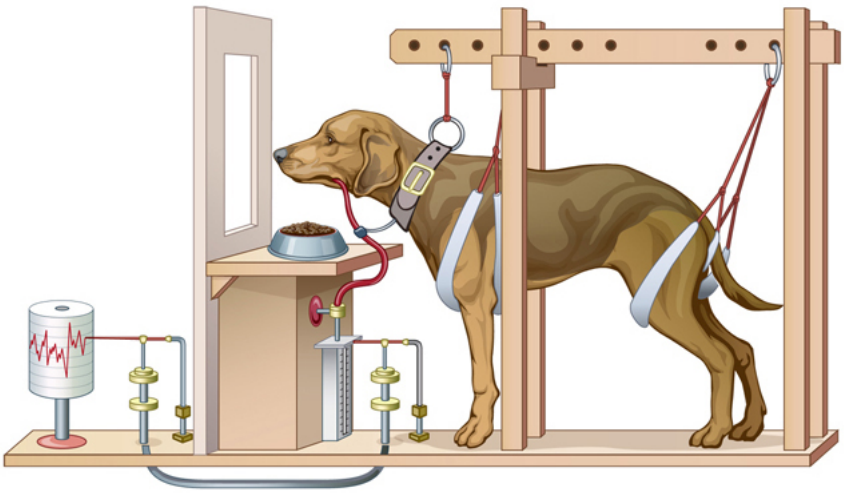


Reinforcement Learning

Travail d'automne

Sergiy Goloviatinski, inf3dlm-b
2018-2019, HE-Arc
Numéro de projet 212

25 janvier 2019

Abstract

Français

Ce rapport présente le déroulement d'un projet de semestre effectué à la HE-Arc qui consistait à développer un module python permettant de faire du Reinforcement Learning avec des approches de deep Q-Learning.

Une grande partie de ce projet était l'apprentissage des techniques de Reinforcement Learning et ensuite de l'exploration afin d'améliorer les résultats dans un scénario donné.

Deux scénarios ont été utilisés pour tester l'efficacité de l'IA ainsi développée :

— Le jeu vidéo Doom, où le but était de survivre le plus longtemps possible face à des vagues d'ennemis sur lesquels l'IA devait tirer.
— Le jeu vidéo Flappy Bird où le but pour l'IA était d'arriver le plus loin possible dans le jeu en sautant pour évoluer dans le niveau en évitant de s'écraser contre les tuyaux.

Les résultats obtenus ont été convaincants : dans Doom, l'IA a développé un comportement où elle visait les ennemis apparaissant et leur tirait dessus, et dans Flappy Bird l'IA a réussi à trouver comment doser les sauts pour passer au mieux entre les tuyaux.

Mots-clés : intelligence artificielle, IA, Reinforcement Learning, RL, Doom, Flappy bird, Q-Learning, python, tensorflow, keras-rl, VizDoom, PLE, Py Learning Environment

English

This report presents the progress of a semester project carried out at HE-Arc, which consisted in developing a python module to make Reinforcement Learning with deep Q-Learning approaches.

The bulk of this project consisted in learning Reinforcement Learning techniques and then exploring to improve results in a given scenario.

Two scenarios were used to test the effectiveness of the developed AI :

— The video game Doom, where the goal was to survive as long as possible against waves of enemies on which the AI had to shoot.
— The video game Flappy Bird where the goal for the AI was to get as far as possible in the game by jumping to evolve in the level and avoiding crashing against the pipes.

The results were convincing : in Doom, the AI developed a behavior where it targeted the enemies appearing and shot at them, and in Flappy Bird the AI managed to find out how to gauge the jumps to pass between the pipes in the best way possible.

Keywords : artificial intelligence, AI, Reinforcement Learning, RL, Doom, Flappy bird, Q-Learning, python, tensorflow, keras-rl, VizDoom, PLE, Py Learning Environment

Table des matières

Chapitre 1

Introduction

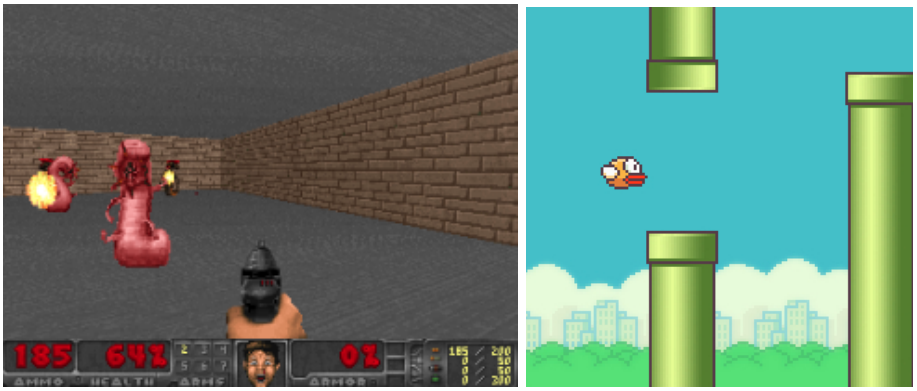


FIGURE 1.1 – A gauche : le jeu vidéo Doom, à droite : le jeu vidéo FlappyBird, tous les deux ont pu être jouables par une IA grâce aux techniques de Reinforcement Learning

1.1 Qu'est-ce que le Reinforcement Learning ?

Le Reinforcement learning (apprentissage par renforcement en français), est une technique utilisée en intelligence artificielle consistant à faire apprendre à une IA quelles actions entreprendre selon l'état de l'environnement au moment d'entreprendre cette action, afin de maximiser un score.[7,]

Pour que cette technique fonctionne, il faut un environnement capable de recevoir des actions à entreprendre et retournant un état et une récompense issue de l'action entreprise précédemment au sein de cet environnement.

Le but de l'agent est de maximiser la somme des récompenses ainsi obtenues, pour ce faire il y a plusieurs algorithmes et façons de prédire la récompense que donnera une action en se basant sur des expériences passées.

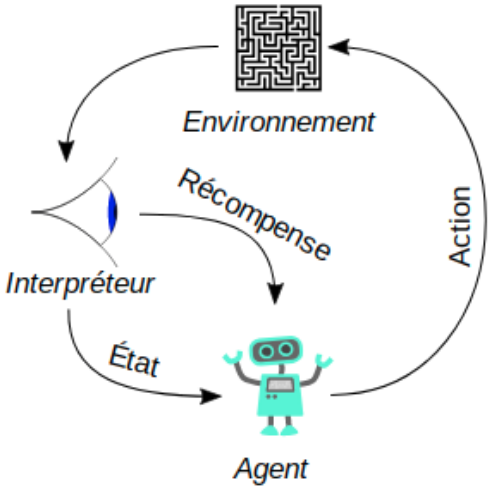


FIGURE 1.2 – Illustration du principe de base du reinforcement learning [7,]

Source: rwikipedia

Quelques précisions relatives aux termes utilisés

Agent : Un acteur qui entreprends des actions en agissant, une IA dans le cas du Reinforcement Learning

Action : Action possible à entreprendre dans un contexte donné, par exemple sauter, avancer, tirer dans un jeu-vidéo

Récompense/score : Comme un score qu'un joueur gagnerait dans un jeu-vidéo ou comme une récompense comme de la dopamine dans le cerveau

Environnement : Entité avec laquelle l'agent interagit en y effectuant des actions (p. ex un jeu-vidéo, une simulation, le monde réel)

Etat : Interprétation de l'environnement en tant que vecteur caractéristique ou les pixels affichés à l'écran

Politique : Règles guidant des décisions afin d'atteindre un résultat voulu

Interpréteur : Entité qui permet d'interpréter l'environnement afin de le représenter avec un état sous la forme qu'attends l'algorithme de Reinforcement Learning, de plus l'interpréteur décide si le changement de l'environnement a provoqué une récompense (positive ou négative) ou non (il est à noter que dans certains frameworks de Reinforcement Learning, l'interpréteur est intégré à l'environnement)

Beaucoup de ces termes viennent de ce qu'on appelle "processus de décision markovien" ou "modèle de Markov", qui est un modèle développé dans les années 1950 dans le cadre d'une discipline appelé "théorie de décision" permettant de modéliser des systèmes qui changent aléatoirement. Dans le cadre de ce modèle, un agent prends des décisions et les résultats de ses actions sont aléatoires. A chaque étape, le processus est dans un certain état, et l'agent choisit une certaine action qui rapportera une certaine récompense à l'agent.

1.2 Utilisations et prouesses du Reinforcement Learning

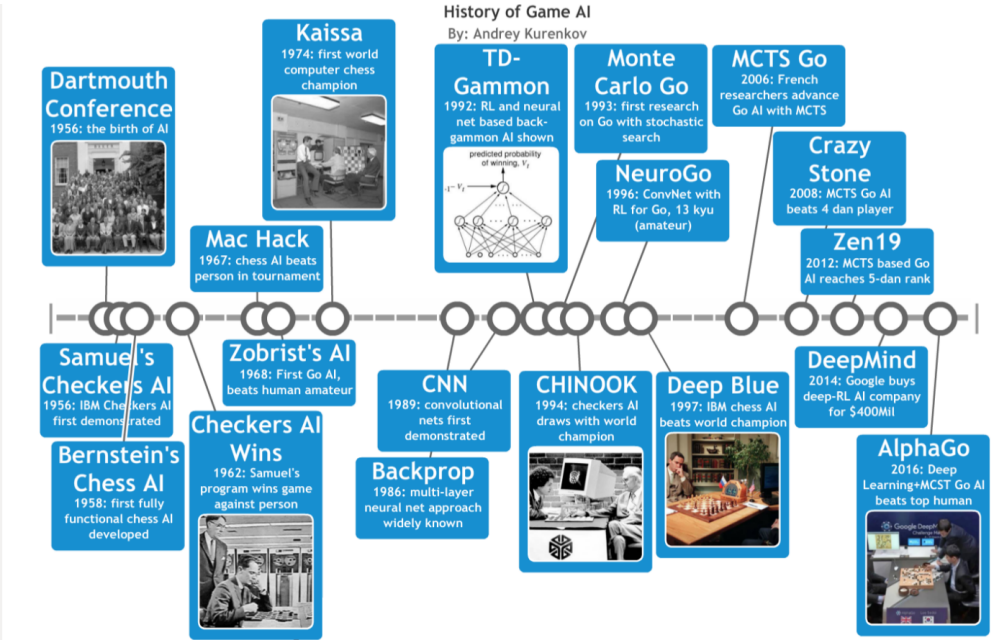


FIGURE 1.3 – Différentes étapes dans l'histoire des IA de jeux jusqu'en 2016, à noter AlphaGo qui se situe tout à droite et que depuis il y a encore eu du progrès [1,]

Le Reinforcement Learning avait beaucoup fait parler de soi quand l'IA mis au point par DeepMind (l'IA s'appelait AlphaGo) a battu le champion du monde dans le jeu de Go en 2016.

Récemment, un projet nommé "OpenAI Five" a mis au point une équipe de 5 bots collaborant entre eux pour jouer au jeu vidéo "Dota 2", pour l'instant ils ont réussi à battre des équipes de joueurs amateurs mais cela est considéré comme une grande prouesse car ce jeu est considéré comme étant complexe pour une IA à cause de la collaboration requise entre joueurs pour gagner une partie.

Le reinforcement learning est aussi utilisé dans le cadre des voitures autonomes, pour cela il est possible d'utiliser un jeu-vidéo comme "GTA V" en tant que simulation avant d'appliquer un modèle ainsi entraîné dans la vraie vie.

1.3 Différents algorithmes de Reinforcement Learning utilisés dans ce projet

1.3.1 Q-Learning

Le Q-Learning est une méthode d'apprentissage utilisé dans le Reinforcement learning. La lettre "Q" (comme "Qualité") désigne la fonction qui prédit la qualité d'une action exécutée dans un état donnée du système en se basant sur des expériences passées. La valeur de sortie de cette fonction est désignée "Q-value" [7,]

Le but du Q-Learning est d'apprendre une politique qui dit à l'agent quelle action entreprendre selon quelle circonstance. Pour cela on n'a pas besoin d'un modèle de l'environnement établi préalablement.

Le Q-Learning trouve une politique qui est optimale dans le sens où elle maximise la valeur attendue de la somme de récompense sur toutes les étapes successives.

Voici la définition formelle de cette fonction : $Q[s,a] := (1-\alpha)Q[s,a] + \alpha(r + \gamma max_{a'}Q[s',a'])$

— s est un état (state)
— a est une action
— r est la récompense liée à ce couple état-action
— α est le facteur d'apprentissage (learning rate), il détermine à quel point la nouvelle information calculée surpassera l'ancienne. Un facteur de 0 ne ferait rien apprendre à l'agent, alors qu'un facteur de 1 ne ferait considérer à l'agent que la dernière information. Une valeur proche de 1 est optimale dans un environnement déterministe
— γ est le facteur d'actualisation (discount rate), il détermine l'importance des récompenses futures. Un facteur 0 ferait considérer que les récompenses courantes à l'agent, et un facteur proche de 1 ferait intervenir les récompenses lointaines.

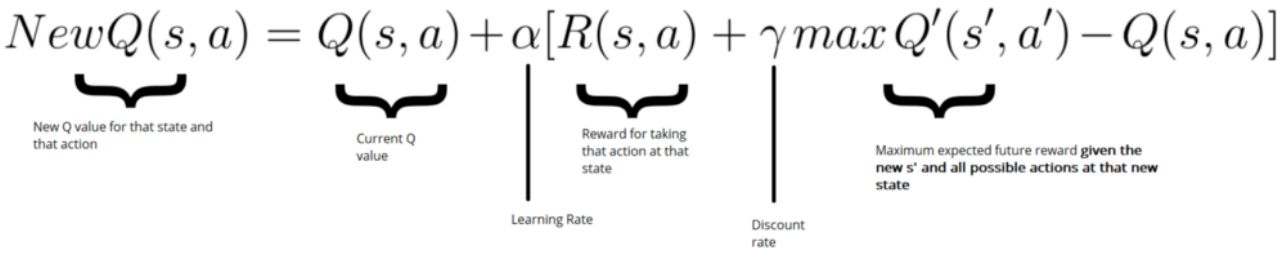


FIGURE 1.4 – Explication visuelle de la fonction Q [5,]

En pratique, pour implémenter cette méthode on utilise un tableau à 2 dimensions (appelée "Q-Table"), où un indice représente un état et l'autre indice représente une action et la case de la matrice liée à une action combinée avec un état contiendra le résultat de la fonction "Q".

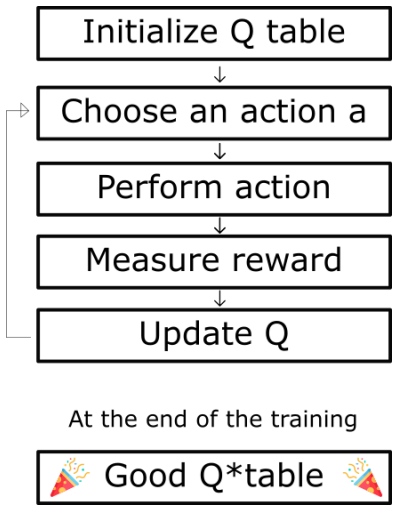


FIGURE 1.5 – Schéma expliquant les actions à entreprendre lors d'un remplissage d'une Q-Table en pratique [5,]

Cette technique est viable dans un cas où l'espace d'actions et d'états possible n'est pas "trop" grand (car le tableau bidimensionnel aura une taille de $A * S$ avec A le nombre d'actions possibles et S le nombre d'états possibles), dans un cas où l'état serait représenté par des pixels, cette solution n'est plus valable car il y aurait beaucoup trop d'états possibles et le tableau bidimensionnel contenant les Q-Values serait trop grand pour être stocké dans la mémoire d'un ordinateur.

1.3.2 Deep Q-Learning (DQN)

Afin de pouvoir appliquer le Q-Learning à des scénarios ayant un vaste espace d'états, une solution est d'utiliser un réseau de neurones qui va approximer pour un état donné, les différentes Q-values pour chaque action possible.[5,]

Grâce à l'utilisation des réseaux de neurones, on n'est plus obligé de stocker tous les tuples état-action dans un tableau bidimensionnel, mais on peut utiliser le réseau de neurones pour approximer une fonction qui donnerait une Q-Value pour un état donné, et le résultat de cette approximation sera stocké dans une mémoire avec les poids des neurones correspondant à chaque état.

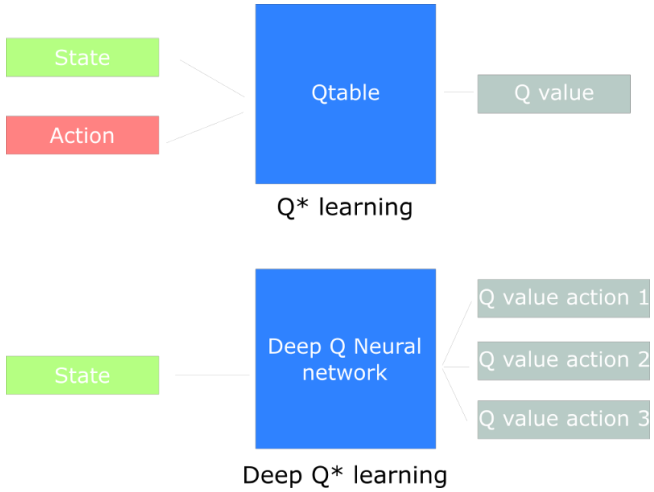


FIGURE 1.6 – Schéma illustrant le changement entre le Q-Learning et le Deep Q-Learning [5,]

Qu'est-ce qu'un réseau de neurones ? Dans le cadre de l'IA, un réseau de neurones est composé de neurones disposés en couches, et les neurones d'une couche sont reliés aux neurones de la couche suivante. Chaque couche traite les informations de la couche précédente et les transmets aux couches suivantes.

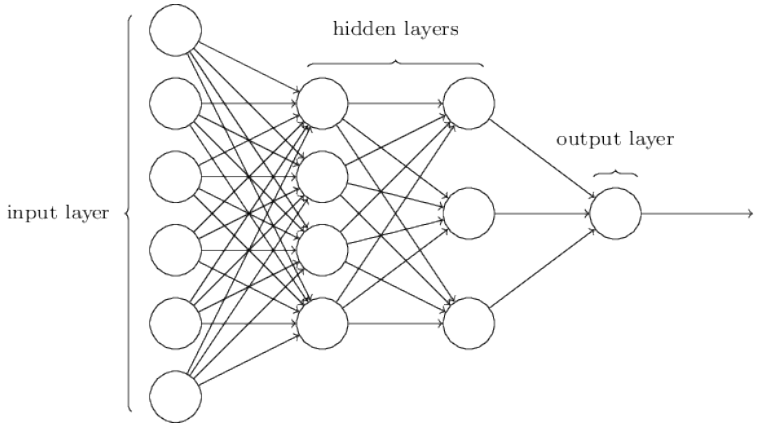


FIGURE 1.7 – Schéma illustrant l'architecture et la propagation d'informations dans un réseau de neurones [6,]

Dans le cas du deep Q-learning, le réseau aura en entrée un état, et en sortie une Q-value associé par état possible.

A force de faire des expériences, l'agent nourri le réseau de neurones et l'approximation de la Q-value devient de mieux en mieux en réduisant l'erreur entre la valeur prédite et la valeur maximale possible du prochain état ("Q-Target").

Cette approche permet d'intégrer des couches de neurones convolutifs à l'entrée du réseau, permettant de traiter des états issus de pixels, comme dans le cas d'un jeu-vidéo comme doom. Sans l'utilisation de réseaux de neurones ceci ne serait pas possible.

Un soucis de cette technique est que les poids reliant les neurones entre eux ne sont que valables pour un état donné, donc quand on change d'état, les poids doivent s'adapter et le réseau "oublie" les poids des états précédents.

Pour corriger ce soucis, nous avons besoin d'une mémoire qui stocke les expériences précédentes, et on utilisera des échantillons pris de façon aléatoire parmi les expériences passées pour nourrir notre réseau de neurones. Ceci permettra au réseau de neurones d'apprendre sur le long terme en se basant sur une large gamme d'expériences effectués par l'agent et ne pas apprendre seulement en se basant sur les expériences les plus récentes. Cette mémoire s'appelle "replay buffer".

1.3.3 Améliorations du deep Q-Learning

Depuis l'introduction du deep Q-Learning en 2014, plusieurs améliorations ont été mises au point pour pallier à certains points faibles que contenait cette technique, les sections suivantes présentent quelques unes qui ont été utilisés dans le cadre de ce projet.

1.3.3.1 Fixed Q-targets

Afin de réduire la marge d'erreur entre la prédiction de la Q-value et la Q-target (valeur maximale possible pour le prochain état), on calcule la différence entre la prédiction et la vraie valeur. Sauf que, la prédiction n'est qu'une estimation de la Q-value. Mais la valeur estimée et la vraie valeur sont issus du même réseau de neurones, donc il y a une forte corrélation entre les deux. Ce qui veut dire qu'à chaque nouvelle étape de l'entraînement, un décalage de la vraie valeur entraîne un décalage de l'estimation ce qui entraîne une oscillation de la marge d'erreur pendant l'entraînement.

Une solution à ce problème à été proposée par DeepMind : utiliser un réseau de neurones séparé avec un paramètre fixe pendant une certaine période pour estimer la Q-value. Grâce à ça, la phase d'apprentissage sera plus stable car la Q-value estimée reste la même pendant un moment.

1.3.3.2 Double DQN

Un autre problème du DQN est la possibilité de surestimer la Q-value, et on n'est plus sûr que la meilleure action à entreprendre dans un état donné est forcément celle qui a la plus haute Q-value.

Ce problème survient surtout au début de l'entraînement, lorsque on ne dispose pas d'assez expériences pour pouvoir choisir la meilleure action à entreprendre.

La solution à ce problème est d'utiliser le 2ème réseau de neurones introduit par la solution des fixed Q-targets afin de découpler le choix de l'action de la génération de l'estimation de la Q-value. Pour cela nous utiliseront le réseau principal pour sélectionner la meilleure action à entreprendre pour un état donné basé sur la plus haute Q-value, et le second réseau pour estimer la Q-Value au prochain état si cette action sera prise. Ceci permet de réduire sur surestimation de la Q-Value et donc permet d'avoir un entraînement plus stable et rapide.

1.3.3.3 Dueling DQN

Une Q-value représente à quel point entreprendre une action dans un état donnée nous sera favorable, donc on peut décomposer $Q(s,a)$ en : $Q(s,a) = A(s,a) + V(s)$, avec $V(s)$ la Q-valeur en étant à cet état et $A(s,a)$ le gain (advantage) qu'on aura en prenant cette action dans cet état, ou à quel point cette action nous rapportera plus que les autres actions possibles à cet état.

Le principe du dueling DQN est de séparer l'estimation de ces deux éléments avec deux flux : un qui estime la valeur lié à un état $V(s)$ et un autre qui estime le gain de chaque action $A(s,a)$

Ensuite, ces deux flux seront combinés pour finalement donner une estimation de $Q(s,a)$.

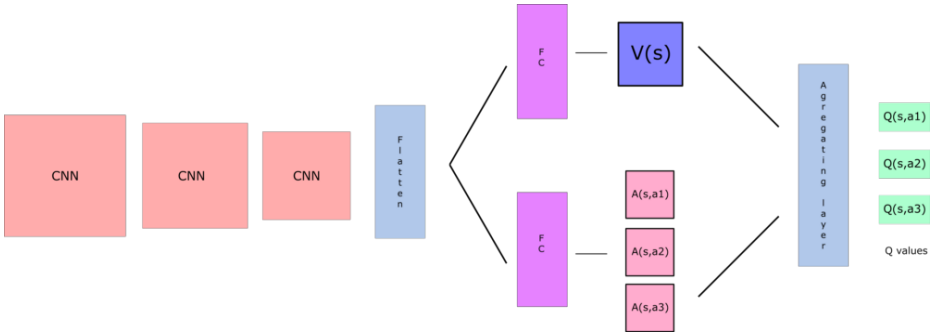


FIGURE 1.8 – Schéma illustrant la décomposition en deux flux pour estimer séparément $A(s,a)$ et $V(s)$ [5,]

En séparant cette estimation, notre réseau de neurones peut apprendre quel état a de la valeur sans apprendre l'effet de chaque action à chaque état. Ceci est particulièrement utile pour des états où les actions qui y sont liés ne provoquent pas un changement conséquent de l'environnement. Ceci accélère encore l'apprentissage du modèle vu qu'on a moins de combinaisons d'états-actions à calculer.

1.3.3.4 Prioritized experience replay

Le Prioritized experience replay (PER) a été introduit en 2015, l'idée est que certaines expériences peuvent être plus importants que d'autres pour notre entraînement mais peuvent survenir rarement.

Vu que l'échantillonage de la mémoire se fait d'une façon aléatoire uniforme, ces expériences importantes qui surviennent pas souvent n'ont pas beaucoup de chances d'êtres sélectionnés.

L'idée du PER est de changer la distribution aléatoire de l'échantillonage de la mémoire en utilisant un critère définissant une priorité pour chaque tuple d'expérience stocké.

Les expériences qui devraient être priorisées sont celles où il y a une grande différence entre la prédiction de la Q-value et la Q-target, ce qui veut dire qu'on a plus à apprendre de cette expérience vu qu'elle a donné une estimation de la Q-value qui était plus fausse que d'habitude.

1.4 Rainbow

Dernièrement (octobre 2017) a été développé et testé par DeepMind [10,] un algorithme s'appelant "rainbow".

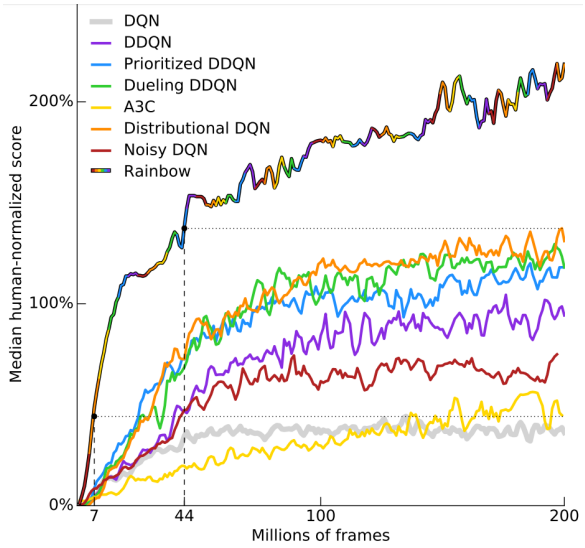


FIGURE 1.9 – Graphique montrant le résultat des tests pour des scénarios où l'agent joue à des jeux Atari, le score est mesuré et comparé à un score qu'un humain aurait fait en moyenne [10,]

La figure 1.9 montre d'où vient le nom "rainbow" : chaque couleur représente une façon de faire du DQN, et "rainbow" est la combinaison de toutes ces méthodes ce qui donne un arc en ciel.

On peut remarquer que rainbow intègre toutes les techniques améliorant le deep Q-Learning expliqué dans la section précédente, et bien plus encore.

Selon la figure 1.9 les résultats issu de l'utilisation de cet algorithme dépassent largement tous les autres dans le cas des jeux Atari, c'est pourquoi il a été jugé intéressant d'essayer un framework implémentant cet algorithme dans le cadre de ce projet, les résultats issu de cet essai sont expliqués dans le chapitre suivant.

Chapitre 2

Analyse

2.1 Comparaison de frameworks de Reinforcement Learning

La première phase du projet hormis la prise en main des algorithmes de Reinforcement Learning, a été de d'explorer et de comparer deux frameworks facilitant la mise en place d'algorithmes de Reinforcement Learning pour pouvoir les appliquer à un scénario donné.

Pour la comparaison a été choisi le jeu-vidéo Doom dont l'interfaçage avec un agent RL a été rendu possible grâce à ViZDoom [11,]

Ce choix a été fait car la seule façon de percevoir l'environnement de Doom pour un agent est d'analyser les pixels à l'écran, ce qui est que possible avec les techniques de Deep Q Learning et c'était donc une façon de tester à pleine puissance les frameworks retenus.

Le scénario utilisé pour la comparaison faisait partie de ceux fournis par ViZDoom [9,], s'appellant "defend the line" il consistait en un niveau de jeu conçu spécialement pour tester la performance d'un agent à survivre le plus longtemps dans un environnement où le but était de tirer sur des ennemis apparaissant en face de lui.

Les actions que pouvait entreprendre l'agent étaient les suivants :

— Tourner à gauche
— Tourner à droite
— Tirer droit devant avec un pistolet

Les ennemis apparaissent à des positions donnés au début de la partie, et à chaque fois qu'un ennemi meurt il réapparaît avec plus de points de vie qu'avant (à chaque réapparition il faut une balle de plus pour le tuer que la fois d'avant)

La fonction de récompense était celle proposée par défaut dans un premier temps : +1 par ennemi tué et -1 à la mort de l'agent. Cette fonction était utilisée pour comparer différents frameworks avec le même algorithme (Table 2.2).

Dans un second temps, après inspiration venant de VizDoom-Keras-rl [8,], il a été jugé intéressant de faire baisser la récompense de 0.1 à chaque fois que l'agent tirait avec son arme et à chaque fois qu'il perdait de la vie. Ceci pourrait lui apprendre à économiser les munition et éviter de se faire toucher par les ennemis. Cette fonction était utilisée pour les résultats de la Table 2.3.

2.1.1 Critères de comparaison

Caractéristiques du framework et son évaluation

La présence ou non des éléments suivants :

— Double DQN
— Dueling DQN
— PER (prioritized experience replay)
— Fixed Q-target
— Rainbow

L'évaluation des points suivants :

— Facilité d'adaptation
— Facilité d'utilisation
— Facilité d'installation
— Facilité d'interfaçage avec un environnement personnalisé
— Qualité de la documentation

Et deux critères "pratiques"

— Systèmes d'exploitations supportés
— Dernière mise à jour sur github (au moment de la comparaison)

Mesures quantifiables pour un scénario donné

— Temps d'apprentissage pour 750 épisodes
— Score moyen sur 100 épisodes de test

Le nombre de 750 épisodes d'apprentissage vient du fait que c'est à partir de là que la version "sans framework" [3,] du algorithme donnait des résultats satisfaisants, c'est à dire que l'agent avait développé un comportement lui permettant de survivre dans le scénario en visant les ennemis explicitement avant de leurs tirer dessus.

Le nombre de 100 épisodes de test a été jugé suffisant pour que l'influence de facteurs aléatoires ne soit pas significative.

2.1.2 Démarche de test

Afin d'avoir la comparaison la plus juste possible entre les frameworks, il a fallu dans un premier temps comparer les résultats pour un même algorithme donné et avec les hyperparamètres se rapprochant le plus possible.

Les hyperparamètres partagés utilisés étaient les suivants :

— Algorithme : dans un premier temps, DQN sans améliorations
— Fonction de score : décrite plus haut
— Nombre de frames empilés : 4
— Taille image originale : 320x240
— Taille image redimensionné : 78x51
— Learning rate : 0.00025
— Nombre d'épisodes d'entraînement : 750
— Taille d'échantillon issus de la mémoire (batch size) : 64
— Epsilon variant de 1 à 0.01 avec un taux de décroissance exponentielle à 0.00005
— Gamma (discounting rate) : 0.95
— Taille de la mémoire : 50000
— Nombre de step d'échauffement (pour remplir la mémoire avec des échantillons aléatoires) : 50000
— Nombre d'épisodes de test : 100
— Réseau de neurones :
  1. 4 images empilées servent à nourrir le réseau
  2. Ils passent par 3 couches de convolution
  3. Ensuite on aplatit
  4. 2 couches entièrement connectées
  5. En sortie, une Q-Value pour chaque action

Dans un second temps, le but était de trouver le meilleur algorithme proposé par les deux frameworks retenus, donc la plupart des hyperparamètres ont été gardés (le réseau de neurones change évidemment si on utilise le double DQN par exemple), et certains hyperparamètres ont été ajoutés étant spécifique à des algorithmes.

2.2 Choix de framework de Reinforcement Learning

Les frameworks à comparer retenus ont été les suivants :

— keras-rl : https://github.com/keras-rl/keras-rl
— dopamine : https://github.com/google/dopamine

De plus, les résultats ont été comparés avec une solution sans framework [3,]

	dopamine	**keras-rl**
Double DQN	Oui	Oui
Dueling DQN	Non	Oui
PER	Oui	Oui (branche github : keras-rl-v0.4.2)
Fixed Q-target	Oui	Oui
Rainbow	Oui	Non
Os supportés	Mac, Linux, (Windows si on n'utilise pas atari-py)	Windows, Mac, Linux
Dernière mise à jour du code sur github	21.11.18	06.11.18

TABLE 2.1 – Comparaison de critères généraux concernant les frameworks

Concernant les mises à jours du code sur github :

— dopamine est mis à jour plus régulièrement mais il est assez récent sorti en Août 2018) donc beaucoup de mises à jour corrigent des bugs et le rendent plus modulaire
— keras-rl est mis à jour plus rarement mais il est déjà mature (sorti en Juillet 2016) donc il a moins de bugs et plus de modularité a été ajouté depuis le temps qu'il a été mis en ligne pour la première fois

Il est à noter que l'implémentation de l'algorithme rainbow dans dopamine diffère de celle du papier scientifique publié par Google DeepMind [10,]

dopamine n'implémente que les éléments suivants en plus de l'algorithme DQN de base :

— Double Q Learning (pas mis sur le README du projet, mais présent et implémenté dans le code)
— n-step Bellman updates (fixed Q-target)
— Prioritized experience replay
— Distributional reinforcement learning (C51)

Dans le papier scientifique, les éléments suivants étaient présent mais ne sont pas implémentés dans dopamine :

— Dueling networks
— Noisy nets

Evaluation de keras-rl

Facilité d'adaptation

— (moyen) Dans `dqn.fit` on peut définir qu'une limite en steps et pas une limite en épisodes par défaut mais c'est facilement modifiable
— (moyen) L'implémentation du PER a dû être récupérée dans une branche séparé sur github car le PER n'est pas supporté dans la version de la branche master
— (bien) Système de callback prévu pour appeler une certaine fonction à des étapes importantes de l'entraînement ou de l'évaluation : par exemple au début de l'entraînement, à la fin de chaque épisode, à la fin de l'entraînement
— (bien) Classes abstraites prévues pour interfacer les agents disponibles dans le framework avec un environnement personnalisé et pour écrire ses propres fonctions callback pour faire du logging

Facilité d'utilisation

— (bien) Si on se base sur les exemples fournis
— (bien) Les réseaux de neurones utilisent keras, qui est plus facile à utiliser que `tf.slim`

Facilité d'installation

— (bien) Pas de soucis particuliers

Facilité d'interfaçage avec environnement existant

— (bien) L'architecture le permet facilement

Qualité de la documentation

— (moyen) la documentation [?,] contient quelques "write me" mais les points importants (quel module sert à quoi et comment) sont bien expliqués, quelques fois se plonger dans le code source était nécessaire mais il est bien commenté et l'utilité de diverses parties du code est claire, de plus on peut facilement se baser sur les quelques exemples disponibles sur github pour bien comprendre le fonctionnement du framework

Evaluation de dopamine

Facilité d'adaptation

— (moyen) Architecture modulaire : **agent**, **environnement**, **"runner"** (gère ce qu'il faut faire entre les épisodes par exemple)
— (bien) Dans `Runner.run_one_phase` on peut définir qu'une limite en steps et pas une limite en épisodes par défaut (ce qui est suffisant selon les besoins d'un scénario, mais dans mon cas la condition d'arrêt d'un entraînement était un nombre d'épisodes), mais c'est facilement modifiable

Facilité d'utilisation

— (bien) Si on veut utiliser l'environnement atari fourni par défaut on n'a rien à faire
— (bien) Si on veut modifier les réseaux de neurones, dopamine utilise `tf.slim` qui offre une couche d'abstraction par dessus tensorflow donc c'est plus facile qu'avec tensorflow de base

Facilité d'installation

— (bien) Pas de soucis particuliers, sauf si on veut utiliser sur windows l'environnement atari fourni par défaut (il n'y a pas de version pour windows du module atari-py)

Facilité d'interfaçage avec environnement existant

— (moyen) L'architecture le permet facilement même s'il n'y a pas de documentation fait pour, il faut étudier le code source pour comprendre comment faire

Qualité de la documentation

— (bien) Les points importants sont bien expliqués [4,] (organisation et utilité des modules du framework), il y a peu d'exemples mais les plus importants sont là (comment créer un agent personnalisé, visualiser résultats avec tensorboard)

2.2.1 Résultats des tests

	dopamine	**keras-rl**	**sans framework**
Temps d'apprentissage pour 750 épisodes	2h	2h	9h19
Score moyen sur 100 épisodes de test	10.84	13.97	3.19

TABLE 2.2 – Comparaison entre frameworks pour un scénario de test et algorithme donné (DQN sans améliorations avec fonction de score par défaut sur le scénario "defend the line")

	dopamine implicit quantile	**dopamine rainbow**	**keras-rl dqn+**	**keras-rql dqn**
Temps d'apprentissage pour 750 épisodes	53 m	48 m	3h52	2h7
Score moyen sur 100 épisodes de test	5.8	6.25	16.837	4.04

TABLE 2.3 – Comparaison entre frameworks et algorithmes pour un scénario de test donné, avec une fonction de score différente que celle utilisée dans le tableau précédent (dqn+ représente l'algorithme dqn avec toutes les améliorations activables dans keras-rl)

L'agent implémentant implicit quantile hérite de celui qui implémente rainbow donc il est sensé ajouter plus d'améliorations à l'algorithme.

Beaucoup de variations des hyperparamètres sur les deux algorithmes disponible dans dopamine n'ont pas donné de différences significatives aux scores.

Plusieurs entraînement ont été effectués, et bien que le score d'évaluation varie d'une fois à l'autre, l'ordre de grandeur de différence entre dopamine et keras-rl reste le même.

Conclusion des tests

Étonnamment, c'est keras-rl qui a largement de meilleurs scores avec son meilleur algorithme DQN par rapport aux deux implémentations de l'algorithme rainbow de dopamine.

Par contre au niveau du temps que prends un entraînement pour le même nombre d'épisodes, les algorithmes implémentés par dopamine (rainbow et dérivées) sont beaucoup plus rapides.

Dans les évaluations des agents entraînés avec dopamine a été observé une certaine "paresse" de l'agent, c'est à dire qu'il se contenait au bout d'un moment (une fois que l'epsilon a fini de diminuer) de ne plus tourner ni à gauche ni à droite et il tirait en face de lui même sans ennemis devant mais il finissait par en toucher ceux qui passaient devant lui.

Par contre l'agent entraîné avec keras-rl se tournait en direction des ennemis quand ils apparaissaient et leurs tirait dessus de façon ciblé, ce qui augmentait sa durée de survie et donc son score, ce qui prouve un certain apprentissage permettant de faire le lien entre les ennemis apparaissant sur l'écran et l'action de leur tirer dessus pour augmenter son score

La seule chose que l'algorithme utilisé par keras-rl avait de plus que dopamine était le dueling network, mais après des tests supplémentaires où cette fonctionnalité était désactivée, keras-rl était quand même meilleur, avec un écart moins important.

Peut être que l'implémentation de dopamine visait à améliorer les performances dans le cadre des jeux Atari, utilisés pour mesurer la performance de leur algorithme dans leur papier scientifique et sur lesquels le framework repose par défaut. Ce qui améliorait les résultats dans les jeux Atari n'est peut être pas valable dans le cadre d'un scénario sur Doom, d'où cette surprise et différence au niveau des scores obtenus.

Plus tard pendant la phase d'implémentation quand le sujet d'étude était le scénario FlappyBird, de nouveaux tests ont été menés pour voir si dopamine serait plus efficace dans un environnement non visuel. L'ordre de grandeur de la différence des scores entre dopamine et keras-rl était le même que pour Doom et donc la conclusion des paragraphes précédents reste inchangé.

2.2.2 Motivation du choix

Au niveau des critères relatifs à l'évaluation (facilité d'utilisation, documentation etc.), les deux frameworks se valent. keras-rl a une certaine maturité par rapport à dopamine ce qui le rends moins risqué au niveau de potentiels bugs.

Il a été jugé que le critère du temps que prends un entraînement est moins important que le score obtenu, donc le choix a été fait d'utiliser keras-rl et non pas dopamine pour la suite de ce travail.

Chapitre 3

Conception

3.1 Choix des critères de mesure

Au début, c'est le score qui a été choisi pour comparer l'efficacité d'une configuration de hyperparamètres par rapport à une autre.

Sauf que ce choix avait un problème : vu que la fonction de score fait partie des hyperparamètres, quand elle est changée, la comparaison avec les configurations précédentes n'est plus pertinente.

La solution finale a été de comparer le nombre de frames que l'agent survit chaque épisode, vu que les deux scénarios consistent à survivre le plus longtemps possible dans un niveau, cette mesure est un indicateur qui est indépendant des hyperparamètres et qui permet justement de comparer efficacement une configuration à une autre.

3.2 Choix des jeux pour tester les algorithmes

3.2.1 Doom

La décision a été prise de continuer d'utiliser Doom, pour avoir un environnement représenté par des pixels. Ceci permettra d'utiliser le plein potentiel du framework, car c'est un problème qui est difficile pour un agent à apprendre avec des techniques conventionnelles à cause du nombre de données qu'engendrent ces pixels pour représenter un état.

Il est à noter qu'en plus des pixels, l'environnement nous fourni diverses variables à chaque step du jeu, voici celles qui étaient utilisés notamment pour la fonction de score :

— Points de vie de l'agent
— Munitions de l'arme actuelle
— Nombre de fois que notre arme a touché un ennemi

3.2.1.1 Choix du scénario doom

Le choix a été pris de continuer avec le même scénario que celui utilisé pour comparer les différents frameworks dans le chapitre précédent (defend the line), car c'était un problème relativement simple où l'optimisation des hyperparamètres tels que la fonction de récompense serait le plus aisé.

Le scénario se compose d'un fichier .wad qui contient la description du niveau : l'agencement en 3d du niveau et les scripts qui décrivent la logique (fonction de récompense par défaut et faire apparaître les ennemis au début du niveau et ensuite quand on les tue en augmentant leurs points de vie)

3.2.2 Flappy Bird

L'interaction entre flappy bird et le jeu était rendu possible grâce à PLE (PyGame Learning Environment) [12,]

PLE propose pour chaque scénario (jeux) une ou deux façons de récupérer les informations relatives à l'environnement : soit les pixels à l'écran soit dans un dictionnaire contenant diverses variables décrivant l'environnement.

C'est pour la présence de cette dernière option que Flappy Bird a été choisi, pour avoir un exemple d'environnement décrit avec un petit nombre de variables, mais qui suffit amplement pour décrire cet environnement dans ce cas.

Les actions que pouvait entreprendre l'agent à chaque frame étaient la possibilité de sauter ou ne rien faire.

3.3 Architecture logicielle

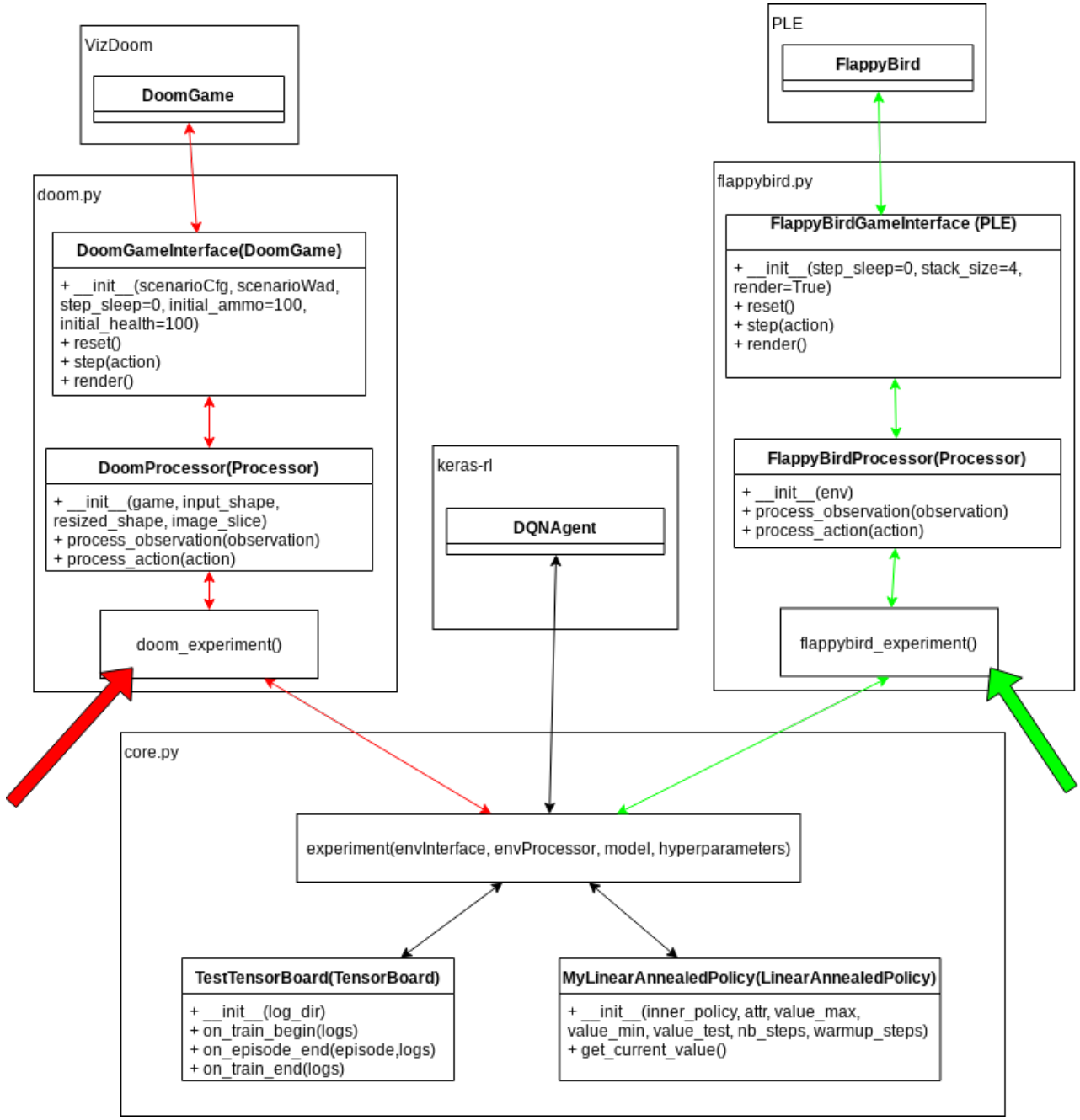


FIGURE 3.1 – Les grosses flèches remplies représentent le points d'entrée des programmes (rouge pour Doom et vert pour FlappyBird)
Les flèches fines ne sont pas de l'héritage au sens UML mais représentent l'interaction entre les divers composants Les flèches noires sont communes à Doom et Flappybird, les flèches rouges ne concernent que Doom et les flèches vertes ne concernent que FlappyBird

Chapitre 4

Implémentation

4.1 Description de l'environnement de développement

Voici un tableau récapitulatif des différentes versions des outils utilisés dans ce projet :

outil	ViZDoom	tensorflow	tensorboard	Keras	keras-rl	ple	scikit-image
version pip	1.1.6	1.12.0	1.12.0	2.2.4			0.14.2
commit github					e94ef88 (master) 0e5dc94(keras-rl-v0.4.2)	ba6ac84 (master)	

TABLE 4.1 – Tableau récapitulatif des différentes versions des outils utilisés dans le projet

A noter que pour keras-rl, l'implémentation du PER vient d'une autre branche que master

4.2 Doom

4.2.1 Réseau de neurones

Les données nourrissant le réseau de neurones sont un empilement de 4 images en niveau de gris recadrés et redimensionnés, les images sont fournis par l'environnement ViZDoom

Explication des étapes et couches du réseau de neurones :

1. L'image venant de Doom est déjà en niveaux de gris et a une taille de 320x240
2. Des parties inutiles ont été coupées de l'image venant du jeu (voir figure 4.1)
3. L'image est ensuite redimensionnée pour prendre moins de place (78x51 dans une première configuration utilisé dans le tableau résumant le réseau de neurones de la page suivante, 160x84 dans une 2ème configuration expliquée plus tard)
4. Ceci s'applique pour 4 images pris à la suite temporellement qui sont ensuite empilés, permettant d'avoir une notion de temporalité
5. Une première couche normalise les niveaux de gris (division par 255)
6. Une seconde couche fait une permutation des paramètres d'entrée pour correspondre au format qu'attends keras-rl
7. Diverses couches de convolution s'inspirant des travaux venant de VizDoomkeras-rl [8,]
8. Finalement, les dernières couches servent à estimer la Q-Value pour l'état passé en entrée dans le réseau et prendre la décision d'effectuer l'action ayant la plus haute Q-Value

9. Il est à noter que si on active de dueling network, keras-rl utilise les dernières couches du réseau de neurones qu'on lui fournit pour les "séparer" en deux flux selon l'algorithme du dueling network, et fait ensuite une agrégation des deux flux séparés, donc le modèle qu'on lui passe est fait comme s'il n'était pas prévu de faire du dueling network

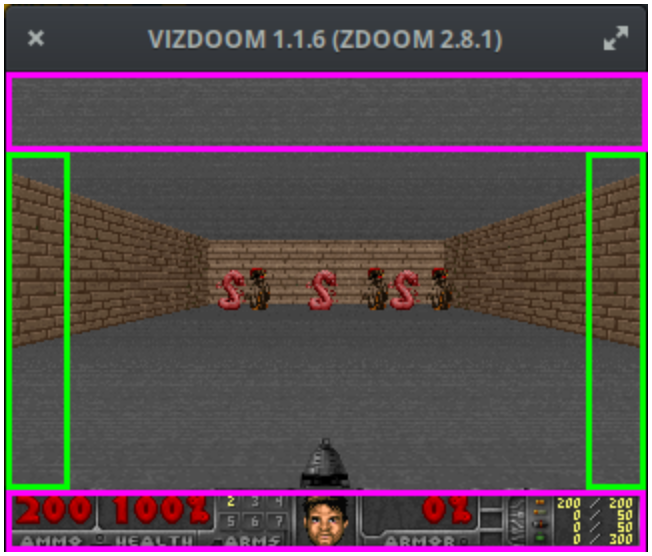


FIGURE 4.1 – En rose et vert : parties coupées pour la 1ère configuration (plus tard redimensionné en 78x51), en rose : parties coupées pour la 2ème configuration (plus tard redimensionné en 160x84)

Voici le résultat de la fonction `model.summary()` appliqué sur le modèle passé à keras-rl :

```
Layer (type)                 Output Shape              Param #
=================================================================
lambda_1 (Lambda)            (None, 4, 51, 78)         0
_________________________________________________________________
permute_1 (Permute)          (None, 51, 78, 4)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 18, 32)        8224
_________________________________________________________________
activation_1 (Activation)    (None, 11, 18, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 8, 64)          32832
_________________________________________________________________
activation_2 (Activation)    (None, 4, 8, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 3, 128)         131200
_________________________________________________________________
activation_3 (Activation)    (None, 1, 3, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 384)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               197120
_________________________________________________________________
activation_4 (Activation)    (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 1539
_________________________________________________________________
activation_5 (Activation)    (None, 3)                 0
=================================================================
Total params: 370,915
Trainable params: 370,915
Non-trainable params: 0
```

4.2.2 Choix et optimisation des hyperparamètres

Les hyperparamètres suivants étaient fixes tout du long

- Nombre de frames empilés : 4
- Taille de l'image de sortie du jeu : 320x240
- Encodage de couleur des pixels : valeurs de gris
- Batch size (taille de l'échantillon issu de la mémoire) : 64
- Max tau (fréquence de mise à jour du réseau cible en nombre de pas, utilisé pour le fixed Q-targets) : 10000
- Nombre de pas d'échauffement (remplissage de la mémoire d'actions aléatoires pour avoir assez d'échantillons au début) : 50000
- Alpha (learning rate) : 0.00025
- Gamma (discount rate) : 0.95
- Epsilon au début de l'entraînement : 1
- Epsilon à la fin de l'entraînement : 0.01
- Nombre de pas pendant décroissance d'epsilon : 80000
- Epsilon utilisé pendant la phase d'évaluation : 0.001
- Nombre d'épisodes de test : 100

Hyperparamètres qui ont évolué et qui ont subi des optimisations

- Recadrage de l'image venant de l'environnement
  - D'abord l'image était coupée de 40 pixels en haut pour ne pas prendre en compte le plafond qui n'apportait pas d'informations utiles, 32 pixels en bas pour ne pas prendre en compte l'interface du jeu, 30 pixels à droite et à gauche
  - Finalement, il a été constaté que les pixels latéraux coupés empêchaient l'agent de voir les ennemis apparaissant du côté droite quand il visait l'ennemis tout à gauche, donc une configuration explorée était de ne pas couper ces pixels. Par contre si on veut avoir un contrôle plus fin sur la taille de l'image redimensionné tout en gardant le même ratio entre l'image recadrée et redimensionnée, il pourrait être judicieux de couper quelques pixels afin que le plus grand diviseur commun soit maximal entre le nombre de pixels vertical et horizontal de l'image recadrée (Par exemple si on coupe 40 pixels en haut, 32 en bas, et 6 à gauche et droite on a une résolution de 308x168 et le plus grand diviseur commun entre ces deux nombre est 28, donc on peut avoir n'importe quelle résolution qui soit un multiple de 11x6 tout en gardant le même ratio entre largeur et hauteur)
- Taille de l'image redimensionné nourrissant le réseau de neurones
  - Avec la première configuration de recadrage : 78x51, avec celle-ci l'agent avait périodiquement des comportements inadéquats où il se tournait face au mur ou à un coin au bout d'un moment et se faisait tuer, une hypothèse était que la résolution de l'image redimensionné était trop basse et empêchait l'agent de percevoir correctement l'environnement dans certains cas
  - Avec le seconde configuration de recadrage : 160x84, les comportements inadéquats de l'agent avaient disparu et les résultats se sont améliorés, mais le temps d'entraînement est devenu extrêmement grand (24h pour 500 épisodes alors qu'avec la configuration d'avant il lui fallait 4h pour 750 épisodes)
- Fonction de décroissance d'epsilon
  - D'abord, la fonction de décroissance était une exponentielle négative
  - Finalement la fonction de décroissance était linéaire, cette dernière semblait donner de meilleurs résultats
- Fonction de score (expliqué dans la prochaine section)
- Taille de la mémoire : quelques changement ont été tentés, mais n'ont pas influencé de façon significative les résultats, la dernière taille retenue était 250000 frames

— Utilisation du dueling network : sa présence ou non n'avait pas beaucoup d'impact sur le résultat (même si avec, le résultat était mieux), d'ailleurs à partir d'un moment du projet, la booléen l'activant a été oublié d'être remis à vrai, ce qui n'a pas empêché d'avoir des résultats convaincants par la suite (et ce qui importait dans l'exploration était la différence sur le score induite par un seul hyperparamètre à la fois, donc les conclusions tirés de la modification des autres hyperparamètres restent quand même cohérents)

— Nombre d'épisodes d'entraînement : avec un bon choix des hyperparamètres précédents, on pouvait se permettre de réduire le nombre d'épisodes d'entraînement pour avoir de bons résultats

4.2.3 Fonction de score

Par défaut, le scénario donne +1 pour chaque ennemi tué et -1 quand l'agent meurt.

Ensuite, a été ajouté un score de -0.1 pour chaque balle tirée et à chaque fois que l'agent se fait toucher pour les mêmes raisons qu'expliqué dans le chapitre concernant la comparaison de keras-rl et dopamine, c'est à dire pour encourager l'agent à économiser ses munitions et éviter de se faire toucher.

Finalement, vu les spécificités du scénario qui font en sorte que les ennemis réapparaissent avec plus de points de vie qu'avant après s'être fait tué, quand l'agent leurs tire dessus sans les tuer cela ne lui apporte aucune récompense donc il aura peut être moins tendance à continuer de leurs tirer dessus vu que la récompense immédiate est moindre.

Pour pallier à cela, la fonction de score a été modifiée de sorte à ce que l'agent reçoit une recompense de +1 à chaque fois qu'il tire et touche un ennemi, qu'il le tue ou pas.

Résumé :

1. +1 par ennemi tué par l'agent
2. +1 par ennemi touché par l'agent
3. -0.1 par balle tirée par l'agent
4. -0.1 par perte de vie de l'agent
5. -1 quand l'agent meurt

L'amélioration de la fonction de score avait significativement amélioré les résultats, surtout la durée de vie de l'agent

4.2.4 Résultats

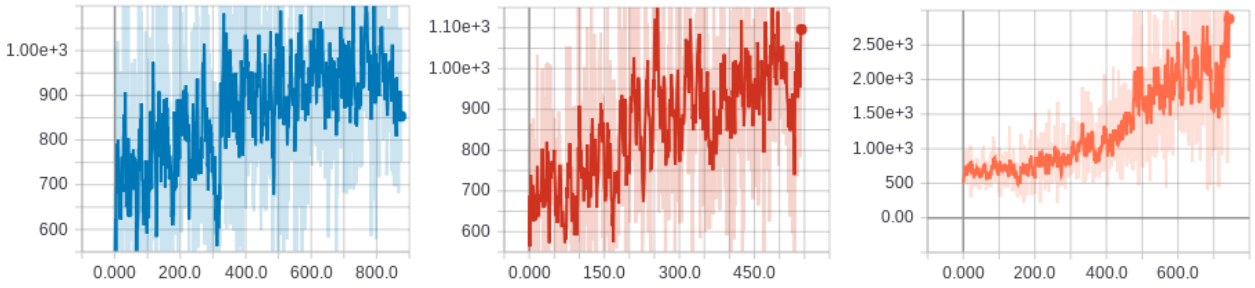


FIGURE 4.2 – Axe x : numéro de l'épisode, axe y : nombre de steps survécu pendant cet épisode
A gauche : version avec la taille de l'image d'entrée du réseau de neurones à 78x51
Milieu : l'image d'entrée est de dimension 160x84
Droite : les ennemis meurent en un tir (l'image d'entrée est de dimension 78x51)

Au travers de différentes phases d'explorations, l'agent entraîné s'est de plus en plus perfectionné.

Une version modifie du .wad a été utilisée pour tester un modèle où les ennemis sont toujours tuables avec un tir, et l'agent survit jusqu'à ce qu'il n'a plus de munitions (il en a 200 au début), ce qui est une façon de prouver l'efficacité du modèle entraîné.

Avec seulement les pixels à l'écran et sans les variables fourni par ViZDoom, un résultat autant bien n'aurait pas pu être possible autant facilement, il aurait fallu analyser les pixels venant de l'interface du jeu pour récupérer le nombre de munitions et les points de vie. Pour ce qui est de savoir si un ennemi est touché, il n'y a aucune information visuelle donc ceci ne serait simplement pas possible (par contre quand un ennemi meurt il y a bien une animation distinctive).

4.3 FlappyBird

4.3.1 Réseau de neurones

Les données nourrissant le réseau de neurones sont représentés par un vecteur à 8 composantes normalisé, issu d'un dictionnaire fourni par l'environment PLE qui est le suivant :

1. position en Y du joueur
2. vitesse du joueur
3. distance du joueur jusqu'au prochain tuyau
4. position en Y de l'ouverture supérieure du prochain tuyau
5. position en Y de l'ouverture inférieure du prochain tuyau
6. distance du joueur jusqu'au après-prochain tuyau
7. position en Y de l'ouverture supérieure du après-prochain tuyau
8. position en Y de l'ouverture inférieure du après-prochain tuyau

L'architecture du réseau de neurones a été construite de façon empirique, notamment en lisant un guide expliquant combien de coches/neurones il faut pour les couches intermédiaires[2,].

L'idée retenue était qu'il fallait incrémenter au début et décrémenter à la fin le nombre de neurones par étage

Voici le résultat de la fonction `model.summary()` appliqué sur le modèle passé à keras-rl :

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 4, 128)            1152
_________________________________________________________________
dense_2 (Dense)              (None, 4, 256)            33024
_________________________________________________________________
dense_3 (Dense)              (None, 4, 512)            131584
_________________________________________________________________
dense_4 (Dense)              (None, 4, 256)            131328
_________________________________________________________________
dense_5 (Dense)              (None, 4, 128)            32896
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 1026
=================================================================
Total params: 331,010
Trainable params: 331,010
Non-trainable params: 0
```

4.3.2 Choix et optimisation des hyperparamètres

Les hyperparamètres suivants étaient fixes tout du long

- Nombre d'empilement de l'entrée : 4
- Taille de la mémoire : 1000000
- Nombre de pas d'échauffement : 100000
- Présence du dueling network
- Nombre d'épisodes : 15000
- Taille d'échantillon pris dans la mémoire : 32
- Nombre d'épisodes de test : 100
- tau maximal (fréquence de mise à jour du réseau cible) : 10000
- Alpha (learning rate) : 0.01
- Gamma (discount rate) : 0.95
- Epsilon au début : 0.15 car le nombre d'actions possibles à explorer est limité (sauter ou ne pas sauter)
- Epsilon à la fin 0.01
- Epsilon pendant phase d'évaluation : 0.001
- Nombre de pas pendant réduction de l'epsilon : 150000

Hyperparamètres qui ont évolué et qui ont subi des optimisations

- Réseau de neurones
  - Au début, le réseau de neurones a été copié d'un exemple présent dans le github de PLE, utilisé pour le jeu "Catcher", mais il a été jugé qu'il comportait trop peu de couches
  - Finalement, le nouveau réseau de neurones utilisait le même nombre de neurones maximales que celui précédemment (512) qui constituait la couche "au milieu" du nouveau réseau et était entouré de couches qui montaient vers ce nombre avant et descendaient après (voir le tableau dans la section au dessus)
- Fonction de score : expliqué dans la section suivante

4.3.3 Fonction de score

Par défaut, la fonction de score proposée par le scénario pénalise l'agent de 5 points s'il perds et le récompense d'un point par tuyau qu'il arrive à passer

Mais avec cette fonction, c'était difficile pour l'agent de corriger ses erreurs après une mort, donc il a été décidé de le récompenser même s'il meurt écrasé contre un tuyau.

Vu que le but est de passer au mieux chaque tuyau, il devrait "viser" le milieu de chaque ouverture verticale.

Grâce aux données numériques fournis de l'environnement, il a été possible de concevoir la fonction de score suivante :

input : nextPipeBottomY, nextPipeTopY, playerPositionY, height
output: reward
gapMiddleYPosition ← (nextPipeBottomY-nextPipeTopY)/2+nextPipeTopY;
if *playerPositionY<gapMiddleYPosition* **then**
  return playerPositionY/gapMiddleYPosition
else
  trueHeight ← height*0.79;
  return 1-(playerPositionY-gapMiddleYPosition)/(trueHeight-gapMiddleYPosition)
end

Algorithm 1: Fonction de score incitant l'agent de viser le milieu de l'ouverture verticale entre tuyaux

Remarques :

— nextPipeBottomY, nextPipeTopY, playerPositionY sont fournis par la représentation de l'environnement à chaque step, height est un attribut de l'objet représentant l'environnement
— l'axe Y est tourné vers le bas
— height représente la hauteur de l'image avec le sol inclus
— trueHeight est la hauteur maximale que peut atteindre l'agent (la limite supérieure du sol)
— le nombre 0.79 était trouvé dans le code source de l'environnement (Donc que le 79% de la hauteur de l'image du jeu est atteignable par l'agent, le 21% restants est le sol en bas de l'image)

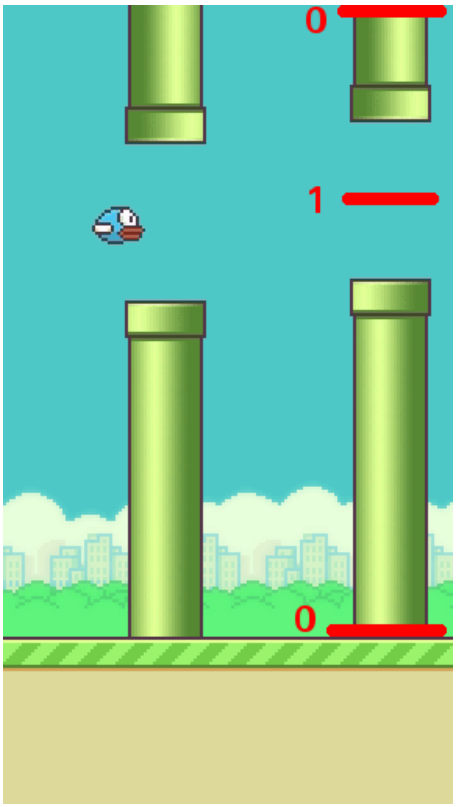


FIGURE 4.3 – Représentation visuelle du score qu'aurait gagné l'agent s'il perds à cause d'une collision avec le prochain tuyau (l'intervalle en haut et en bas sont linéaires et leurs linéarité est différente pour le bas et le haut)

4.3.4 Résultats

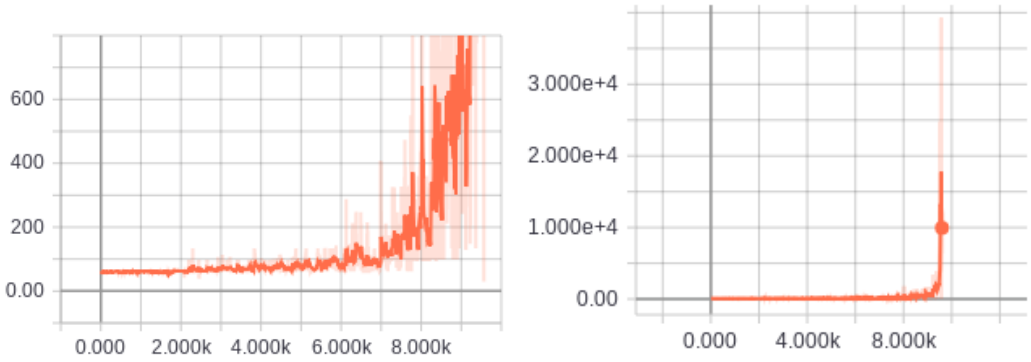


FIGURE 4.4 – Axe x : numéro de l'épisode, axe y : nombre de steps survécu pendant cet épisode
A gauche : Echelle du graphique permettant de voir l'évolution de la durée des épisodes
A Droite : Echelle du graphique permettant de voir la valeur maximale ($\sim$ 39200) et à quel point le modèle est devenu efficace

Donc avoir accès à des données numériques fourni par l'environnement est un atout considérable car on peut utiliser ces données pour diriger l'agent vers une solution souhaitée, ce qui ne serait pas possible si on n'avait accès qu'aux pixels du jeu.

Finalement on peut considérer que l'agent a "gagné" le jeu, vu qu'il arrive à faire des épisodes durant plus qu'une heure avant de perdre, ce résultat n'aurait pas pu être possible en analysant les pixels, ou alors après une durée d'entraînement beaucoup trop conséquente par rapport à la méthode actuelle.

Chapitre 5

Conclusion

Les modèles entraînés dans les deux jeux ont atteint des résultats très convaincants, surtout pour flappy bird.

D'autant bons résultats n'auraient pas pu être atteint avec autant d'aisance relative sans techniques de deep Q-learning pour flappy bird et n'aurait pas pu être possible du tout dans le cas de Doom.

L'hyperparamètre le plus important est sans doute la fonction de score, mais pour la définir on a besoin de variables issus d'un environnement qui étaient dans notre cas mises à disposition exprès pour être exploité dans un cadre de Reinforcement Learning.

Mais la question qu'on pourrait se poser est : est-ce que le Reinforcement Learning serait une solution miracle pour entraîner des modèles dans des simulations et ensuite les appliquer dans la vraie vie ?

La réponse est nuancée, pour avoir des bons résultats il faut donc avoir accès à des données pas forcément mises à disposition dans n'importe quel environnement.

Dans une simulation il est relativement facile d'avoir accès à ces données, pour autant qu'on aie accès au code source de la simulation ou une API nous fournissant des données utiles.

Mais pour appliquer ensuite ce modèle entraîné dans la vraie vie, il faudrait mettre en place divers capteurs mesurant des données utiles à notre modèle, une simple caméra nous fournissant des pixels ne suffirait pas d'avoir des résultats convaincants autant facilement qu'avec les méthodes utilisées dans le cadre de ce projet.

Donc non, ce n'est pas une solution miracle car il faut un travail d'adaptation conséquent selon avec quel environnement on veut interfacer notre agent. De plus, il faudrait avoir accès aux mêmes données que celles utilisées dans l'entraînement du modèle si on voudrait l'appliquer en dehors d'un simulateur.

Apport personnel grâce au projet

Ce projet m'a permis d'acquérir/améliorer les compétences suivantes, en plus de ce qui touche au Reinforcement Learning :

- Découverte de quelques astuces en python et de nouveaux modules comme numpy
- L'étude en détail et la compréhension des codes sources sur github en essayant de comprendre comment il marche (surtout pendant la phase de comparaison dopamine vs keras-rl)
- Tout ce qui touche aux réseaux de neurones en général, CNN, tensorflow, keras, tensorboard
- La mesure et l'interpretation de données d'expériences
- Quelques fonctionnalités et modules de LaTeX

Chapitre 6

Annexes

6.1 Manuel d'utilisation

Installation

— navigate to `src` folder
— `python3 -m venv venv_rl`
— `source venv_rl/bin/activate`
— `pip install -r requirements.txt`
— All the commands in this user guide must be executed in the *src* folder (where *doom.py* and *flappybird.py* are located)

Tensorboard logs location

They are located in $xxx\_saves/yyy/logs/zzz$

— xxx is either "doom" or "flappybird"
— yyy is the experiment name
— zzz is either "train" or "test$N$" with N empty or a number

Model weights location

they are located in $xxx\_saves/yyy/weights.h5f$

— xxx is either "doom" or "flappybird"
— yyy is the experiment name

Doom (doom.py)

Command line arguments

All those arguments have default values except "weights"

- mode : "test" "train" or "watch", if set to "watch" or "test" and no weights argument specified, it will use the last trained model (default : train)
- weights : path to trained model weights file (*.h5f* format) (only test and watch mode, default : none)
- step-sleep : number of ms to sleep between each step (useful to observe for a human the agent playing, default : 0)
- test-episodes : number of episodes to test (only test and watch mode, default : 100)
- train-episodes : number of episodes to train (only train mode, default : 780)
- warmup-steps : number of steps for warmup (only train mode, default : 50000)
- wad : path to wad file containing doom scenario (default : *doom_scenarios/defend_the_line.wad*)
- image-shape : shape of the image to be put in the neural network, 78x51 or 160x84 (default : 78x51)

How to watch interesting trained models at 60 fps without tensorboard logging by loading weight file

- One shot ennemy kill with 160x84 image shape :
- `./doom.py --mode=watch --weights=doom_saves/160x84/weights.h5f --step-sleep=15 --wad=doom_scenarios/defend_the_line-oneShotKill.wad --image-shape=160x84`
- one shot ennemy kill with 76x51 image shape (sometimes the agent last longer than in the 160x84 version and sometimes his behaviour is messy) :
- `./doom.py --mode=watch --weights=doom_saves/oneShotKill/weights.h5f --step-sleep=15 --wad=doom_scenarios/defend_the_line-oneShotKill.wad --image-shape=78x51`
- normal scenario with 160x84 image shape :
- `./doom.py --mode=watch --weights=doom_saves/160x84/weights.h5f --step-sleep=15 --image-shape=160x84`
- normal scenario with 76x51 image shape :
- `./doom.py --mode=watch --weights=doom_saves/78x51/weights.h5f --step-sleep=15 --image-shape=78x51`

How to train + test a new model

- creates a new folder named $doom\_saves/\{saveFolder\} - \{x\}$ where x is a number and *saveFolder* "78x51" or "160x84"
- example : 78x51 image size, normal scenario, 100 train episodes, 20000 warmup steps, 20 test episodes :
- `./doom.py --mode=train --train-episodes=100 --warmup-steps=20000 --test-episodes=20 --image-shape=78x51`

Re test previously trained model (from previous example)

- creates a new folder named $doom\_saves/\{saveFolder\}/logs/test\{x\}$ folder where x is a number and saveFolder "78x51-1" for example
- example : re test previous trained model :
- `./doom.py --mode=test --test-episodes=20 --image-shape=78x51`

Test a model by loading weight file and log test

- creates a new folder named $doom\_saves/\{saveFolder\}/logs/test\{x\}$ folder where x is a number and saveFolder "78x51-1" for example, the saveFolder is parsed from the weight file path (if the weight file is located at $foo/toto/tutu/weights.h5f$, saveFolder : $foo/toto/tutu$)
- example : load and test previously trained model "$doom\_saves/78x51-1/weights.h5f$"
- `./doom.py --mode=test --test-episodes=2 --image-shape=78x51 --weights=doom_saves/78x51-1/weights.h5f`

Flappy bird (flappybird.py)

Command line arguments

All those arguments have default values except "weights"

- mode : "test" "train" or "watch", if set to "watch" or "test" and no weights argument specified, it will use the last trained model (default : train)
- weights : path to trained model weights file ($.h5f$ format) (only test and watch mode, default : none)
- step-sleep : number of ms to sleep between each step (useful to observe for a human the agent playing, default : 0)
- test-episodes : number of episodes to test (only test and watch mode, default : 100)
- train-episodes : number of episodes to train (only train mode, default : 16700)
- warmup-steps : number of steps for warmup (only train mode, default : 100000)
- render : render the graphics of the game or not, if set to "0" do not render, if set to anything else, render

How to watch interesting trained models at 60 fps without tensorboard logging by loading weight file

- To test the model that "won" the game :
- `./flappybird.py --mode=watch --step-sleep=15 --weights=flappybird_saves/best/weights.h5f`

How to train + test a new model

- creates a new folder named $flappybird\_saves/experiment-\{x\}$ where x is a number, "$flappybird\_saves/experiment-1$" for example
- `./flappybird.py --mode=train --test-episodes=50 --train-episodes=250 --warmup-steps=1000`

Re test previously trained model (from previous example)

- creates a new folder named $flappybord\_saves/experiment-\{y\}/logs/test\{x\}$ folder where x and y are numbers, "$flappybird\_saves/experiment-1/logs/test2$" for example
- `./flappybird.py --mode=test --test-episodes=50`

Test a model by loading weight file and log test

- creates a new folder named $flappybird\_saves/\{saveFolder\}/logs/test\{x\}$ folder where x is a number and saveFolder : "best" for example, the saveFolder is parsed from the weight file path (if the weight file is located at $foo/toto/tutu/weights.h5f$, saveFolder : $foo/toto/tutu$)
- `./flappybird.py --mode=test --weights=flappybird_saves/experiment-1/weights.h5f`

6.2 Cahier des charges

Filière informatique

haute école neuchâtel berne jura arc ingénierie saint-imier le locle delémont

Cahier des charges pour travail de semestre

Titre:	**ReinforcementLearning**
N° projet:	**820**
Etudiant:	**Sergiy Goloviatinski**
Professeur:	**Stefano Carrino**

Situation initiale

Le *Reinforcement Learning* (RL) est sur le point de révolutionner le domaine de l'IA et représente une étape vers la construction de systèmes autonomes avec une meilleure compréhension du monde. Le RL a beaucoup fait parler de soi en 2016 quand l'algorithme développé par DeepMind a battu le champion du monde dans le jeu du Go.

Actuellement, le deep learning permet au reinforcement learning (cela donne le "Deep Reinforcement Learning" ou DRL) d'évoluer vers des problèmes auparavant intraitables, comme apprendre à jouer à des jeux vidéo directement à partir de pixels ou être appliqué en robotique, pour le contrôle de robots complexes.

Buts du projet:

Ce projet a pour objectif de:

- Prendre en main des solutions de (deep) Reinforcement Learning
- Développer un module Python de Reinforcement Learning avec des approches de (deep) Q-Learning
- Évaluer les résultats obtenus en les comparants avec des "approches traditionnelles"
- Optionnel – implémenter un deuxième algorithme de reinforcement learning (*Agent-Critique*)

Démarche proposée:

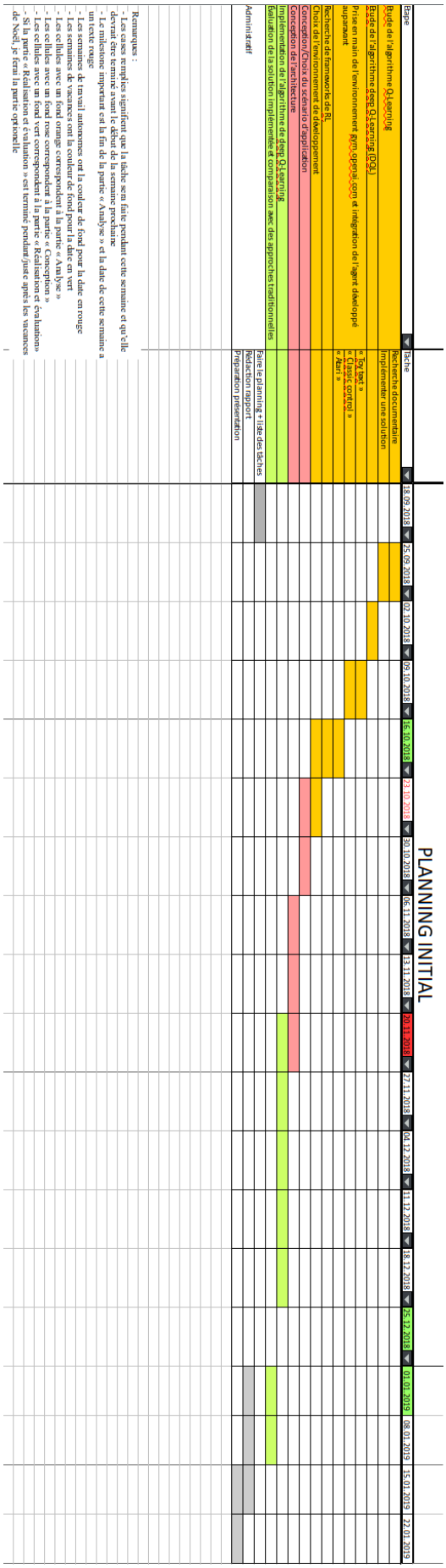
- Analyse :
  - Etude de l'algorithme Q-Learning et implémentation
  - Etude de l'algorithme deep Q-Learning (DQL)
  - Prise en main de l'environnement gym.openai.com en implémentant des exemples
  - Recherche de frameworks de RL
  - Choix de l'environnement de développement (milestone !)
- Conception :
  - Conception/Choix du scénario d'application
  - Conception de l'architecture
- Réalisation et évaluation
  - Implémentation de l'algorithme de deep Q-Learning
  - Évaluation de la solution implémentée et comparaison avec des approches traditionnelles

Contraintes:

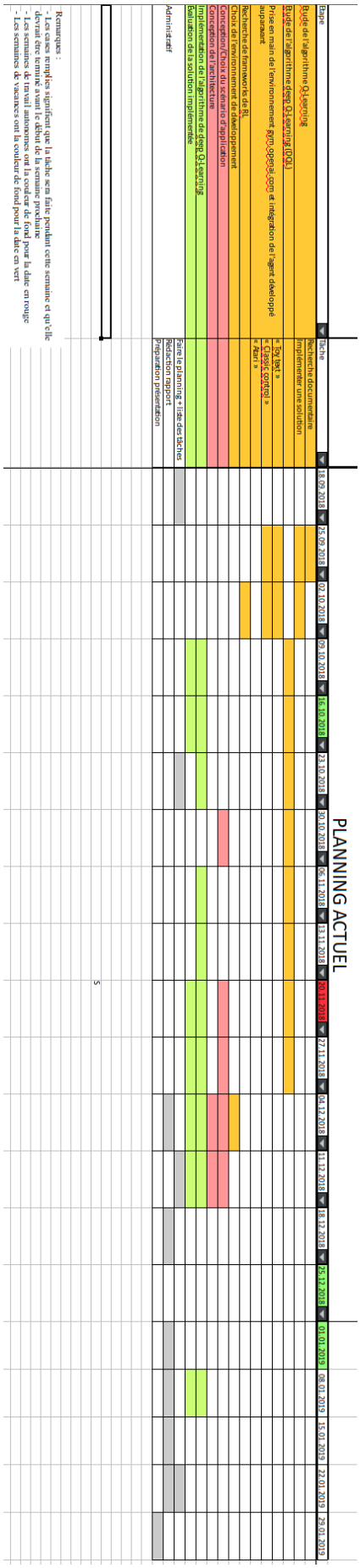
-

Les directives de travail sont détaillées dans le document annexe.

6.3 Planning initial



6.4 Planning final



6.5 Journal de laboratoire

Total d'heures travaillées : 153h

```
# Semaine 15
## 25.01.09 ~5h
- Redaction du rapport
## 24.01.19 ~8h
- Redaction du rapport
- Refactorisation du code
## 22.01.19 ~6h
- Redaction du rapport
- Refactorisation du code
- Mise a jour du planning
# Semaine 14
## 21.01.19 ~5h
- Redaction du rapport
- Refactorisation du code
## 20.01.19 ~5h
- Redaction du rapport
- Calcul du nombre d'heures travailles jusqu'a aujourd'hui: 124h
## 19.01.19 ~8h
- Redaction du rapport
## 17.01.19 ~2h
- Redaction du rapport
## 16.01.19 ~1h
- Redaction du rapport
## 15.01.19 ~2h
- Redaction du rapport
# Semaine 13

## 14.01.19 ~2h
- Redaction du rapport
- Amelioration du code
- Re test scenario avec nouvelle methode de test ou les ennemis ont toujours 1 point de vie:
    - test sur 100 episodes
    - score moyen: 50.34
    - nb step episode moyen: 1450

## 09.01.19 ~2h
- Tentative d'experience en augmentant le nombre de pixels ((51, 78) => (84, 160))
    - Entrainement beaucoup trop long (23h pour 544 episodes...)
    - Observation: l'agent n'a pas de problemes du genre "se tourne vers le mur et ne fait rien"
        ou quand il "reste coince"
    - test sur 100 episodes
        - nb step episode moyen: 835
        - score moyen: 61.5



## 08.01.19 ~3h
- Creation d'une classe tensorboard pour le test, meme que celle de base de keras mais qui ecrit
    aussi les moyennes a la fin de l'entrainement
- Gestion d'arguments supplementaires depuis la ligne de commande (nb episodes test + train, nb
    steps warmup, sleep time entre chaque step)
- Resultat experience ou la variable hitcount est disponible et les ennemis qui voient leur vie
    augmenter a chaque spawn:
    - 1574 episodes de train pour 15h
    - Observation: des fois l'agent se tourne vers un mur/coin derriere lui et ne fait rien
    - test sur 100 episodes
        - nb step episode moyen: 732
        - score moyen: 45.5

# Semaine 12
```

```
## 07.01.19 ~1h
- En fait c'est plus coherent de comparer les stepCount donc j'implemente une moyenne de
    stepCount dans la methode test

## 06.01.19 ~3h
- Redaction de la structure du rapport
    (https://forge.ing.he-arc.ch/projects/he-arc-inf-1718/wiki/Directives_TA)
- Mise a jour des binaries windows pour utiliser la derniere version de vizdoom, donc la variable
    contenant le compteur de coups touches est disponible
    - Entrainement avec "ancienne" version du .wad ou les ennemis ont leur points de vie qui
        augmentent a chaque respawn


## 18.12.18 ~1h
- Lien par rapport a RL asynchrone: https://arxiv.org/pdf/1602.01783.pdf
- Mise en place structure rapport
    - Utilisation package latex pour ecrire des algos
        https://en.wikibooks.org/wiki/LaTeX/Algorithms
- Petite discussion avec thibault: finalement ils ont fait le choix de rendre le simulateur
    synchrone, et ils vont me redire quand leurs code sera pret

# Semaine 11

## 17.12.18 ~30 min
- Mise a jour du planning

## 15.12.18 ~2h
- Finalement la solution que j'ai retenu est de modifier le script acs du .wad pour que les
    ennemis ne voient pas leurs sante augmenter a chaque nouveau respwan
- Entrainement de la version dqn+ de keras-rl avec .wad modifie:
    - J'ai arrete l'entrainement au bout de 11 heures car les episodes devenaient trop longs
    - Score moyen pour 100 episodes de test: 66.78 avec certains episodes qui ont un score negatif
        (l'agent se tourne contre le mur) et d'autres avec un score de 148 (l'agent a vide toutes
        les munitions)

## 14.12.18 ~4h
- Modification du .wad pour mettre l'ammo de depart a 200
- Probleme: sur la version windows de vizdoom, le package pip n'est pas disponible et les bin
    precompiles sont en version 1.1.5pre
    - Tentative de compiler sous windows pour la version 1.1.6
        - Trop complique
    - Tentative de modifier le script acs dans le .wad pour augmenter le reward a chaque fois
        qu'on touche un ennmi
        - Trop complique

## 11.12.18 ~2h
- Modification de fonction de recompense pour recompenser agent a chaque coup tire sur ennemi
- TODO: modifier le wad sur ma machine windows pour mettre l'ammo de depart a 200 (sur mon linux
    ca a l'air de bug)
- Thibault va mettre sur un google docs les input/output qu'ils ont besoin pour le simulateur

# Semaine 10

## 10.12.18 ~1h
- Mise en place structure rapport
- Re test flappy-bird avec dopamine sur 60k episodes:

## 09.12.18 ~1h
- Re test keras-rl dqn+ avec nouveau neural network et fonction de score, pour 15k episodes:
    - Il n'arrive pas a atteindre 15k episodes car ils durent trop long maintenant, du coup il ne
        perds jamais donc c'est excellent

## 08.12.18 ~2h
- Test keras-rl+ non visuel avec nouveau neural network:
    -  1h16 de train pour un score de -4.29 (100 test ep)
- Test keras-rl+ non visuel avec nouveau neural network + nouveau systeme de reward a 5k episodes:
```

```
    -  1h16 de train pour un score de -3.16 (100 test ep)
- Test keras-rl+ non visuel avec nouveau neural network + nouveau systeme de reward a 10k
    episodes:
    - 3h4 de train pour un score de 3.56 (100 test ep)
- Test keras-rl+ non visuel avec nouveau neural network + nouveau systeme de reward a 15k
    episodes:
    - 3h15 de train pour un score de -3.58 (100 ep test) => bizare, l'experience precedante etait
        "de la chance"
- Ajout de https://github.com/google/dopamine/pull/54 au code de dopamine
- Implementation du nouveau neural network + reward dans dopamine
    - Test a 10k episodes: 29m de train pour un score de -4.99


## 05.12.18 ~2h
- Implementation du reward base sur distance verticale entre game over du player et pipe le plus
    proche

## 04.12.18 ~2h
- Recherche sur reseaux de neurones afin d'ameliorer celui du flappy bird :
    https://towardsdatascience.com/beginners-ask-how-many-hidden-layers-neurons-to-use-in-artificial-neural-net
- Modification du reseau de neurones de l'agent keras-rl de flappy bird
- Ajout d'une recompense pour chaque frame "survecue" dans flappy bird

# Semaine 9

## 03.12.18 ~2h
- Fin implem flappybird sur dopamine et train+test: -4.73 pour 41m
    - re test flappybird sur dopamine avec pretrain_length a 100k et epsilon_decay_steps a 150k:
        -5.0 ???
    - re test flappybird sur keras-rl avec memes param: -4.64 pour 1h45
- Re test vanilla dqn: 14.26 pour keras-rl vs 6.027 pour dopamine

## 01.12.18 ~3h
- Suite et fin implem flappybird non visuel sur keras-rl (dqn+)
- Re test vanilla dqn pour keras-rl: 2h14 train pour 750 ep avec nouveau sys de score: 1.465
- Debut implem flappybird non visuel sur dopamine (implicite quantile)
- Test flappybird pour 10'000 eps avec kerasl-rl dqn+: score moyen -4.83

## 30.11.18 ~1h
- Re test vanilla dqn pour dopamine: 1h29 de train pour 750 ep avec nouveau sys de score: 6.151

## 29.11.18 ~2h
- Implem flappy bird avec keras-rl

## 27.11.18 ~2h
- Amelioration/nettoyage/optimisation code dqn+ keras-rl
- Recherches environnements sans pixels :
    https://pygame-learning-environment.readthedocs.io/en/latest/user/games/flappybird.html ou un
    autre
    - exemple avec keras non visuel:
        https://github.com/ntasfi/PyGame-Learning-Environment/blob/master/examples/keras_nonvis.py
        => adapter dqn+ agent (changer interface env + convnet)

# Semaine 8

## 26.11.18 ~3h
- Redaction tableau comparatif frameworks
- Implementation du PER pour implicit quantile et train + test => pas mieux
- Modifications des hpyerparametres pour coller a 100% a la version dqn+ de keras-rl => pas mieux
- Test sur scenario "defend the center" avec implicit quantile => score de -1.373 pour 33m de
    train (l'agent avait adopte une strategie "paresseuse" et faisait que tourner en rond sans
    tirer)
- Test sur scenario "defend the center" avec dqn+ de keras-rl => 4.521 de moyenne pour 2h34 de
    train
- Idee: tester si la difference de score et de temps vient vraiment de la presence des dueling
    dqn dans keras-rl
```

```
## 25.11.18 ~5h
- Changement gamma ‘0.99‘-> ‘0.95‘ pour rainbow (avec nouveau systeme de score):
    - 750 ep de train, 5.615 de moyenne pour 57 minutes de train
- Train + test keras-rl vanilla_dqn avec nouveau systeme de score:
    - 750 ep de train, 4.04 de moyenne pour 2h7 de train
        - observations: l’agent fait n’importe quoi, se tourne direction le mur gauche et tire
            dans le vide
- Train + test keras-rl improved_dqn avec nouveau systeme de score:
    - pour 3h52, score moyen 16.837
- implem implcit quantile
    - version avec network par defaut sur 750 episodes:
        - 53m de train: 5.798
    - version avec meme network que les dqn+ keras-rl sur 750*3 episodes:
        pour 2h27 5.912
- train + test keras-rl dqn+ pour 750*3 episodes:
    pour 12h, 8.38

## 24.11.18 ~2h
- Changement pour scenario: perdre de la vie et utiliser des munitions diminue de 0.1 la
    recompense + les munitions ne sont plus infinies mais on commence un episode avec 100
    munitions
    - moyenne de 5.747 pour 750 episodes et 47m de train
    - constatation:
        - sur les graph de comparaison du papier scientifique parlant du rainbow, on voit pas bien
            a partir de combien de frame il surpasse les autres (le 1er repere est apres 7
            millions de frames) mais avec 750 episodes on fait que 428k frames => essayer de faire
            tourner beaucoup plus longtemps?
        - j’ai l’impression que l’agent "n’ose pas trop" tourner a droite et gauche et attend que
            les ennemis passent devant lui pour tirer => bloque dans maximum local?
    - du coup faudrait refaire le test pour keras-rl (autant version vanilla dqn que dqn+) avec ce
        nouveau systeme de recompense
- Changement hyper-param pour rainbow: ‘vmax‘ ‘10‘->‘20‘, (‘vmax: float, the value distribution
    support is [-vmax, vmax].‘)
    - moyenne de 6.25 pour 750 episodes et 48m de train (1ere iteration, ~428k frames)
    - 2eme iteration: 5.684 pour 750+750 episodes et 53 minutes (~841.5-428= 413 k frames)

## 23.11.18 ~1h
- train + test du scenario avec rainbow
    - apres 750 et meme apres 1500 episodes le score moyen pour la phase de test est autour de
        10...
    - faut essayer de tweaker les hyper-parametres demain

## 21.11.18 ~3h
- implem vanilla dqn sur dopamine
- train et test de vanilla dqn sur dopamine
    - observation pendant l’entrainement: le voyant "fan stop" de ma carte graphique est eteint,
        ce qui veut dire que par defaut dopamine exploite mieux la carte graphique ?
        - en fait oui vu qu’on specifie explicitement le "tf_device"
    - pour 2h de train: moyenne de score sur 100 test episodes a 10.84, en regardant la courbe de
        score on apercoit qu’elle est aussi bruitee et "peine a decoller"
- implem scenario avec rainbow de dopamine

## 20.11.18 ~1h
- train + test exemple freecodecamp
- resultat train 750 episodes
    - pour 9h19 de train : moyenne score sur 100 test episodes a 3.19 ... vraiment nul mais c’est
        bizare car la courbe du score a l’air de monter bien plus haut pendant l’entrainement
        (mais elle est aussi "bruitee") => peut etre qu’en faisant des actions random l’agent est
        plus efficace qu’apres entrainement ?

## 19.11.18 ~2h
- adaptation de keras-rl pour dqn vanilla
- resultat train 500 episodes:
    - pour 1h26 de train: moyenne de score sur 100 test episodes a 5.04 ... c’est tres bas et plus
        rapide qu’avant, du coup je retest avec 750 episodes
```

```
- resultat train 750 episodes:
    - pour 2h de train: moyenne de score sur 100 test episodes a 13.97, c'est deja mieux par
        contre la courbe du score ne monte pas du tout d'une facon prevue...

# Semaine 7

## 10.11.18 ~5h
- a mettre dans comparatif frameworks: dopamine est moins evident a utiliser vu qu'il utilise tf
    et pas keras
- Continuation d'implem sur dopamine, c'est presque bon il y a juste pas de double dqn + dueling
    dqn dans le dqn_agent (ils sont tous dans le rainbow), faut voir ce que ca donne quand meme
- TO-DO: ecrire le score apres chaque episode dans tensorboard (surtout trouver comment integrer
    ca "facilement")
- Idee:
  - comparer les 3 methodes avec version DQN "vanilla" (sans PER, ni dueling DQN ni double DQN,
      ni fixed targets) => ca serait plus facile de "downgrader" les 2 solutions precedentes
      plutot que d'implementer le double dqn + dueling dqn dans dqn_agent de chez dopamine (faut
      vraiment aller "grailler" profondement, j'ai peur de tout casser) => important pour avoir
      une comparaison equivalente
  - Se lancer aussi dans implem avec rainbow agent du meme scenario pour voir a quel point il est
      superieur aux 2 autres solutions avec PER, dueling DQN, double DQN et fixed targets

## 09.11.18 ~3h
- Debut d'implem du scenario sur dopamine

## 08.11.18 ~1h
- Test de `keras-rl.py` avec la EpsGreedyQPolicy: score de 17.58 (500 ep train, 100 ep test) donc
    ca change pas enormement

## 06.11.18 ~2h
- Fix de l'implem `keras-rl.py` pour y integrer une EpsGreedyQPolicy, sauf que pour faire
    descendre epsilon on a de disponible que une LinearAnnealedPolicy qui fait descendre un
    parametre de facon lineaire, et on a pas de Policy pour faire descendre un parametre de facon
    exponentielle comme dans l'autre exemple
- Lecture : https://arxiv.org/pdf/1710.02298.pdf (Article scientifique sur l'algo rainbow)

# Semaine 6

## 05.11.18 ~1h
- Entrainement de la version keras-rl
- Constatations:
  - keras-rl: pour 100 episodes de test: Score/nbEpisodes = 21.68 (500 episodes d'entrainement,
      duree: 2h40)
  - version freecodecamp: pour 100 episodes de test: Score/nbEpisodes = 11.01 (500 episodes
      d'entrainement, duree: 8h)
  - la version keras-rl est bien plus rapide pour l'entrainement et a de meilleurs scores
  - la seule chose negative c'est que j'ai du modifier le code pour qu'on puisse faire un train
      sur un nombre d'episodes (sinon c'etait que pour un nombre de steps), sinon l'ajout de
      callback et l'heritage des Processor pour faire l'interface entre l'agent et un
      environnement custom est pratique

## 04.11.18 ~5h
- Fin d'implementation du scenario sous keras-rl
- Entrainement de la version freecodecamp

## 02.11.18 ~1h
- Reduction de la taille des frames et "nettoyage" du code implemente venant de freecodecamp

## 01.11.18 ~1h
- Ecriture du script bash pour installer diverses dependances sur le serveur de DL de l'ecole
- Test du script bash sur VM Ubuntu 18.04 LTS

## 30.10.18 ~2h
- lecture de https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf qui est un document qui a inspire
    l'implementation de l'exemple `dqn_atari.py` duquel je vais m'inspirer a mon tour pour
    implementer le scenario doom sur keras-rl
```

```
- debut d'implementation du scenario avec keras-rl

# Semaine 5

## 29.10.18 ~2h
- Changement de scenario, passage de "deadly corridor" a "defend the center" et adaptation de
    l'exemple de freecodecamp en consequence, j'ai des `CUDA_ERROR_OUT_OF_MEMORY` pour l'instant
    j'ai essaye de reduire le `pretrain_length` et `memory_size` (a 80'000 ca va bien)

## 28.10.18 ~5h
- Lecture partie 3+ du guide :
    https://medium.freecodecamp.org/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-r
- Implementation/adaptation de l'exemple du guide ci-dessus
- Ajout de quelques criteres pour comparer frameworks sur le google docs
- Debut d'adaptation de l'exemple pour keras-rl

## 26.10.18 ~1h
- Mise a jour du planning "reel" en se referant aux anciens JT/PV

## 23.10.18 ~2h
- Debut de redaction du document de comparaison de differents frameworks de RL, surtout le
    listage de differents criteres qu'on pourrait utiliser pour les comparer entre eux, sachant
    que certains criteres auraient du sens d'etres compares entre les 2 groupes alors que
    d'autres auraient du sens qu'on comparant avec le meme exemple
- Lecture de l'implementation en tf du dqn pour doom
- Installation des dependances pour vizdoom


# Semaine 4

## 20.10.18 ~5h
- Implementation, tests, "bidouillages" du code du 1er lien (DQN avec keras)
- Comparaison entre solutions "avant" et "apres" deep learning:
  - L'agent depasse largement la version d'avant au niveau du score, atteignant souvent (mais pas
      toujours) le maximum de 500
  - L'apprentissage prends beaucoup plus de temps (meme sur une machine utilisant tensorflow-gpu
      avec une GTX 1070), mais a partir de 2000 episodes d'entrainement l'agent atteint deja
      souvent le score maximum, avec 10000 episodes il l'atteint presque toujours
- Installation et tests de dopamine
  - Probleme: pas prevu pour Windows donc je n'ai pas acces a une machine puissante a la maison
  - Tests dopamine: le dqn agent basique fonctionne (```dopamine/agents/dqn/configs/dqn.gin```)
      mais je n'ai pas trouve comment recuperer le resultat de l'entrainement pour le voir "en
      live"/render

## 19.10.18 ~1h
- Installation de tensorflow-gpu + CUDA + cudnn sur machine windows fixe a la maison pour
    accelerer la phase d'apprentissage

## 17.10.18 ~1h
- continuation de lecture

## 09.10.18 ~1h
- lecture:
  - https://medium.com/@gtnjuvin/my-journey-into-deep-q-learning-with-keras-and-gym-3e779cc12762
  - https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8


# Semaine 3

## 08.10.18 ~2h30
- Recherche de framework de RL
- Implementation d'une methode permettant de "tatonner" pour savoir quelle valeur pour quelle
    hyperparametre prendre afin de maximiser le score

## 04.10.18 ~2h
- Tunage des hyperparametres
```

```
## 02.10.18 ~3h
- Tunage des hyperparametres
- Ajout d'une limite en terme d'episodes avant laquelle le param alpha et epsilon ne baissent pas
- Reecriture de la discretisation de l'espace continu de l'environnement


# Semaine 2

## 01.10.18 ~2h30
- fin implementation environnements continus pour classe QAgent
- tests et tunage de hyperparametres

## 28.09.18 ~1h
- debut d'implementation des environnements continus pour la classe QAgent notamment pour le cas
    "CartPole" de openai gym

## 27.09.18 ~1h30
- ecriture d'une classe "QAgent" pour environnements discrets + implem sauvegarde/chargement
    qtable

## 26.09.18 ~1h30
- implementation d'un exemple "taxi" sur openai gym


## 25.09.18 ~4h
- lecture:
  - https://medium.freecodecamp.org/an-introduction-to-reinforcement-learning-4339519de419
  -
      https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe

- debut d'implementation du Q-Learning avec l'exemple "CartPole" de openai gym
  - https://medium.com/@tuzzer/cart-pole-balancing-with-q-learning-b54c6068d947
  - https://ferdinand-muetsch.de/cartpole-with-qlearning-first-experiences-with-openai-gym.html

# Semaine 1

## 23.09.2018 ~30min
- Mise en page du planning + revision


## 21.09.2018 ~2h
- Redaction du planning
- prevision de la liste de taches
- precision du cahier de charges
- mise en place du projet sur la forge

## 18.09.2018 ~2h
- Redaction du planning
```

Bibliographie

[1] Andrey kurenkov's website. http://www.andreykurenkov.com. Accessed : 2019-01-25.

[2] Beginners ask "how many hidden layers/neurons to use in artificial neural networks?". https://towardsdatascience.com/beginners-ask-how-many-hidden-layers-neurons-to-use-in-artificial-neural-networks-51466afa0d3e. Accessed : 2019-01-25.

[3] Deep q learning with doom. https://gist.github.com/simoninithomas/7611db5d8a6f3edde269e18b97fa4d0c#file-deep-q-learning-with-doom-ipynb. Accessed : 2019-01-25.

[4] dopamine documentation. https://github.com/google/dopamine/tree/master/docs. Accessed : 2019-01-25.

[5] A free course in deep reinforcement learning from beginner to expert. https://simoninithomas.github.io/Deep_reinforcement_learning_Course/. Accessed : 2019-01-25.

[6] neuralnetworksanddeeplearning.com. http://neuralnetworksanddeeplearning.com. Accessed : 2019-01-25.

[7] Reinforcement learning - wikipedia. https://en.wikipedia.org/wiki/Reinforcement_learning. Accessed : 2019-01-25.

[8] VizDoom keras-rl. https://github.com/flyyufelix/ViZDoom-Keras-rl. Accessed : 2019-01-25.

[9] VizDoom scenarios. https://github.com/mwydmuch/ViZDoom/tree/master/scenarios#defend-the-line. Accessed : 2019-01-25.

[10] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. Rainbow : Combining improvements in deep reinforcement learning. *CoRR abs/1710.02298* (2017).

[11] Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. ViZDoom : A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games* (Santorini, Greece, Sep 2016), IEEE, pp. 341–348. The best paper award.

[12] Tasfi, N. Pygame learning environment. https://github.com/ntasfi/PyGame-Learning-Environment, 2016.

Table des figures