Rapport travaux d'études dirigés

Gizel Hadji - Frederik Voigt - Quentin Cognard

May 22, 2020

Apprendre à jouer à Space Invaders en C++

Contents

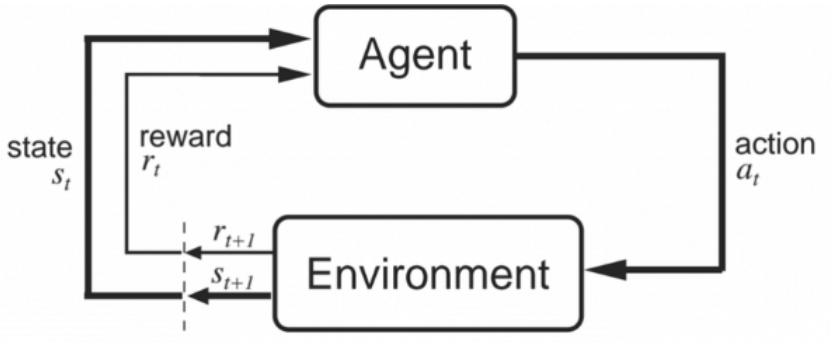
1 Compréhension du sujet

1.1 Généralités & Apprentissage par renforcement

Space Invaders est un jeu d'arcade apparu sur les bornes d'arcade en 1978. Grâce à un émulateur de la console Atari 2600, notre but est de réaliser un programme en C++ qui, grâce aux pixels de l'image et du score courant, apprend à jouer à Space Invaders. Pour cela nous utiliserons une technique qui a souvent fait ses

preuves depuis AlphaGo quand il s'agit d'apprendre à jouer à un jeu vidéo par une machine, l'apprentissage par renforcement.

> En intelligence artificielle, l'apprentissage par renforcement (Reinforcement Learning, RL) consiste, pour un agent autonome, à apprendre les actions à prendre, à partir d'expériences, de façon à optimiser une récompense quantitative au cours du temps. L'agent est plongé au sein d'un environnement, et prend ses décisions en fonction de son état courant. En retour, l'environnement procure à l'agent une récompense, qui peut être positive ou négative. Grâce aux récompenses l'agent va modifier sa stratégie (ou politique) afin que les prochaines actions prises soient plus rémunéré. (Wikipédia)



1.2 Policy Gradient

De manière plus précise, dans notre projet nous utiliserons le Policy Gradient (PG). Nous allons essayer de donner une approche intuitive : Avec cet algorithme notre but est de mettre à jour un état θ pour s'approcher de l'état $\pi\theta$, la politique optimal. Si on choisi l'action optimal a^*, alors on veut que $\pi\theta\left(a^*|s\right)$ (la probabilité de choisir l'action optimal à l'état s) soit aussi proche de 1 que possible. On va pouvoir déterminer la fonction suivante :

$$\theta_{t+1} = \theta_t + \alpha\nabla\pi\theta_t\left(a^*|s\right)$$

θ_t est notre réseaux de neurones à un moment t

α représente une constante qui détermine la vitesse d'apprentissage, plus elle est grande plus l'apprentissage sera rapide mais en dépit de la précision.

$\nabla\pi\theta_t\left(a^*|s\right)$ est le gradient que nous pouvont voir comme la direction qui permettra de s'approcher de la valeur optimal $\pi\theta_t\left(a^*|s\right)$

Grâce à cette formule notre politique va converger vers le résultat optimal, car nous avons choisi l'action a^*(l'action optimale pour la situation), et donc seulement celle-ci va évoluer et par conséquent les autres probabilités vont décroitre. Mais en réalité nous ne savons pas quelle action est la meilleure, le

choix de l'action se fait de manière stochastique, seulement notre politique va nous indiquer la probabilité qu'une action est meilleure qu'une autre. Alors nous utiliserons le gradient $\nabla\pi\theta_t\left(a|s\right)$ que nous allons lier à une fonction $Q\left(s,a\right)$ qui permet d'estimer la valeur de l'action a dans l'état s, autrement dit c'est une fonction qui grâce aux observations de l'environnement nous indique si l'action a été efficace ou pas.

$$\theta_{t+1} = \theta_t + \alpha Q\left(s,a\right)\nabla\pi\theta_t\left(a|s\right)$$

Cependant avec cette fonction il y a anguille sous roche. En effet à la différence des généralités du RL, le PG ne fait qu'augmenter la probabilité de l'action choisi pour un état donné. Or sait que les actions les plus probable vont être prise plus souvent, ainsi, si notre réseau neuronal est mal initialisé, une action peut converger vers 1 non pas a cause de son efficacité, mais seulment a cause du fait qu'elle est choisi bien plus souvent que les autres. Toutefois il est assez simple de résoudre ce problème de manière naturelle, il suffit de faire une division de notre calcul d'ajout par la probabilité de l'action, ainsi les actions qui sont choisies plus souvent reçoivent une modification moins importante.

$$\theta_{t+1} = \theta_t + \frac{\alpha Q(s,a)\nabla\pi\theta_t(a|s)}{\pi\theta(a|s)}$$

Et voilà pour une explication intuitive et synthétique du fonctionnement du PG.

1.3 RMSProp

Au lieu d'utiliser seulement les gradients du PG à un moment t pour guider notre apprentissage, RMSProp va nous permettre de prendre en compte les précédents gradients afin d'affecter le pas d'apprentissage sans en changer la direction.

$$v_t = \rho v_{t-1} + (1-\rho) * g_t^2$$

$$\nabla w_t = -\frac{\eta}{\sqrt{v_t+\epsilon}} * g_t$$

$$w_{t+1} = w_t + \nabla w_t$$

ρ est le taux de décroissance

η est le taux d'apprentissage

g_t représente notre gradient qui vient de notre algorithme d'apprentissage (ex : PG)

∇w_t le nouveau gradient obtenue par le RMSProp

Imaginons, nous obtenons un ensemble de gradients g grâce à notre PG, pour appliquer le RMSProp nous devons commencer par obtenir v_t qui représente la moyenne exponentielle des gradients au carré, pour faire cela on multiplie la moyenne exponentielle précédente par ρ (souvent compris entre 0,9 et 0,99) puis on y ajoute le carré du gradient obtenu par le PG multiplié par $(1 - \rho)$. Le calcul de v_t est là, afin d'aider nos poids les plus récents d'augmenter plus que les précédents. Puis avec la seconde équation on cherche à obtenir un nouveau gradient ∇w_t qui se déplace dans la direction de celui du PG mais qui sera affecté par notre moyenne exponentielle v_t. Enfin la 3éme équation nous ajoutons simplement à nos poids le gradient obtenu.

2 Fonctionnement de l'algorithme

2.1 Entrée et traitement

Afin de fonctionner, l'algorithme à besoin de valeurs renvoyées par l'émulateur, sous forme de matrice. Chaque frame du jeu est renvoyée sous forme de matrice, et récupérée après l'initialisation du jeu ou l'exécution d'une action. Ensuite, un traitement est éffectué sur la matrice afin d'enlever les valeurs non-interressantes (comme les bordures par exemple), et de la transformer en un vecteur de taille 6400 (80x80 étant la taille choisie de l'écran). On calcule ensuite la différence entre la frame récupérée à la frame actuelle et celle d'avant, afin de n'avoir que les changements présent dans le vecteur.

2.2 Policy forward

À partir du vecteur, il nous faut maintenant calculer la probabilité de chaque action d'être choisie, cela consiste notament à faire des multiplications entre matrices.

```
def policy_forward(x):
  h = np.dot(model['W1'], x)
  h[h<0] = 0 # ReLU nonlinearity, fonction d'activation
  logp1 = np.dot(model['W2'][0], h)
  logp2 = np.dot(model['W2'][1], h)
  p1 = sigmoid(logp1)
  p2 = sigmoid(logp2)
  return p1, p2, h # return probability of taking action 2, and hidden state
```

Les multiplications se font avec la fonction np.dot. Il y en a ici au nombre de trois, dans la première on multiplie donc le vecteur récupéré avec une matrice préalablement initialisée nommée W1. Elle représente une couche cachée (hidden layer), qui permet notamment d'ajuster en fonction de la taille choisie, les valeurs que l'on obtiendra plus tard en sortie. Comme précisé précédemment,

notre objectif ici est de calculer les probabilités de chaque action. Ainsi, il va nous falloir une autre matrice afin de représenter chaque neurone (ou probabilité), une matrice possédant un nombre de lignes égal au nombre de neurones de sortie, soit 2 dans notre cas (W2 ici). Chaque ligne de cette matrice représentera donc une probabilité, et il nous faudra multiplier chaque ligne par la valeur obtenue précédemment, afin d'influencer nos valeurs de sortie. Ainsi, on obtient de cette manière nos probabilités (réagustées entre 0 et 1 grâce à la fonction sigmoid, et représentées par $\pi\theta(a|s)$), que l'on retourne en même temps que notre hidden state (h). Notons aussi que chaque matrice plus tard être modifié avec les valeurs du gradient.

2.3 Tirage des actions et labels

Il va maintenant nous falloir tirer les actions à partir des probabilités. Dans notre implémentation, nous avons choisi de "combiner" nos tirages. En effet, on tire une fois pour savoir si on tire et dans quel sens on bouge, car dans space invaders, on peut à la fois tirer et bouger. Ainsi on tire deux valeurs entre 0 et 1 et on regarde si on est tombé entre 0 et la probabilité de l'action, on choisit en fonction des deux résultats de l'action à faire. Ensuite, nous allons calculer ce qu'on appelle le loss, qui est en fait $y - proba$ pour les deux actions, où y est un label valant 1 si on a tiré entre 0 et proba, et 0 sinon. Le loss nous permettra aussi d'influencer le gradient, on va donc le garder en mémoire dans un tableau de hauteur du nombre de probabilité (2). Le loss est calculé de cette manière, car nous voulons augmenter la probabilité pour les actions choisies, et baisser celles non choisies.

2.4 Rewards

Afin de pouvoir savoir si nos actions étaient les bonnes, nous avons besoin de connaitre la récompense (reward) obtenue, qui change avec le score. Afin d'obtenir ce reward, nous avons simplement à rentrer l'action choisie précédemment dans la fonction prévue par gym. On obtient donc notamment une nouvelle frame et le reward et on sait si on a terminé le jeu. On calcule ensuite la somme des rewards obtenus et on les gardes en mémoires.

2.5 Policy Backward et policy gradient

Comme mentionné dans la partie compréhension du sujet, nous avons besoin d'une fonction qui nous permet d'estimer la valeur de l'action a dans l'état s. Nous devons donc d'abord calculer le discounted reward pour toutes les frames (Représenté par le $Q(s, a)$ dans la partie policy gradient). On calcule le discounted reward de la frame i avec le reward de la frame i et le discounted reward de la frame suivante.

```
def discount_rewards(r):
    #
    discounted_r = np.zeros_like(r)
    discounted_r[r.size-1] = r[r.size-1]
    for i in reversed(range(0, r.size-1)):
        discounted_r[i] = r[i] + gamma*discounted_r[i+1]
    return discounted_r
```

Ceci permet d'attribuer un reward plus important aux dernières actions. Ici on calcule donc le discounted reward et, d'après le document "pong from pixel", on l'ajuste afin qu'il encourage et décourage seulement environ la moitié des actions effectuées, cela permettrait de mieux controler le gradient.

```
# compute the discounted reward backwards through time
discounted_epr = discount_rewards(epr)
# standardize the rewardsto be unit normal (helps control the gradient estimator variance)
discounted_epr -= np.mean(discounted_epr)
deviation = np.std(discounted_epr)
not_all_zeros = np.any(deviation)
if(not_all_zeros):
    discounted_epr /= deviation
```

On calcule ensuite finalement le gradient pour chaque matrice dont on a parlé précédemment, soit W1 et W2.

```
epdlogp *= discounted_epr # modulate the gradient with advantage (PG magic happens right he
grad = policy_backward(eph, epdlogp)
epdlogp = [[],[]]
grad_buffer['W1'] += grad['W1'][0]
grad_buffer['W1'] += grad['W1'][1]
grad_buffer['W2'][0] += grad['W2'][0]
grad_buffer['W2'][1] += grad['W2'][1]
```

Le grad_buffer contient donc notre gradient accumulé sur le nombre d'épisode choisi au préalable. On choisit un nombre au bout duquel on appliquera nos modifications. On utilise ensuite le calcul du RMSProp défini au-dessus et on modifie W1 et W2 en fonction des valeurs obtenues.

```
# perform rmsprop parameter update every batch_size episodes
if episode_number % batch_size == 0:
  for k,v in iter(model.items()):
    g = grad_buffer[k] # gradient
    rmsprop_cache[k] = decay_rate * rmsprop_cache[k] + (1 - decay_rate) * g**2
    model[k] += learning_rate * g / (np.sqrt(rmsprop_cache[k]) + 1e-5)
    grad_buffer[k] = np.zeros_like(v) # reset batch gradient buffer
```

On recommence ensuite en relançant une partie avec ces données.

3 Implémentation en C++

3.1 Contexte et objectif

Actuellement, le programme tourne en python grâce à la librairie Gym. Cependant, notre objectif est de faire fonctionner l'algorithme d'intelligence artificielle ainsi que sa visualisation en C++. L'implémentation du programme actuellement python vers le C++ nous permettra par la suite d'optimiser les performances notamment via la parallélisation. Ceci nous permettra d'augmenter le nombre de neurones pour rendre notre algorithme plus efficace (un apprentissage plus rapide et un meilleur taux d'apprentissage).

3.2 Arcade Learning Environment

Pour parvenir à nos fins, nous utiliserons le framework Arcade Learning Environment . Il s'agit d'un outil nous permettant de travailler dans un environnement propice à l'utilisation d'algorithmes d'intelligence artificielle via l'utilisation de l'émulateur Stella (On peut aussi noter que l'utilisation de Gym repose aussi sur ALE d'où la pertinence de son utilisation). ALE, propose une série de fonctionnalité via son interface permettant par exemple de visualiser l'écran ou bien récupérer les valeurs de la RAM ou alors de l'écran...

Dans le cas du Policy Gradient, l'état de la RAM ne nous intéresse pas, on souhaite pouvoir récupérer l'ensemble des valeurs RGB représentées par l'écran : fonction -> getScreenRGB. Cette fonction prend en paramètre un vecteur à remplir qui représentera notre matrice de représentation des pixels affichés. On aura aussi besoin de la fonction -> act pour pouvoir récupérer un objet de type reward_t correspondant explicitement au reward. Le paramètre pris par la fonction -> act est de type Action.L'ensemble des actions disponibles est récupérable dans un vecteur grâce à la fonction -> getLegalActionSet. On utilisera cependant la fonction -> getMinimalActionSet pour récupérer seulement les actions dont nous aurons besoin.

Les fonctions que nous utiliserons seront donc les fonctions précédemment citées ainsi que d'autre permettant de vérifier et de gérer l'état du jeu.

La fonction -> game_over : vérifie si la partie est terminée (utilisé pour vérifié si l'épisode n'est pas terminé).

La fonction -> game_reset : recommence une partie (utilisé en fin des calculs de la terminaison de l'épisode).

3.3 Spécifications du C++

Le souhait principal de l'implémentation en C++ réside dans l'utilisation de la librairie de base et seulement celle de base pour la construction de l'algorithme (hormis ALE évidemment) . En effet, si nous souhaitons paralléliser notre algorithme, il est plus sage d'avoir la main mise sur l'entièreté du code et sur un maximum d'opérations effectuées.

Ainsi, il a été crée une librairie spéciale nommé Moon (car celle ci envoie ses utilisateurs dans l'espace). Cette librairie est une implémentation C++ de ce que propose numpy au niveau des calculs matriciels. Cette librairie permet également de définir un objet Matrix facilement manipulable et instanciable à partir des données procurées par ALE. Cela améliore ainsi fortement l'éfficacité de l'implémentation et permet de garder une cohérence par rapport à la version Python. Exemple : à droit la version python, à gauche son implémentation grâce à Moon.

```
Matrix h = dot(*model["W1"],x);          h = np.dot(model['W1'], x
```

Au final la plupart de l'algorithme est donc assez ressemblant, les différences se situent au niveau des facilités accordés par Python (visible notammennt dans la fonction -> prepro) et la communications avec l'environnement d'apprentissage. Python étant un langage interprété il fallait être sur de la nature des variables manipulées, notamment les dimensions des matrices qui ont été vérifié plusieurs fois. Ainsi, les types et les variables ont été respecté et son conformes à ce qu'on pourrai attendre en Python.

3.4 Premier test

Avec une première implémentation complète et compilée, le premier test a été effectué. On a pu deja constater que le jeu se lancais bien mais avec de grosse pertes de frame et un crash dés le premier épisode. Les raisons, une mauvaise gestion de la mémoire (involontaire) et un aucun calculs parallèles (volontaire en premier lieu). Le premier résultat est donc peux concluant mais néanmoins encourageant car les données récupérés dans la matrice prepro sont bonnes et les calculs effectué sur celle ci sont juste.

3.5 Gestion de la mémoire

Notre programme possède un réel problème de fuite mémoire, autant dans l'implémentation que pour la librairie à cause des nombreux objet Matric crée. Une première idée a donc été d'utiliser des pointeurs pour pouvoir y attribuer les adresses des matrices et les supprimer quand on ne s'en sert plus.

En effet, l'apprentissage par réseaux neuronal requiert une très bonne gestion de la mémoire, il s'agit d'éffectuer de nombreux calculs sur de grandes matrices à chaque frame sans pour autant que cela ne puisse se voir à l'écran (du moins c'est le cas dans la version Python). Ainsi les poineurs crée sur la boucle répété à chaque frame (celle qui ne gère pas la fin d'un épisode) ont été testé, fonctionne mais ne sont pas suffisant, il faut aussi paralléliser.

3.6 Parallélisation

La façon la plus efficace pour nous permettre de gagner en performance est de paralléliser nos calculs. Cette parallélisation doit avoit lieu pour tout ce qui concerne la manipulation de tableau ou les calcules matriciels. On utilisera OpenMP : une interface de programmation pour le calcul parallèle (ce qui implique également une bonne gestion de la mémoire en premier temps). Un premier test de parallélisation à eu lieu sur la fonction -> prepro.

La fonction -> prepro à pour but de traiter puis redimensionner l'observation correspondant aux valeurs des pixels de l'écran afin de ne manipulé que des valeurs intérrésant pour l'algorithme, en théorie. En pratique, il s'agit de récuperer le vecteur "observation" obtenu grâce à la fonction de ALE -> getScreenRGB (remplis un vecteur d'element de type 'char'). Ensuite on "crop" les éléments grâce à plusieurs boucles for et on stock les valeurs voulus dans un nouveau vecteur de 'float' cette fois ci. Enfin, on assigne les valeurs du vecteur obtenu dans un tableau de 'float' qu'on utilise pour instancier la nouvelle matrice qui sera le résultat du prepro.

La parallélisation a donc eu lieu sur les boucles for et sur une partie spécifique qui ne devait être traité que par un seul thread comme ce qui suit

```
#pragma omp parallel
#pragma omp for
for (i = 35; i < 195; i ++){
  if (i % 2 == 0){
    for (j = 0; j < width; j ++){
      if ( j % 2 == 0){
        for (k = 0; k < 1; k ++){
          pos = j + width*i + width*height*k;
          #pragma omp critical
          if ((float)I.at(pos) != 0){
            prepro_colors.push_back(1.0);
          }
          else {
            prepro_colors.push_back((float)I.at(pos));
          }
        }
      }
    }
  }
}

float** t;
t = new float*[D];
  #pragma omp parallel for
  for(int i = 0; i < D; ++i){
    t[i] = new float[1];
  }

#pragma omp parallel for
for (int i= 0; i<prepro_colors.size(); i++){
      t[i][0] = prepro_colors.at(i);
}
```

3.7 Deuxieme test

Avec ce début de parallélisation sur un calcul qui s'effectue chaque frame (prepro) et la création de pointeur pour les premieres matrices on obtient le résultat suivant.

On observe une amélioration des performances avec un ralentissement "quasi" nul, les 2 premiers épisodes s'effectuent de manière fluide mais nous avons un ralentissement "exponentiel " en milieu ~ fin du 3 ème épisode (problème de mémoire encore une fois)

3.8 Bilan de l'implémentation

Les test ont permis de constater que le programme fonctionne sans erreurs de segmentation mais que nous avons un souci de gestion de la mémoire. Il faudrai donc s'y pencher avec de continuer la parallélisation pour les calculs suivants. La gestion d'une fin d'épisode n'a donc pas pu être testé proprement en rai-

son des problèles cités et nous ne pouvons donc pour le moment pas savoir si l'algorithme apprend bel et bien à jouer (même si les parties individuelle de cette partie de l'algorithme ont été vérifié). Cependant, les résultat sont plutôt rassurant et montre que notre implémentation correspond en grande partie à la version Python. La librairie Moon à grandement aidé, son principe et son fonctionnement sont parfaitement en accords avec ce que nous voulons. Cependant, il faudra peut être la remanier au niveau de la gestion de la mémoire et la paralléllisation de ses calculs. Une fois cela fais, cette grande avancée nous permettra d'effectuer des tests plus long sur l'entièreté du programme. Ainsi, nous pourrons enfin voir notre implémentation fonctionner et observer ses résultats d'apprentissage au fil des épisodes.

Conclusion

Ce sujet nous a permis d'avoir une première approche sur l'apprentissage par renforcement ainsi qu'une premère expérience dans des travaux de recherche. Cette branche de l'intelligence artificielle est très récente et très en vogue, les articles sont donc suffisamment abondants. Toutefois, il nous a été difficile de faire le pont entre la vulgarisation des articles et le code du Pong en python qui nous a été fourni. Cepandant grâce aux efforts fournis, il nous a été possible de surmonter cette difficulté, ce qui nous a conduit a la seconde étape du projet, l'implémentation en python du Space Invaders. Les résultats de nos implémentations restent mitigés puisque bien que l'agent s'améliore, on atteint visiblement un plafond à 250 points en moyenne. Nos travaux nous ont conduit a établir un lien entre les hyperparametres et les performances de l'IA. (voir graphiques en annexes)

La dernière partie de nos travaux a consité à implémenter l'algorithme en C++. Pour cela une librarie reproduisant le comportement de numpy à été écrite, ce qui a donc facilité l'implémentation, cela a permi de garder une cohérence avec la version en Python. Cependant le passage de Python à C++ a été d'autant plus difficile à cause de la différence de typage entre les deux languages. De plus si la gestion de mémoire est automatique en Python elle ne l'est pas en C++. Dû a ces difficultés nous n'avons pas pu obtenir une implémentation satisfaisante en C++.

References

[1] Wkipédia : Apprentissage par renforcement

[2] Andrej Karpathy, Deep Reinforcement Learning: Pong from Pixels, 2016

[3] Adrien Lucas Ecoffet, An Intuitive Explanation of Policy Gradient, 2018

[4] Michael Klear, Understanding Evolved Policy Gradients, 2018

[5] Vlad Mnih, et al, Human Level Control Through Deep Reinforcement Learning, 2015

[6] Jesse Farebrother, Marc G. Bellemare, Arcade Learning Environment (ALE) – a platform for AI research. 2019

[7] Richard S. Suttonn, Andrew G. Barto, Reinforcement Learning: An Introduction, 2018

[8] Yao, Haipeng, Jiang, Chunxiao, Qian, Yi, Developing Networks using Artificial Intelligence, page 187. (hand crafted loss function), 2019

[9] Niels Justesen, Philip Bontrager, Julian Togelius, Sebastian Risi, Deep Learning for Video Game Playing, 2019

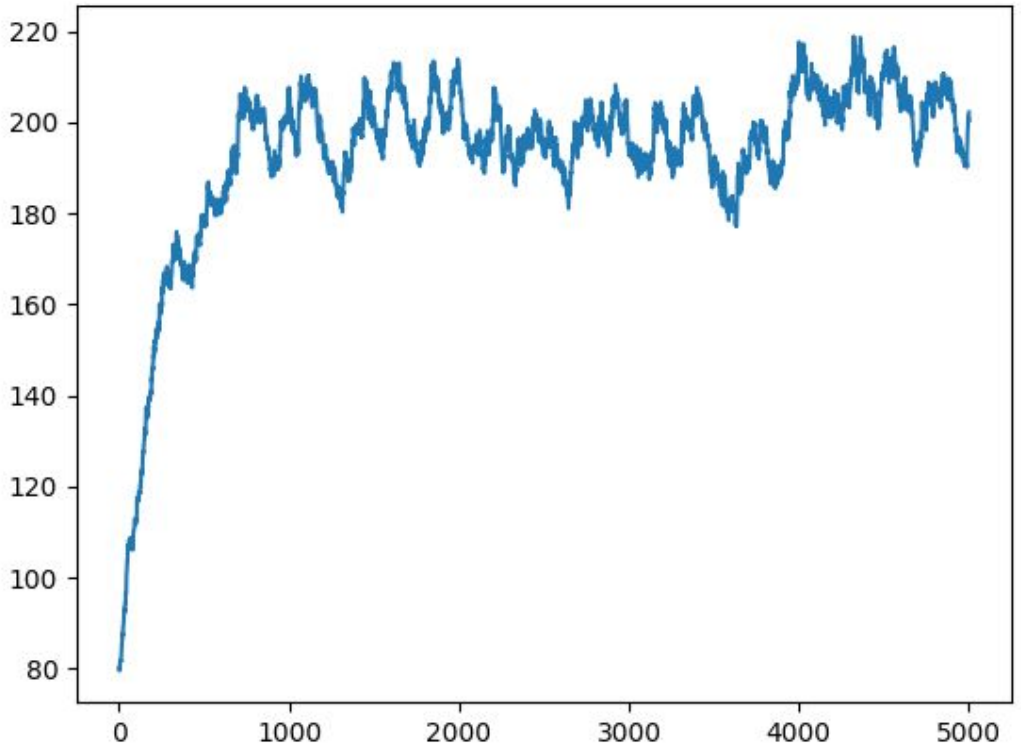
[10] Ayoosh Kathuria, Intro to optimization in deep learning: Momentum, RMSProp and Adam, 2018

[11] Vitaly Bushaev, Understanding RMSprop —{} faster neural network learning, 2018

ANNEXE

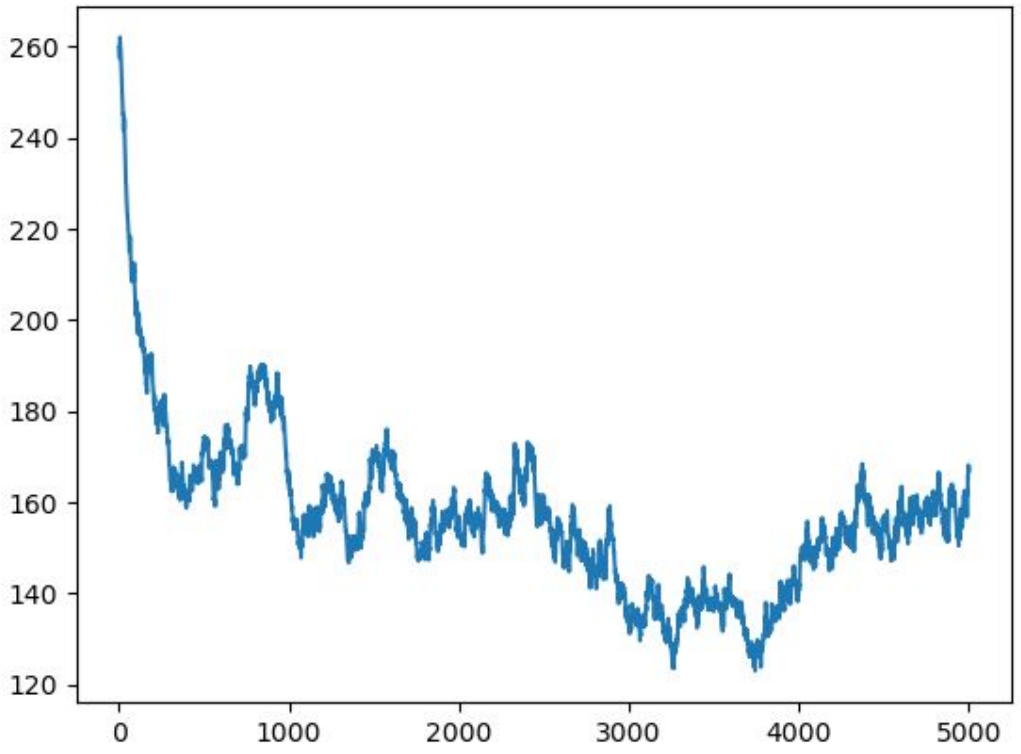
Les gifs de l'agent qui joue sont hébérgé sur imgbb.com
Agent a 150 points en moyenne : https://ibb.co/9gZkrQn
Agent a 250 points en moyenne : https://ibb.co/hLvvpKz

Tests sur les hyperparamètres et la fonction Loss
Tous les tests ont été réalisés avec 4 actions, qui eux sont décidées par 2 neurones et 1 hidden layer. A noter pour voir une réelle progression de l'agent il faut attendre plus de 20 mille épisodes, ce qui prend plusieurs heures même avec un CPU performant. Faute de temps et d'énergie nous ne pouvions faire plus.
On retrouve les rewards en ordonnées et le nombre d'épisode en abscisses



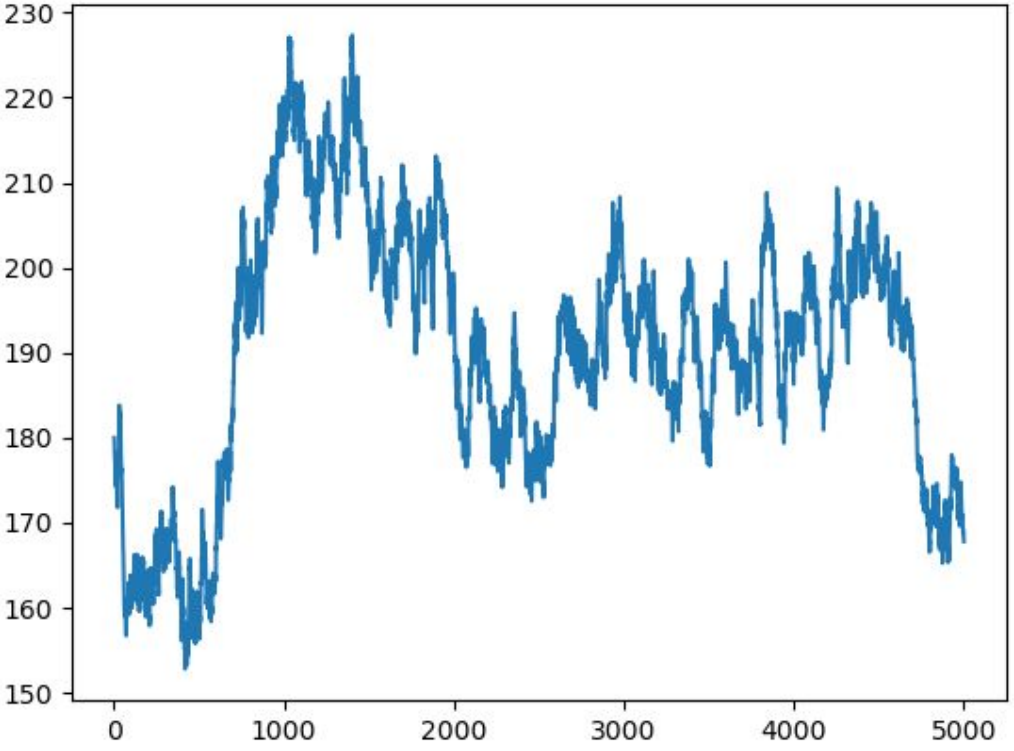
Réalisé avec les données suivantes :

- Fonction loss : hand crafted loss (y-prob)
- gamma : 0,99
- learning rate : 1e-4



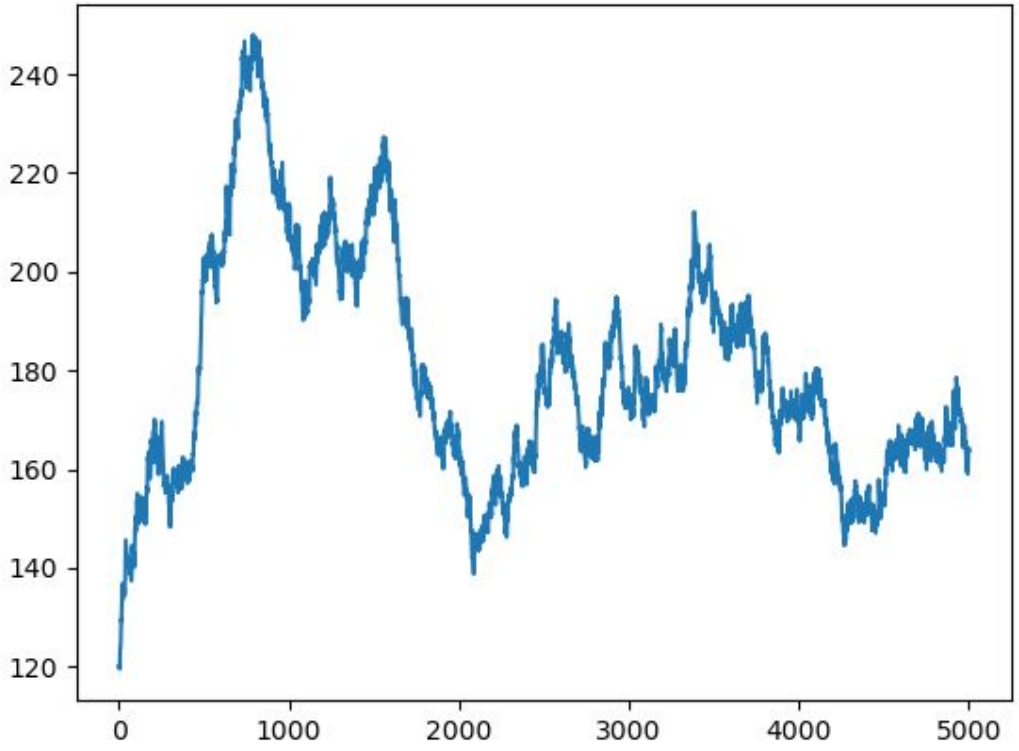
Réalisé avec les données suivantes :

- Fonction loss : $(y\text{-}prob)^2$
- gamma : 0,9
- learning rate : 1e-3



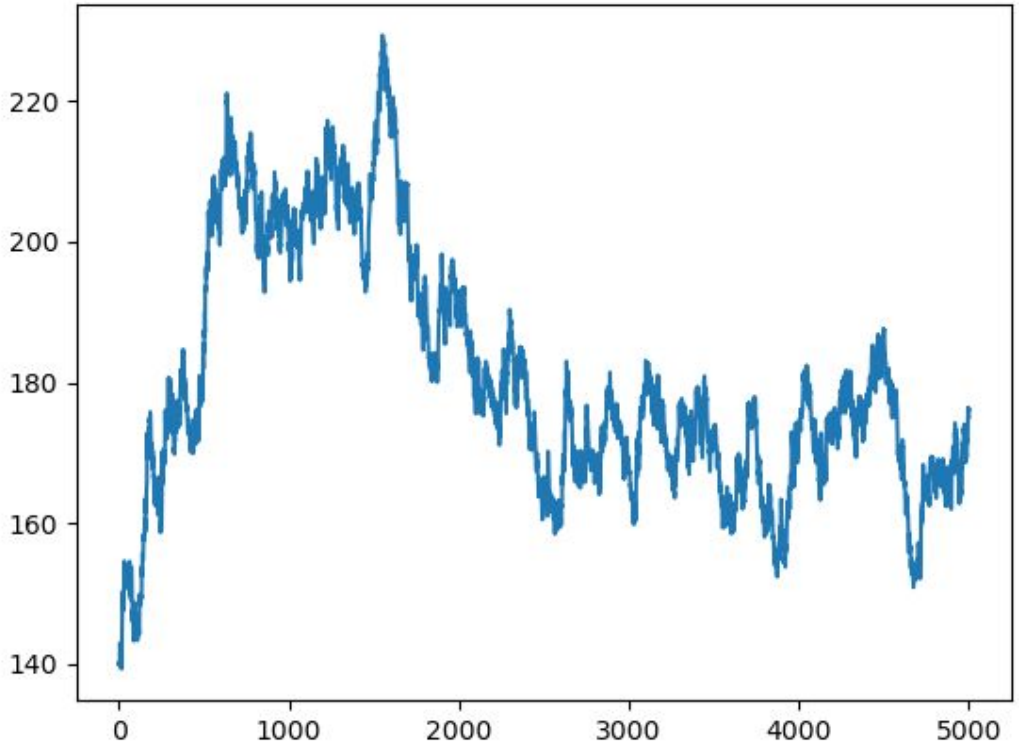
Réalisé avec les données suivantes :

- Fonction loss : ½ * (y-proba)²
- gamma : 0,9
- learning rate : 1e-4



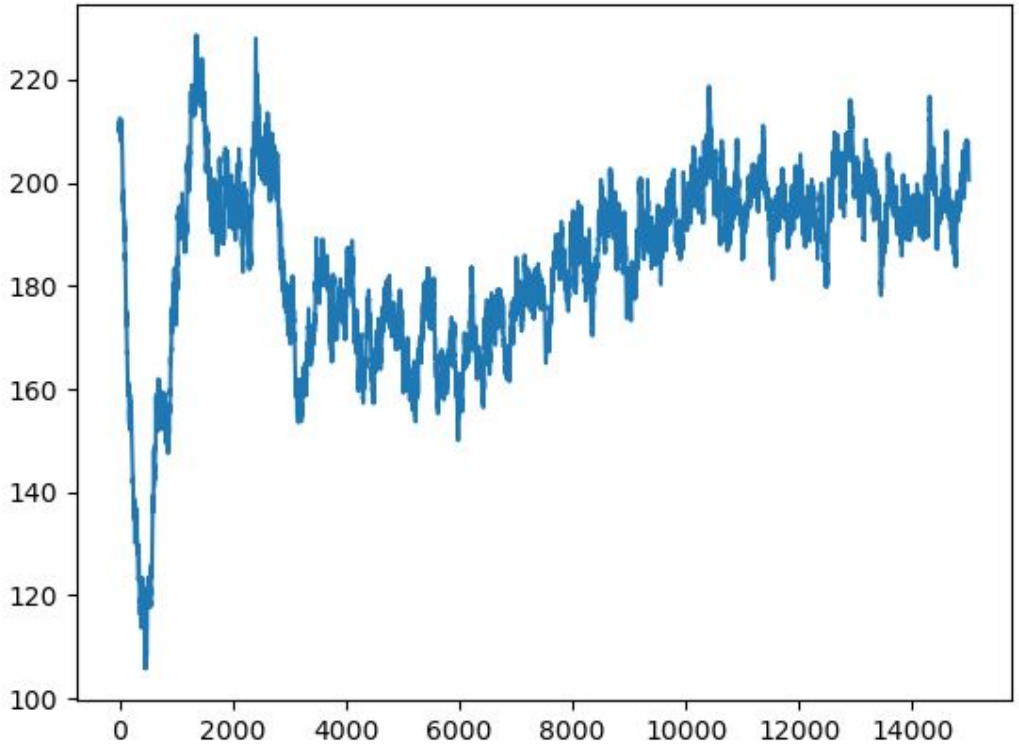
Réalisé avec les données suivantes :

- Fonction loss : Hubert Loss
- gamma : 0,9
- learning rate : 1e-3



Réalisé avec les données suivantes :

- Fonction loss : Hubert Loss
- gamma : 0,9
- learning rate : 1e-4



Réalisé avec les données suivantes :

- Fonction loss : Hubert Loss
- gamma : 0,99
- learning rate : 1e-4