Eléments logiciels pour le traitement de données massives

Asynchronous Methods for Deep Reinforcement Learning

Zakarya Ali
Quentin Spalla

5 février 2018

Introduction

"Asynchronous Methods for Deep Reinforcement Learning" est un article publié en 2016 par Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver et Koray Kavukcuoglu, chercheurs chez Google DeepMind. Les auteurs y présentent les plus récentes avancées en ce qui concerne l'apprentissage renforcé et ses algorithmes comme le Deep Q-learning (DQN) et surtout l'Asynchronous Actor-Critic Agents (A3C). Dans le cadre de notre projet, nous nous concentrons sur l'implémentation de ce dernier et sa parallèlisation. Nous allons le définir, expliquer sa mise en place et présenter les résultats obtenus suite à son entrainement sur le jeu Atari Breakout.

1 Asynchronous Actor-Critic Agents

Alors que le Q-Learning a longtemps été considéré comme un très bon outil, la sortie de cet article et les résultats mis en avant ont boulversé le paysage de l'apprentissage renforcé. Nous n'expliquons pas ici cette méthode, seulement l'A3C, qui montre de bien meilleurs performances avec une puissance de calculs beaucoup moins importante. Elle peut simplement être éxecuté sur plusieurs CPU.

La figure 1 résume le fonctionnement de la méthode. Dans l'A3C il y a un réseau global, et de multiples agents qui ont chacun leur ensemble de paramètres. Ils interagissent tous avec une copie de l'environnement d'apprentissage en même temps. Cette approche asynchrone permet d'explorer une très grande diversité de situations étant donné que l'expérience de chaque agent est indépendante.

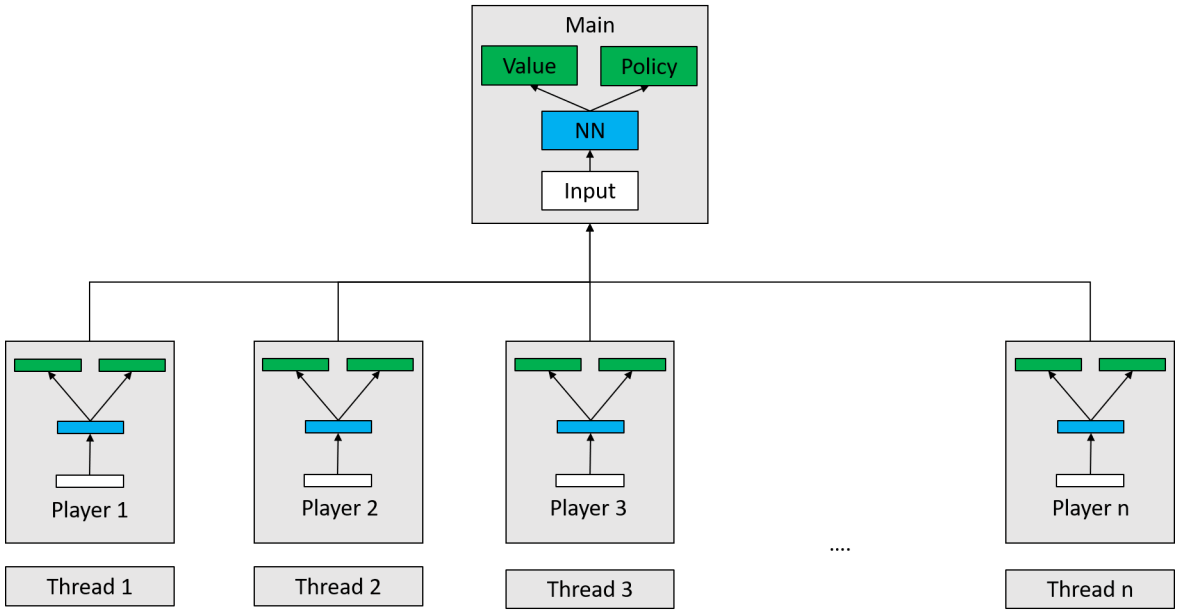

FIGURE 1: A3C - Schéma

On note 2 approches dans le domaine de l'apprentissage renforcé : "value-iteration" (exemple : Q-learning), "policy-iteration" (Policy Gradient). A3C combine ces 2 approches.

En effet, on estime à la fois $V(s)$ (la valeur de se trouver dans un état s) et la policy $\Pi(s)$ (quel futur état s choisir). Ces 2 parties sont les couches de sorties d'un réseau de neurones. Chaque agent utilise la valeur estimée (la critique) pour mettre à jour la policy.

La figure 2 présente l'algorithme A3C tel qu'écrit dans l'article. Les points importants à noter sont :

— θ : poids de la Policy
— θ_v : poids de la Value
— s_t : état (image) à la date t
— a_t : action choisie à la date t grâce à la policy $\Pi(a_t|s_t;\theta)$
— R : reward obtenu en choisissant l'action a_t à l'état s_t

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

FIGURE 2: Algorithme A3C issu de l'article

L'algorithme A3C pour un thread est assez simple. Le thread possède un copie des poids du réseau de neurones (NN). Il va procéder à une descente de gradient au bout de t_max itérations (ou que s_t soit un état terminal) sur ces itérations qui formeront sont minibatche. Le thread met alors à jour les poids partagés par tous les threads de manière asynchrone.

2 Implémentation

Le code développé pour ce projet est disponible sur GitHub.

2.1 Arcade Learning Environment

Un prérequis nécessaire pour éxécuter l'entrainement de notre algorithme est The Arcade Learning Environment. Cet outil nous permet d'entrainer le modèle sur une version émulée

du jeu Atari 2600 Breakout (voir figure 3). La plateforme nous renvoie les images du jeu, la liste des actions disponibles à chaque image, l'état du jeu (game over ou non, score...) et on peut exécuter les actions retenues par nos agents.

FIGURE 3: Atari Breakout

2.2 Structure du code

Le projet est développé en Python. Les librairies utilisées sont : multiprocesssing, numpy, scikit-image, pickle et ctypes.

Pour simplifier notre projet, nous l'avons divisé en plusieurs fichiers :

- ale.py possède la classe ALEGame qui permet de faire le lien entre Python et l'émulateur de jeux Atari. C'est grâce à cette classe et notamment sa méthode `preprocess_image`() que l'on va pouvoir récupérer l'image du jeu. Cette image est l'input de notre réseau de neurones.

- constants.py référence tous les paramètres nécessaires du projet. On peut notamment modifier le nombre de threads que l'on souhaite utiliser, ou bien le taux d'apprentissage (learning rate).

- tools.py est un fichier support référençant des fonctions utiles. On y trouve par exemple des fonctions permettant aux couches de convolutions de s'exécuter à la fois en forward et en backward. Le fichier présente également des fonctions de transformation (de liste à vecteur ou inversement).

- layer.py est le fichier qui recense toutes les classes représentant les couches de notre réseau de neurones. Nous y trouvons donc une classe ConvLayer pour les couches de convolution mais également des classes FCLayer (pour les couches fully connected) ou ReluLayer,

etc. Ces classes ont des fonctions communes telles que `backward`() ou `forward`() qui permettent au réseau de neurones de s'exécuter. Notons que nous avons choisi dans ce projet de coder l'intégralité du réseau de neurones nous même. Ainsi, aucun package type tensorflow, pyTorch, Keras ou autre n'a été utilisé. En faisant cela, nous avons pu bien comprendre le fonctionnement interne d'un réseau de neurones et l'ensemble de ses rouages. Ce fut un travail long et potentiellement source d'erreurs, mais fût également très intéressant et extrêmement formateur.

- network.py contient la classe NNetwork qui permet la création d'un réseau de neurones profond ainsi que diverses méthodes utiles au réseau telles que `update_weights_bias`() qui met à jour les poids du NN, `get_all_weights_bias`() qui récupère les poids, `backpropag_pi`() qui fait la backpropagation, `get_intermediate_values`() qui récupère les valeurs à chaque neurone (cela nous permet de faire fonctionner plus rapidement la backpropagation en minibatches).

- player.py possède la classe ActorA3C qui représente en fait le joueur d'une partie d'un jeu Atari. La méthode `process`() va représenter l'algorithme S3 (figure 2) et est donc la phase d'apprentissage pour un actor-thread. C'est cette méthode qui, utilisée avec plusieurs threads, va entraînter notre NN et mettre à jour les poids de manière asynchrone. On présente aussi la méthode `test_play`() qui permet simplement au code de jouer au jeu à partir de poids que nous lui proposons pour le NN. On peut checker ainsi les scores obtenus à chaque partie par le NN entraîné.

- async.py est le fichier recensant la mémoire partagée par tous nos threads. Cette mémoire se trouve dans la classe SharedWeights sous la forme d'un multiprocessin.sharedctypes.Array. Grâce à cette mémoire partagée, l'algorithme A3C peut fonctionner normalement. Ainsi chaque, thread, mettra à jour par une descente de gradient ces poids partagés sans qu'intervienne un seul lock. En effet, l'algorithme A3C a la particularité d'être lock-free et de tourner sur CPU. Cette classe possède de plus la méthode `gradient_descent`() qui est une descente de gradient classique, sans aucune optimisation de type RMSProp ou Adam.

- train.py est le fichier main qui va permettre l'apprentissage. On peut initialiser le NN à partir de poids aléatoires ou bien, si on a des poids enregistrés sous la forme d'un vecteur dans un fichier .pkl, les utiliser pour initialiser le NN .

- test.py est le second fichier main qui permet de récupérer les scores obtenus en jouant. Il faut lui indiquer les poids avec lesquels le NN doit fonctionner. Il va appeler la méthode `test_play`() que nous avons présentée plus haut. Qui est semblable à la méthode d'apprentissage `process`() mis-à-part qu'aucune backpropagation ou descente de gradient n'est effectuée.

2.3 Parallélisation

CPython, la version que nous utilisons de Python, est basé sur le Global Interpreter Lock (GIL). Cet outil permet un gain de vitesse pour les programmes en single-thread, en

revanche, il empêche la possibilité de faire du multithreading. On utilise la librairie multiprocessing pour mettre en place la parallèlisation. Elle contourne ce problème en utilisant de sous-processus à la place des threads. Nous appelons donc dans ce rapport des thread les joueurs qui vont mettre à jour de manière asynchrone les poids du NN, mais dans le code il s'agit en réalité de process car les thread en Python ne peuvent tourner que sur un seul coeur. Tandis que le package multiprocessing nous permet d'utiliser plusieurs coeurs de la machine pour ainsi retrouver le but de l'algorithme A3C développé par DeepMind.

Comme énoncé dans l'article, chaque thread possède un agent qui calcule indépendemment des autres les gradients des poids de la policy et la value. Toutes les t_{max} images ces gradients vont permettre la mise à jour des poids au niveau global. Ces mises à jour n'ont pas de condition, c'est-à-dire que n'importe quel thread peut mettre à jour les poids à n'importe quel moment. Ainsi ces poids représentés dans un vecteur sont une varibale partagée que l'on retrouve dans notre code comme `self.shared_theta` dans la classe SharedWeights du fichier async.py. Cette variable partagée par tous nos threads sera mise à jour régulièrement sans qu'aucun verrou ne soit utilisé. En effet, l'algorithme A3C étant asynchrone, il est ici lock-free et donc aucun thread ne peut s'approprier les poids le temps qu'il intervienne dessus. Une fois qu'un thread a terminé sa descente de gradient et qu'il a mis à jour la variable partagée, il va récupérer une copie de cette dernière (les poids de notre NN) qui sera indépendante de tout futur changement jusqu'à ce qu'il récupère à nouveau une copie de la variable partagée. Cet algorithme asynchrone implique également que dès lors qu'un thread a mis à jour la variable partagé, tous les autres thread exécutant leur apprentissage sur une copie de l'ancienne variable partagée se retrouvent donc avec une copie obsolète.

3 Résultats

Notre code permet d'enregistrer les poids entraînés dans un fichier .pkl pour après faire des tests. Malheureusement, nous n'avons jusqu'à présent pas eu assez de temps pour parvenir à entraîner suffisamment l'algorithme pour obtenir des résultats satisfaisants. En effet, quelques erreurs de code ou bien de compréhension de l'algorithme (notamment sur l'équation de la fonction de perte du NN) nous ont retardés pour faire des tests concluants. De plus, nous avons volontairement omis quelques optimisations techniques de l'article telles que l'optimisation de la descente de gradient grâce à RMSProp, l'entropy de la fonction de perte ou bien l'amoindrissement de la learning rate en fonction du nombre d'itérations. Cela implique donc un algorithme plus lent que celui original de DeepMind et explique donc que nous ne pouvons présenter des résultats.

Conclusion

Nous avons vu la force de l'A3C, une méthode d'apprentissage renforcé dont l'implémentation se révèle accessible, et la parallélisation relativement simple. Malgré tout, nos résultats sont encore perfectibles. On peut, par exemple, modifier notre méthode de descente de gradient, pour utilser la méthode optimisé de l'article : RMSprop. Il est également possible faire

tourner l'algorithme sur plus de temps, ou plus de threads... Pour gagner en vitesse, il est également possible de réécrire le projet en C++. On peut également essayer d'appliquer cet outil à d'autres domaines, par exemple, le secteur du médical et le traitement des patients. Enfin en dernier point il convient de mentionner les difficultés que nous avons rencontrées. Tout d'abord, en décidant de coder de A à Z le réseau de neurones, nous avons du beaucoup nous documenter car l'article de base ne suffisait pas. De plus, pour la parallélisation, Python étant un langage haut niveau et l'A3C ne nécéssitant pas l'utilisation de machine virtuelle, il était trop compliqué pour nous de distribuer nous même notre algorithme par l'intermédiaire de Python. Nous avons essayé mais nos tentatives sont toutes restées infructeuses. Enfin, comme expliqué précédemment, nous ne sommes pas encore parvenus à obtenir des résultats concluants et cela notamment car il est nécessaire de laisser travailler plusieurs heures le réseau de neurones.