le cnam
Paris

I. Table des matières

II. Contexte

Jeune ingénieur diplômé de l'école nationale supérieure des arts & métiers, j'ai toujours été intéressé par les outils informatiques et l'aide que ces derniers apportent dans notre quotidien. Avec des amis nous menons un projet qui implique l'utilisation de réseau de neurones à des fins de meilleures consommations énergétiques. Afin d'approfondir mes connaissances sur ces outils, et de m'initier au monde de la programmation informatique, je me suis inscrit au C.N.A.M. dans l'ensemble des modules constituant le certificat de spécialisation d'intelligence artificielle sur l'année 2020-21.

III. Objectifs

Les objectifs dans ce rapport sont de mettre en pratique les connaissances acquises lors de l'enseignement du module RCP 211 : Intelligence artificielle avancée, autour d'un cas d'étude. Parmi ceux proposés j'ai sélectionné le sujet « Deep A3C » qui consiste à apprendre un agent à jouer à Pac-Man avec l'utilisation de la technique de dite « Asynchronous Advantage Actor Critic » introduite par une des équipes de recherche de google en 2016.

IV. Synthèse du papier

La filiale de Google nommé DeepMind a en 2016 introduit dans son papier nommé « Asynchronous Methods for Deep Reinforcement Learning » un nouveau paradigme dans la catégorie de l'apprentissage par renforcement.

A. Avantages de l'asynchronisation

Il était pensé que la combinaison de simples algorithmes d'apprentissage par renforcement et de réseaux neuronaux ne serait pas suffisamment stable pour être fiable. C'est une des raisons qui a introduit la technique de « replay memory », une solution qui permet de réduire la non-stationnarité et les mises à jour corrélées qui sont deux éléments qui empêchent les modèles de progresser dans leurs apprentissages. Toutefois ces techniques ne sont pas optimales, car elles :

- Apprennent en se mettant à jour avec des données générées par une politique antérieure
- Nécessite un entraînement hors ligne et donc beaucoup de mémoire
- Utilisent plus de calculs pour chaque interaction

Afin de pallier à ces problèmes, le papier introduit l'utilisation de méthodes asynchrones à la place du « replay memory ». Ce qui est asynchrone c'est la mise à jour d'agents, exécuter en parallèle et chacun d'eux ayant leur propre instance de l'environnement. Je reviendrai plus en détail sur cette asynchronisation dans la suite de cette section. Le papier applique ce concept général aux algorithmes existants : « One-step Sarsa », « One-step Q-learning », « N-step Q-learning », « Advantage actor critic », et en fait leur déclinaison asynchrone.

L'idée est d'améliorer ces algorithmes de façon à les rendre plus robustes, plus performants tout en consommant moins de ressources. Pour arriver à cet objectif, les concepteurs se basent sur deux principes :

- Développer des agents d'apprentissage asynchrones sur un même processeur informatique (CPU) à raison d'un agent par cœur, ceci afin de minimiser les transferts de données
- Que la multiplicité des agents permette d'explorer différentes parties de l'environnement, et que cette diversité peut être explicitement utilisée pour être maximisée.

Grâce à l'abandon du « replay memory » et l'utilisation de l'asynchronisation, il est possible de développer des variants stables « en ligne » d'apprentissage par renforcement. Un des avantages est le gain de temps d'entraînement qui est un peu près proportionnel aux nombres d'agents, ce qui assure l'évolutivité du modèle. Le papier démontre également que malgré des initialisations aléatoires, il existe des pas d'apprentissage qui donne de bons résultats, ce qui témoigne la robustesse et la stabilité des modèles asynchrones.

B. Résultats démontrés

L'équipe de recherche a mené de très nombreux essais de ses méthodes asynchrones sur les jeux Atari2600, qui servent désormais de benchmark pour l'apprentissage par renforcement, et elles s'avèrent être plus performantes que la plupart des méthodes de l'état l'art, tel que « Deep Q-learning » Network (DQN), au moment de la sortie du papier.

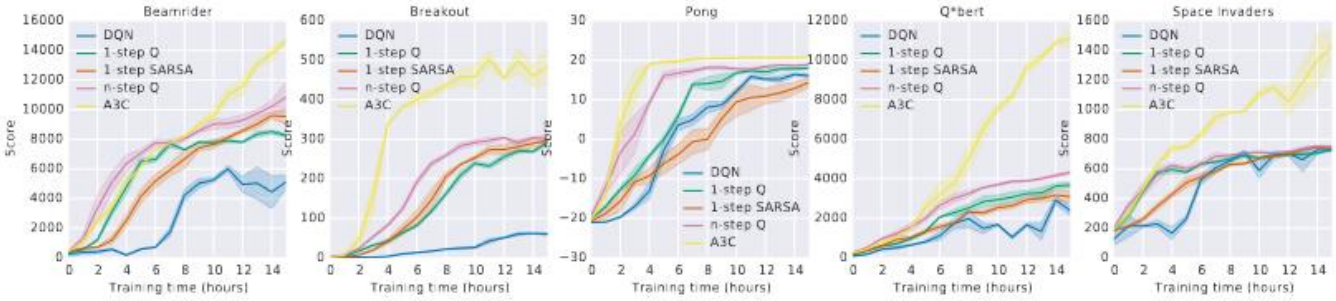

Figure 1 Comparaison des variants asynchrones avec DQN

Sur ces graphiques issus du papier, il se dégage une tendance générale des versions asynchrones à converger plus rapidement, de plus ces variants sont entraînés sur un CPU de 16 cœurs tandis que DQN est entraîné sur un GPU. Comme en témoigne le tableau de comparaison ci-dessous, également issu du papier, ces méthodes asynchrones présentent de réels avantages en termes de calculatoire, puis qu'elles s'achèvent de bien meilleurs résultats que leurs concurrents, sur des matériels moins puissants, pour des temps très réduits.

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

Tableau 1 Scores moyen et médian normalisé selon les performances de l'Homme sur 57 jeux Atari

Il est également remarquable que l'asynchronisation semble profiter plus à l'A3C qu'aux autres, sachant ceci les chercheurs l'ont également testé sur :

- Un environnement continu, tel que la simulation de physique des solides dans un milieu dynamique
- Sur sa capacité de généralisation, à l'aide d'un jeu « Labyrinth » qui a la particularité de changer le parcours du labyrinthe à chaque épisode

Sur chacun des tests, l'algorithme A3C montre sa capacité à apprendre plus rapidement et de généraliser des concepts en ayant que des images en entrée. C'est donc un algorithme qui se montre prometteur pour l'apprentissage par renforcement, et que je vais mettre en application dans cette étude.

C. Méthode A3C

Après cette description des avantages généraux de l'asynchronisation et des résultats plus qu'intéressants de l'« asynchronous advantage actor critic », je vais rentrer plus en détail sur cette technique avant de l'implémenter par moi-même.

Cette méthode est basée sur le principe d'acteur-critique, une méthode de mise à jour simultanée de la politique et la valeur. En voici la formule et ses correspondances :

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)]$$

- La "Critique" qui estime la fonction de valeur qui peut être la valeur d'action (la valeur Q) ou la valeur d'état (la valeur V)
- L' « Acteur » qui met à jour la distribution de la politique dans la direction suggérée par la critique

Cette technique d'acteur critique présente un inconvénient majeur : elle est généralement instable ce qui la rend lente à converger. Une façon de réduire la variance et d'augmenter la stabilité consiste à y soustraire une « Baseline ». Plusieurs formules sont proposées, et notamment celle de « advantage actor critic » avec l'utilisation de la fonction de valeur d'état (V) comme fonction de base. Intuitivement, cela se résume à savoir à quel point il est préférable de choisir cette action par rapport à l'action générale moyenne de l'état donné. Ci-dessous la formule de l'avantage :

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

Une fois combiné avec la formule précédente, il en résulte la fonction « Advantage Actor Critic » (A2C) ci-après :

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

Pour ajouter le « A » manquant à la technique de A3C il faut mettre en place l'asynchronisation. Dans ce but, les agents (« Worker » ci-dessous) vont itérer avec la technique A2C dans leurs environnements, pour un nombre d'actions $\tau_{max,}$ ou lorsqu'un état final est atteint. Après quoi ils transmettent leurs gradients à l'agent global, et itérer son optimisateur, ce qui revient à mettre à jour le modèle global avec l'exploration de l'agent local. Une fois la mise à jour faîte, l'agent local charge les paramètres globaux et recommence le cycle. Ci-dessous un schéma qui illustre les interactions :

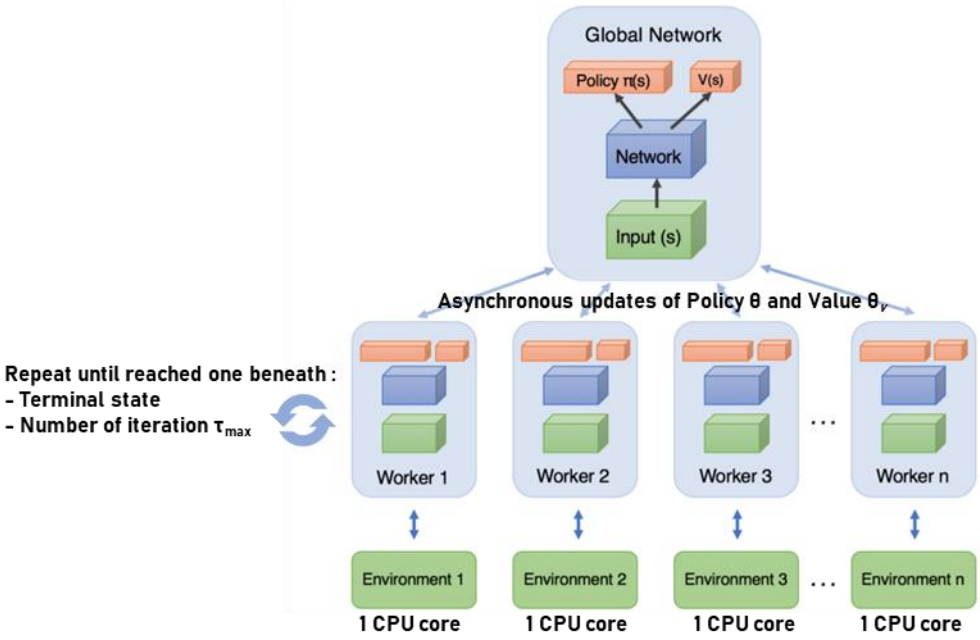

Figure 2 Illustration des interactions de la méthode A3C entre les agents locaux performant A2C et l'agent global

Les chercheurs derrière le papier ont également appliqué un système d'entropie de la politique dans la fonction objective, ceci dans le but d'améliorer l'exploration et d'éviter des convergences vers des solutions sous optimales. Cette entropie prend la forme suivante :

$$\beta \nabla_{\theta'} H(\pi(s_t; \theta'))$$

Avec H l'entropie, et β un facteur de prise en compte de cette régularisation. La forme finale de la fonction objective est la suivante :

$$\nabla_{\theta'} \log \pi(a_t|s_t; \theta')(R_t - V(s_t; \theta_v)) + \beta \nabla_{\theta'} H(\pi(s_t; \theta'))$$

En ce qui concerne l'optimisation, les chercheurs ont mené des essais avec :

- Stochastic Gradient Descent (SGD), avec un momentum
- Root Mean Square Propagation (RMSProp), sans statistiques partagées
- RMSProp, avec statistiques partagées

RMSProp est une optimisation par variation du pas d'apprentissage. Dans le papier, il est annoncé que la technique la plus efficace est RMSProp non centrée et avec les statistiques partagée, car elle présente plus de robustesse. C'est donc cette technique qui sera implémentée lors de la mise en place du réseau.

Maintenant que la synthèse du papier est faite, que les avantages et résultats sont présentés, il est temps de mettre en place l'algorithme que je viens de détailler. Pour cela, il me faut un environnement de travail adéquat.

V. Préparation de l'environnement de travail

Bien que cette section ne soit pas celle qui apporte le plus de valeur ajoutée vis-à-vis de l'étude et la retranscription des compétences acquises ce semestre, elle représente un temps investi *très conséquent* pour mener à bien cette étude. Je vais donc brièvement introduire les problèmes rencontrés ainsi que les solutions apportées.

A. Windows Subsystem for Linux (WSL)

N'ayant pas de compétences de développeur logiciel, j'ai rencontré de nombreuses difficultés pour installer correctement l'environnement, car certains modules sur lesquels reposent les simulateurs des jeux Atari, également utilisé par gym n'ont pas de support Windows.

Afin contourner cette difficulté, j'ai utilisé l'outil récemment introduit de Microsoft : Windows Subsystem for Linux, qui permet d'émuler un noyau sans avoir la nécessité d'avoir une machine virtuelle, ou bien d'intégrer un dual boot. Cette méthode à l'avantage de mettre à disposition un environnement de développement sous Linux tout en consommant moins de ressources que les alternatives existantes.

B. Installation de gym[all]

Après m'être familiarisé avec l'environnement de Linux et avoir mis en place des environnements virtuels de python, j'ai tenté d'installer le module gym[all]. Cependant le module retournait constamment des erreurs des librairies Mujoco et Box2d, qui demandaient une intervention manuelle. De plus Mujoco est une bibliothèque a priori nécessaire pour enregistrer les rendus des simulations des jeux.

À la suite de nombreuses recherches, et puis avoir compris que les licences n'étaient pas libres d'accès, j'ai créé un compte en ligne, et ai reçu une clé d'activation et des fichiers nécessaires à l'installation. Une première étape était franchie, et de nombreuses itérations d'ajouts des librairies manquantes plus tard, la mission de mise en place de l'environnement était terminée.

C. Outils supplémentaires

Ce qui est mis en place dans les sections précédentes n'est qu'un terminal linux avec les librairies utiles pour le fonctionnement de gym et des systèmes d'émulations. En revanche il n'y pas d'outils facilitant l'édition de code, ni de retour visuel qui permette de suivre l'évolution de Mrs Pac-Man dans son univers.

En ce qui concerne l'édition de code, j'ai suivi la recommandation de Microsoft en utilisant l'extension « Remote – WSL » du logiciel Visual Studio Code. Cet outil permet d'utiliser le logiciel comme à son habitude, en sélectionnant l'environnement virtuel de python souhaité, tout en exécutant le code au travers du noyau Linux.

Il a également fallu faire quelques tours de passe-passe pour obtenir un retour visuel. Le sous-système précédemment créé est considéré comme un ordinateur sur le réseau local de Windows. Avec l'utilisation de l'outil « VcXsrv » qui met en place un serveur sur la machine hôte, ce dernier permet de « streamer » les applications graphiques du noyau. Après une première tentative ratée et de nouvelles recherches, il m'a fallu introduire des règles d'exceptions dans la sécurité du pare-feu pour autoriser la communication entrante de l'adresse IP du noyau vers l'hôte, avant d'obtenir gain de cause.

Cette installation a été un grand défi, car elle m'a demandé d'apprendre en détail des notions très diverses de l'univers informatique tel quel la virtualisation, l'environnement linux, et la communication au sein d'un réseau local, sans compter les différents processus de parallélisation intrinsèque à la méthode développés dans cette étude. Avec beaucoup de persévérance, j'ai fini par arriver à bout des problèmes les uns après les autres, et exploiter ces outils pour la conception du réseau.

VI. Mise en place du réseau

Pour développer ce réseau, il est nécessaire de se tenir au principe du papier, et donc de faire fonctionner un agent par cœur de CPU. La plupart des ordinateurs modernes ont au moins 8 cœurs, ce qui permet l'utilisation de 8 agents en parallèle.

La parallélisation est un sujet longtemps étudié, car il permet généralement de raccourcir les temps de calcul, cependant il existe de multiples formes de parallélisation, et elle nécessite de comprendre les fonctionnements basiques d'un processeur, ce qui peut s'avérer être difficilement manipulable. Fort heureusement le Framework PyTorch a sa propre implémentation d'utilisation de multiprocesseur, qui facilitera notre travail pour le traitement des tâches parallèles.

Je vais commencer par introduire la structure générale du code, qui reprend le schéma de la section précédente, puis je rentrerai en détail dans les différentes sous-parties du code. À noter que tous les paquets utilisés dans le code sont issus du Framework Pytorch.

A. La structure générale

Voici ci-dessous la hiérarchie globale des classes et des fonctions appelées lors de l'exécution du script « Asynchronous_Advantage__Actor_Critic.py » :

```
Main------------------------------------Global function
|__evaluate-----------------------------Global function
|     |__A3C_Functions------------------Class
|     |     |__A3C_Neural_Network--------Class
|     |___model_testing-----------------Global function
|     |     |__ action------------------Local function from A3C_Functions
|     |     |__ refresh_state-----------Local function from A3C_Functions
|     |___plot_learning_curve-----------Global function
|
|__ train-------------------------------Global function
      |__ A3C_Functions-----------------Class
            |__ A3C_Neural_Network-------Class
            |__ action------------------Local function from A3C_Functions
            |__ update------------------Local function from A3C_Functions
            |__ refresh_state-----------Local function from A3C_Functions
```

Figure 3 Schéma synthétisant la hiérarchie de l'algorithme A3C tel que conçu dans cette étude

Ce schéma permet de donner la structure générale de l'algorithme tel qu'il est créé ici, ainsi que les relations entre les différentes fonctions et classes éditées pour l'application de la méthode A3C. Ci-dessous, une section descriptive du déroulement.

B. Fonctionnement des classes et fonctions

La fonction `main` va initier le modèle global partagé entre tous les agents locaux, ainsi que les processus en parallèle avec :

- Un cœur réserver à la fonction d'évaluation `evaluate`
- Et les autres cœurs pour héberger les agents et leurs instances de l'environnement qui vont progresser avec la fonction `train`.

La fonction `evaluate` vient évaluer périodiquement le modèle global partagé et créer des sauvegardes des poids du réseau. Pour cela, elle crée une instance de l'environnement, ainsi qu'un agent local qui télécharge les paramètres globaux. L'agent local sera transmis à la fonction `_model_testing` qui retourne le score obtenu pour un essai, ceci est répété pour un nombre de cycles `evaluate_episodes` défini par l'utilisateur. Une fois l'ensemble des scores obtenus, il en est extrait la moyenne avec laquelle est tracé la courbe d'apprentissage via la fonction `_plot_learning_curve`. Cette dernière trace également la moyenne totale et la moyenne sur 5 valeurs pour faire ressortir la tendance du modèle

Pour terminer la boucle, les poids du réseau sont enregistrés sous le nom de « `last_model.pt` » et réalisent une pause pour un temps défini par `time_sleep` avant de recommencer en chargeant les nouveaux paramètres du modèle général. De manière périodique (toutes les 25 évaluations) les paramètres du modèle sont enregistrés sous un nom spécifique afin de pouvoir comparer l'évolution des comportements de Mr PacMan dans son environnement.

La fonction `train` fait exécuter un nombre d'actions τ_{max}, paramétrable, à un agent local dont les paramètres ont été préalablement synchronisés avec ceux de l'agent global. Les actions sont choisies en utilisant la fonction `action` inhérente à la classe `A3C_Functions`, cette dernière permet de calculer l'action selon la politique, la valeur de l'état, la log probabilité et l'entropie. Ces données précédemment créées sont transmises à la fonction `update`, également une classe de `A3C_Functions`, qui calcule les gradients et la rétropropagation. Le gradient est ensuite partagé avec l'agent global, avec lequel l'optimisateur itère. Pour terminer le cycle, une dernière fonction `refresh_state` est appelée pour réinitialiser les éléments associés à la couche LSTM de la classe `A3C_Neural_Network`.

L'ensemble des fonctions inhérentes à la classe `A3C_Functions` s'appuie sur le réseau constitué dans la classe `A3C_Neural_Network`. Cette dernière initie le réseau avec l'ensemble de couches, dans l'ordre suivant (nom de la couche, et dimensions associées) :

- Convolution 2 dimensions : 3 x 210 x 160 -> 16 x 51 x 39
- Normalisation par batch 2 dimensions
- Convolution 2 dimensions : 16 x 51 x 39 -> 32 x 16 x 13
- Normalisation par batch 2 dimensions
- Convolution 2 dimensions : 32 x 16 x 13 -> 32 x 7 x 5
- Normalisation par batch 2 dimensions
- Flatten : 1120

- Long Short Term Memory Cells : 1120 -> 256
- 2 couches linéaires :
  - Une pour la critique : 256 -> 1
  - Une pour la politique : 256 -> 9 (Nombre d'actions possible pour PacMan)

L'idée de ce réseau et de s'appuyer sur la force d'extraction de « features » des réseaux convolutifs pour prendre des décisions sur un espace plus condensé. De plus ce réseau est combiné avec des couches de « LSTM » pour donner de la mémoire sur les conséquences des actions choisies.

Après avoir déroulé le fonctionnement de l'algorithme, ci-dessous les paramètres nécessaires à chaque classe / fonction.

C. Détails et paramètres des classes et des fonctions

La classe `A3C_Neural_Network` qui crée le réseau précédent, s'appuie sur le module de « neural network » du framework PyTorch, pour s'initialiser il lui faut les paramètres suivants :

- `n_channels` : entier, nombre de canaux dans l'image d'entrée, ici 3 pour RGB
- `n_actions` : entier, nombre d'actions possibles, ici 9

La classe `A3C_Functions` qui découle des fonctions pour manipuler le réseau n'a pas d'héritage, pour s'initialiser le premier paramètre est nécessaire, les suivants sont par défaut :

- `env` : l'environnement du jeu enveloppé par la bibliothèque gym
- `lr` : flottant, nombre du pas d'apprentissage initial pour l'optimisateur,
- `gamma` : flottant, facteur de réduction de la récompense immédiate

Maintenant, les paramètres des fonctions inhérentes à cette dernière classe, avec en premier la fonction locale `action` qui a besoin de :

- `state` : tenseur, l'observation de l'environnement, c'est-à-dire l'image
- `training` : booléen, paramètre pour déterminer le retour de la fonction

Suivi de la fonction locale `update`, qui s'appuie essentiellement sur les valeurs retourner par la fonction `action` précédente :

- `config` : dictionnaire, ensemble des paramètres défini par l'utilisateur
- `rewards` : liste, ensemble des récompenses
- `values` : liste, ensemble des valeurs d'états
- `log_probs` : liste, ensemble des log probabilités
- `entropies` : liste, ensemble des entropies
- `R` : liste, ensemble des valeurs

La fonction locale `refresh_state` n'a pas besoin de paramètres, et réinitialise les items des couches LSTM.

La fonction globale `_model_testing`, qui retourne la récompense obtenue pour un modèle donné, fonctionne avec :

- `controller` : classe `A3C_Functions`, c'est un agent local
- `env` : l'environnement local du jeu enveloppé par la bibliothèque gym

La fonction globale `_plot_learning_curve`, qui trace l'évolution de l'apprentissage du modèle, prend en charge :

- `rewards` : liste, ensemble des récompenses
- `config` : dictionnaire, ensemble des paramètres défini par l'utilisateur

La fonction globale `evaluate`, qui teste périodiquement le modèle a besoin des paramètres suivants :

- `config` : dictionnaire, ensemble des paramètres défini par l'utilisateur
- `shared_model` : classe `A3C_Functions`, c'est un agent global

Avant de finir avec la fonction principale, la fonction globale train se base sur les mêmes paramètres que la fonction précédente, cependant elle a besoin du numéro du processeur sur lequel elle s'entraîne.

- `config` : dictionnaire, ensemble des paramètres défini par l'utilisateur
- `shared_model`: classe `A3C_Functions`, c'est un agent global
- `rank` : entier, numéro du processeur sur lequel la fonction s'exécute

La fonction globale `main` enveloppant l'ensemble des autres fonctions prend en variable le dictionnaire qu'elle passera à l'ensemble des autres fonctions :

- `config` : dictionnaire, ensemble des paramètres défini par l'utilisateur

Voici les variables essentielles ajustables dans le dictionnaire config :

- `max_workers` : entier, nombre d'agents en parallèle
- `evaluate_episodes` : entier, nombre d'épisodes évalués pour la fonction `evaluate`
- `t_max` : entier, nombre maximal d'actions exécutable par les agents
- `value_loss_factor` : flottant, facteur de réduction de la valeur dans la fonction loss
- `entropies_factor` : flottant, facteur de réduction de l'entropie dans la fonction loss

Cette synthèse permet d'expliciter les variables attendues pour un déroulement sans accroc de l'algorithme. Ci-après une comparaison des résultats obtenus pour différents entraînements.

VII. Résultats

Dans l'ensemble, les résultats obtenus ne sont pas aussi prometteurs que le papier l'a annoncé, dans la mesure où il n'y a pas d'erreur majeure dans la conception du code. Il faut également plusieurs heures d'entraînement pour tirer une conclusion sur la progression des modèles, de plus l'entraînement mobilise une grande partie des ressources de l'ordinateur, ce qui rend l'apprentissage en ligne pas très évidente à « fine-tuner » pour un particulier.

Ci-dessous je vais comparer 3 modèles entraînés. À noter que pour chaque entraînement, une dizaine d'évaluations prend environ 1 heure, et chaque évaluation se fait sur une moyenne de 10 épisodes et intervalle de pause d'1 minute. L'ensemble des figures sont disponibles en annexe, dans le dossier « Trained model » et le sous-dossier correspondant au modèle.

A. Premier modèle

Tout d'abord, voici les paramètres utilisés et à côté la courbe d'apprentissage obtenue :

- `max_workers` : 10
- `t_max` : 150
- `value_loss_factor` : 0.5
- `entropies_factor` : 0.1

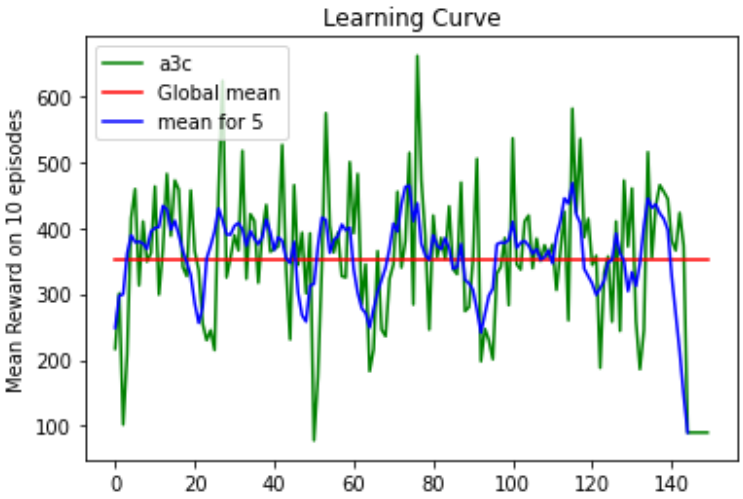

Figure 4 Courbe des moyennes des scores pour chaque évaluation du premier modèle

Il est notable que la fin de la courbe plonge et stagne, c'est un problème que je n'ai su identifier et que je retrouve lors de l'entraînement du second modèle. Malgré ceci, le modèle ne semble pas progresser énormément, et atteint vite une position stationnaire autour de la moyenne, toutefois cette dernière sera la plus grande obtenue lors de ces trois tests. À partir de ce constat, et pour tenter d'améliorer les performances, pour le second modèle j'ai ajouté deux agents et ai laissé plus d'actions pour explorer l'espace des solutions. J'ai également augmenté la prise en compte des valeurs dans la fonction « loss », afin d'observer l'influence de ce paramètre sur la progression du modèle.

B. Second modèle

Voici les paramètres utilisés et la courbe d'apprentissage du second modèle :

- `max_workers` : 12
- `t_max` : 200
- `value_loss_factor` : 0.8
- `entropies_factor` : 0.1

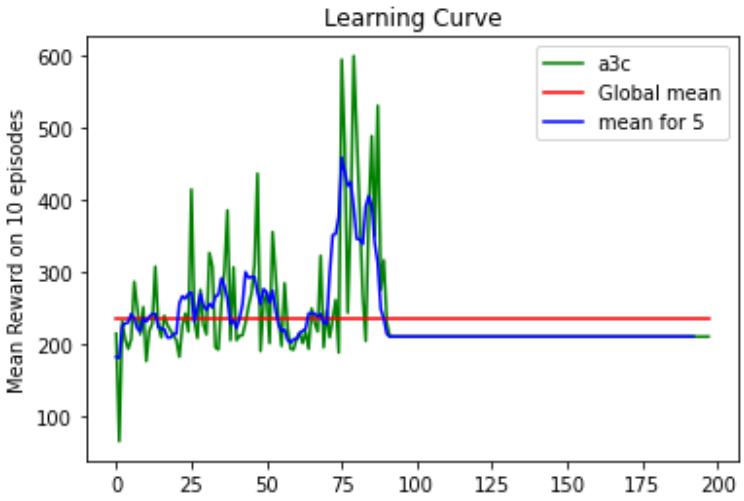

Figure 5 Courbe des moyennes des scores pour chaque évaluation du second modèle

Lors de l'entraînement j'ai une nouvelle fois rencontré ce problème inexplicable d'évaluation stagnante, si cette section est retirée la moyenne globale se situe autour de 280 points. Hormis cet incident, l'ensemble des changements apportés semble donner une courbe d'apprentissage qui ressemble plus à une pente ascendante, avec des pics et des minimums qui ont une tendance à progresser au fur et à mesure des évaluations, comme l'en témoigne la moyenne glissante sur 5 valeurs.

Il semble également logique que plus on laisse d'actions aux l'agents plus ceux-ci sont en mesure d'aller chercher des récompenses. De plus, l'ajout de 2 agents permet théoriquement d'augmenter l'ensemble des explorations réalisées, ce qui peut n'être que bénéfique dans le cas de la méthode A3C. Dans la continuité de cette réflexion, le troisième modèle aura deux agents de plus, et un nombre d'actions doublé.

Il est encore difficile de comprendre l'effet du facteur de la « loss » de la valeur sur l'apprentissage du modèle, c'est pourquoi je vais garder une valeur similaire lors du troisième test.

En relisant le papier, je me suis aperçu que le facteur d'entropie utilisé lors de leurs études était 10 fois plus petit que celui choisi et j'ai donc appliqué cette même valeur, toujours dans l'objectif de trouver les paramètres d'apprentissage adéquats.

C. Troisième modèle

Voici les paramètres utilisés et la courbe d'apprentissage du troisième modèle :

- `max_workers` : 14
- `t_max` : 400
- `value_loss_factor` : 0.9
- `entropies_factor` : 0.01

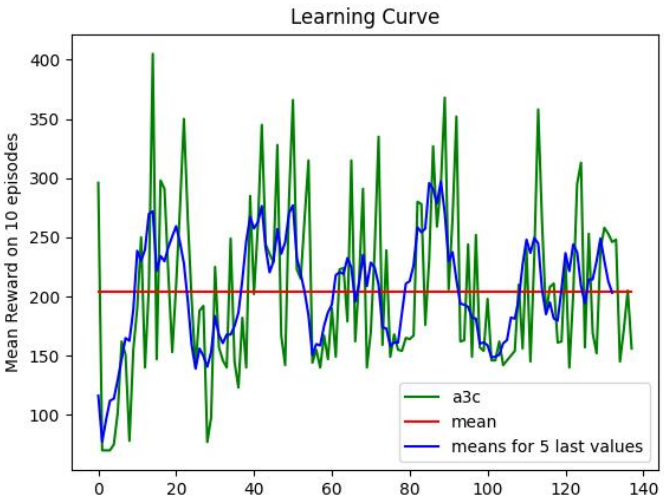

Figure 6 Courbe des moyennes des scores pour chaque évaluation du troisième modèle

Dans ce cas de figure, malgré l'ajout de 2 agents supplémentaires, la moyenne totale est la plus basse, ce qui me laisse penser que le nombre d'actions à exécuter est trop grand pour que le modèle commence à converger sur un temps d'entraînement similaire aux deux autres tests. De plus, les valeurs gravitent autour de la moyenne, et la moyenne glissante laisse indiquer une très légère pente ascendante de progression, cependant ce serait une donnée à vérifier avec plus d'entraînement. Le fait d'avoir baissé le facteur d'entropie laisse supposer que le modèle encourage moins la progression, ce qui se témoignerait par la variance des résultats : pour deux résultats consécutifs, il y a souvent une valeur qui pointe vers le haut, tandis que la suivante pointe vers le bas, ce qui conclut un manque de stabilité.

En ce qui concerne le facteur de la « loss » de la valeur, il est encore une fois difficile d'en déterminer son impact à cause des changements majeurs du nombre d'actions, et du facteur d'entropie.

D. Bilan

L'ensemble de ces tests ne sont pas concluants s'ils sont comparés aux résultats du papier après une dizaine d'heures d'entraînement, cependant le second modèle laisse à penser qu'avec plus de temps il serait en mesure de faire des performances convenables. Peut-être que le temps d'entraînement n'est pas suffisamment long pour que les agents apprennent une solution viable. Une amélioration à apporter de ce point de vue là serait d'initialiser les poids du réseau. Il n'est également pas possible d'évaluer l'aspect linéaire de progression par rapport au nombre de cœurs en regard de la différence des tests menés dans cette étude. Idéalement, il faudrait réaliser différentes expériences pour trouver des paramètres qui permettent une meilleure progression des agents au travers de la méthode A3C.

En conclusion de cette section, les promesses tenues par le papier ne s'appliquent pas très bien dans ce cas d'étude avec le jeu Atari2600 : Mrs PacMan.

Conclusions

L'ensemble de cette étude m'a énormément enseigné que ce soit sur la théorie de l'apprentissage par renforcement, de virtualisation, de parallélisation des taches, mais également beaucoup de mise en pratique, j'ai pu affiner ma connaissance du framework PyTorch et en manipuler des concepts abstraits. C'était donc un défi intellectuel de taille et très intéressant.

Cependant le modèle longuement étudié ici est critiquable, en effet il faut tout de même une capacité calculatoire importante pour être en mesure de faire tourner l'entraînement, et pour des résultats quelque peu mitigés. Si toutefois les résultats étaient concluants, combien de temps faudrait-il pour que l'ordinateur soit en mesure de jouer à Mrs Pac-Man correctement ? Et n'est-il pas question ici de surapprentissage ? Quelle est la capacité d'un modèle entraîné sur un jeu à jouer aux autres ? Je trouve également qu'il manque de temporalité au sein du réseau constitué, et que l'introduction d'un réseau de neurones récurrent serait intéressante.

Lors de recherches, je suis également tombé sur un article des mêmes équipes de recherche de google qui faisait une critique de leur papier : Le variant synchrone, c'est-à-dire que la synchronisation des paramètres se fait une fois que tous les agents ont fini leurs cycles et non pas individuellement comme c'est le cas ici, ne présentent pas plus d'inconvénients et permet d'être exploité sur des cartes graphiques, ce qui en fait d'après l'article une méthode plus performante que l'A3C.

Autant d'éléments de réflexions qui me sont venus tout au long de mon travail, et qui nécessiteraient d'être exploré pour approfondir mes connaissances dans ce domaine.

Pour terminer sur une ouverture, voici quelques-unes des nombreuses pistes d'améliorations envisageables pour poursuivre ce projet :

- L'initialisation des poids du réseau comme mentionné dans le bilan de la section précédente
- Donner des paramètres différents pour chaque agent, avec par exemple un agent qui va exécuter un petit nombre d'actions, et un autre agent qui en exécute un grand nombre
- En mettant en place une version asynchrone du « Duelling Deep Q Network », qui d'après le papier est la technique « naturellement » plus performante pour notre jeu