces unités peut combattre, mais seuls les Worker peuvent récolter des ressources.

Les Light, Heavy et Ranged servent à combattre le joueur adverse et sont bien plus efficaces pour cela que les Worker. Une dualité est induite par les Light et les Heavy :

d'une manière générale le Light est plus rapide mais tapera plus faiblement que le Heavy. Le Ranged quant à lui, peut attaquer à distance.

Il existe deux types de bâtiments (unités qui ne bougent pas) appartenant à un unique joueur (celui-ci peut en détenir plusieurs) :

Les "Base"


Les "Barracks"

Les "Base" permettent de produire des "Worker". Les "Barracks" elles, produisent les unités d'attaque : les "Light" les "Heavy" et les "Ranged".

Il existe finalement deux types de structures sur la carte dont le joueur ne choisit pas la position :


Les "Wall"

Les "Minerals"

Les Minerals peuvent être récoltés par les Worker (le nombre au milieu de celles-ci indique la quantité de ressources restantes). Les Wall eux ne bloquent pas la vision du joueur, mais empêchent toute construction sur la case où ils sont, et les unités ne peuvent pas les traverser. Cas particulier, les Ranged peuvent attaquer à travers.

Personnalisation des parties

La partie se termine quand l'un des deux joueurs n'a plus d'unités, ou si le nombre de cycles maximum à été atteint. Le vainqueur dans ce cas-là est celui à qui il reste le plus d'unités.

Pour pouvoir avancer dans la partie, les Worker doivent exploiter les Minerals afin de fournir au joueur les ressources pour produire de nouvelles unités.

Comme chaque carte est personnalisable, le déroulement d'une partie peut commencer à des stades plus ou moins avancés, voire inégaux entre les joueurs. On peut par exemple avoir un joueur dont la base est juste à côté de toutes les ressources, et l'autre joueur se trouvant au loin. Il est également tout à fait possible de commencer une partie déjà peuplée d'une multitude d'unités, pour par exemple entrainer une IA à un certain scénario bien défini.

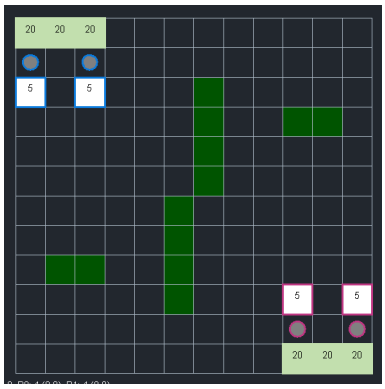

FIGURE 1 – Un exemple de map

La carte peut être entièrement visible, ou seulement en partie visible par chaque joueur. Chaque unité dispose alors d'une portée et permet au joueur de voir toutes les cases à portée de ses unités.

Des comportements prédéfinis sont déjà fournis par µRTS (des IA codées en dur), afin d'être utilisées pour entraîner nos futures IA. Par exemple, WorkerRush est une IA codée en dur qui dispose de la stratégie suivante :
- Si le joueur dispose de plus d'un Worker, directement envoyer tous les autres attaquer l'unité ennemie la plus proche.
- Si le joueur dispose d'une base, entraîner sans arrêt des Worker. Avec le Worker restant, construire une Base si nécessaire, sinon exploiter les Minerals.

Cela produit une IA hyper agressive qui attaquera dès qu'une unité qui le peut est vacante. On imagine bien que ce n'est pas forcément efficace, mais il faut peut-être une stratégie pour la contrer dans le cas où elle survient.

Options de lancement de μRTS

Il existe des paramètres afin d'utiliser µRTS pour faire du reinforcement learning, à savoir :

`-l` : Spécifie le mode de lancement. L'argument STANDALONE permet de lancer automatiquement la partie sur la machine.

`-m` : Chemin d'accès au fichier de la carte du jeu.

`-c` : Le nombre de cycles maximum.

`--headless` : Nous permet de ne pas afficher la partie afin de ne pas dégrader la vitesse des calculs.

`--partially_observable` : Permet d'activer ou non le brouillard de guerre.

`-u` : Permet de modifier le type de table d'unité à prendre (toutes les informations relatives aux unités : leurs dégats, leur vie, etc). Par défaut, il en existe trois. Deux d'entre elles sont déterministes (les dégâts des unités sont fixes), l'autre non.

`--ia1` et `--ia2` : Le nom de la classe à utiliser pour l'IA 1 ou 2.

Ces paramètres permettront d'entraîner l'IA plus efficacement mais aussi différemment. Une IA entraînée avec le brouillard de guerre aura un comportement différent de celle entraîner sans et d'une entraîner au deux.

L'environnement $\mu RTS - Py$

MicroRTS-Py

MicroRTS-Py (µRTS-Py) est une surcouche (un wrapper) python de MicroRTS qui intègre des mécanismes de ML. Il nous servira d'exemple afin de concevoir notre propre wrapper de MicroRTS, et ainsi pouvoir générer, entraîner et tester dans celui-ci des IA qui implémentent des algorithmes PPO que nous avons nous-même réalisés.

Organisation des données

MicroRTS-Py fournit une structuration intéressante pour le stockage et le traitement des informations du jeu. Chaque cellule peut alors être décrite par une liste de bits.

Cette structuration est présentée de la manière suivante :

Observation Features	Planes	Description
Hit Points	5	0, 1, 2, 3, ≥ 4
Resources	5	0, 1, 2, 3, ≥ 4
Owner	3	player 1, -, player 2
Unit Types	8	-, resource, base, barrack,worker, light, heavy, ranged
Current Action	6	-, move, harvest, return, produce, attack
Action Components	**Range**	**Description**
Source Unit	$[0, h \times w - 1]$	the location of unit selected to perform an action
Action Type	$[0, 5]$	NOOP, move, harvest, return, produce, attack
Move Parameter	$[0, 3]$	north, east, south, west
Harvest Parameter	$[0, 3]$	north, east, south, west
Return Parameter	$[0, 3]$	north, east, south, west
Produce Direction Parameter	$[0, 3]$	north, east, south, west
Produce Type Parameter	$[0, 6]$	resource, base, barrack, worker, light, heavy, ranged
Relative Attack Position	$[0, a_r^2 - 1]$	the relative location of unit that will be attacked

FIGURE 2 – Répartition des bits d'une observation de case

En prenant le cas d'une cellule sur laquelle se trouve un "Heavy" du joueur 2 avec 6 points de vies en train de se déplacer on obtient la liste suivante :

[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]

Les cinq premiers bits [0 0 0 0 1] correspondent aux "Hit Points" du tableau. Ici c'est le dernier bit qui est mis à 1, on sait donc que l'unité a 4 points de vie ou plus.

Les cinq suivants [1 0 0 0 0] correspondent aux ressources que l'unité transporte. Puisque c'est le premier bit qui est à 1, on sait que l'unité décrite ne transporte aucune ressource.

Ensuite viennent les trois bits [0 0 1] correspondants au joueur à qui appartient l'unité. Ici, puisque c'est le 3e bit qui est à 1, on sait que l'unité appartient au joueur 2. Si c'était le 2e bit alors l'unité n'aurait appartenu à personne (exemple : Mineral)

Les huit suivants [0 0 0 0 0 0 1 0] représentent un le type d'unité. Ici, on sait donc que c'est un Heavy.

Les six derniers [0 1 0 0 0 0] indiquent l'action que réalise l'unité. Ici, elle se déplace simplement.

Les actions que peut réaliser une unité sont encodées avec la même logique, et simplement des ranges différentes.

A l'aide d'une structuration similaire, il est alors possible d'utiliser un algorithme implémentant PPO avec un réseau neuronal qui a comme nombre d'entrées le nombre total de bits, et d'en récupérer des informations permettant la construction et l'envoi d'une action pertinente face à la situation.

Le wrapper

Afin d'utiliser µRTS avec des IA que nous avons nous-même développé, deux façon de procéder sont proposées. L'une d'elle est en Java, et nécessite de créer un nouveau fichier `.jar` contenant notre IA héritant d'une classe `AIWithComputationBudget`.

La deuxième façon nous permettra de, en plus d'utiliser une nouvelle IA, créer un wrapper et nous permettra aussi d'utiliser Python comme langage de programmation comme c'est l'un de ceux ayant d'une part, beaucoup de modules existant permettant d'utiliser rapidement de l'intelligence artificielle et d'une autre part, il possède une grande communauté active autour de celle-ci.

Cette façon de créer un wrapper utilise des sockets ce qui nous permettra de faire transiter les informations via le réseau grâce à la classe `SocketAI`. En nous aidant de 3 classes déjà présentes mais codées uniquement en Java `JSONSocketWrapperAI`, `XMLSocketWrapperAI`, `GameVisualSimulationWithSocketAI` et principalement grâce à la dernière il sera possible d'obtenir rapidement quelque chose d'exploitable.

Notre wrapper consistera donc à lancer grâce à python µRTS, en mode serveur, et par la même lancer notre propre IA en mode client. Il faudra ainsi créer la connexion entre le jeu et l'IA et faire ensuite transiter les informations grâce au formats JSON ou XML mais cela n'a pas de grande importance car µRTS gère les deux.

Le serveur sera donc codé en Java et aura pour but de créer l'environnement du jeu. Il sera intéressant de pouvoir, via les lignes de commandes ou via un fichier de configuration, fournir à ce serveur des cartes personnalisées afin d'entraîner notre IA sur plusieurs terrains. Nous pourrons également faire varier les UnitTypeTable correspondantes aux statistiques propres à chaque unité, et même créer plus de deux IA afin de créer des tournois, etc.

Nos IAs seront créés grâce à l'un des modules d'IA et de réseaux neuronaux via Python.

Une fois la connexion établie, il s'agira d'une suite d'échanges entre µRTS et l'IA. Le serveur enverra le `GameState` à savoir les différentes informations concernant la carte, les unités présentes sur celle-ci. L'IA devra ensuite retourné une action à effectuer puis attendre de nouveau de le `GameState`. Il faudra pouvoir créer et entraîner un réseau neuronal, le sauvegarder puis pouvoir le réutiliser.

C'est alors qu'entre en jeu l'apprentissage par renforcement, ou reinforcement learning en anglais ainsi que l'algorithme d'optimisation de la politique proximale, ou proximal policy optimization.

L'apprentissage par renforcement

Le RL est un type d'apprentissage automatique qui fait appel aux réseaux de neurones artificiels. En quelques mots, c'est un algorithme qui va chercher à modifier un réseau de neurones afin de maximiser une récompense.

Les réseaux de neurones artificiels

Les réseaux de neurones artificiels sont des modèles de calcul mathématique et informatique dont la conception est inspirée du fonctionnement des neurones biologiques capables de traiter les entrées reçues d'un ensemble de connexions caractérisées par des poids synaptiques. La somme pondérée des entrées devient un signal d'entrée vers la fonction d'activation pour donner une sortie.

Ces poids d'entrée sont réglables afin que le réseau neuronal puisse s'adapter au fur et à mesure de son apprentissage et ainsi maximiser le nombre de sorties jugé correcte comme le montre la figure suivante.

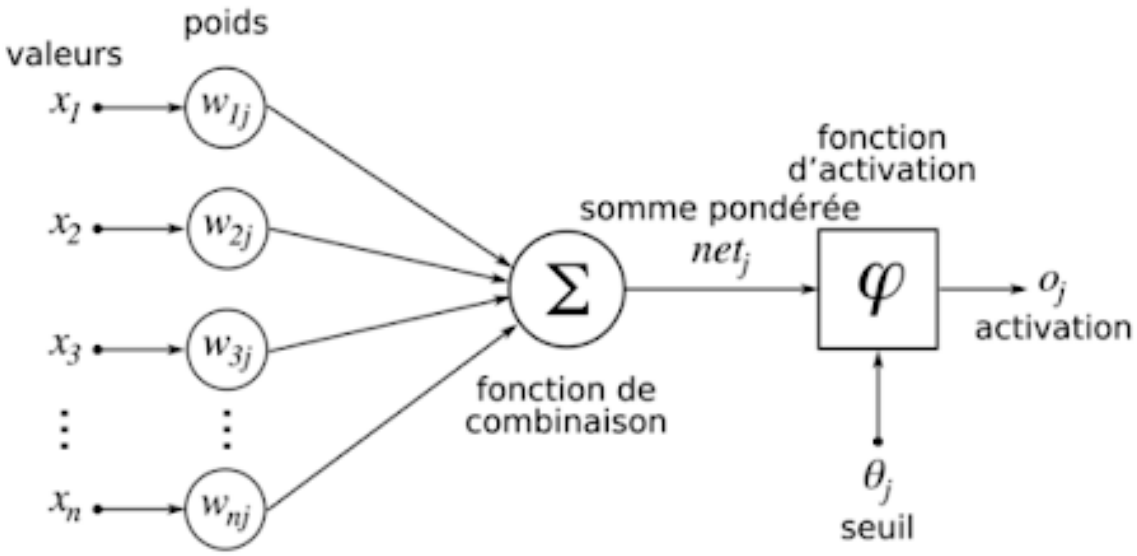

FIGURE 3 – Présentaion d'un réseau neuronal

Il y a diverses architectures de réseaux neuronaux artificiels. Notre algorithme s'applique à toutes les structures de réseaux. La seule condition est que celui-ci est autant d'entrées que de données que nous devons lui passer et autant de sorties que d'actions disponibles que l'on souhaite intégrer.

Le RL via Python

En utilisant Python, les développeurs peuvent facilement importer et utiliser des collections de modules pré-écrits appelées bibliothèques pour implémenter des fonctionnalités. Python est particulièrement utile pour la data science car, les réseaux neuronaux étant

traités sur carte graphique, la vitesse de calcul ne dépend pas uniquement du langage. La préparation du réseau neuronal sera certes plus lente sur Python que sur un autre langage comme le C mais cela ne représente qu'une infime partie du temps d'exécution. Passer sur Python permet de coder et d'apporter des modifications beaucoup plus facilement et rapidement pour une perte négligeable..

Il existe de nombreuses bibliothèques de machine learning disponibles pour Python, qui permettent de développer des modèles de machine learning de manière efficace comme : TensorFlow, PyTorch, Keras, NumPy ,Pandas et Scikit-Learn.

Les bibliothèques populaires pour l'apprentissage par renforcement sont Tensorflow et PyTorch. Celles-ci possèdent de grandes communautés et permettent sensiblement la même chose.

TensorFlow

TensorFlow est une bibliothèque de machine learning qui offre une grande flexibilité grâce à ses nombreux outils et ressources ainsi qu'une communauté comme dit précédemment. Ces outils permettent aux développeurs de mener efficacement des recherches en machine learning et en deep learning, ainsi que de facilement créer et déployer des solutions basées sur le machine learning.

TensorFlow peut être utilisé pour mettre en production des modèles de machine learning sur diverses plateformes, y compris dans le cloud, sur site, dans un navigateur ou sur un appareil.

La bibliothèque "tf-agents" de TensorFlow est une bibliothèque de haut niveau qui permet de facilement implémenter des algorithmes de reinforcement learning avec TensorFlow. Elle fournit une large gamme d'algorithmes pré-implémentés, ainsi que des outils pour collecter et analyser les données de l'agent pendant l'apprentissage. Voici un exemple d'utilisation de la bibliothèque "tf-agents" :

PyTorch

PyTorch est une bibliothèque de machine learning populaire développée par l'équipe de recherche en intelligence artificielle de Facebook. Elle est souvent utilisée pour les tâches de vision par ordinateur, de traitement du langage naturel et d'autres tâches complexes similaires. De nombreux systèmes utilisent le côté back-end distribué de PyTorch pour améliorer les performances lorsqu'ils traitent de grandes quantités de données.

PyTorch offre un certain soutien pour l'apprentissage par renforcement. On peut utiliser PyTorch pour implémenter nos propres algorithmes d'apprentissage par renforcement en utilisant ses outils de base pour la définition et l'exécution de modèles de machine learning.

```
import tensorflow as tf
from tf_agents.networks import q_network
from tf_agents.agents.dqn import dqn_agent

q_net = q_network.QNetwork(
  train_env.observation_spec(),
  train_env.action_spec(),
  fc_layer_params=(100,))

agent = dqn_agent.DqnAgent(
  train_env.time_step_spec(),
  train_env.action_spec(),
  q_network=q_net,
  optimizer=optimizer,
  td_errors_loss_fn=common.element_wise_squared_loss,
  train_step_counter=tf.Variable(0))

agent.initialize()
```

Importation de tensorflow et les outils tf-agents

Créer une instance de la classe QNetwork avec les arguments spécifiés.

Créer une instance d'un objet DqnAgent. DqnAgent est une classe qui semble définir un agent d'apprentissage par renforcement (RL) qui utilise l'algorithme de Q-learning profond pour apprendre une politique pour interagir avec un environnement.

Initialiser l'état interne de l'agent et le préparer à l'interaction avec un environnement.

FIGURE 4 – Extrait commenté d'un code utilisant tensorflow

Notations et concepts du RL

Le RL est une des branches de l'intelligence artificielle, au même titre que les apprentissages supervisés et non supervisés. Mais même si ces deux catégories, à cause de leur nom, paraissent faire une partition totale des types d'apprentissage, il n'en est en réalité rien. Les deux types précédemment cités prennent en entrée des descriptions de l'état, et dans le cas du supervisé, associés à un label, binaire ou non, pour essayer d'en dégager une règle permettant une catégorisation selon les labels, afin de généraliser à des descriptions d'état non rencontrés. Cependant, le reinforcement learning n'a aucun besoin de données déjà récoltées pour l'apprentissage (qui peuvent être très difficiles à recueillir dépendant du cas d'application), car il est basé sur un système de récompenses, qui interagit avec son environnement en réalisant des actions, et en apprenant des changements d'états et des réactions (erreurs ou succès, avantage gagné, etc).

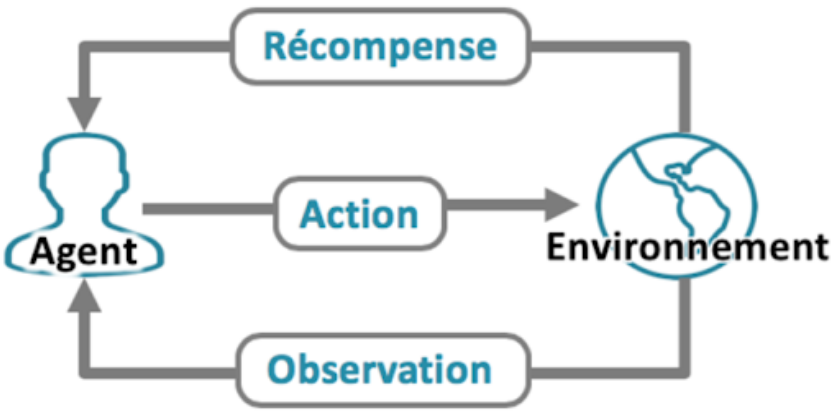

FIGURE 5 – Visualisation de l'apprentissage par renforcement

Les Actions et les Etats

Les actions et les états représentent tout ce que saura l'agent de l'environnement avec lequel il interagit. Les actions sont typiquement notées a ou a_t pour l'action réalisée au temps t. Elles représentent le moyen pour l'agent d'interagir avec l'environnement, et sont sélectionnées par l'agent suivant une certaine méthode.

Les états (states), quant à eux, notés s ou s_t pour l'état reçu au temps t, sont le moyen pour l'agent de percevoir son environnement. Ce sont toutes les données du monde, telles que perçues par l'agent. Elles sont soit complètes soit partielles, auquel cas on peut également les trouver notées sous la forme o ou o_t pour observation au temps t. Ici, nous n'utiliserons que s ou s_t.

Le tout premier état du monde, noté s_0, est pris dans la distribution des états de base, notée $\rho_0(\cdot)$. Ainsi la probabilité qu'a l'état s d'être l'état de base est $\rho_0(s)$

Les policies (stochastiques)

Les policies sont la représentation du mécanisme qu'a l'agent pour déterminer quelle action il devrait prendre en fonction de l'état de l'environnement. Il en existe deux types, soit déterministes soit stochastiques. Les déterministes sont notées sous la forme

$$a_t = \mu_\theta(s_t)$$

c'est-à-dire l'action déterminée par la policy déterministe avec les paramètres quand l'état de l'environnement est s_t.

Les policy stochastiques sont elles notées

$$a \sim \pi_\theta(\cdot|s_t)$$

c'est-à-dire l'action a a une probabilité d'être choisie parmi toutes les actions disponibles de $\pi_\theta(\cdot|s_t)$ (la policy avec les paramètres θ sachant s_t). On peut également noter cette probabilité sous la forme

$$\pi_\theta(a|s_t)$$

probabilité de réaliser l'action a sachant s_t sous la policy π_θ. En pratique, on n'utilise quasiment que des policy stochastiques et on les notera toujours donc π_θ.

Les trajectoires et les transitions

Les trajectoires (autrement appelées épisodes, ou rollouts), sont un élément essentiel de la théorie derrière le reinforcement learning, mais ne sont pas toujours explicitées comme des objets à part entière, notamment dans le livre de Sutton et Barto, où même si elles sont mentionnées, elles ne font pas l'objet d'une définition claire. Ce sont une liste d'associations état-action réalisables dans l'environnement, elles représentent la "trajectoire" qu'a pu prendre l'environnement en fonction des actions de l'agent :

$$\tau = (s_0, a_0, s_1, a_1, ...)$$

Les transitions entre différents états sont, quant à elles, déterminées par l'environnement où évolue l'agent et peuvent être déterministes ou stochastiques, et ce indépendamment des policy :

si les transitions sont déterministes $s_{t+1} = f(s_t, a_t)$

si les transitions sont stochastiques $s_{t+1} \sim P(\cdot|s_t, a_t)$ (aussi notées $P(s_{t+1}|s_t, a_t)$)

Le principe de rewards et de return

Les rewards et les return sont la manière qu'aura l'agent de juger de la qualité de ses actions (définir si elles étaient bonnes ou mauvaises), aussi bien sur le court terme que sur le long terme. Elles sont modélisées par des fonctions de reward, qui prennent entre 1 et 3 arguments. Soit uniquement l'état actuel de l'environnement ($r_t = R(s_t)$), soit l'état ainsi que l'action réalisée à ce moment-là ($r_t = R(s_t, a_t)$) ou finalement, la version la plus complexe, qui prend en plus des deux autres l'état suivant de l'environnement :

$$r_t = R(s_t, a_t, s_{t+1})$$

Après avoir pu modéliser la reward pour un moment t donné, il devient possible de créer des fonctions de return, qui elles tenteront de calculer la valeur d'une trajectoire entière. Il en existe deux types principaux :

- Les return à horizon finie sans discount, qui sont la somme simple des reward qu'a pu obtenir un agent sur une période définie
$$R(\tau) = \sum_{t=0}^{T} r_t$$
- Les return à horizon infinie avec discount, qui sont la somme des reward qu'a pu obtenir un agent tout au long de la trajectoire, mais avec un facteur dépendant d'à quel moment il a pu obtenir ladite reward
$$R(\tau) = \sum_t \gamma^t r_t$$

Le paramètre $\gamma \in [0, 1[$ permet de définir avec quel poids les reward qui sont plus éloignées dans le temps seront jugées.

La maximisation du retour dans l'espace des policies

Le problème central dans le reinforcement learning se fait de plus en plus clair, il s'agit de choisir la policy qui maximisera le return moyen sur l'ensemble des trajectoires que la policy pourra prendre

$$\pi^* = argmax_\pi(J(\pi))$$

avec π^* la policy optimale, et $J(\pi)$ le return estimé pour la policy donnée.

Pour calculer cela, il est nécessaire de connaître la probabilité qu'une policy puisse emprunter une trajectoire donnée, multiplier par le return de la trajectoire, et finalement intégrer (faire la moyenne) sur toutes les trajectoires :

$$J(\pi) = \int_\tau P(\tau|\pi)R(\tau) = E_{\tau\sim\pi}[R(\tau)]$$

avec

$$P(\tau|\pi) = \rho_0(s_0) \sum_{t=1}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

la probabilité pour la trajectoire donnée d'être réalisée par la policy donnée.

Les fonctions de valeurs et les fonctions d'avantage

Les fonctions de valeurs permettent d'estimer la "valeur" des états, ainsi que des paires état-action en fonction d'une policy, ce qui s'avère très utile quand il s'agit, par exemple, de mettre à jour nôtre policy actuelle vers une "meilleure" version. Il existe 4 de ces fonctions :

- $V^\pi(s) = E_{\tau\sim\pi}[R(\tau)|s_0 = s]$ qui estime le return en partant de s, puis en suivant la policy à l'infini. Elle se nomme la fonction de valeur On-Policy.
- $Q^\pi(s) = E_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$ qui estime le return en partant de s, en effectuant l'action a, puis en suivant la policy à l'infini. Elle se nomme la fonction d'action-valeur On-Policy.
- $V^*(s) = max_\pi E_{\tau\sim\pi}[R(\tau)|s_0 = s]$ qui estime le return en partant de s, puis en suivant la policy optimale à l'infini. Elle se nomme la fonction de valeur optimale.
- $Q^*(s) = max_\pi E_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$ qui estime le return en partant de s, en effectuant l'action a puis en suivant la policy optimale à l'infini. Elle se nomme la fonction d'action-valeur optimale.

La fonction d'avantage, quant à elle, permet à partir des fonctions de valeur et d'action-valeur de déterminer si une action a un "avantage" sur les autres en moyenne dans un état précis. Elle est définie comme la différence entre la fonction d'action-valeur On-Policy et la fonction de valeur On-Policy :

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

La notation $\hat{A}_t$ apparaîtra souvent dans les algorithmes, et il est essentiel pour implémenter PPO de comprendre comment calculer cette approximation de A.

Pour cela, nous utiliserons l'approche présentée dans l'article d'OpenAI décrivant PPO qui est une version tronquée permettant l'utilisation de Generalized Advantage Estimator (présenté dans cet article) sur un temps T fini :

$$\hat{A}_t = \sum_{l=0}^{T-1} (\gamma\lambda)^l \delta_{t+l}$$

avec :

γ un hyper-paramètre décrivant le discount rate (comme dans le calcul de return)
λ un hyper-paramètre, mis à 0.95 dans les expérimentations présentées dans l'article décrivant PPO
δ_t un raccourci pour $r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$

L'algorithme PPO

PPO est un algorithme s'inscrivant dans le contexte du reinforcement learning. Puisqu'il est assez spécifique et que le domaine en lui-même est complexe, il paraît bon de faire une présentation de celui-ci. Cela permettra une meilleure compréhension du sujet et de la théorie, mais également d'apporter les notations usuelles afin de comprendre les différentes formules disponibles décrivant l'algorithme. La source de ces informations est majoritairement l'article d'OpenAI, qui est une très bonne ressource, permettant de bien comprendre les concepts essentiels au RL.

Les principes de PPO

PPO vise uniquement à réaliser de l'optimisation de policy, c'est-à-dire qu'il maximisera un certain objectif. Il est dit "on-policy" (comme la plupart des algorithmes d'optimisation de policy), et ce en opposition aux algorithmes "off-policy", qui appartiennent plutôt à la famille des algorithmes de Q-Learning, qui tentent d'approximer $Q^*(s,a)$ la fonction calculant le return optimal en améliorant $Q^\theta(s,a)$, permettant de choisir ensuite l'action à réaliser avec $a_t = argmax_a Q^\theta(s_t,a)$, l'action avec la plus grande valeur estimée depuis l'état actuel. Pour ce faire, un algorithme de Q-Learning utilisera généralement toutes les données à sa disposition, quelle que soit la policy utilisée pour les récolter.

PPO lui, comme la majorité des algorithmes "on-policy", n'utilisera que les données récoltées par la dernière version de la policy pour mettre à jour celle-ci vers une version considérée comme meilleure. Il tentera de déterminer ladite nouvelle version en maximisant un objectif de "substitution" (nommé ainsi car l'objectif principal reste $J(\pi)$) qui est dérivé de l'objectif de l'algorithme TRPO.

Résumé rapide de TRPO

TRPO est un algorithme qui établit une procédure justifiée par la théorie, et qui obtient des résultats d'optimisation qui approchent de manière monotone les policy optimales théoriques. Pour accomplir cela, il utilise un objectif de substitution nommé $L^{CPI}(\theta)$ dans l'article d'OpenAI présentant PPO, défini comme :

$$L^{CPI}(\theta) = \hat{\mathrm{E}}_t[r_t(\theta)\hat{\mathrm{A}}_t]$$

avec :

$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ le ratio entre la probabilité précédente de choisir a_t et la nouvelle probabilité (à ne pas confondre avec r_t la reward en t)

$\pi_{\theta_{old}}$ la policy précédente et qui va être changée

$\hat{\mathrm{A}}_t$ l'avantage estimé au temps t (calculable de différentes manières)

TRPO inclut également une contrainte sur , et ce car il veut pénaliser des changements de policy qui résulteraient en un trop grand changement dans le comportement de la policy. Pour cela, il introduit la contrainte suivante :

$$\hat{\mathrm{E}}_t[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta$$

avec :

δ un hyper-paramètre (0.01 par exemple, comme dans les expérimentations qu'a présenté l'article proposant TRPO)

$KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]$ le calcul de la divergence de Kullback-Leibler entre les anciennes probabilités de réaliser chaque action et les nouvelles

La divergence de Kullback-Leibler

La divergence de Kullback-Leibler est une mesure de la dissimilarité entre deux distributions de probabilités, ici utilisée pour déterminer à quel point la nouvelle policy est différente de l'ancienne dans son comportement.

Puisque les distributions de probabilités utilisées ici sont discrètes, la formule est :

$$KL[P, Q] = \sum_i P(i) log \frac{P(i)}{Q(i)}$$

PPO

Dans PPO, un nouvel objectif dérivé de celui de TRPO est utilisé. Il est motivé par la volonté de réduire la complexité de l'implémentation de TRPO, tout en conservant les avancées monotones produites par celui-ci et le fait de réaliser plusieurs optimisations de la policy par batch de données récupérées.

Ainsi, PPO est un algorithme d'optimisation de premier ordre uniquement, contrairement à TRPO, et est utilisable dans plus de situations, permettant une meilleure flexibilité.

La nouveauté dans l'objectif proposé par OpenAI pour PPO dans leur article est l'utilisation, à la place de la contrainte qui n'existe maintenant plus, est le fait de clip le ratio dans l'objectif L^{CPI} à maximiser. Ils proposent ainsi l'objectif dénommé L^{CLIP} suivant :

$$L^{CLIP}(\theta) = \hat{\mathrm{E}}_t[min(r_t(\theta)\hat{\mathrm{A}}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{\mathrm{A}}_t)]$$

avec ϵ un hyper-paramètre, généralement très petit.

Ce nouvel objectif permet de ne pas réaliser des changements qui modifieront trop la policy et ce dans le but, tout comme TRPO, de ne pas trop modifier le comportement de celle-ci, et de garantir une avancée vers l'optimal.

Algorithme principal

Comme mentionné précédemment, l'algorithme prend plusieurs petits pas vers la policy qu'il pense optimale à la fois par batch de données. Ces plusieurs petits pas sont réalisés un nombre N de fois, cependant il est intéressant, comme suggéré dans l'article de documentation Spinup d'OpenAI d'utiliser la divergence KL entre l'ancienne et la nouvelle policy afin d'arrêter plus tôt que prévu les plusieurs petits pas. Cela résulte logiquement en une meilleure stabilité, puisqu'il y aura une limite sur chaque pas mais également une limite globale.

Dans notre cas, voici l'algorithme qui servira de référence, et qui implémente la version de PPO utilisant un clip, qui a obtenu les meilleures performances sur les différentes versions) :

initialiser la policy π_0
pour chaque iteration $k \in [1, K]$ **faire** :

utiliser la policy π_k T fois
collecter les donnees crees

calculer les avantages A_t
calculer $L^{CLIP}(\pi_k)$

initialiser $k+1 = k$
initialiser $n = 0$

tant que $E_t[KL[\pi_k(\cdot|st), \pi_{k+1}(\cdot|st)]] \leq \delta$ et $n < N$ **faire** :
optimiser π_{k+1} (ascension de gradient, gradients = L^{CLIP})

calculer L_t^{CLIP} **pour** la policy π_{k+1}

$n = n + 1$
fin tant que
fin pour

L'explicitation des étapes d'optimisation devrait permettre une implémentation peu fastidieuse, et ce car l'optimisation du gradient une fois L_t^{CLIP} calculé, est gérée par un algorithme tel qu'Adam, comme suggéré dans l'article présentant PPO.

Sources

GitHub MicroRTS :
https://github.com/Farama-Foundation/MicroRTS

GitHub de MicroRTS-Py :
https://github.com/Farama-Foundation/MicroRTS-Py

AI Competition MicroRTS :
https://sites.google.com/site/micrortsaicompetition/getting-started?authuser=0

L'apprentissage par renforcement :
https://www.quantmetry.com/blog/reinforcement-learning-conduite-autonome-essais-aws-deep-racer/

Article OpenAI sur l'apprentissage par renforcement :
https://spinningup.openai.com/en/latest/spinningup/rl_intro.html

Livre de Sutton et Barto, "Reinforcement Learning : An introduction" :
https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

Techtarget, définition de neurone artificiel :
https://www.techtarget.com/searchcio/definition/artificial-neuron

Les bibliothèques Python à utiliser pour le machine learning :
https://mobiskill.fr/blog/conseils-emploi-tech/les-bibliotheque-python-a-utiliser-pour-le-machine-learning/