Deep A3C
Projet pour l'UE RCP211

PHELIZOT Yvan
2021 - 2022 / Semestre 1
RCP211 - Intelligence artificielle avancée

le cnam

Au cours de ces dernières années, des progrès importants ont eu lieu dans le domaine de l'Intelligence Artificielle (IA), notamment des réussites dans les jeux. On peut penser à l'algorithme développé par Google Deepmind qui a été capable de battre le meilleur joueur humain, prouesse que l'on pensait irréalisable jusqu'alors. De même, Deepmind a aussi conçu une IA très efficace, AlphaStar, pour le jeu StarCraft2 qui a montré des résultats très impressionnants face aux meilleurs joueurs de la discipline. Dans les deux cas, ces programmes s'appuient sur l'apprentissage par renforcement (*Reinforcement Learning* en anglais) et les dernières avancées dans les réseaux de neurones profonds ("*Deep*"). Plusieurs modèles issus de ce domaine de recherche existent. Dans le cadre du projet pour l'UE RCP211, j'ai mis en œuvre un algorithme basé sur le modèle "Acteur/Critique" mettant à jour les paramètres de manière asynchrone. Comme pour AlphaGo ou AlphaStar, cet algorithme a été appliqué à un jeu très connu: Pacman. Nous présenterons, dans un premier temps, le problème que nous cherchons à résoudre. Puis, l'architecture retenue, Deep A3C, sera décrite. Enfin, les différents résultats obtenus seront présentés et analysés.

Le code source du projet est disponible ici: https://github.com/cotonne/rcp211

Table des matières

Bibliographie

Mnih, Volodymyr. "Asynchronous Methods for Deep Reinforcement Learning." *ICML*, vol. 2016, 2016, https://arxiv.org/abs/1602.01783.

Mnih, Volodymyr. "Human-level control through deep reinforcement learning." *Nature*, vol. 518, 2015, pp. 529–533, https://www.nature.com/articles/nature14236.

Paszke, Adam. *REINFORCEMENT LEARNING (DQN) TUTORIAL*. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

Wang, Ziyu. "Dueling Network Architectures for Deep Reinforcement Learning." https://arxiv.org/abs/1511.06581.

I) A3C appliqué à PacMan

I.1) Apprentissage par renforcement, A3C

L'apprentissage par renforcement est un domaine de l'intelligence artificielle où un agent doit apprendre la meilleure politique possible pour maximiser son gain à partir de ces expériences passées. Plusieurs méthodes existent qui permettent d'évaluer les gains et de choisir la meilleure politique.
Sur un temps discret, à chaque instant t, l'agent reçoit un état S_t en entrée. De cet état, il déduit la meilleure action a_t possible selon la politique π en cherchant à trouver l'équilibre en exploration des états possibles (pour éviter de suivre toujours le même chemin) et maximisation du gain. Une fois l'action choisie et appliquée à l'environnement, l'agent reçoit une récompense r_{t+1} et l'état évolue.
La formule du gain que l'on souhaite maximiser est la suivante:

$$G_t = \Sigma_{k=0} \gamma^k r_{t+k}$$

Une fois un épisode terminé, il est possible de connaître le gain final. Pour guider les choix de l'agent, on utilise une estimation de ce gain avec la fonction valeur $v_\pi(S_t)$ (réciproquement la fonction état-valeur $q_\pi(S_t, a_t)$. Plusieurs formules peuvent être utilisées, notamment TD(0):

$$v_\pi(S_t) \leftarrow v_\pi(S_t) + \alpha(r_{t+1} + \gamma * v_\pi(S_{t+1}) - v_\pi(S_t))$$

On appelle "Erreur TD", la partie $\delta_t = r_{t+1} + \gamma * v_\pi(S_{t+1}) - v_\pi(S_t)$.
Lors de ce projet, nous avons implémenté la méthode "Acteur/Critique" (Actor/Critic en anglais). Cette méthode apprend à la fois la fonction valeur et la politique. Cette méthode dérive d'un ensemble de méthodes s'appuyant sur le théorème du gradient de la politique (Policy Gradient Theorem en anglais) qui cherche à optimiser la fonction état par montée de gradient des paramètres de la politique, pondéré par le gain. Pour réduire la variance lors de l'estimation de la politique, on remplace le gain par l'avantage. Or, l'erreur TD est un estimateur non biaisé de l'avantage, que l'on pénalise avec l'entropie pour régulariser le résultat.
L'algorithme de la méthode Advantage Actor/Critic (A2C) est:

```
def A2C():
  Initialiser s, θ, w
  Sélectionner l'action a
  Pour chaque pas:
    Appliquer l'action a à l'état s
    Obtenir la récompense r et l'état s'
    Sélectionner l'action suivante a'
    δ = r + γ * q_w(s',a') - q_w(s, a)
    Θ = θ + α * ∇_θ log π_θ(s,a) * q_w(s,a)
    W = W + β*δΦ(s,a)
    S = s', a = a'
```

La méthode Actor/Critic implique que la fonction valeur et la politique soit dérivable. Dans le cas où ces fonctions n'ont pas d'écriture analytique évidente ou que le nombre d'états/actions soit trop important, il est possible de s'appuyer sur les réseaux de neurones qui sont des fonctions dérivables.
Dans cet algorithme, un seul agent exécute un épisode. Pour accélérer le calcul, il est possible d'avoir plusieurs agents et d'avoir plusieurs épisodes en parallèle. Chacun des agents possède des poids partagés qui sont mis à jour de manière asynchrone.
L'algorithme devient:

```
def A3C():
  Initialiser θ, w à partir des poids partagés
  Initialiser s
  Sélectionner l'action a
  Pour chaque pas:
    Appliquer l'action a à l'état s
    Obtenir la récompense r et l'état s'
    Sélectionner l'action suivante a'
    δ = r + γ * q_w(s',a') - q_w(s, a)
    dΘ = dθ + α * ∇_θ log π_θ(s,a) * q_w(s,a)
    dW = dW + β*δΦ(s,a)
    S = s', a = a'
  Retourner dΘ, dW
Exécuter en parallèle A3C()
Mettre à jour de manière asynchrone Θ avec dΘ et W avec dW
```

Nous avons défini brièvement l'apprentissage par renforcement et la méthode A3C qui est mise en œuvre dans le cadre de ce projet. Nous allons voir comment nous pouvons mettre concrètement en œuvre cette méthode dans le cadre du jeu Pac-Man.

I.2) Pacman

Pac-Man est un jeu vidéo sorti au Japon en mai 1980. Il a d'abord été conçu pour les bornes d'arcade et a rapidement été porté sur de nombreuses consoles dont l'Atari 2600. Le joueur contrôle un personnage, Pac-Man, qu'il déplace dans un labyrinthe. Pour avoir un maximum de points, Pac-Man doit manger des "pac-gommes" et des fruits comme des cerises. Sur le plateau se trouvent aussi quatre fantômes qui poursuivent Pac-Man. Si un des fantômes touche Pac-Man, le jeu s'arrête. Cependant, si Pac-Man mange une cerise, les fantômes peuvent alors être mangés par Pac-Man pendant un court laps de temps.

Le lien avec l'apprentissage par renforcement est direct. Les récompenses sont les points accumulés quand Pac-Man réussit à manger un pac-gomme, un fruit ou un fantôme. Les actions possibles sont celles de Pac-Man (haut, bas, gauche, droite). La politique à trouver est celle qui nous permettra de manger le plus sans se faire toucher par un fantôme.

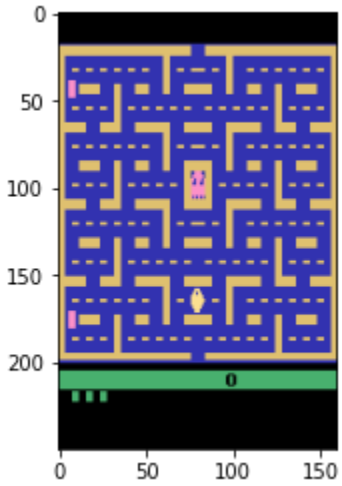
Pour implémenter notre agent ou programme jouant à Pac-Man, nous avons utilisé la librairie Atari Learning Environment qui facilite le développement d'applications d'IA pour les jeux Atari. Cette librairie facilite l'intégration de jeux Atari dans un programme Python. Elle met à disposition une API pour charger un jeu, effectuer une action et obtenir le score.
Le code suivant présente un programme qui joue de manière aléatoire à un jeu Atari:

```
from ale_py import ALEInterface
from numpy.random import choice

ale.loadROM("path/to/rom")
possible_actions = ale.getLegalActionSet()

total_reward = 0
while not ale.game_over():
     action = choice(possible_actions)
     reward = ale.act(a)
     total_reward += reward
print("Final score: %d" % (total_reward))
```

La librairie fournit deux fonctions qui retourne une représentation du jeu: soit les valeurs de la mémoire avec getRAM, soit l'écran de jeu avec **getScreenRGB** ou **getScreenGrayscale**.



Exemple d'écran pour Pac-Man.

II) Implémentation

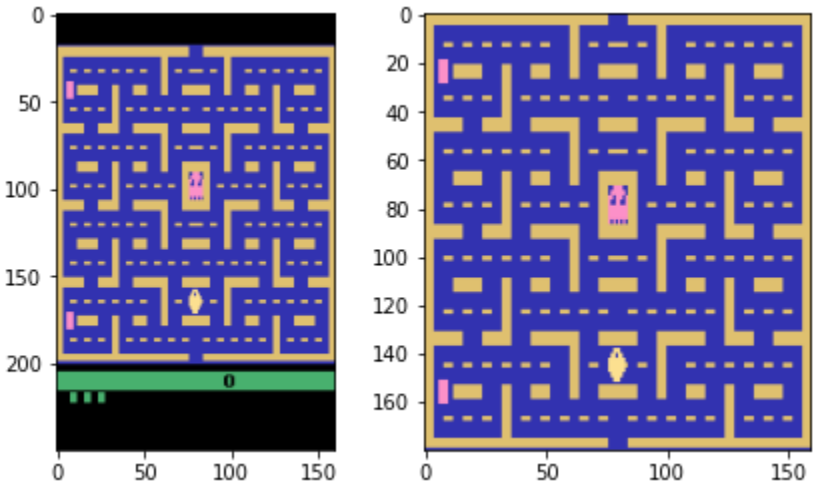
Comme indiqué précédemment, les réseaux de neurones sont des bonnes fonctions candidates lorsque les fonctions valeur ou politique ne peuvent être facilement définies.

II.1) Architecture Deep A3C

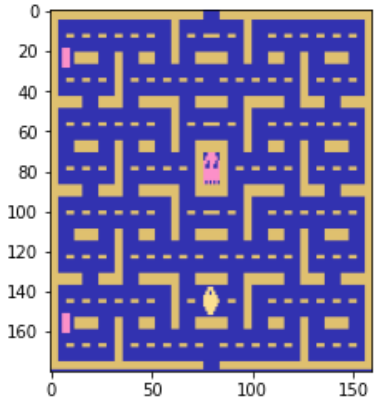
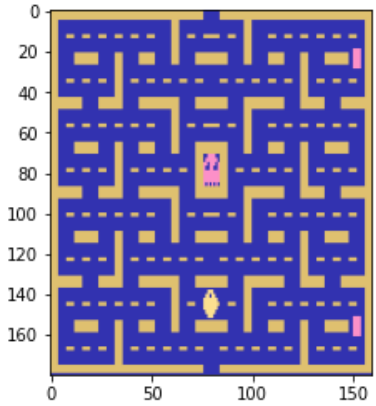
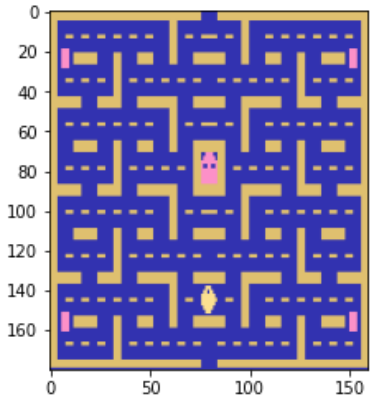
Dans le cadre du projet, les deux fonctions d'états **getRAM** et **getScreenRGB** ont été utilisées. Des réseaux de neurones sont utilisés pour représenter les fonctions état et politiques. Il diffère selon le type de données en entrée.

a) Pré-traitement

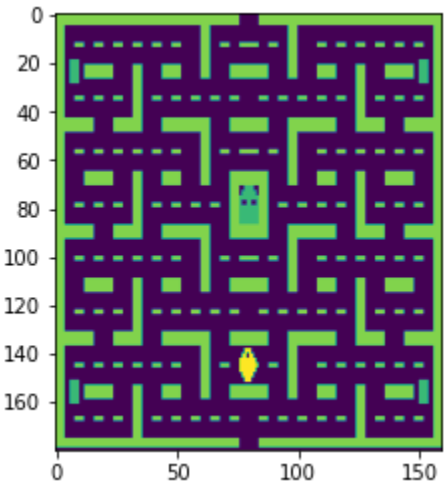
Cette partie décrit le pré-traitement effectué pour simplifier l'image. L'image en entrée est une image RGB de 250 pixels de hauteur sur 160 pixels de large. Pour accélérer les traitements, il est préférable de réduire le plus possible les données en entrée. Une image optimisée concentre mieux l'information et permet ainsi une convergence plus rapide du réseau.
Plusieurs actions ont été menées. Tout d'abord, l'image comprend deux barres horizontales qui n'apportent pas beaucoup d'informations. L'image est donc coupée (crop en anglais).



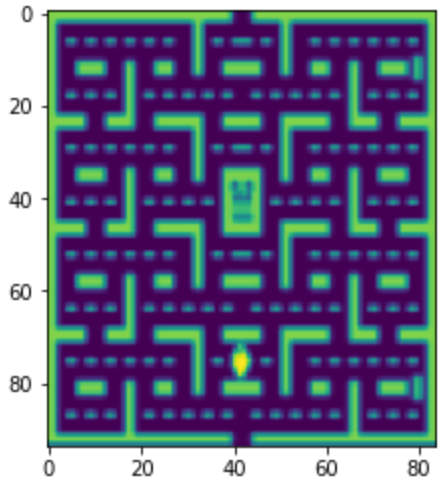
Ensuite, pour des raisons d'optimisation liées à la plateforme Atari, toutes les images ne contiennent pas tous les éléments. Pour afficher tous éléments, les deux dernières images sont fusionnées:

		
Première frame avec les cerises à gauche	Frame suivante avec les cerises à droite	Image fusionnée

Chaque pixel de l'image est composé à partir d'un mélange des couleurs Rouge, Bleu et Vert (les trois canaux RGB). On retrouve sur chaque canal les différents éléments composant le jeu. Pour optimiser encore plus l'image, il est possible de ne garder qu'une dimension. Plusieurs solutions sont possibles. On peut ne conserver qu'un canal ou utiliser des niveaux de gris. Ici, les canaux ont été remplacés par la luminance selon la formule suivante: Y=0.2126R+0.7152G+0.0722B. On obtient alors le rendu suivant:

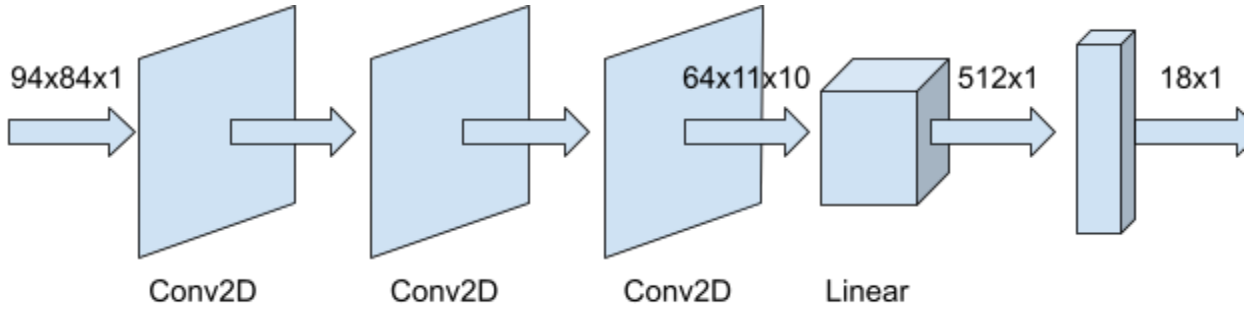


Enfin, l'image est redimensionnée à la taille 94x84:



b) Fonctions Actor et Critic pour l'image

L'état en entrée est une image, les réseaux vont intégrer une couche de convolution. Les fonctions Actor et Critic sont construites selon les architectures de réseaux de neurones convolutifs. Leurs architectures sont similaires à celles décrites dans l'article (Mnih #).. Les fonctions ont trois couches de convolution avec une fonction d'activation ReLU disposées de manière séquentielle suivie d'une couche entièrement connectée disposée de manière séquentielle.
Pour la fonction valeur, la sortie est de dimension 1 et correspond à la valeur du gain.
Pour la fonction politique, la sortie est de dimension 18, soit le nombre d'actions possibles sur une console Atari. Chaque valeur du vecteur est égale à la probabilité de sélectionner cette action. On utilise la fonction softmax pour conditionner les valeurs du vecteur de sortie. On choisit alors l'action selon une distribution multinomiale construite à partir du vecteur de sortie.

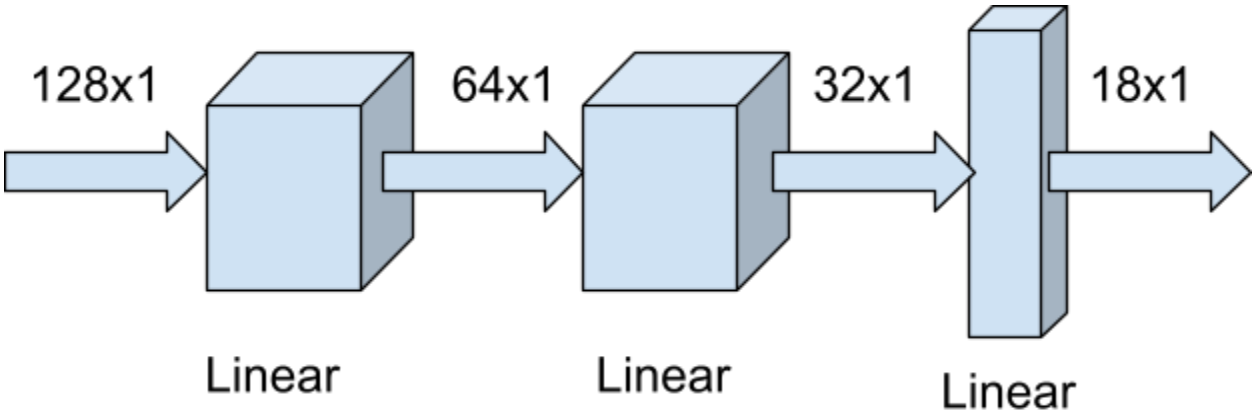


Architecture de la fonction politique pour l'image

Il faut aussi noter que la mise à jour de la politique se fait par montée de gradient. Ce principe n'est pas nativement présent dans PyTorch. En multipliant par -1 la fonction d'erreur, on réalise une montée avec l'implémentation existante de descente de gradient de PyTorch.

c) Fonctions Actor et Critic pour la mémoire

On peut utiliser la mémoire vive (RAM) pour représenter l'état. Dans ce cas, on a en entrée un vecteur de 128 valeurs, bien plus petit que le vecteur de données issues des images (7896 valeurs). Ces données sont aussi plus simples à traiter puisqu'il n'y a pas l'aspect à deux dimensions d'une image.
L'architecture des fonctions politique et valeur est un réseau de neurones composées de plusieurs couches cachées entièrement connectées avec une fonction non-linéaire de type ReLU. Le choix du nombre de couches est arbitraire. Trois couches sont présentes pour donner un pouvoir d'apprentissage important au réseau.



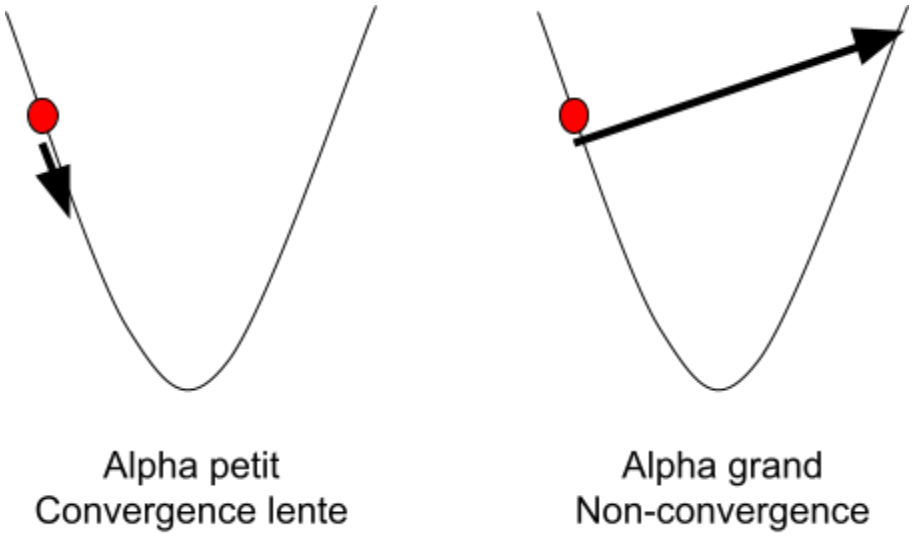
Architecture de la fonction politique pour la RAM

d) Processus asynchrone

Pour mettre en œuvre le comportement asynchrone, plusieurs composants ont été mis en place. Tout d'abord, une variable partagée contenant les deux jeux de poids de la fonction acteur et critiques est créée et donnée aux autres processus exécutant les épisodes. Lorsqu'un épisode est fini, les gradients des poids sont envoyés dans une queue. Un processus léger ("thread") en charge du calcul est démarré. Ce processus écoute la queue. La queue sert à stocker les poids reçus. Plusieurs poids peuvent être reçus en parallèle. Chaque fois que de nouveaux gradients sont reçus, le processus de calcul met à jour les valeurs des poids à partir des gradients.
Dans un premier temps, le processus de calcul intégrait une mise à jour des poids avec un paramètre α.

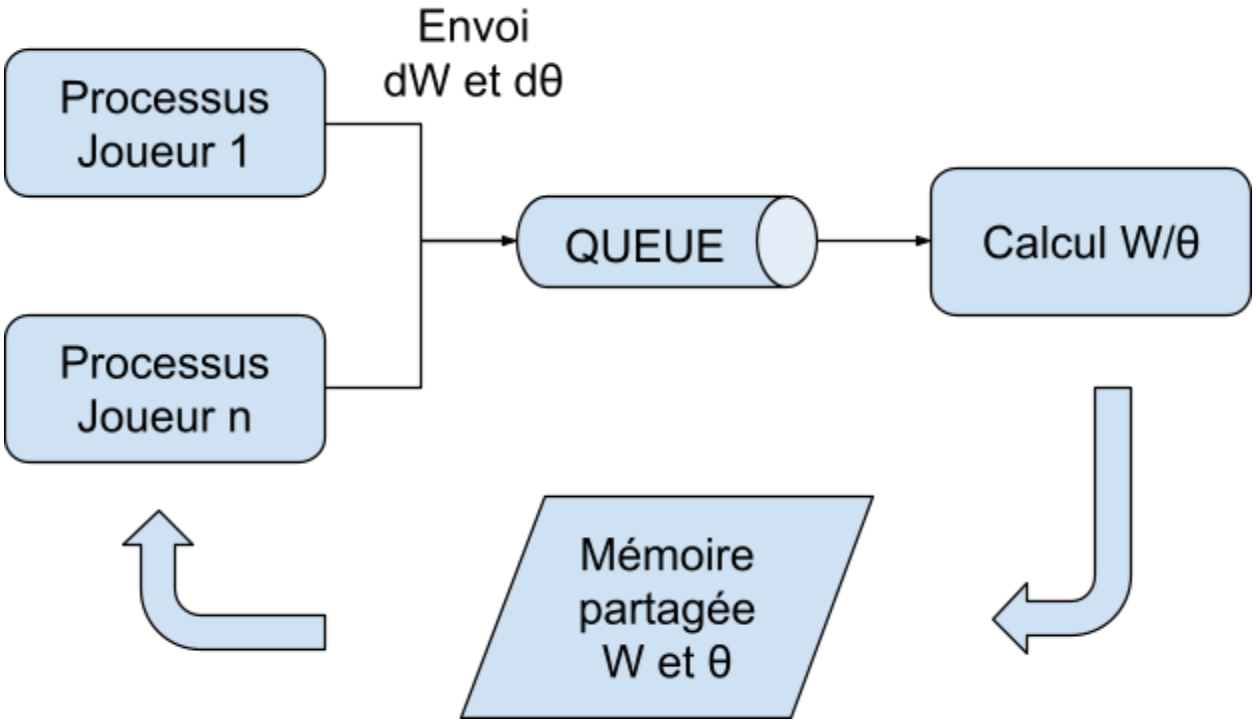
$$\theta = \theta - \alpha * d\theta$$
$$W = W - \alpha * dW$$

Le problème de la descente de gradient stochastique est le choix du paramètre α. Une valeur trop grande entraîne une divergence des poids. Une valeur trop petite ralentit la convergence.



Pour résoudre ce problème, j'ai ré-implementé l'algorithme Adam[1] (*Adagrad*) dans le processus de calcul. Cet algorithme prend en compte la valeur du gradient. Si le gradient est trop important, le paramètre α est atténué. S'il est trop faible, α est amplifié.

Le diagramme suivant présente l'architecture globale du système:



Architecture globale

En extension, le processus de calcul incorpore la partie "Experience Replay".

[1] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

II.2) Experience Replay

Un des éléments sensibles dans la modèle Apprentissage par Renforcement est la fonction Critique. Pour que l'algorithme soit efficace, il est nécessaire d'avoir une fonction qui retourne de manière correcte la valeur d'un état. Lors de l'exécution, cette valeur est mise à jour après chaque action, un état à la fois.
L'idée de l'"Experience Replay" est de faire réapprendre le réseau sur un ensemble de couple (état courant, action, récompense, état suivant). Ces différents couples sont conservés au cours de l'exécution du modèle. L'acteur peut ainsi apprendre sur plusieurs exemples en un fois ce qui lui permet de mieux généraliser sur la valeur à retourner pour un état donné.

II.3) Choix techniques

Ce projet a été réalisé en utilisant plusieurs librairies reconnues dans le monde de l'Intelligence Artificielle. Le code est écrit en python. Pour la partie réseau de neurones, le code appelle les fonctions fournies par la librairie PyTorch, comme vu durant les travaux pratiques.
Pour la partie système de jeu, j'ai utilisé la librairie Atari Learning Environment qui permet d'interagir facilement avec le jeu.
Enfin, pour pouvoir visualiser le comportement de l'agent dans le cas où l'état est une image, j'ai utilisé OpenCV qui m'a permis de transformer les différentes images après chaque action en une vidéo.
Plusieurs développements ont été réalisés, en s'inspirant des travaux pratiques ou de tutoriaux. Cette partie décrit l'organisation du code du projet.

`README.md`: documentation du projet

`main.py` : point d'entrée de l'application. Contient le code pour le processus asynchrone qui effectue la mise à jour des poids du modèle à partir des gradients et d'experience replay

`recorder.py` : classe en charge de la génération de vidéo

`Preprocessing.py`: fonction de pré-traitement de l'image

`Experience_replay.py` : mémoire pour l'expérience replay

```
rgb/
      ├── actor.py : réseau de l'acteur
      ├── critic.py : réseau du critique
      └── runner.py : processus qui réalise un épisode
ram/
      ├── actor.py
      ├── critic.py
      └── runner.py
```

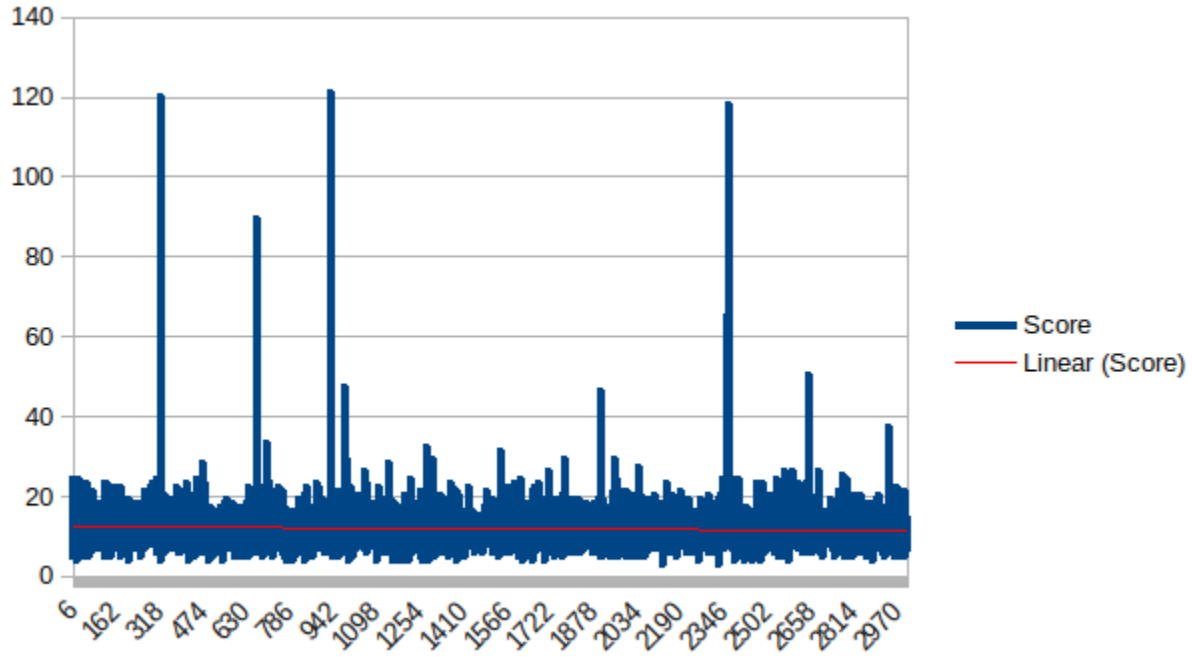
III) Résultats et analyses

Lors des tests, la valeur γ choisie était de 0.9.

III.1) Résultats

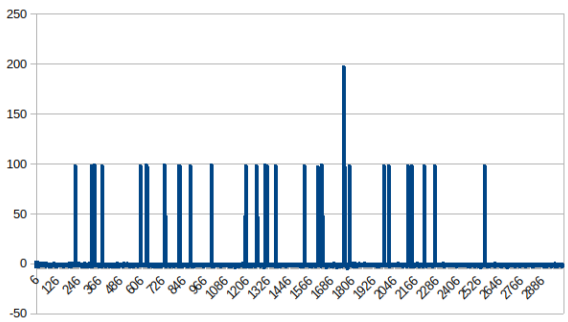
a) Image comme état d'entrée sans experience replay

Le programme a été testé avec 3 processus en parallèle et 3000 épisodes avec l'image en tant qu'état. L'exécution a duré environ 6 heures. Le graphe suivant représente l'évolution du score au cours des épisodes.

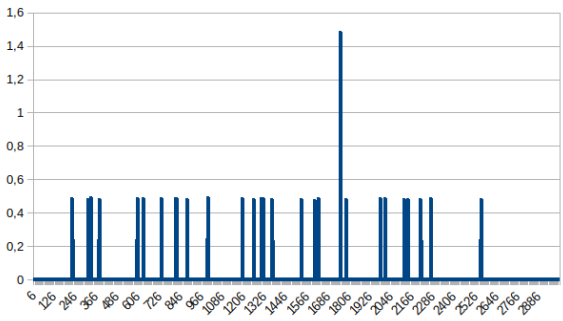


La courbe rouge représente la tendance. La pente est proche de zéro. Le score maximum obtenu est de 122.

Les graphes suivants sont l'évolution du score de la politique et du critique au cours du temps:



Policy Loss

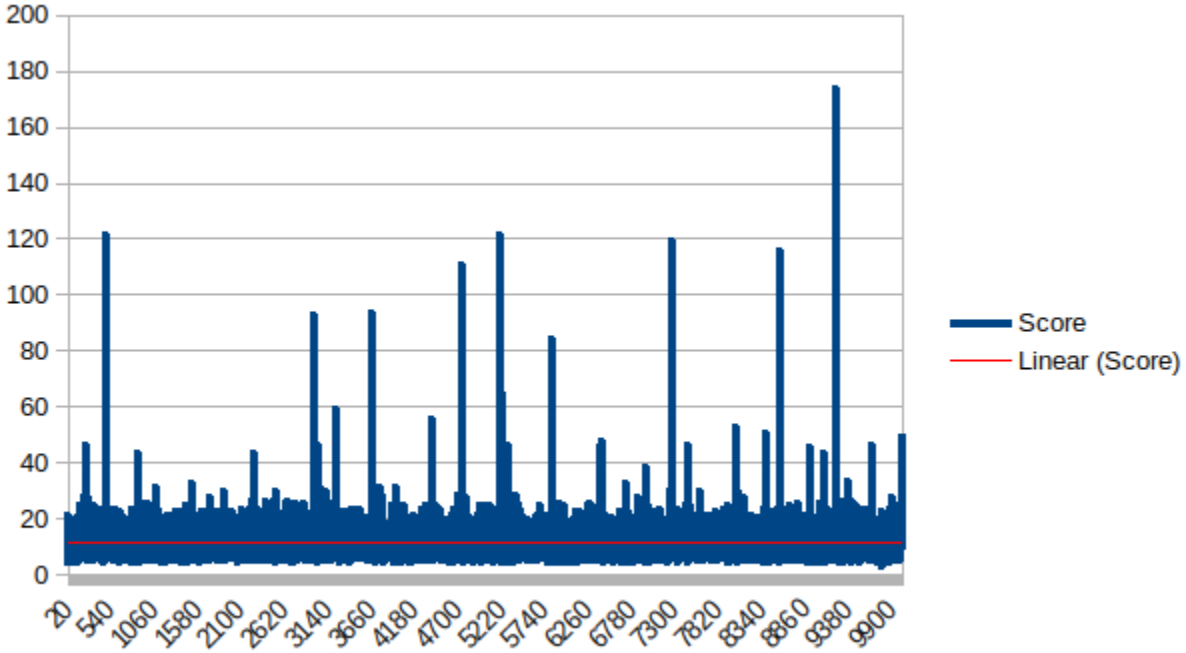


Value Loss

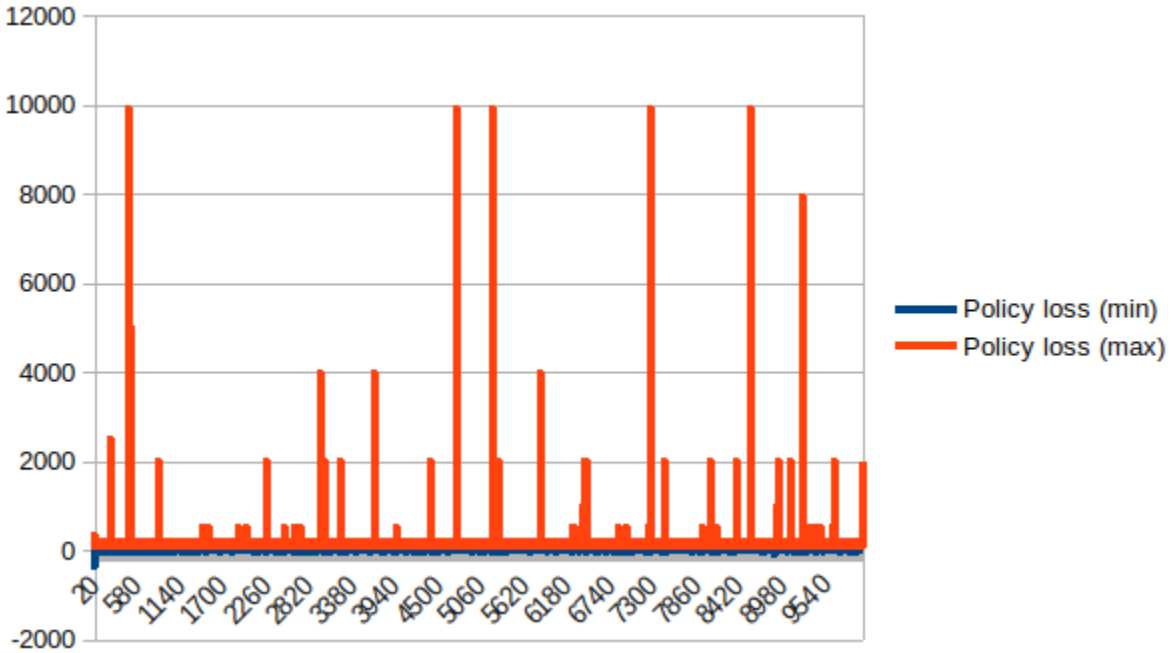
Il s'agit de la valeur à la fin de l'épisode.

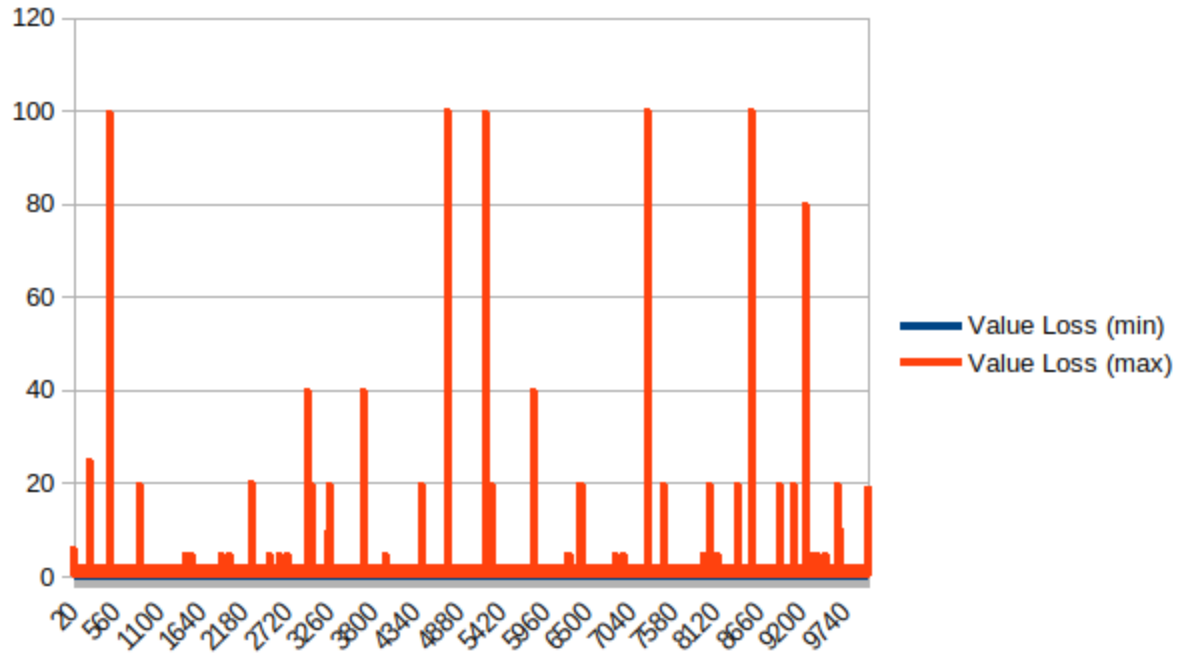
b) Mémoire comme état d'entrée sans experience replay

Le programme a été testé avec 4 processus en parallèle et 10000 épisodes avec l'image en tant qu'état. L'exécution a duré environ 12 heures. Le graphe suivant représente l'évolution du score au cours des épisodes.



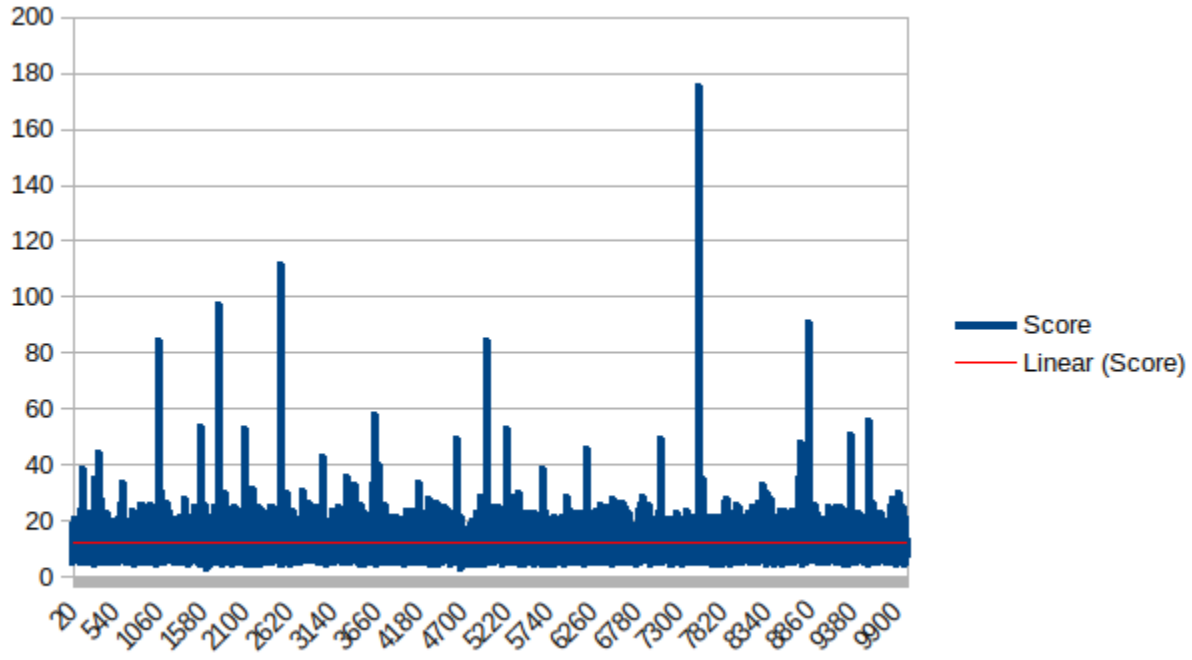
La courbe rouge représente la tendance. La pente est proche de zéro. Le score maximum obtenu est de 176. Pour les valeurs du score de la politique et du critique, il a semblé plus intéressant de récupérer les minimums et maximum au cours des épisodes:

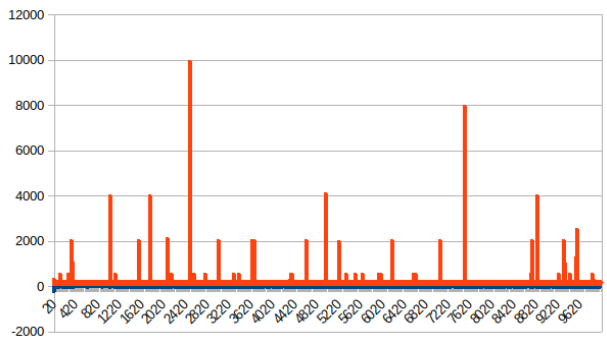




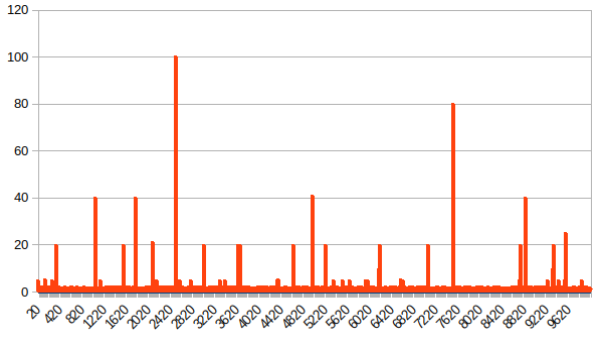
c) Mémoire comme état d'entrée avec experience replay

Le programme a été testé avec 4 processus en parallèle et 10000 épisodes avec l'image en tant qu'état. Le modèle pour la critique ré-apprend tous les 10 résultats sur un batch de 128 couples dans un ensemble des 10000 derniers couples états/action/réponses. L'exécution a duré environ 12 heures. Le graphe suivant représente l'évolution du score au cours des épisodes.





Policy Loss



Value Loss

III.3) Analyses des résultats

Les différents tests ont montré que le modèle n'a pas réussi à s'améliorer au cours du temps. Les mesures de score montrent malgré tout que le modèle est bien influencé par les actions.

Plusieurs causes peuvent être à l'origine de ces résultats. Tout d'abord, une erreur d'implémentation peut être présente dans le code malgré tout le soin apporté. Une solution pour cette hypothèse serait de tester l'implémentation sur un problème plus simple comme "CartPole" et voir s'il retourne un résultat intéressant. Cela n'a pas pu être fait par manque de temps.
Ensuite, les méthodes à base d'Apprentissage par Renforcement nécessitent beaucoup d'épisodes pour produire un modèle utilisable. Le nombre d'épisodes a été d'environ 10000. Par manque de temps, il n'a pas été possible d'effectuer de nombreux tests.
Enfin, le modèle possède plusieurs hyper-paramètres:

- L'architecture des réseaux de neurones utilisés. L'architecture est semblable aux différents articles scientifiques qui ont servi de support à ce projet
- La valeur γ qui sert à calculer l'avantage
- Les paramètres pour la descente de gradient. Des valeurs trop fortes ou faibles peuvent empêcher le réseau de se stabiliser ou de progresser.

Conclusion

Lors de ce projet, j’ai tenté d'implémenter une Intelligence Artificielle basée sur le modèle A3C issu du domaine de l’Apprentissage par Renforcement dans le but de jouer au jeu PacMan. Pour ce faire, j’ai étudié les articles cités dans la bibliographie et je me suis appuyé sur les travaux réalisés lors de l’UE RCP211 et plusieurs tutoriaux et codes disponibles sur Internet.
Plusieurs modèles ont été implémentés avec la librairie PyTorch qui utilisaient la RAM et l’image du jeu fourni via la librairie ALE.
Les résultats obtenus n’ont pas permis de conclure que le modèle avait appris à jouer à PacMan. J’ai néanmoins acquis une bonne compréhension du modèle “Acteur/Critique” ainsi que des extensions avec le mode asynchrone. Le projet m’a aidé à consolider les connaissances techniques de la librairie PyTorch.